SPARK SQL

PREPARED BY RAJASEKHAR

Limitations of RDD

- **No input optimization engine**
  - There is no provision in RDD for automatic optimization. We can optimize RDD manually
- **Runtime type safety**
  - There is no **Static typing** and **run-time type safety** in RDD. It does not allow us to check error at the runtime.
- **Degrade when not enough memory**
  - The RDD degrades when there is not enough memory to store RDD **in-memory** or on disk.
- **Overhead of serialization & garbage collection**
  - Since the RDD are in-memory JVM object, it involves the overhead of **Garbage Collection** and **Java serialization** this is expensive when the data grows.
- **Handling structured data**
  - RDD does not provide schema view of data. It has no provision for handling structured data.

OVERVIEW

- Spark SQL is a module for Structured data processing
- Interfaces provided by Spark SQL provide spark with more information about the structure of both data and the computation being performed and this helps Spark SQL to perform extra optimizations
- Spark SQL can be interacted with DataFrame and Dataset API
- Spark SQL is used to execute SQLs by making use of Spark in-memory distributed processing
- Spark SQLs can be used to read data from existing Hive tables
- Spark SQL can also be interacted with command-line or JDBC/ODBC

In Apache Spark SQL we can use structured and semi-structured data in three ways:

- To simplify working with structured data it provides DataFrame abstraction in *Python, Java, and* ***Scala***. DataFrame is a distributed collection of data organized into named columns. It provides a good optimization technique.

OVERVIEW

- The data can be read and written in a variety of structured formats. For example, JSON, **Hive** Tables, and Parquet.
- Using SQL we can query data, both from inside a Spark program and from external tools. The external tool connects through standard database connectors (JDBC/ODBC) to Spark SQL.
- The best way to use Spark SQL is inside a Spark application. This empowers us to load data and query it with SQL. At the same time, we can also combine it with "regular" program code in Python, Java or Scala.

Advantages of Spark SQL:

Integrated

- Apache Spark SQL mixes SQL queries with Spark programs.
- With the help of Spark SQL, we can query structured data as a distributed dataset (RDD).

OVERVIEW

Integrated

- We can run SQL queries alongside complex analytic algorithms using tight integration property of Spark SQL.

Unified Data Access

- Using Spark SQL, we can load and query data from different sources.
- The Schema-RDDs lets single interface to productively work structured data.
- For example, Apache Hive tables, parquet files, and JSON files.

High compatibility

- In Apache Spark SQL, we can run unmodified Hive queries on existing warehouses.
- It allows full compatibility with existing Hive data, queries and UDFs, by using the Hive fronted and MetaStore.

Standard Connectivity

- It can connect through JDBC or ODBC. It includes server mode with industry standard JDBC and ODBC connectivity.

OVERVIEW

Scalability

To support mid-query fault tolerance and large jobs, it takes advantage of RDD model. It uses the same engine for interactive and long queries.

Performance Optimization

The query optimization engine in Spark SQL converts each SQL query to a logical plan. Further, it converts to many physical execution plans. Among the entire plan, it selects the most optimal physical plan for execution.

For batch processing of Hive tables

We can make use of Spark SQL for fast batch processing of Hive tables.

DATASET and DATAFRAME

- DATASET:
  - A Dataset is a distributed collection of data
  - Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (Strong typing and powerful lamda functions) with the benefits Spark SQL's optimized execution engine
  - A Dataset can be constructed from JVM objects and then manipulated using functional transformations(map,flatMap,filter etc.)
  - Datasets are similar to RDDs, however they use a specialized Encoder to serialize the objects for processing or transmitting the objects over the network
  - Encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into objects. This helps in improving the performance
  - Dataset API available in Scala and Java but this feature not added in Python

DATASET and DATAFRAME

Encoder:

- The **encoder** is primary concept in *serialization* and *deserialization* (**SerDe**) framework in Spark SQL.
- Encoders translate between JVM objects and Spark's internal binary format.
- Spark has built-in encoders which are very advanced. They generate bytecode to interact with **off-heap** data.
- An encoder provides on-demand access to individual attributes without having to de-serialize an entire object.
- To make input output time and space efficient, Spark SQL uses SerDe framework. Since encoder knows the schema of record, it can achieve serialization and deserialization.

DATASET and DATAFRAME

Features of Dataset

Optimized Query:

- Dataset in Spark provides Optimized query using Catalyst Query Optimizer and **Tungsten**.
- It represents and manipulates a data-flow graph. Data flow graph is a tree of expressions and relational operators.
- By optimizing the Spark job Tungsten improves the execution. Tungsten emphasizes on the hardware architecture of the platform on which Apache Spark runs.

Analysis at compile time

- Using Dataset we can check syntax and analysis at compile time. It is not possible using Dataframe, RDDs or regular SQL queries.

Persistent Storage

Spark Datasets are both serializable and Queryable. Thus, we can save it to persistent storage.

DATASET and DATAFRAME

Features of Dataset

Inter-convertible:

We can convert the Type-safe dataset to an "untyped" DataFrame. To do this task Datasetholder provide three methods for conversion from *Seq[T]* or *RDD[T]* types to *Dataset[T]*:

toDS(): Dataset[T]

toDF(): DataFrame

toDF(colNames: String*): DataFrame

Faster Computation:

The implementation of Dataset is much faster than the RDD implementation. Thus increases the performance of the system.

Less Memory Consumption

While caching, it creates a more optimal layout. Since Spark knows the structure of data in the dataset.

Single API for Java and Scala

It provides a single interface for **Java** and **Scala**. This unification ensures we can use Scala interface, code examples from both languages.

DATASET and DATAFRAME

- DATAFRAME:
  - A DataFrame is a Dataset organized into named columns
  - DataFrame is conceptually same as relational database but with richer optimizations under the hood.
  - DataFrames can be constructed from variety of sources such as structured data files, tables in Hive, external databases, or existing RDDs
  - The features common to RDD and DataFrame are **immutability**, in-memory, resilient, distributed computing capability. It allows the user to impose the structure onto a distributed collection of data. Thus provides higher level abstraction.
  - DataFrame API is available in Scala, Java, Python and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame simply a type alias of Dataset[Row]

DATASET and DATAFRAME

Why DataFrame?

DataFrame one step ahead of RDD. It provides memory management and optimized execution plan.

Custom Memory Management:

- This is also known as Project **Tungsten.** A lot of memory is saved as the data is stored in off-heap memory in binary format. There is no Garbage Collection overhead
- Expensive Java serialization is also avoided. Since the data is stored in binary format and the schema of memory is known.

Optimized Execution plan:

- This is also known as the **query optimizer**.
- Using this, an optimized execution plan is created for the execution of a query. Once the optimized plan is created final execution takes place on RDDs of Spark.

DATASET and DATAFRAME

- Catalyst supports optimization. It has general libraries to represent trees. DataFrame uses **Catalyst tree transformation** in four phases:
  - Analyze logical plan to solve references
  - Logical plan optimization
  - Physical planning
  - Code generation to compile part of a query to Java bytecode.

Creating DataFrame:

The entry point into all functionality in Spark is the SparkSession class

```
Import org.apache.spark.sql.SparkSession
val spark = SparkSession
            .build()
            .appName("Spark SQL Example").config("spark-some-config","some-value")
            .getOrCreate()
```

DATASET and DATAFRAME

```
// For implicit conversions like converting RDDs to Dataframes
  import spark.implicits._
// create dataframe from Json file
 val df = spark.read.json("examples/src/main/resources/people.json")
//Displays the content of the DataFrame to stdout
 df.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

DATASET and DATAFRAME

```
// Print the schema in a tree format
df.printSchema()
 root
  |-- age: long (nullable = true)
  |-- name: string (nullable = true)
// Select only the "name" column
df.select("name").show()
 +-------+
  |    name |
 +-------+
 |Michael |
  |    Andy |
  |  Justin |
 +-------+
```

DATASET and DATAFRAME

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     null|
|   Andy|       31|
| Justin|       20|
+-------+---------+
// Select people older than 21
df.filter($"age" > 21).show()
 +---+----+
 |age|name|
 +---+----+
 | 30|Andy|
 +---+----+
```

DATASET and DATAFRAME

```
// Select everybody, but increment the age by 1
df.select($"name", $"age" + 1).show()
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     null|
|   Andy|       31|
| Justin|       20|
+-------+---------+
// Select people older than 21
df.filter($"age" > 21).show()
 +---+----+
 |age|name|
 +---+----+
 | 30|Andy|
 +---+----+
```

DATASET and DATAFRAME

```
// Count people by age
df.groupBy("age").count().show()
+----+-----+
| age |count |
+----+-----+
|  19 |    1 |
|null |    1 |
|  30 |    1 |
+----+-----+
```

Running SQLs Programmatically:

```
//Register the DataFrame as a SQL Temporary View
df.createOrReplaceTempView("people")
val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
+----+-------+
| age |   name |
+----+-------+
|null |Michael |
|  30 |   Andy |
|  19 | Justin |
```

DATASET and DATAFRAME

Global Temporary View

- Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates.
- Global Temporary views can be shared among all sessions and keep alive until the Spark application terminates
- Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g Select * from global_temp.view1

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")
// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

DATASET and DATAFRAME

```
// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Creating Datasets:

```
case class Person(name: String, age: Long)
```

DATASET and DATAFRAME

```
// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
 +----+---+
  |name|age|
 +----+---+
  |Andy|  32|
 +----+---+
// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

DATASET and DATAFRAME

```
// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name
val path = "examples/src/main/resources/people.json"
val peopleDS = spark.read.json(path).as[Person]
peopleDS.show()
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

DATASET and DATAFRAME

Interoperating with RDDs

- *Spark SQL supports two different methods for converting existing RDDs into datasets*
- The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.
- The second method for creating Datasets is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct Datasets when the columns and their types are not known until runtime.

Inferring the Schema Using Reflection

- The Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame.
- The case class defines the schema of the table.
- The names of the arguments to the case class are read using reflection and become the names of the columns.

DATASET and DATAFRAME

```
// For implicit conversions from RDDs to DataFrames
import spark.implicits._
// Create an RDD of Person objects from a text file, convert it to a Dataframe
val peopleDF = spark.sparkContext
  .textFile("examples/src/main/resources/people.txt")
  .map(_.split(","))
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))
  .toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")
// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age
BETWEEN 13 AND 19")
```

DATASET and DATAFRAME

```
// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
// or by field name
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]("name")).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

DATASET and DATAFRAME

Programmatically Specifying the Schema

- A dataset can be created in three steps programmatically
  - Create an RDD of Rows from the original RDD
  - Create the schema represented by StructType matching structure of Rows of the RDD created in step1
  - Apply the schema to the RDD of Rows via createDataFrame method provided by SparkSession

Example:

```
//create an RDD
Val peopleRDD = spark.SparkContext.textFile("examples/src/main/resources/people.txt")
//The schema encoded in a String
Val schemaString = "name age"
//Generate the schema based on the String of Schema
Val fields = schemaString.map(_.split(" ")).map(fieldname =>
StructField(fieldname,StringType,nullable=true))
Val schema = StructType(fields)
 //Convert records of RDD (people) to rows
```

DATASET and DATAFRAME

```
//Convert records of RDD (people) to rows
Val rowRDD = peopleRDD.map(_.split(“,”)).map(attributes => Row(attributes(0),attributes(1).trim)
//apply the schema to the RDD
Val peopleDF = spark.createDataFrame(rowRDD,schema)
//create temporary view using the dataframe
peopleDF.createOrReplaceTempView(“people”)
//SQL can be run over a temporary view created using DataFrames
Val results = spark.sql(“select * from people”)
//Results of SQL queries are DataFrames and support all normal RDD operations
//The columns of a row in the result can be accessed by field index or field name
results.map(attributes => “Name: ” + attributes(0)).show()
-------------------------------
|                  value  |
-------------------------------
 Name:Michael
 Name:     Andy
Name:    Justin
```

DATA SOURCES

Generic Load/Save Options:

- In the simplest form, the default data source (parquet unless otherwise configured by spark.sql.source.default) will be used for all operations

Val usersDF = spark.read.load("examples/src/main/resources/users.parquet")

usersDF.select("name","favorite_color").write.save("namesAndFavcolors.parquet")

Manually Specifying Options

- You can also manually specify the data source that will be used along with any extra options that you would like to pass to the data source
- Data sources are specified by their fully qualified name (i.e org.apache.spark.sql.parquet), but for built in sources, their short names can be used(json,parquet,jdbc,orc,csv,text)
- DataFrames loaded from any data source type can be converted into other types using this syntax

Val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")

peopleDF.select("name","age").write.format("parquet").save("nameAndAges.parquet")

DATA SOURCES

Val peopleDFCsv = spark.read.format("csv")

.option(sep,";")

.option("inferSchema","true")

.option("header","true")

.load("examples/src/main/resources/people.csv")

peopleDFCsv.select("name","age").write.format("parquet").save("nameAndAges.parquet")

Run SQL on files directly:

- Instead of using read API to load a file into DataFrame and query it, you can also query the file directly using SQL

Val sqlDF = spark.sql("select * from parquet.`examples/src/main/resources/people.parquet`")

Save operations can optionally take a SaveMode

DATA SOURCES

Save Modes:

- Save operations can optionally take a SaveMode, that specifies how to handle existing data if present
- Save mode do not utilize any locking and are not atomic
- When performing an overwrite, the data will be deleted before writing out the new data

Scala/Java	Any Language	Meaning
`SaveMode.ErrorIfExists` (default)	`"error"` or `"errorifexists"` (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
`SaveMode.Append`	`"append"`	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
`SaveMode.Overwrite`	`"overwrite"`	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
`SaveMode.Ignore`	`"ignore"`	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL.

Save operations can optionally take a SaveMode

DATA SOURCES

Saving to Persistent Tables

- DataFrames can also be saved as persistent tables into Hive meta store using the saveAsTable command.
- An Existing hive deployment is not necessary to use this feature, Spark will create a default local hive metastore (using Derby)
- saveAsTable will materialize the contents of the DataFrame and create a pointer to the data in the hive metastore
- For file based data source, e.g text, parquet, json, etc. you can specify a custom table path via the path option.

  eg: df.write.option("path","/some/path").saveAsTable("t")

Bucketing, Sorting and Partitioning

- For file based data source, it is also possible to bucket and sort or partition the output.
- Bucketing and sorting is applicable only for persistent tables

peopleDF.write.bucketBy(42,"name").sortBy("age").saveAsTable("people_bucketed")

Save operations can optionally take a SaveMode

DATA SOURCES

Bucketing, Sorting and Partitioning

- Partitioning can be used for both save and saveAsTable

usersDF.write.partitionBy("favourite_color").format("parquet").save("namePartByColor.parquet")

usersDF.write.partitionBy("favourite_color").bucketBy(42,"name").saveAsTable("users_partitioned_bucketed")

Partition Discovery

- In a partitioned table, data are usually stored in different directories, with partitioning column values encoded in the path of each partitioned directory
- All built-in data file sources (including Text/csv/JSON/ORC/parquet) are able to discover and infer partitioning information automatically
- For example, we can store all previously population data into a partitioned table using the following directory structure, with two extra columns, gender and country as partitioning columns

Save operations can optionally take a SaveMode

DATA SOURCES

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

- By passing path/to/table to either SparkSession.read.parquet or SparkSession.read.load, the Spark SQL will automatically extract partitioning information from the paths. Now the schema of the returned dataframe is

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

Save operations can optionally take a SaveMode

DATA SOURCES

- The data types of the partitioning columns are automatically inferred . Currently, numeric data types, date, timestamp and string type are supported.
- The automatic type inference can be configured by spark.sql.sources.partitionColumnTypeInference.enabled, which is default to true
- When type inference is disabled, string type will be used for the partitioning columns

Parquet Files

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

Save operations can optionally take a SaveMode

DATA SOURCES

//Encoders for most common types automatically provided by importing spark.implicits._

Import spark.implicits._

Val peopleDF = spark.read.json("examples/src/main/resources/people.json")

//Dataframes saved as parquet files, maintaining the Schema information

peopleDF.write.parquet("people.parquet")

//Read in the parquet file created above

Val parquetfileDF= spark.read.parquet("people.parquet")

//parquet files can also be used to create a temporary view and then used in SQL statements

parquetfileDF.createOrReplaceTempView("parquetTable")

Val namesDF=spark.sql("SELECT name FROM parquetTable where age BETWEEN 13 and 19")

namesDF.map(attributes => "Name:"+attributes(0)).show()

Save operations can optionally take a SaveMode

DATA SOURCES

Schema Merging

- Users can start with simple schema and gradually add more columns to the schema as needed. In this way, users may end up with multiple parquet files but mutually compatible schemas
- The parquet data source is now able to automatically detect this case and merge schema for all these files
- Schema merging is a relatively expensive operation and it is not a necessity in most of the cases, hence it is turned it off by default (1.5.0)
- Schema merging can be enabled in following ways
  - setting data source option mergeSchema to true when reading parquet files
  - Setting the global SQL option spark.sql.parquet.mergeShema to true

```
//This is used to implicitly convert RDD to DataFrames
Import spark.implicits._
//Create a simple DataFrame, store into a partition directory
Val squaresDF = spark.SparkContext.makeRDD(1 to 5).map(i => (i, i * i)).toDF("value","square")
squaresDF.write.parquet("data/test_table/key=1")
```

Save operations can optionally take a SaveMode

DATA SOURCES

```
//create another DataFrame in a new partition directory
//adding a new column and drop an existing column
Val cubeDF = spark.SparkContext.makeRDD(6 to 10).map(x =>
(x,x*x*x)).toDF("value","cube")
cubeDF.write.parquet("data/test_table/key=3")
//read the partitioned table
Val mergedDF=spark.read.option("mergeSchema","true").parquet("data/test_table")
mergedDF.printSchema()
// The final schema consists of all 3 columns in the Parquet files together
// with the partitioning column appeared in the partition directory paths
// root
//   |-- value: int (nullable = true)
//   |-- square: int (nullable = true)
//   |-- cube: int (nullable = true)
//   |-- key: int (nullable = true)
```