



# USING SPARK SQL FOR ETL

Bin Jiang

04/22/2017

# What is Spark SQL

- Spark SQL is a Spark module for structured data processing
- Spark SQL programs are executed in a distributed fashion by the SQL query engine
- Spark SQL queries can be mixed with Spark programs written in Scala, Python and Java
- Several ways to interact with Spark SQL including SQL, the DataFrames API and the Datasets API
- You also get a tight integration with Hive
- In its server mode, Spark SQL offers connectivity to ODBC/JDBC clients

# Uniform Data Access with Spark SQL

- Spark SQL provides uniform connectivity API to any data source, such as: Avro, Parquet, ORC, JSON, Hive, JDBC/ODBC-enabled sources
- Joining data across the above data source is also supported

# Integration with BI Tools

- Spark SQL offers ODBC/JDBC connectivity for various business intelligence tools

# Starting Point: SQL Session

- SparkSession is the entry point to Spark SQL. It is one of the very first objects you create while developing a Spark SQL application using the typed Dataset (or untyped Row-based DataFrame) data abstractions
- SparkSession has merged SQLContext and HiveContext in one object in Spark 2.0
- Use the SparkSession.builder method to create an instance of SparkSession.

# Starting Point: SQL Session

```
import org.apache.spark.sql.Session

val spark: Session = Session.builder
  .appName("My Spark Application") // optional and will be autogenerated if not specified
  .master("local[*]")              // only for demo and testing purposes, use spark-submit instead
  .enableHiveSupport()              // self-explanatory, isn't it?
  .config("spark.sql.warehouse.dir", "target/spark-warehouse")
  .getOrCreate
```



# SQLContext

- The entry point into all functionality in Spark SQL
- Need a SparkContext to create a basic SQLContext
- Rich APIs for data query, DataFrame management and DDL
- Only dialect available is “sql” for a SQLContext
- Simple SQL parser provided by Spark SQL

# Spark SQL Structure APIs

- **SQL**

```
select dept, avg(age) from data group by dept
```

- **RDD**

```
rdd.map {case (dept, age) => dept -> (age,1)}  
  .reduceByKey {case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}  
  .map {case (dept, (age, c)) => dept -> age /c}
```

- **DataFrame**

```
df.groupBy("dept").avg("age")
```



# What is a DataFrame

- DataFrame is a distributed collection of data organized into named columns
- Conceptually equivalent to a table in a relational database
- Dataframes can be created from Hive tables, structured data files or existing Spark RDDs, external databases
- The DataFrame API is available in Scala, Java, Python and R

# Creating DataFrames

- create DataFrames from an existing RDD

```
val df = existingRDD.toDF()
```

- create DataFrames from a Hive table

```
val df1 = sqlContext.sql("select * from sample_07")
```

- create DataFrames from from data sources

```
val df2 = sqlContext.read.json("/root/labs/datasets/labs/people.json")
```

# DataFrame Operations

- DataFrames provide a domain-specific language for structured data manipulation in Scala, Java, Python and R
- Examples:

```
val df = sqlContext.read.json("/root/labs/datasets/labs/people.json")
```

```
df.show()
```

```
df.printSchema()
```

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

```
df.select(df("name"), df("age") + 1).show()
```

```
df.filter(df("age") > 21).show()
```

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

# DataFrame Operations

- **Expression Builder Style**

```
df.filter(df.col("age").gt(30)).show()
```

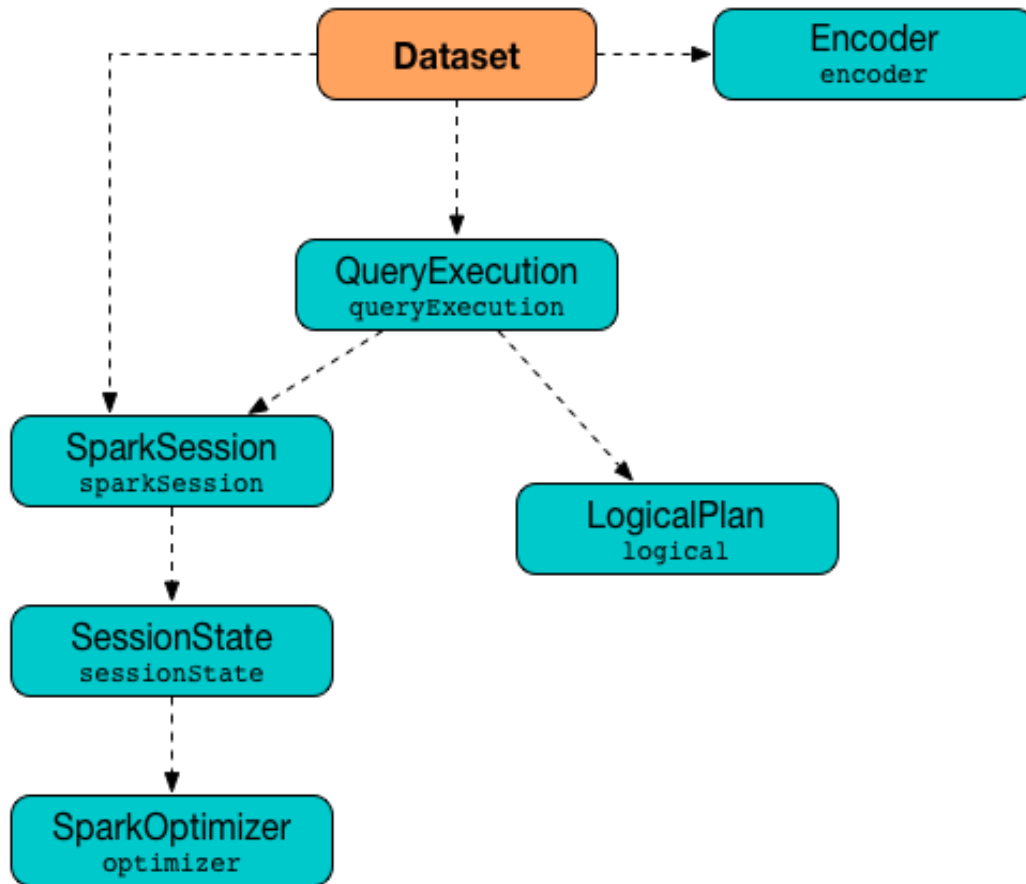
- **SQL Style**

```
df.filter("age > 30"),show()
```

# What is a DataSet

- New experimental interface added in Spark 1.6
- Provide the benefits of RDDs (strong typing, ability to use powerful lambda functions)
- Benefits of Spark SQL's optimized execution engine
- Dataset can be created from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.)
- The unified Dataset API can be used both in Scala and Java.
- Dataset is a strongly-typed data structure in Spark SQL that represents a structured query with encoders.

# What is a DataSet





# What is a DataSet

- Datasets are lazy and structured query expressions are only triggered when an action is invoked
- Dataset is the result of executing a query expression against data storage like files, Hive tables or JDBC databases
- Dataset API offers declarative and type-safe operators that makes for an improved experience for data processing
- Can convert a type-safe Dataset to a "untyped" DataFrame or access the RDD that is generated after executing the query

# What is a DataSet

- Dataset offers convenience of RDDs with the performance optimizations of DataFrames and the strong static type-safety of Scala
- Datasets use the Catalyst Query Optimizer and Tungsten to optimize query performance
- Dataset object requires a SparkSession, a QueryExecution plan, and an Encoder (for fast serialization to and deserialization from InternalRow)

# What is a DataSet

**Dataset operations are available in three variants.**

- SQL query
- Column-based SQL expression
- Scala/Java lambda function

```
// Variant 1: filter operator accepts a Scala function  
dataset.filter(n => n % 2 == 0).count
```

```
// Variant 2: filter operator accepts a Column-based SQL expression  
dataset.filter('value % 2 === 0).count
```

```
// Variant 3: filter operator accepts a SQL query  
dataset.filter("value % 2 = 0").count
```

# Creating Datasets

- Datasets are similar to RDDs
- Examples

```
val ds = Seq(1, 2, 3).toDS()
```

```
ds.map(_ + 1).collect()
```

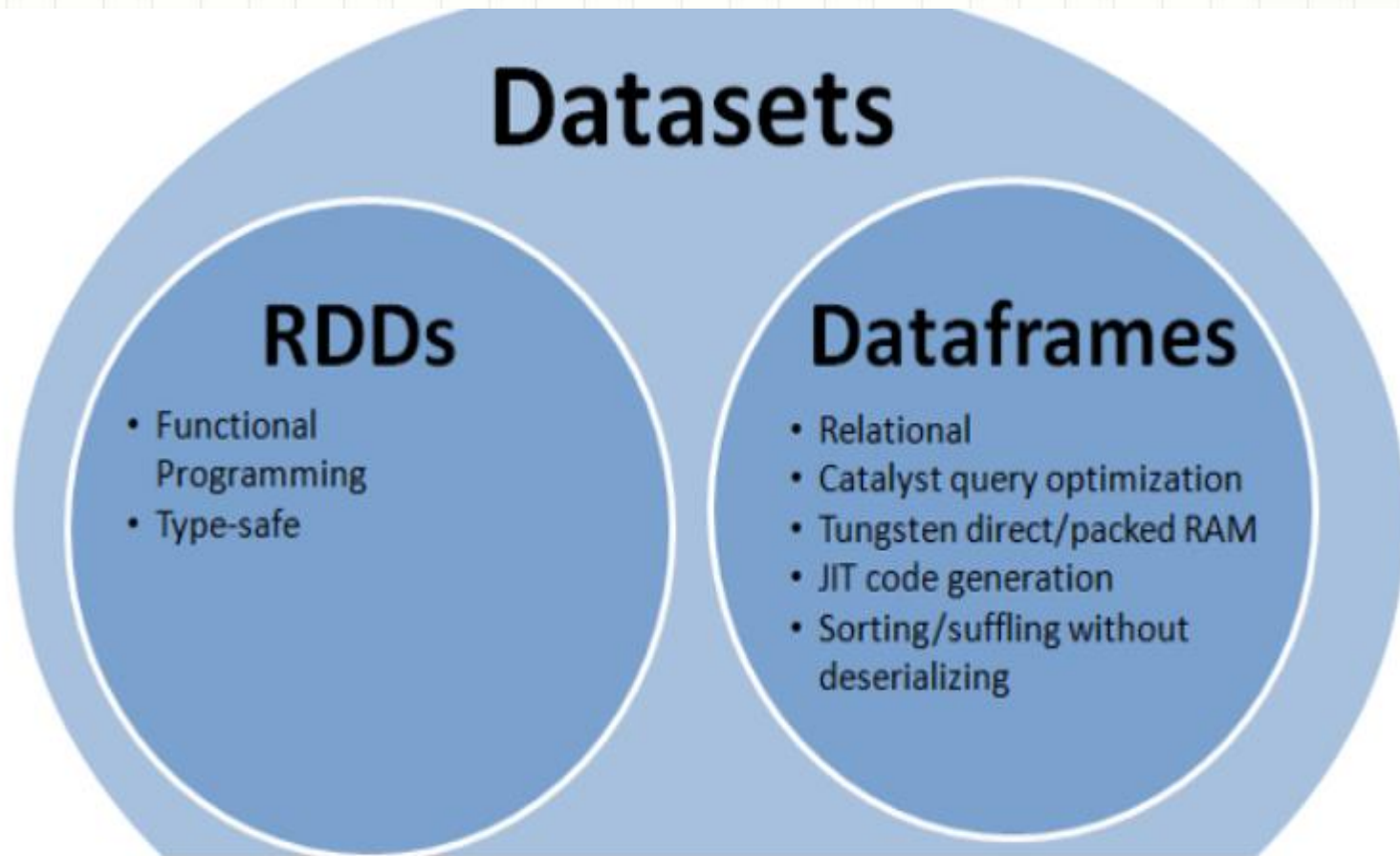
```
case class Person(name: String, age: Long)
```

```
val ds = Seq(Person("Andy", 32)).toDS()
```

```
val path = "/root/labs/datasets/labs/people.json"
```

```
val people = sqlContext.read.json(path).as[Person]
```

# RDD vs DataFrame vs Dataset



# RDD vs DataFrame vs Dataset

- RDD
  - Blocks of spark, Dataframe or Dataset eventually are converted into RDD to run
  - Collection of objects are immutable and triggered lazily
  - Type Safety
  - Function Programming and OOP style
  - Performance limitations, e.g. GC and Serialization are expensive



# RDD vs DataFrame vs Dataset

- Dataframe
  - Schema view of data
  - Execution of Dataframe is also lazily triggered
  - Huge performance improvement over RDDs because of Optimized Execution Plans (Catalyst) and Custom Memory management (Tungsten)
  - Lack of Type Safety and OOP style

# RDD vs DataFrame vs Dataset

- Dataframe
  - Extension to Dataframe
  - OOP style and performance boosting features of Catalyst and Tungsten
  - Encoder to convert Dataframe to Dataset, act as interface between JVM objects and
  - Off-heap custom binary format data
  - Encoder makes the sort and shuffling without de-serializing
  - case class makes it works as RDD but underneath as Dataframe
  - Dataframe is special Dataset, like `DataFrame=Dataset[Row]`

# Interoperating with RDDs

- Uses reflection to infer the schema of an RDD that contains specific types of objects
- Programmatic interface that allows you to construct a schema and then apply it to an existing RDD

# Spark SQL - Schema

- A schema is the description of the structure of your data. It can be implicit (and inferred at runtime) or explicit (and known at compile time)
- A schema is described using StructType which is a collection of StructField objects (that in turn are tuples of names, types, and nullability classifier)
- StructType and StructField belong to the `org.apache.spark.sql.types` package

# Spark SQL – Create Schema

- Create schema

```
import org.apache.spark.sql.types.StructType
val schemaUntyped = new StructType()
    .add("a", "int")
    .add("b", "string")
```

- Create schema using Schema DSL

```
val schemaUntyped_2 = new StructType()
    .add($"a".int)
    .add($"b".string)
```

# Spark SQL – Create Schema

- Create schema using the canonical string representation of SQL types

```
// it is equivalent to the above expressions
import org.apache.spark.sql.types.{IntegerType, StringType}
val schemaTyped = new StructType()
    .add("a", IntegerType)
    .add("b", StringType)
```

- Create schema using the singleton DataTypes class with static methods

```
import org.apache.spark.sql.types.DataTypes._
val schemaWithMap = StructType(
    StructField("map", createMapType(LongType, StringType), false) :: Nil)
```



# Spark SQL – Data Type

Spark SQL and DataFrames support the following data types:

- Numeric types
  - ByteType
  - ShortType
  - IntegerType
  - LongType
  - FloatType
  - DoubleType
  - DecimalType
- String type
- Binary type

# Spark SQL – Data Type

Spark SQL and DataFrames support the following data types:

- Boolean type
- Datetime type
  - TimestampType
  - DateType
- Complex types
  - ArrayType
  - MapType
  - StructType
  - ObjectType
- NullType
- UserDefinedType

# Inferring the Schema Using Reflection

- 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
- Case classes can also be nested or contain complex types such as Sequences or Arrays
- This RDD can be implicitly converted to a DataFrame and then be registered as a table
- Tables can be used in subsequent SQL statements

# Inferring the Schema Using Reflection

- Examples

```
import sqlContext.implicits._
```

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

```
val people = sc.textFile("/root/labs/datasets/labs/people.txt").map(_._split(",")).map(p => Person(p(0),  
p(1).trim.toInt)).toDF()
```

```
people.registerTempTable("people")
```

```
val teenagers = sqlContext.sql("SELECT name, age FROM people WHERE age >= 13 AND age <= 19")
```

```
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

```
teenagers.map(t => "Name: " + t.getAs[String]("name")).collect().foreach(println)
```

```
teenagers.map(_._getValuesMap[Any](List("name", "age"))).collect().foreach(println)
```

# Programmatically Specifying the Schema

- Use this approach whenever case classes cannot be defined ahead of time
- Create an RDD of Rows from the original RDD
- Create the schema represented by a StructType matching the structure of Rows in the RDD created
- Apply the schema to the RDD of Rows via createDataFrame method provided by SQLContext

# Programmatically Specifying the Schema

- Examples:

```
val people = sc.textFile("/root/labs/datasets/labs/people.txt ")
val schemaString = "name age"
import org.apache.spark.sql.Row;
import org.apache.spark.sql.types.{StructType, StructField, StringType};
val schema = StructType(schemaString.split(" ").map(fieldName => StructField(fieldName,
StringType, true)))

val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

peopleDataFrame.registerTempTable("people")

val results = sqlContext.sql("SELECT name FROM people")
results.map(t => "Name: " + t(0)).collect().foreach(println)
```



# HiveContext

- HiveContext provides a superset of the functionality provided by the basic SQLContext
- Spark SQL supports interfaces with Apache Hive for reading and writing data stored in Apache Hive
- Write queries using the more complete HiveQL parser
- Access to Hive UDFs
- Read data from Hive tables
- HiveContext is only packaged separately to avoid including all of Hive's dependencies in the default Spark build
- In a HiveContext, the default dialect is "hiveql"
- Spark SQL also supports Hive serialization and deserialization libraries

# Hive Integration

- Hive support is enabled by adding the -Phive and -Phive-thriftserver flags to Spark's build
- Configuration of Hive is done by placing your hive-site.xml, core-site.xml (for security configuration), hdfs-site.xml (for HDFS configuration) file in conf/
- Example

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

```
sqlContext.sql("LOAD DATA LOCAL INPATH '/training/apps/spark/datasets/labs/kv1.txt' INTO TABLE src")
```

```
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

# Data Sources

- Spark SQL supports operating on a variety of data sources through the DataFrame interface
- DataFrame can be operated on as normal RDDs
- DataFrame can be registered as a temporary table
- Registering a DataFrame as a table allows you to run SQL queries over its data

# Generic Load/Save Functions

- Default data source (parquet unless otherwise configured by `spark.sql.sources.default`) will be used for all operations
- Example

```
val df = sqlContext.read.load("/root/labs/datasets/labs/users.parquet")  
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

# Manually Specifying Options

- Manually specify the data source that will be used along with any extra options
- Data sources are specified by their fully qualified name (i.e., `org.apache.spark.sql.parquet`)
- Built-in sources you can also use their short names (`json`, `parquet`, `jdbc`).
- Example

```
val df = sqlContext.read.format("json").load("/root/labs/datasets/labs/people.json")
df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

# Run SQL on files directly

- Query that file directly with SQL
- Example

```
val df = sqlContext.sql("SELECT * FROM parquet.`/root/labs/datasets/labs/users.parquet`")
```



# Save Modes

- Save operations can optionally take a SaveMode
- Specifies how to handle existing data if present
- Save modes do not utilize any locking and are not atomic
- When performing a Overwrite, the data will be deleted before writing out the new data
- Example
  - SaveMode.ErrorIfExists
  - SaveMode.Append
  - SaveMode.Overwrite
  - SaveMode.Ignore

# Saving to Persistent Tables

- With a HiveContext, DataFrames can also be saved as persistent tables using the saveAsTable command
- SaveAsTable will materialize the contents of the dataframe and create a pointer to the data in the HiveMetastore
- Persistent tables will still exist even after your Spark program has restarted
- A DataFrame for a persistent table can be created by calling the table method on a SQLContext with the name of the table
- By default saveAsTable will create a “managed table”

# What is Spark ETL

- Extract: Dealing with Dirty Data (Bad Records or Files)
- Extract: Multi-line JSON/CSV Support -
- Transformation: High-order functions in SQL
- Load: Unified write paths and interfaces

# What is a Data Pipeline

- Sequence of transformations on data
- Source data is typically semi-structured/unstructured (JSON, CSV etc.) and structured (JDBC, Parquet, ORC, the other Hive-serde tables)
- Output data is integrated, structured and curated. – Ready for further data processing, analysis and reporting

# ETL Goal

- Retrieve data from sources (EXTRACT)
- Transform data into a consumable format (TRANSFORM)
- Transmit data to downstream consumers (LOAD)

# Why is ETL Hard

- Too complex
- Error-prone
- Too slow
- Too expensive
- Various sources/formats
- Schema mismatch
- Corrupted files and data
- Schema evolution
- Continuous ETL



# Data Sources Support

- Built-in connectors in Spark
  - JSON, CSV, Text, Hive, Parquet, ORC, JDBC
- Third-party data source connectors
- Build custom data source connectors

# Schema Support

- Schema Inference
  - Semi-structured files
- User-specific Schema
- User-specific DDL-format Schema

# Choosing Data Formats

- Data is available in a myriad of different formats. Spreadsheets can be expressed in XML, CSV, TSV; application metrics can be written out in raw text or JSON
- Every use cases have a particular data format tailored for it. In the world of Big Data, there comes across formats like parquet, ORC, Avro, JSON, CSV, SQL and NoSQL data sources, and plain text files
- Three categories: structured, semi-structured, and unstructured data

# Choosing Data Formats

- New Format in DataframeWriter API
  - ☐ Users can create Hive-serde tables using DataframeWriter APIs
  - ☐ `df.write.format("hive").option("fileFormat", "avro").saveAsTable("tab")`
  - ☐ CREATE Hive-serde tables
  - ☐ CREATE data source tables
- Unified Create Table
  - ☐ `CREATE TABLE t1(a INT, b INT) USING ORC`
  - ☐ `CREATE TABLE t1(a INT, b INT) USING hive OPTIONS(fileFormat 'ORC')`
  - ☐ `CREATE TABLE t1(a INT, b INT) STORED AS ORC`

# Data Transformation

- Better JSON and CSV Support
  - ❑ Multi-line JSON and CSV Support
- Higher-order Function in SQL Transformation on complex objects like arrays, maps and structures inside of columns
  - ❑ UDF - Expensive data serialization

# Quality Assurance

- Skip Corrupt Files
  - ☐ `spark.sql.files.ignoreCorruptFiles = true`
- Skip Corrupt Records
  - ☐ TextFile formats (JSON and CSV) support 3 different ParseModes while reading data: 1. PERMISSIVE 2. DROPMALFORMED 3. FAILFAST
  - ☐ `spark.sql.columnNameOfCorruptRecord`
  - ☐ Mode
- Better Corruption Handling
  - ☐ `badRecordsPath`: a user-specified path to store exception files for recording the information about bad records/files.
  - ☐ A unified interface for both corrupt records and files - Enabling multi-phase data cleaning
  - ☐ DROPMALFORMED + Exception files



# Processing JSON DataSet

- Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame
- Using `SQLContext.read.json()` on either an RDD of String, or a JSON file
- The file that is offered as *a json file* is not a typical JSON file
- Each line must contain a separate, self-contained valid JSON object

# JSON Options

The following options are supported:

- `samplingRatio`
- `primitiveAsString` – default is false
- `prefersDecimal` – default is false
- `allowComments` – default is false
- `allowUnquotedFieldNames` – default is false
- `allowSingleQuotes` – default is true
- `allowNumericLeadingZeros` – default is false
- `allowNonNumericNumbers` – default is true
- `allowBackslashEscapingAnyCharacter` – default is false
- `compression`

# JSON Options

The following options are supported:

- mode
  - PERMISSIVE – permissively parses the records
  - DROPMALFORMED – ignores the whole corrupted records
  - FAILFAST- throws an exception when it meets corrupted records
- columnNameOfCorruptRecord
- dateFormat – default is yyyy-MM-dd
- timestampFormat – default is yyyy-MM-dd'T'HH:mm:ss
- multiline – true or false

# Example of Working with a JSON DataSet

```
val rdd =  
sc.parallelize("""{"name":"Bin","address":{"city":"Toronto","state":"ON"}}""") :: Nil)  
val df = sqlContext.read.json(rdd)  
val df1 = df.select("name")  
df1.show()  
df1.write.format("parquet").save("/user/root/users.parquet")
```

# Example of Working with a JSON DataSet

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val path = " /root/labs/datasets/labs/people.json"
val people = sqlContext.read.json(path)
people.printSchema()
people.registerTempTable("people")
val teenagers = sqlContext.sql("SELECT * FROM people WHERE
age >= 13 AND age <= 19")
teenagers.write.format("parquet").save("/user/root/people.parquet")
```

# 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
- Preserves the schema of the original data automatically
- All columns are automatically converted to be nullable when writing Parquet files



# Example of Working with a Parquet File

```
val parquetFile =  
sqlContext.read.parquet("/user/root/people.parquet")  
parquetFile.registerTempTable("parquetFile")  
val teenagers = sqlContext.sql("SELECT * FROM parquetFile")  
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

# Partition Discovery

- The Parquet data source is able to discover and infer partitioning information automatically

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

# Partition Discovery

- By passing path/to/table to either `SQLContext.read.parquet` or `SQLContext.read.load`, Spark SQL will automatically extract the partitioning information from the paths

# Using Spark with RDBMS

- Big data use cases for relational database include the following
  - Complex OLTP transactions
  - Applications or features that need ACID compliance
  - Support for standard SQL
  - Real-time ad hoc query functionality
  - Systems implementing many complex relationships

# Using Spark with RDBMS

- Spark SQL also includes a data source that can read data from other databases using JDBC
- Preferred over using jdbcRDD
- Some databases convert all schema names to uppercase
- JDBC data source is also easier to use from Java or Python as it does not require the user to provide a ClassTag
- Include the JDBC driver for your particular database on the spark classpath.e.g.

`SPARK_CLASSPATH= mysql-connector-java.jar bin/spark-shell`

Or `spark-shell --jars /usr/hdp/2.6.3.0-235/hive/lib/mysql-connector-java.jar`

- Tables from the remote database can be loaded as a DataFrame or Spark SQL

# JDBC Connection Example

- Using Scala:

```
val dataframe_mysql = sqlContext.read.format("jdbc").option("url",  
"jdbc:mysql://localhost:3306/test").option("driver",  
"com.mysql.jdbc.Driver").option("dbtable", "DRIVERS").option("user",  
"root").option("password", "").load()  
dataframe_mysql.show()
```

- Using SQL:

```
CREATE TEMPORARY TABLE jdbcTable USING org.apache.spark.sql.jdbc  
OPTIONS ( url "com.mysql.jdbc.Driver", dbtable "schema.tablename")
```



# JDBC Connection Options

The following options are supported:

- url
- dbtable
- Driver
- Optional parameters only for reading
  - fetchSize
  - partitionColumn
  - lowerBound
  - upperBound
  - numPartitions

# JDBC Connection Options

The following options are supported:

- Optional parameters only for writing
  - truncate
  - createTableOptions
    - E.g., “CREATE TABLE t (name string) ENGINE=InnoDB DEFAULT CHARSET=utf8”
  - createTableColumnTypes
  - batchsize
  - isolationLevel
    - NONE, READ\_UNCOMMITTED, READ\_COMMITTED, REPEATABLE\_READ, SERIALIZABLE

# CSV Options

The following options are supported:

- delimiter – default is comma
- charset – default utf8
- quote
- escape
- comment
- header – true or false
- inferSchema
- ignoreLeadingWhiteSpace
- ignoreTrailingWhiteSpace

# CSV Options

The following options are supported:

- columnNameOfCoorruptRecord
- nullValue
- nanValue
- positiveInf
- negativeInf
- compression
- mode
  - PERMISSIVE – permissively parses the records
  - DROPMALFORMED – ignores the whole corrupted records
  - FAILFAST- throws an exception when it meets corrupted records

# CSV Options

The following options are supported:

- dateFormat – default is yyyy-MM-dd
- timestampFormat – default is yyyy-MM-dd'T'HH:mm:ss
- multiline – true or false
- maxColumns – default is 20480
- maxCharsPerColumn
- escapeQuotes – true or false
- quoteAll – true or false

# Processing Complex Data Types

## Advanced Spark SQL functions

- `get_json_object()`
  - Extract json object from a json string based on json path specified, and returns a json string as the extracted json object
- `from_json()`
  - Uses schema to extract individual columns
- `to_json()`
  - Convert a JSON struct into a string
- `selectExpr()`
- `explode()`
  - Extract nested structures
- Create pivot and unpivot tables using the functions above



# Spark SQL Performance Tuning

- Optimizing data serialization
- Catalyst optimizations
- Understanding Spark Application Execution
- Cost-based Optimization
- Data Lineage Tracking
- Iterative Broadcast
- JOIN ordering optimization
- Whole-stage code generation
- Debugging query execution
- Tungsten execution

# Performance of Spark SQL

- Caching data in memory
  - `sqlContext.cacheTable("tableName")`
  - `dataFrame.cache()`
  - `sqlContext.uncacheTable("tableName")` to remove the table from memory.
- Configuration of in-memory caching
  - `setConf` method on `SQLContext`
  - SET key=value commands using SQL
    - ❖ `spark.sql.inMemoryColumnarStorage.compressed`
    - ❖ `spark.sql.inMemoryColumnarStorage.batchSize`
- Tune the performance of query execution
  - `spark.sql.autoBroadcastJoinThreshold`
  - `spark.sql.tungsten.enabled`
  - `spark.sql.shuffle.partitions`

# Spark SQL JOIN

Supported join types include:

- inner
- outer
- full
- fullouter
- leftouter
- left
- rightouter
- right
- leftsemi
- leftanti
- cross

# Spark SQL JOIN

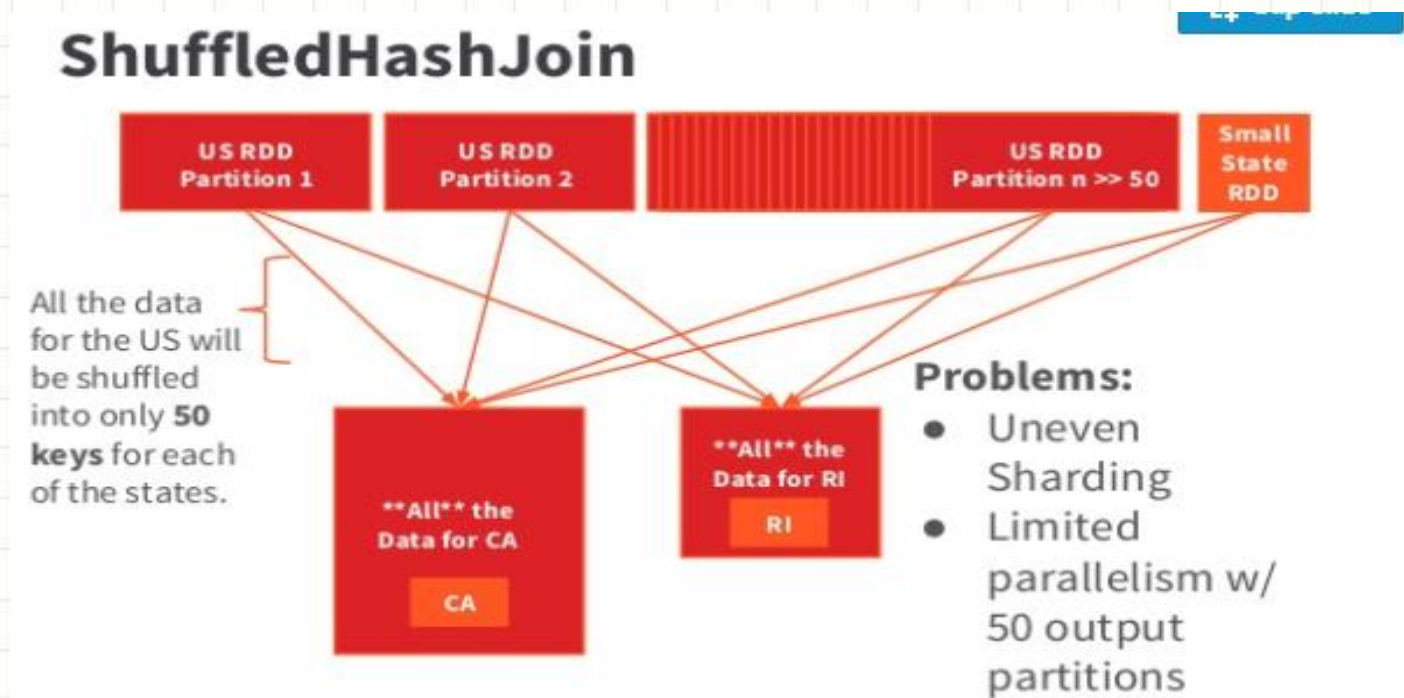
## Types of Spark SQL JOIN

- ShuffledHashJoin
- BroadcastHashJoin
- SortMergeJoin
- BroadcastNestedLoopJoin
- CartesianProduct
- Iterative Broadcast Join

# Spark SQL JOIN

## ShuffledHashJoin

- Uneven sharding
- limited parallelism



# Spark SQL JOIN

## BroadcastHashJoin

- No shuffling
- Join large data set with small data set
- `spark.sql.autoBroadcastJoinThreshold` - currently statistics are only supported for Hive Metastore tables where the command `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` has been run
- For dataframe, using broadcast function
- For hive sql query, using mapjoin hint
- SET `spark.sql.autoBroadcastJoinThreshold` = -1 to force disabling broadcast

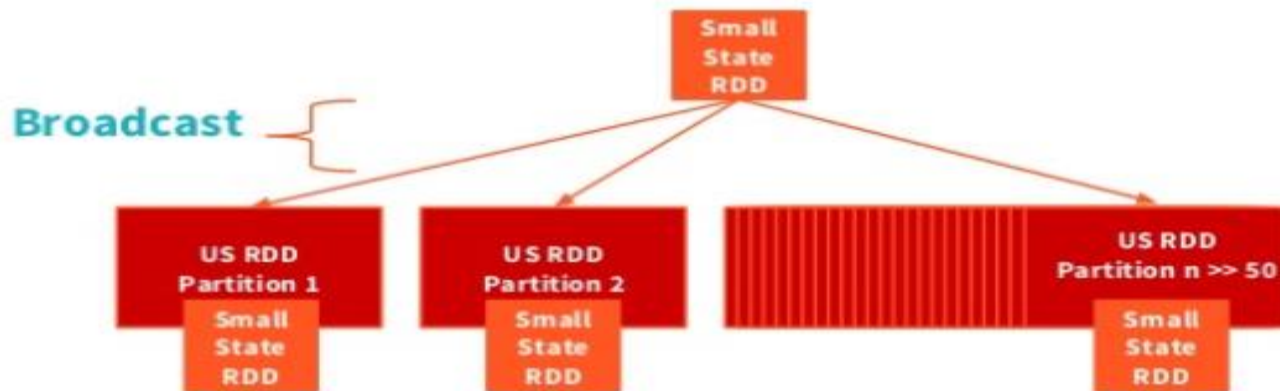


# Spark SQL JOIN

BroadcastHashJoin

## BroadcastHashJoin

**Solution: Broadcast the Small RDD to all worker nodes.**



**Parallelism of the large RDD is maintained (n output partitions), and shuffle is not even needed.**

# Spark SQL JOIN

## SortMergeJoin

- sort-merge join is composed of 2 steps. The first step is the ordering operation made on 2 joined datasets. The second operation is the merge of sorted data into a single place by simply iterating over the elements and assembling the rows having the same value for the join key.

**Customers table**

Id	Login
2	User#2
1	User#1
4	User#4
3	User#3

**Orders table**

Id	User id
1001	2
1002	4
1003	4
1004	1

*Sorting*

Id	Login
1	User#1
2	User#2
3	User#3
4	User#4

Id	User id
1004	1
1001	2
1002	4
1003	4

*Merging*

# Spark SQL JOIN Strategies

- if the size of one of tables is smaller than `spark.sql.autoBroadcastJoinThreshold` then use **broadcastHashJoin**
- if `spark.sql.join.preferSortMergeJoin` is not true, one table size is much smaller than another, and one table size is smaller enough to build local hash map, or the keys are not orderable then use **ShuffledHashJoin**
- if keys in both table are sortable then use **SortMergeJoin**
- if there is no join key and one table is smaller than `spark.sql.autoBroadcastJoinThreshold`, then use **BroadcastNestedLoopJoin**
- Pick **CartesianProduct** for inner join

# Join Big Table with Medium Table

- Left Join
  - Medium Table in the left
  - Limited by the size of large table
- Filter large table for only entries that match in the medium, then join
  - Less data shuffled
  - More transformation
- Iterative Broadcast

# Work with skew data

- BroadcastHashJoin
- Local Hash Map + UDF
- Iterative Broadcast

# Iterative Broadcast

- Divide the smaller table into "pass"
- Broadcast a pass and left join to the larger dataset
- Clear the broadcast partition from memory
- Repeat until passes are processed

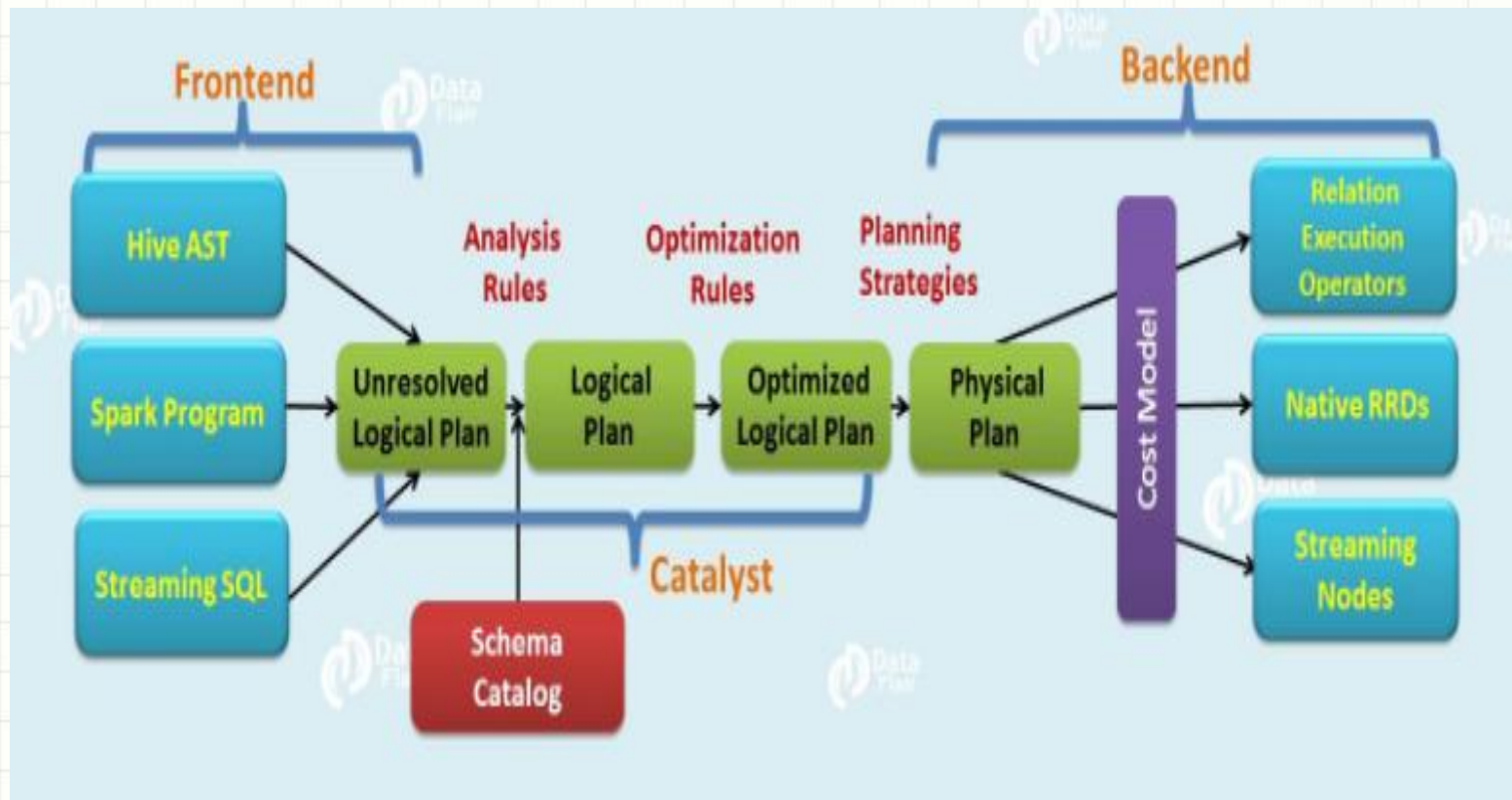


# Spark SQL's Execution Plan

Five Phrases:

- Parsing
- Analysis Rule
- Optimization Rule
- Planning Strategies
- Code Generation

# Spark SQL's Execution Plan



# Spark SQL's Execution Plan

Five Phrases:

## ➤ Parsing

- ✓ Hive Query, SQL Query and DataFrame are parsed to AST
- ✓ AST to Unresolved Logical Plan

# Spark SQL's Execution Plan

Five Phrases:

## ➤ Analysis Rule

- ✓ Use Catalyst Rules and Schema Catalog that tracks the table in all data sources to resolve the unresolved attributes, find where datasets and columns are coming from and types of columns
- ✓ Transform Unresolved Logical Plan to resolved Logical Plan

# Spark SQL's Execution Plan

Five Phrases:

## ➤ Optimization Rule

- ✓ Applies standard rule based optimization to the logical plan
- ✓ Transform resolved Logical Plan to Optimized Logical Plan by Rule Executor
- ✓ Note: Rule Executor transforms a tree to another same type Tree by applying many rules defined in batches

# Spark SQL's Execution Plan

Five Phrases:

## ➤ Planning Strategies

- ✓ It generates one or more physical plans, using physical operators that match the spark execution engine, then select a physical plan using Cost Based Optimization
- ✓ Transform Optimized Logical Plan to Selected Physical Plan



# Spark SQL's Execution Plan

Five Phrases:

## ➤ **Code Generation**

- ✓ Generating java bytecode to run on each machine
- ✓ Selected Physical Plan becomes RDDs

# Spark SQL's Catalyst Optimizer

- Spark Catalyst Optimizer's goal is to minimize end-to-end query response time
- It is based on two key ideas:
  - Pruning unnecessary data as early as possible, for example, filter pushdown and column pruning
  - Minimizing per-operator cost, for example, broadcast versus shuffle and optimal join order
- Rule-based and Cost-based

# Spark SQL's Catalyst Optimizer

- **Logic Plan**

- Describes computation on datasets without defining how to conduct the computation
- Output is a list of attributes generated by the Logical Plan
- Constraints are a set of invariants about the rows generated by the plan
- Statistics is the size of the plan in rows/bytes

# Spark SQL's Catalyst Optimizer

- **Optimized Execution Plans – Rule Executor**
  - Query plans are created for execution using Spark catalyst optimiser. After an optimised execution plan is prepared going through some steps, the final execution happens internally on RDDs only but that is completely hidden from the users
  - Transforms a resolved logical plan to an optimized logical plan

# Spark SQL's Catalyst Optimizer

```
users.join(events, users("id") === events("uid"))  
      .filter(events("date") > "2015-01-01")
```



filter pushdown to data source so that it can apply that filter on the disk level rather than bringing all data in memory

# Spark SQL's Catalyst Optimizer

- **Physical Plan – Strategies + Rule Executor**
  - Describes computation on datasets with specific definitions on how to conduct the computation
  - Transforms an optimized logical plan to a physical plan
  - Rule Executor is used to adjust the physical plan to make it ready for execution
  - Physical Plan is executable



# How Catalyst Transformation Works

- **Transformations**

Transformations without changing the tree type (Transform and Rule Executor)

- Expression  $\Rightarrow$  Expression
- Logical Plan  $\Rightarrow$  Logical Plan
- Physical Plan  $\Rightarrow$  Physical Plan

# How Catalyst Transformation Works

- **Transformations**

Transforming a tree to another kind of tree

➤ Logical Plan => Physical Plan

# How Catalyst Transformation Works

- **Transform**

A function associated with every tree used to implement a single rule

A transformation is defined as Partial Function, which is defined for a subset of its possible arguments

# How Catalyst Transformation Works

- **Rule Executor**

Combining multiple rules, e.g. Predicate Pushdown, Column Pruning

Rule Executor transforms a tree to another same type Tree by applying many rules defined in batches

# How Catalyst Transformation Works

- **Strategies**

A Logical Plan is transformed to a Physical Plan by applying a set of Strategies

Every Strategy uses pattern matching to convert a Logical Plan to a Physical Plan

# Extend Spark Catalyst optimizer

- **Custom Rules and Strategies**

Using the following way to add your functionality to the SparkPlanner

- `SparkSession.experimental.extraStrategies`
- `SparkSession.experimental.extraOptimizations`



# Extend Spark Catalyst optimizer

- **Custom Rules and Strategies**

```
object MyStrategy extends Strategy {  
  def apply(plan: LogicalPlan): Seq[SparkPlan] = {  
    println("Hello world!")  
    Nil  
  }  
}
```

```
spark.experimental.extraStrategies = Seq(MyStrategy)  
val q = spark.catalog.listTables.filter(t => t.name == "five")  
q.explain(true)
```

# Rule-Based Optimizer

- Most of Spark SQL optimizer's rules are heuristics rules
  - PushDownPredicate
  - ColumnPruning
  - ConstantFolding
- Does NOT consider the cost of each operator
- Does NOT consider selectivity when estimating join relation size
- Join order is mostly decided by its position in the SQL queries
- Physical Join implementation is decided based on heuristics

# Cost-Based Optimizer

- **Goals**

- Helps the optimizer choose a better query plan
- Obtain more efficient queries

- **Implementation**

- Collects, infers, and propagates table/column statistics on source / intermediate data
- Calculates the cost of each operator in terms of the number of output rows, the output size
- Picks the most optimal query execution plan based on these cost calculations

# Cost-Based Optimizer

- **Table Statistics Collected**

- ❑ `ANALYZE TABLE table_name COMPUTE STATISTICS`

- ❑ Collect the following table-level statistics and save them into meta-store

- The number of rows,

- Number of files (or HDFS data blocks)

- Table size (in bytes).

# Cost-Based Optimizer

- **Column Statistics Collected**

- ❑ `ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS column-name1,column-name2`

- ❑ Collect the following column level statistics for the specified columns and save them into meta-store

- Maximal column value
    - Minimal column value
    - Number of distinct values
    - Number of null values
    - Average length
    - Max length

# Cost-Based Optimizer

- **Other Operator Estimation**
  - ☐ Filter Cardinality Estimation
  - ☐ Join Cardinality Estimation
  - ☐ Project: does not change row count
  - ☐ Aggregate: consider uniqueness of group-by columns
  - ☐ Limit
  - ☐ Sample



# Cost-Based Optimization

- **Build Side Selection – Two-way Join**
  - Choose one operand as build side and the other as probe side
  - Choose lower-cost child as build side of hash join
    - ✓ Before CBO - build side was selected based on original table sizes
    - ✓ With CBO - build side as selected based on estimate cost of various operators before join

# Cost-Based Optimization

- **Multi-way JOIN Ordering Optimization**

Reorder the joins using a dynamic programming algorithm

- First pull all items (basic joined nodes) into level 0
- Build all 2-way joins at level 1 from plans at level 0
- Build all 3-way joins from plans at previous levels
- Build all 4-way joins etc until build all n-way joins and pick the best from them
- For example, {A,B,C} -> (A Join B) Join C, (A Join C) Join B or (B Join C) Join A, choose the best from them

# Advanced Spark SQL - Encoder

- Internal Row Converters
- Encoder is the fundamental concept in the serialization and deserialization (SerDe) framework in Spark SQL 2.0
- Spark SQL uses the SerDe framework for IO to make it efficient time- and space-wise
- Spark has borrowed the idea from the Hive SerDe library
- Encoder is also called "a container of serde expressions in Dataset"
- Used to convert (encode and decode) any JVM object or primitive of type T (that could be your domain object) to and from Spark SQL's InternalRow which is the internal binary row format representation (using Catalyst expressions and code generation)

# Advanced Spark SQL - Encoder

- ExpressionEncoder
  - Expression-Based Encoder
- LocalDateTimeEncoder
  - Custom ExpressionEncoder for `java.time.LocalDateTime`
- RowEncoder
  - Encoder for DataFrames

# Advanced Spark SQL - Row

- Row is a generic row object with an ordered collection of fields that can be accessed by an ordinal / an index (aka generic access by ordinal), a name (aka native primitive access) or using Scala's pattern matching
- Row is also called Catalyst Row
- Row may have an optional schema
- Row field access by index
- Row companion object offers factory methods to create Row instances from a collection of elements (apply), a sequence of elements (fromSeq) and tuples (fromTuple)

# Advanced Spark SQL – SQL Conf

- SQLConf is an internal key-value configuration store for parameters and hints used in Spark SQL
- Access a session-specific SQLConf using SessionState
- SQLConf is an internal part of Spark SQL and is not meant to be used directly



# Advanced Spark SQL – SQL Conf

## SQL Conf Parameters and Hints

- `spark.sql.adaptive.enabled`
- `spark.sql.autoBroadcastJoinThreshold`
- `spark.sql.broadcastTimeout`
- `spark.sql.cbo.enabled`
- `spark.sql.cbo.joinReorder.enabled`
- `spark.sql.cbo.starSchemaDetection`
- `spark.sql.codegen.fallback`
- `spark.sql.codegen.maxFields`
- `spark.sql.codegen.wholeStage`
- `spark.sql.defaultSizeInBytes`
- `spark.sql.execution.useObjectHashAggregateExec`
- `spark.sql.hive.convertMetastoreOrc`
- `spark.sql.hive.convertMetastoreParquet`
- `spark.sql.inMemoryColumnarStorage.batchSize`
- `spark.sql.inMemoryColumnarStorage.compressed`

# Advanced Spark SQL – SQL Conf

## SQL Conf Parameters and Hints

- `spark.sql.join.preferSortMergeJoin`
- `spark.sql.limit.scaleUpFactor`
- `spark.sql.optimizer.maxIterations`
- `spark.sql.orc.impl`
- `spark.sql.pivotMaxValues`
- `spark.sql.retainGroupColumns`
- `spark.sql.runSQLOnFiles`
- `spark.sql.selfJoinAutoResolveAmbiguity`
- `spark.sql.shuffle.partitions`
- `spark.sql.statistics.fallBackToHdfs`
- `spark.sql.statistics.histogram.enabled`
- `spark.sql.statistics.histogram.numBins`
- `spark.sql.statistics.size.autoUpdate.enabled`
- `spark.sql.streaming.schemaInference`

# Advanced Spark SQL - Parsing

SQL Parser Framework in Spark SQL uses ANTLR to translate a SQL text to a data type, Expression, TableIdentifier or LogicalPlan

- **SparkSqlParser** that is the default parser of the SQL expressions into Spark's types
- **CatalystSqlParser** that is used to parse data types from their canonical string representation

# Advanced Spark SQL - Functions

- `org.apache.spark.sql.functions` object defines many built-in functions to work with Columns in Datasets
- You can access the functions using the following import statement:

```
import org.apache.spark.sql.functions._
```

- There are over 200 functions in the functions object.

```
scala> spark.catalog.listFunctions.count
```

# Advanced Spark SQL - Functions

- **Aggregate Functions**

- count
- grouping
- grouping\_id

- **Collection Functions**

- explode
- explode\_outer
- from\_json

- **Regular Functions**

- col
- expr
- struct
- broadcast

# Advanced Spark SQL - Functions

- **String Functions**

- split
- upper

- **Window Functions**

- row\_number
- ntile
- cume\_dist
- lag
- lead
- rank,
- dense\_rank
- percent\_rant



# Advanced Spark SQL - UDF

- User-Defined Functions (aka UDF) is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL for transforming Datasets.
- UDFs are a blackbox for Spark SQL and it cannot optimize them
- Define a new UDF by defining a Scala function as an input parameter of udf function. It accepts Scala functions of up to 10 input parameters

```
val upper: String => String = _.toUpperCase  
val upperUDF = udf(upper)
```

# Advanced Spark SQL - UDF

- Register UDFs to use in SQL-based query expressions via UDFRegistration

```
spark.udf.register("myUpper", (input: String) => input.toUpperCase)
```

- Query for available standard and user-defined functions using the Catalog interface

```
spark.catalog.listFunctions.filter('name like "%upper%").show(false)
```

# Custom Memory management - Tungsten

Data is stored in off-heap memory in binary format. This saves a lot of memory space. Also there is no Garbage Collection overhead involved. By knowing the schema of data in advance and storing efficiently in binary format, expensive java Serialization is also avoided

# Distributed SQL Engine

- Spark SQL can also act as a distributed query engine
- Run Spark Queries via using JDBC/ODBC or command-line interface similar to run any SQL queries against RDBMS
- Running the Spark SQL CLI
  - `spark-sql`
- Running the Thrift JDBC/ODBC server
  - `start-thriftserver.sh`
  - the server listens on `localhost:10000`
- Use beeline to test the Thrift JDBC/ODBC server
  - `beeline`
  - `!connect jdbc:hive2://localhost:10000`

# Building JIT Data Warehouse

Common pain points of traditional data warehouse:

- Inelasticity of compute and storage resources
- Rigid architecture that's difficult to change
- Limited advanced analytics capabilities

# Building JIT Data Warehouse

## Spark Solution

- Unified platform for a variety of data sources
- On-schema reads for direct data access
- Scale on-demand for maximum elasticity
- Support for advanced data analytics



# Spark SQL for EDA

## What is EDA

An approach to data analysis that attempts to maximize insight into data

- Assess the quality and structure of the data
- Calculate summary or descriptive statistics
- Plot appropriate graphs
- Uncover underlying structures
- Suggest how the data should be modeled
- Detect outliers, errors, and anomalies in the dataset

# Spark SQL for EDA

## Why use Spark for EDA

- In-memory computations
- A high degree of parallelism to achieve interactivity with large distributed data
- Capable of handling petabytes of data
- Provides a set of versatile programming interfaces and libraries

# Spark SQL for EDA

Use Spark SQL for basic data analysis

- Identifying missing data
- Computing basic statistics
- Identifying data outliers
- Visualizing data with Zeppelin
- Sampling data with Spark SQL APIs
- Creating pivot tables

# Spark SQL for EDA

## APIs for computing basic statistics

- **describe** – dataset

Compute statistics for numeric and string columns, including count, mean, stddev, min and max. if no columns are given, this function computes statistics for all numerical or string columns

If you want to programmatically compute summary statistics, use the 'agg' function instead

# Spark SQL for EDA

## APIs for computing basic statistics

- **stat** – dataset

Return DataFrameStatFunctions for working statistic functions support

### ➤ **approxQuantile**

Calculate the approximate quantiles of a numerical column of a DataFrame

### ➤ **cov**

Calculate the sample covariance of two numerical columns of a DataFrame

# Spark SQL for EDA

## APIs for computing basic statistics

- **stat** – dataset

Return DataFrameStatFunctions for working statistic functions support

### ➤ **corr**

Calculates the Pearson Correlation Coefficient of two columns of a DataFrame

### ➤ **crosstab**



# Spark SQL for EDA

## APIs for computing basic statistics

- **stat** – dataset

Return DataFrameStatFunctions for working statistic functions support

### ➤ **crosstab**

Computes a pair-wise frequency table of the given columns. Also known as a contingency table. The first column of each row will be the distinct values of 'col1' and the column names will be the distinct values of 'col2'. The name of the first column will be 'col1\_col2'. Counts will be returned as 'Long's pairs that have no occurrence will have zero as their counts Null elements will be replaced by 'null', and back ticks will be dropped from elements if they exist

# Spark SQL for EDA

## APIs for computing basic statistics

- **stat** – dataset

Return DataFrameStatFunctions for working statistic functions support

### ➤ **freqItems**

Finding frequent items for columns, possibly with false positives

### ➤ **sampleBy**

Returns a stratified sample without replacement based on the fraction given on each stratum

# Spark SQL for EDA

## APIs for computing basic statistics

- **stat** – dataset

Return DataFrameStatFunctions for working statistic functions support

### ➤ **countMinSketch**

Builds a count-min sketch over a specified column

### ➤ **bloomFilter**

Builds a bloom filter over a specified column

# Spark SQL for EDA

## APIs for computing basic statistics

- **na** – dataset

Return DataFrameNaFunctions for working with missing data

### ➤ **drop**

Returns a new 'DataFrame' that drops rows containing any null or NaN values

### ➤ **fill**

Returns a new 'DataFrame' that replaces null or NaN values in numeric columns with 'value'

# Spark SQL for EDA

## APIs for computing basic statistics

- **na** – dataset

Return DataFrameNaFunctions for working with missing data

### ➤ **replace**

Replaces values matching keys in 'replacement' map with the corresponding values

# Spark SQL for Data Munging

What is data munging

Raw data is typically messy, has missing data, duplicate or bad data and requires a series of transformation before it comes into usable format. Data munging is such kind of transformation



# Spark SQL for Data Munging

## Data munging techniques

- Cleaning
- Transformation
  - subset
  - filter
  - aggregate
  - sort
  - merge
  - reshape

# Spark SQL for Data Munging

## Data munging techniques

- Transformation
  - type conversion
  - add new field
  - rename columns
- Statistical analysis
- Visualization

# Spark SQL for Data Munging

## Data munging techniques - Example

- Pre-process the data
- Compute basic statistics and aggregation
- Augment the data with new information
- Analyze missing data
- Dealing with variable length records
- Combine the data using JOIN and analyze the results
- Prepare the data set for machine learning
- De-duplicate the data set

# Spark SQL for Data Munging

Textual data

Data munging techniques for typical text analysis situations include computing word counts, removing stop words, stemming etc.

# Spark SQL for Data Munging

## Time-series data

Time series data is a sequence of values linked to a timestamp, Cloudera provided a new library for analyzing time-series data with Spark, which use spark-ts to define data-time index and create TimeSeriesRDD, handling missing time-series data

# Spark SQL for Data Munging

Dealing with variable length records

- Make the number of fields equal to the maximum number of possible fields by adding empty fields
- Use the `explode()` function



# Spark SQL for Fuzzy Matching

## Fuzzy Matching

- Approaches
  - Good blocking key to choose to reduce the complexity of  $N \times M$  problem
  - Learning based approach like a decision tree. E.g. name = 10% + email = 20% => match
  - Matching with the highest precision

# Spark SQL for Data Munging

## Fuzzy Matching

- Approaches

- Word or Sentence similarities

- ✓ Character based

- ❑ Levensthein, JasoWinkler etc. but punish if there is more info in the middle of name

- ✓ Vector based approach

- ❑ Cosine similarity between two vectors, can use n-grams vector of names and TF-IDFs for them

- ✓ Training based string similarities approaches

- ❑ Word2Vec, Sentence2Vec

# Spark SQL for Data Munging

## Fuzzy Matching

- Approaches

- Soundex Phonetic Algorithm

- ☐ Based on English pronunciation – words that are pronounced the same, but spelled differently

- Other Phonetic Algorithms

- ☐ double\_metaphone
    - ☐ nysiis
    - ☐ refined\_soundex

# Spark SQL for Data Munging

## Fuzzy Matching

- Approaches

- Levenshtein Similarity

- ❑ String Metric for measuring the difference between two sequences

- Other Similarity Functions

- ❑ cosine\_distance
    - ❑ fuzzy\_score
    - ❑ jaccard\_similarity
    - ❑ jaro\_winkler

- Sentence Similarity using Word2Vec

# Enterprise Spark SQL Solutions

## Spark Data Sources APIs for reading the data

### ➤ **BaseRelation**

The abstraction of a collection of tuples read from the data source. It provides the schema of the data

### ➤ **TableScan**

Reads all the data in the data source

### ➤ **PrunedScan**

Eliminates unneeded columns at the data source

### ➤ **PrunedFilteredScan**

Applies predicates at the data source

### ➤ **PushDownScan**

New API that applies complex operations such as joins at the data source

# Enterprise Spark SQL Solutions

## Extending Spark SQL Data Source with Push Down

- Spark SQL provides Data Source APIs for query structured data
- Spark Catalyst Optimizer allows optimizations such as Filter push down and Column pruning, which some of functionalities can be pushed down to the data source
- Push down can improve the query performance significantly by reducing the amount of data transfer and take advantage of the capabilities of the data sources such as index
- Implement the push down via execution strategy
- Spark Catalyst Optimizer uses a combination of heuristics and cost model to support push down



# Enterprise Spark SQL Solutions

## Hive Bucketing in Spark

- When to use bucketing
  - Tables used for frequently in JOINS with same key
  - Loading daily cumulative tables
  - Indexing capability

# Enterprise Spark SQL Solutions

## Hive Bucketing in Spark

- Spark's bucketing support
  - ✓ `df.write.bucketBy(numBuckets, "col").sortBy("col").saveAsTable("bucketed_table")`
  - ✓ create table `bucket_table(col int)` using parquet clustered by (col) sorted by (col) into n buckets
  - ✓ set `spark.sql.sources.bucketing.enabled=true`

# Enterprise Spark SQL Solutions

## Hive Bucketing in Spark

- Bucketing semantics of Spark vs Hive

- ❑ **Hive**

- ✓ Optimizes read, write are costly
    - ✓ A single file represents a single bucket
    - ✓ Each bucket is sorted globally
    - ✓ Hive's inbuilt hash

- ❑ **Spark**

- ✓ Write are cheaper, reads are costlier
    - ✓ A collection of files together comprise a single bucket
    - ✓ Each file is individually sorted but not globally sorted
    - ✓ Murmur3Hash

# Big Data Application Architecture

- Batch Processing
- Stream Processing
- Lambda Architecture

# Big Data Application Architecture

- **Batch Processing**

Batch processing is done on huge volumes of data to create batch views in order to support ad hoc querying and MIS reporting functionality, and/or to apply scalable machine learning algorithms for classification, clustering, collaborative filtering, and analytics applications

# Big Data Application Architecture

- **Stream Processing**

Deal with high data volumes, combined with low-latency processing requirements. In stream processing, data is not collected over significant time periods before triggering the required processing. Commonly, the incoming data is moved to a queuing system, such as Apache Kafka or Amazon Kinesis. This data is then accessed by the stream processor, which executes certain computations on it to generate the resulting output



# Big Data Application Architecture

- **Lambda Architecture**

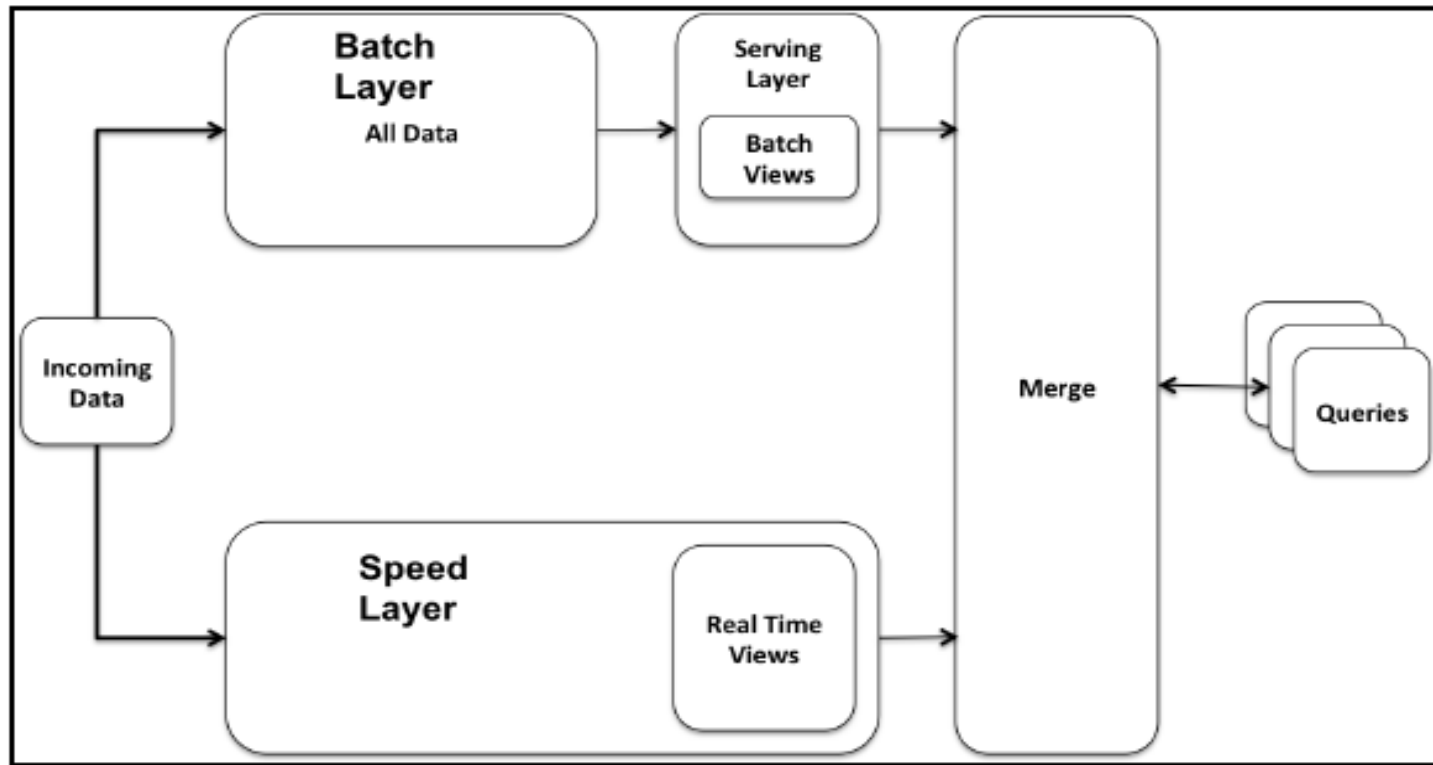
The Lambda architectural pattern attempts to combine the best of both worlds—batch processing and stream processing

This pattern consists of several layers:

- Batch Layer - ingests and processes data on persistent storage such as HDFS and S3
- Speed Layer - ingests and processes streaming data that has not been processed by the Batch Layer yet
- Serving Layer - combines outputs from the Batch and Speed Layers to present merged results

# Big Data Application Architecture

- Lambda Architecture



# Developing Big Data Applications

- Using Spark SQL in ETL Applications
- Using Spark SQL in Streaming Applications
- Using Spark SQL in Text Analysis Applications
- Using Spark SQL in Machine Learning Application
- Using Spark SQL in Deep Learning Applications