Spark SQL

* spark module for structured data processing
* component on top of Spark Core that introduces a new data abstraction called SchemaRDD.
* first release ⇒ Spark 1.0 (May, 2014)
* Spark introduces programming module for structured data processing called Spark SQL
* It provides a programming abstraction called Dataframe and can act as distributed SQL query engine

Challenges

1) Perform ETL to and from various data sources

⇓

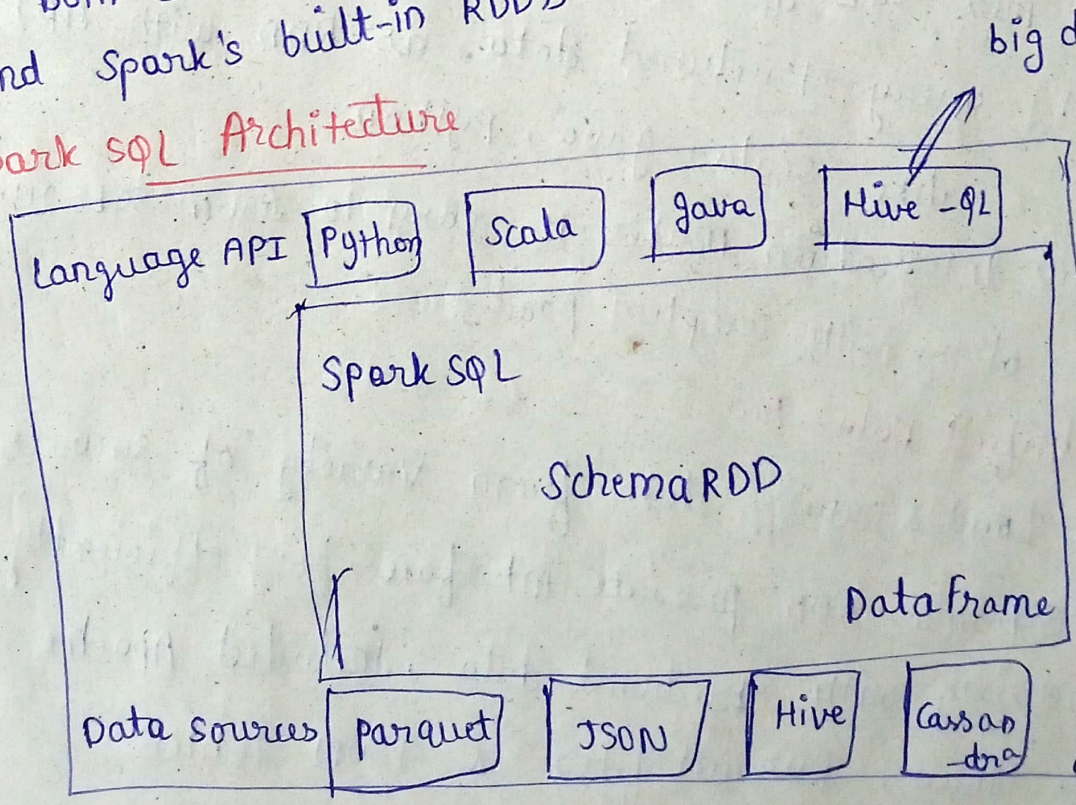
Soln ⇒ Dataframe API that can perform relational operations on both external data sources and Spark's built-in RDDs

2) perform advanced analytics (ML etc.,)

⇓

highly extensible optimizer, catalyst that uses features of Scala.

Spark SQL Architecture



1) Language API

* Spark is compatible with different languages & Spark SQL
* API ⇒ Python, Scala, Java, Hive SQL

2) Schema RDD :-

* Spark Core is designed with special data structure called RDD
* Spark SQL works on schemas, tables, records
* ∴ We can use Schema RDD as temporary table
* We can call this Schema RDD as Dataframe

3) Data Sources :-

* Data source for spark-core ⇒ text file, Avro file
* " " for spark-SQL ⇒ parquet file, JSON doc, HIVE tables, Cassandra db

Features Of Spark SQL

1) Integrated

(i) mix SQL queries with spark programs

(ii) query structured data as RDD with APIs like Scala, Java, Python

(iii) Integration makes it easy to run SQL queries along with complex prob.

2) Unified Data-Access

1) Load & query data from variety of sources
2) Schema-RDD provide interface for efficiently working with structured data, included Apache Hive tables, parquet files & JSON tables

3) Hive compatibility

* Run unmodified Hive queries on existing warehouses

* Hive => distributed, fault-tolerant data warehouse system that enables analytics at massive scale.

4) Standard connectivity :- => Connect through JDBC & ODBC
Ex:- Tableau, Qlik (Java-based servers)

5) Scalability :- use same engine for both interactive & long queries, support mid-query fault tolerance

User-Defined Functions => own processing code & invoke it from Hive query.

- UDF => for single row.
- UDAF => User defined Aggregate Function (multiple rows i/p)
- UDTF => " " Table-generating function (single i/p row)

Spark RDD

* fundamental data structure
* immutable distributed collection of objs that can be stored in memory or disk across cluster
* Each dataset in RDD → divided into logical partitions
  ⇓
  Computed on diff nodes of cluster
* Automatically rebuilt when failure occurs
* Parallel functional transformations (map, filter)
* RDDs can contain any type of Python, Java or Scala objs including user-defined classes
* read-only, partition partitioned collection of records

* RDDs can be created through deterministic operations on either data
* fault-tolerant collection of elements that can be operated on in parallel.

* Two ways to create RDD
  (i) parallelizing an existing collection in driver program.
  (ii) referencing data in external storage system such as a shared file system, HDFS.

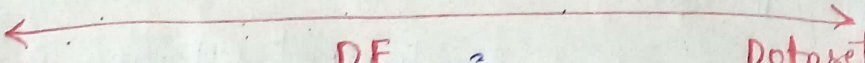
** RDD is used to achieve faster and efficient mapreduce operations.

Dataset and Dataframe

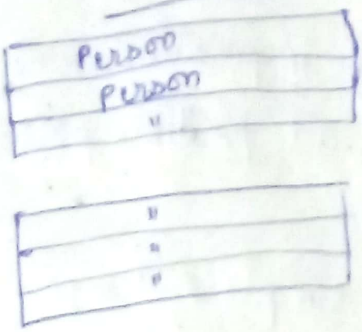
* distributed collection of data, which is organized into named columns.
* It is equivalent to relational tables
* DF can be constructed from an array of different sources like Hive tables, existing RDDs, external files
* used for Big data and data science applications etc.;

Dataframe :- Data is organized into named columns like table in relational database

Dataset :- distributed collection of data
  - added in Spark-1.6
  - static typing & runtime = safety



	SQL	DF	Datasets
Syntax E	Runtime	compile Time	CT
Analysis E	RT	RT	CT



Dataframe

Name	Age	Height
string	int	double
"	"	"
"	"	"
"	"	"
"	"	"
"	"	"

Features of DF :-

1) ability to process data in KB to PB on single node cluster to large cluster.
2) supports different data formats → Avro, csv etc, and storage sys (Mysql, Hive tables)
3) provides API for Python, Java, Scala & R.
4) can be easily integrated with all Big data tools and frameworks
5) Provides API for Python, Java, Scala & R.

Spark SQL :-

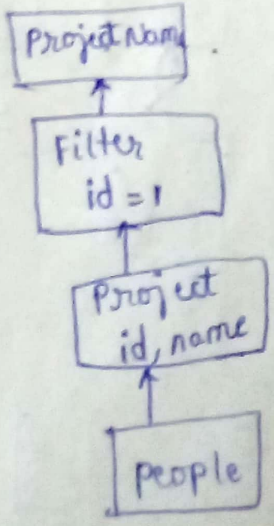
1) Write less code
2) Read less data
3) Let optimizer do more work

Ex:-

```
SELECT name
FROM (
    SELECT id, name
    FROM People) p
    where p.id = 1
```

~~optimized~~ ~~Execution~~

Logical plan



Plan optimization & Execution

Optimized execution

1) Writing imperative code to optimize all possible patterns is hard

2) So, write simple rules :-
   (i) Each rule makes one change
   (ii) Run many rules together to fixed pt.