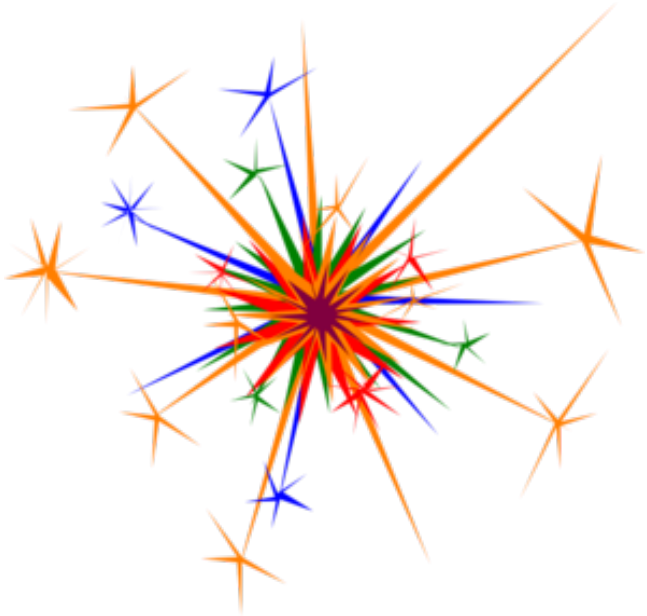


Master Spark SQL

Apache Spark Training Series

iT 21 LEARNING

About Me

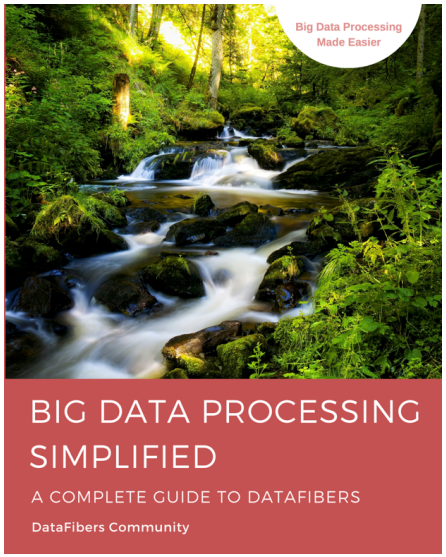
Will Du

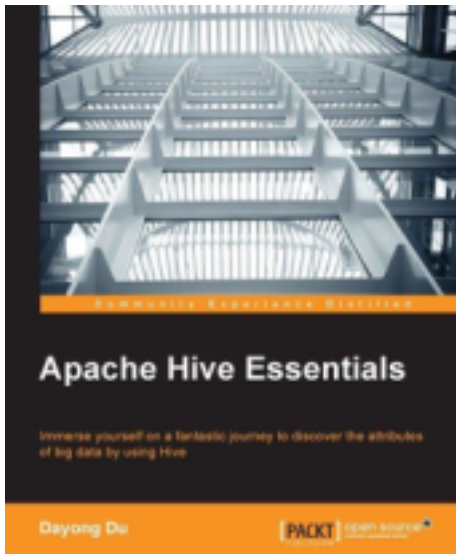
Big Data Practitioner | Author | Coach

TD • Dalhousie University

Toronto, Canada Area • 500+

- ✓ Big data since 2009
- ✓ Author of Apache Hive and Data Stream books
- ✓ Teacher of *Master Big Data/Spark Essential* @ IT21
- ✓ Founder of DataFibers
- ✓ Co-funder and advisors of few associations and start-ups





Agenda of Today

- Spark SQL Overview
  - ✓ What is Spark SQL and Why
  - ✓ Spark SQL Architecture
  - ✓ Spark SQL Features and Advantages
  - ✓ Spark SQL Internal – Catalyst
- Spark SQL Core API
  - ✓ Data Source API
  - ✓ UDF
- Spark SQL Exercise

Spark SQL Overview

A Quick Review - What is Apache Spark?

Fast and general cluster computing system, interoperable with Hadoop, included in all major distributions

- Improves efficiency through:
  - In-memory computing primitives
  - General computation graphs
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell

Write programs in terms of transformations on distributed datasets

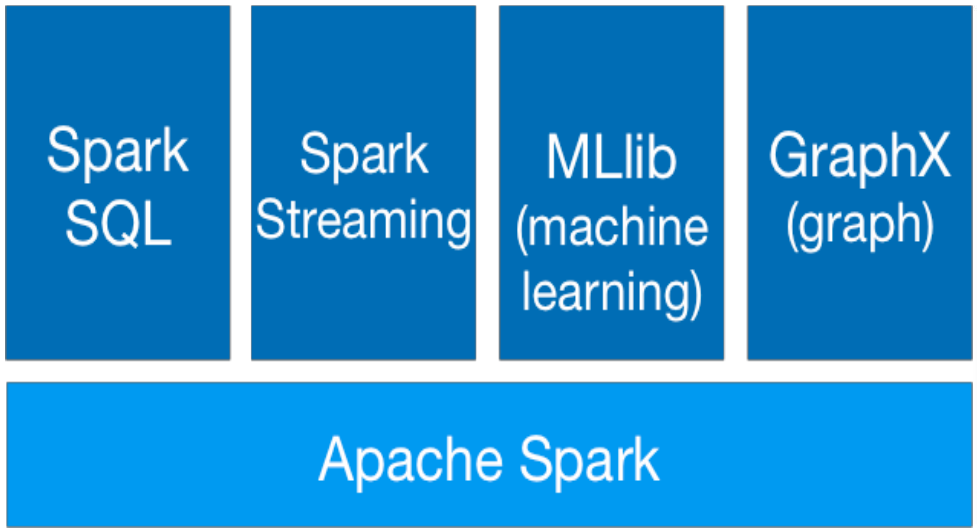
Resilient Distributed Datasets (RDDs)

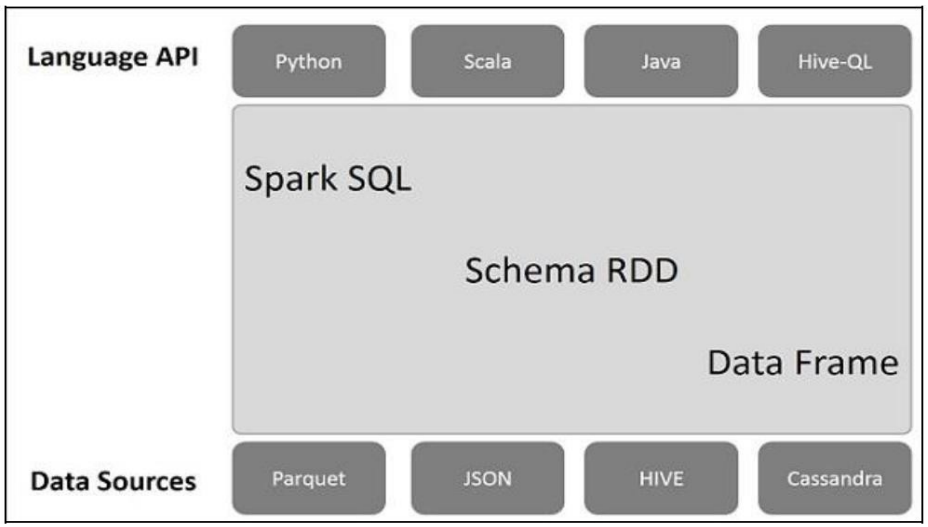
- Collections of objects that can be stored in memory or disk across a cluster
- Parallel functional transformations (map, filter, …)
- Automatically rebuilt on failure

Up to 100× faster (2-10× on disk)
2-5× less code

What is Spark SQL

- Spark SQL is a Spark module for structured data processing.
- Spark SQL is a component on top of Spark Core that introduces a new data abstraction called SchemaRDD.





Why it comes to SQL again? - Challenges and Solutions

Challenges

- Perform ETL to and from various (semi- or unstructured) data sources faster
- Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems faster and flexible
- Easy migration from hive data warehouse

Solutions

- A *DataFrame/DataSet* API that can perform relational operations on both external data sources and Spark's built-in RDDs.
- A highly extensible optimizer, *Catalyst*, that uses features of Scala to add composable rule, control code gen., and define extensions.

Spark SQL Success Stories

Twitter Sentiment Analysis With Spark SQL

Trending Topics can be used to create campaigns and attract larger audience

Sentiment helps in crisis management, service adjusting and target marketing

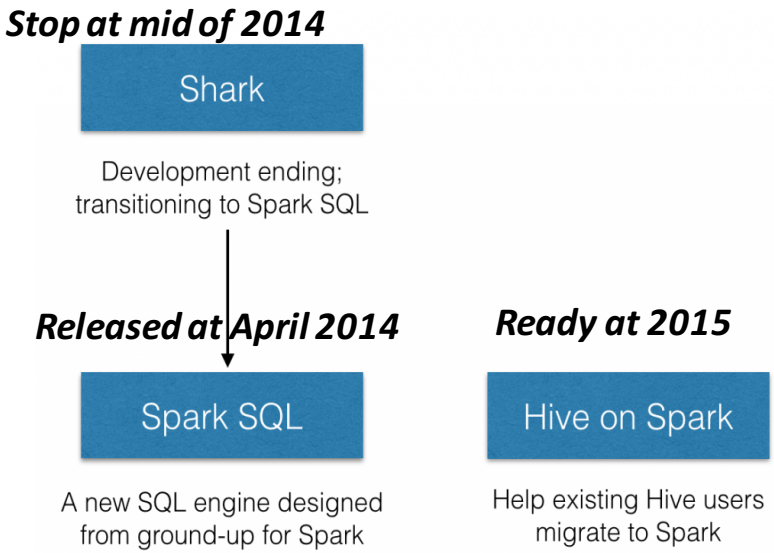
NYSE: Real Time Analysis of Stock Market Data

Banking: Credit Card Fraud Detection

Genomic Sequencing

History About Spark SQL

- Spark SQL was first released in Spark 1.0 (April, 2014).
- Initial committed by Michael Armbrust & Reynold Xin from Databricks.
- Spark introduces a programming module for structured data processing called DataFrame
- It provides a programming abstraction and can act as distributed SQL query engine.



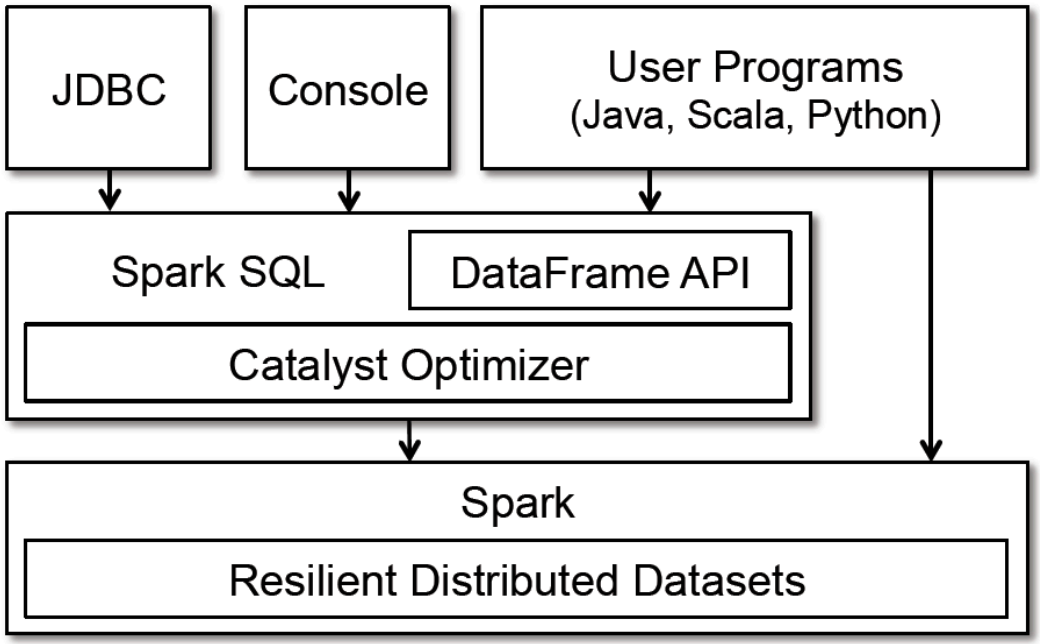
Spark SQL Architecture

- **Language API:**
  - ✓ Spark is compatible with different languages and Spark SQL.
  - ✓ It is also supported by these languages - API (python, scala, java, R, SQL).

- **Schema RDD:**
  - ✓ Spark Core is designed with special data structure called RDD.
  - ✓ Generally, Spark SQL works on schemas, tables, and records.
  - ✓ Therefore, we can use the Schema RDD as temporary table.
  - ✓ We can call this Schema RDD as Data Frame/Set.

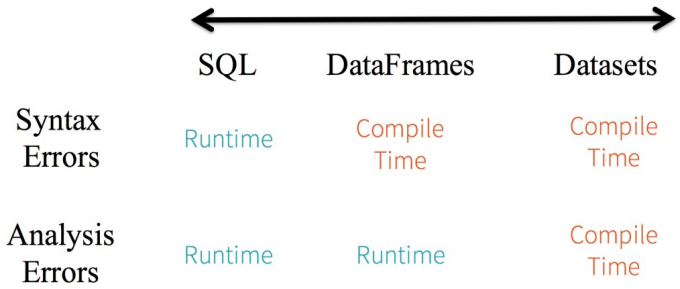
- **Data Sources:**
  - ✓ Spark SQL supported many type of source, such as CSV file, Avro file, Parquet file, JSON document, Hive tables, and Cassandra database.

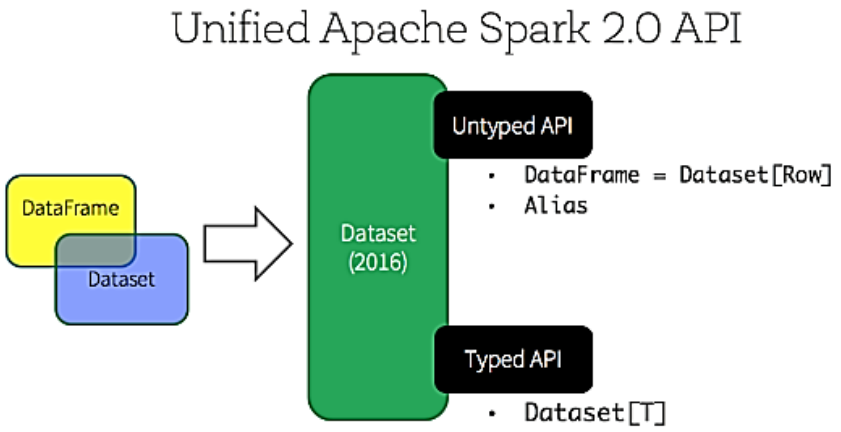
Spark Programming Interface



Review: DataFrame vs. DataSet

- DataFrame is an immutable distributed collection of data which is organized into named columns, like a table in a relational database.
- Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (compile time error, strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine.
- In Spark 2.0, DataFrame APIs will merge with Datasets APIs, unifying data processing capabilities across libraries.
- Now – Only DataSet



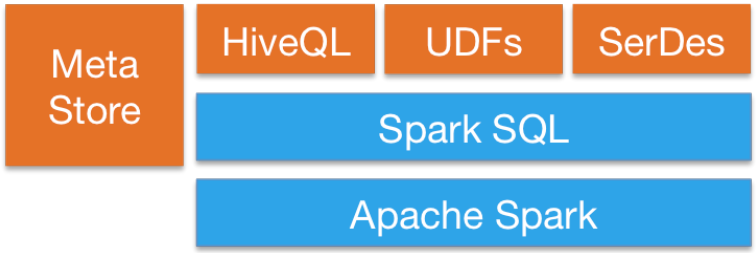


Spark SQL Features

1. Integrated:
   - Seamlessly mix SQL queries with Spark programs.
   - Spark SQL lets you query structured data as a distributed dataset (RDD) in Spark, with integrated APIs in Python, Scala and Java.
   - This tight integration makes it easy to run SQL queries alongside complex analytic algorithms.
2. Unified Data Access:
   - Load and query data from a variety of sources.
   - Schema-RDDs provide a single interface for efficiently working with structured data, including Apache Hive tables, parquet files and JSON files.
3. Hive Compatibility:
   - Run unmodified Hive queries on existing warehouses.
   - Spark SQL reuses the Hive frontend and MetaStore, giving you full compatibility with existing Hive data, queries, and UDFs.
   - Simply install it alongside Hive.

Spark SQL Features – More about Spark SQL on Hive

- Spark SQL reuses the Hive frontend and metastore
- Spark SQL gives you full compatibility with existing Hive data, queries, and UDFs.
- Support almost all hive query – HiveContext, which is superset of SQLContext.
  Use SparkSession.builder() for all after 2.0.0
- Spark SQL also uses Hive UDF and SerDe




```
val spark = SparkSession
  .builder()
  .appName("Spark  Hive Example")
  .config("spark.sql.warehouse.dir",  warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()

...

spark.sql("CREATE  TABLE IF NOT EXISTS src (key INT, value STRING)")
Spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
```

Spark SQL Features *Cont.*

4. Standard Connectivity:
   - Connect through JDBC or ODBC.
   - Spark SQL includes a server mode with industry standard JDBC and ODBC connectivity.

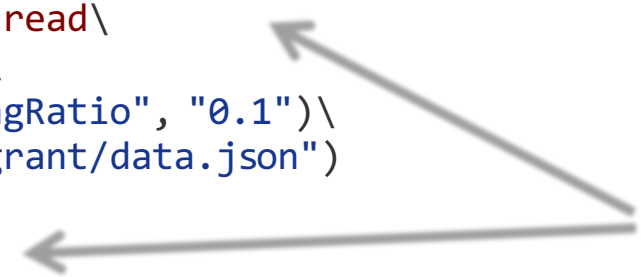
5. Scalability:
   - Use the same engine for both interactive and long queries.
   - Spark SQL takes advantage of the RDD model to support mid-query fault tolerance, letting it scale to large jobs too.
   - Do not worry about using a different engine for historical data.

Spark SQL Advantages – Write Less Code: Input & Output

- Unified interface to reading/writing data in a variety of formats:

```
df = sqlContext.read\
.format("json")\
.option("samplingRatio", "0.1")\
.load("/home/vagrant/data.json")

df.write\
.format("parquet")\
.mode("append")\
.partitionBy("year")\
.saveAsTable("fasterData")
```



read and write functions create new builders for doing I/O

Spark SQL Advantages – Write Less Code: Input & Output

- Unified interface to reading/writing data in a variety of formats:

```
df = sqlContext.read\
.format("json")\
.option("samplingRatio", "0.1")\
.load("/home/vagrant/data.json")

df.write\
.format("parquet")\
.mode("append")\
.partitionBy("year")\
.saveAsTable("fasterData")
```

Builder methods are used to specify:

- Format
- Partitioning
- Handling of existing data
- and more

Spark SQL Advantages – Write Less Code: Input & Output

- Unified interface to reading/writing data in a variety of formats:

```
df = sqlContext.read\
.format("json")\
.option("samplingRatio", "0.1")\
.load("/home/vagrant/data.json")

df.write\
.format("parquet")\
.mode("append")\
.partitionBy("year")\
.saveAsTable("fasterData")
```

load(…), save(…) or saveAsTable(…) functions create new builders for doing I/O

Spark SQL Advantages – Read Less Data with Efficient Formats

- Parquet is an efficient columnar storage format:
  - ✓ Compact binary encoding with intelligent compression (delta, RLE, etc).
  - ✓ Each column stored separately with an index that allows skipping of unread columns.
  - ✓ Data skipping using statistics (column min/max, etc).

Spark SQL Advantages – Write Less Code: Operation Example

- Using RDDs

```
data = sc.textFile(...).split("\t")  data.map(lambda x: (x[0], [int(x[1]), 1]))
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]])
    .map(lambda x: [x[0], x[1][0] / x[1][1]])
    .collect()
```

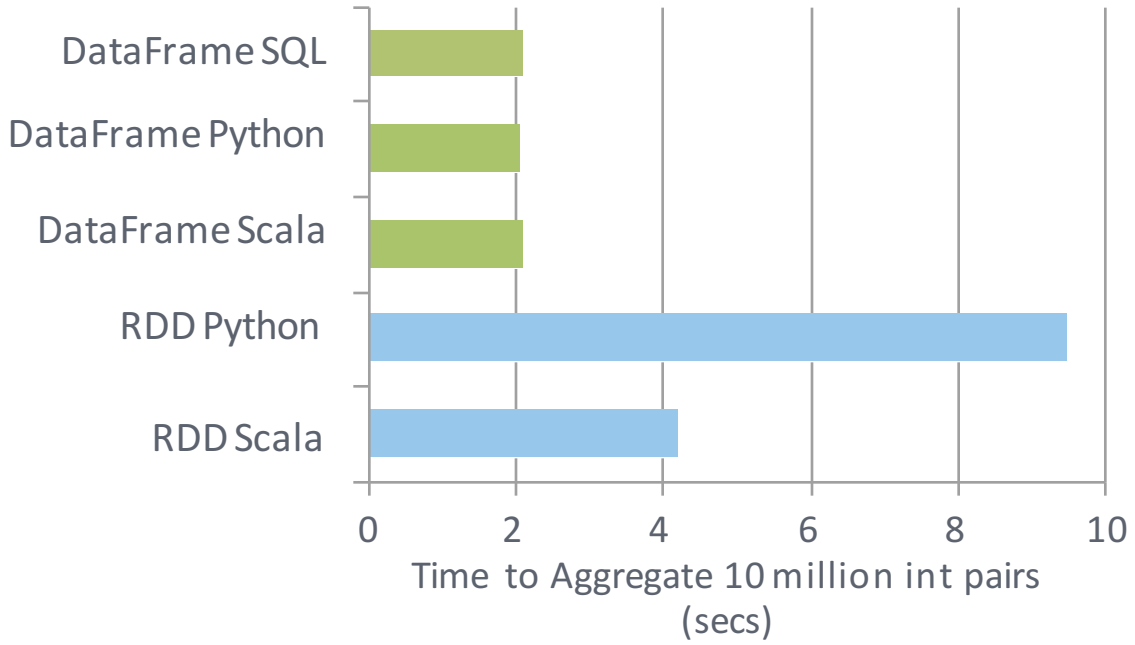
- Using DataFrames

```
sqlCtx.table("people").groupBy("name").agg("name", avg("age")).collect()
```

- Using SQL

```
SELECT name, avg(age) FROM people GROUP BY name
```

Spark SQL Advantages – Less Code but Fast Run



Side Note: Spark SQL for DataFrame/DataSet

- A distributed collection of rows with the same schema (RDDs suffer from type erasure)
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects (SchemaRDDs as of Spark < 1.3)
- Evaluated lazily → unmaterialized *logical* plan

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

Side Note: Spark SQL for DataSet Cont.

- Relational operations (select, where, join, groupBy) via a DSL
- Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

Side Note: Spark SQL on DataSet API vs. SQL API

Advantages

- Holistic optimization across functions composed in different languages.
- Control structures (e.g. *if*, *for*) is more flexiable
- Identify code errors associated with data *schema* issues on the fly.

Disadvantages

- Dataset API requires more coding and more learning curve
- Static string passing SQL is more powerful
- Business logic to implementation process is difficult to understand
- Power of SQL optimization for now and future

Spark SQL Advantages – Querying Native Datasets

- Infer column names and types directly from data objects (via reflection in Java and Scala and data sampling in Python, which is dynamically typed)

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

- Native objects accessed in-place to avoid expensive data format transformation.

- Benefits:
  - Run relational operations on existing Spark programs.
  - Combine RDDs with external structured data
  - Columnar storage with *hot* columns cached in memory

Spark SQL Advantages – User-Defined Functions (UDFs)

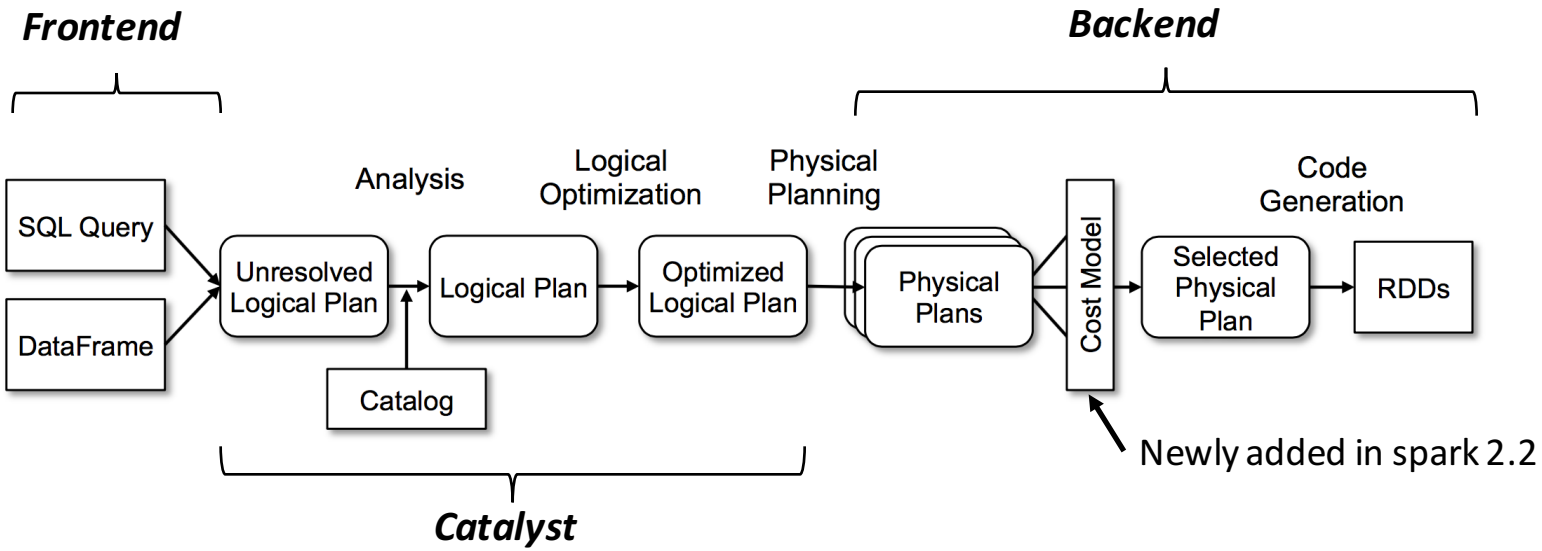
- Easy extension of limited operations supported.
- Allows inline registration of UDFs
  - Compare with Pig, which requires the UDF to be written in a Java package that's loaded into the Pig script.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

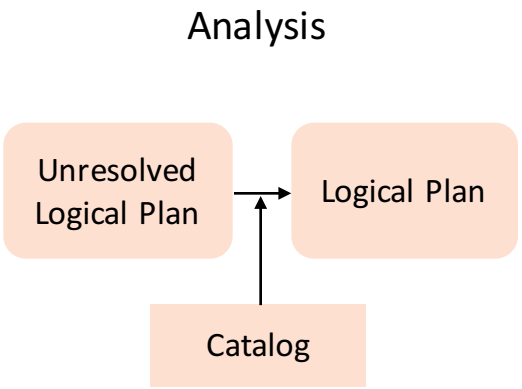
Spark SQL – How Optimization Works with Catalyst



DataFrames/DataSet and SQL share the same optimization/execution pipeline

Spark SQL – How It Works with Catalyst - Analysis

- An attribute is *unresolved* if its type is not known or it's not matched to an input table.
- To resolve attributes:
  - Look up relations by name from the catalog.
  - Map named attributes to the input provided given operator's children.
  - UID for references to the same value
  - Propagate types through expressions (e.g. 1 + *col*)



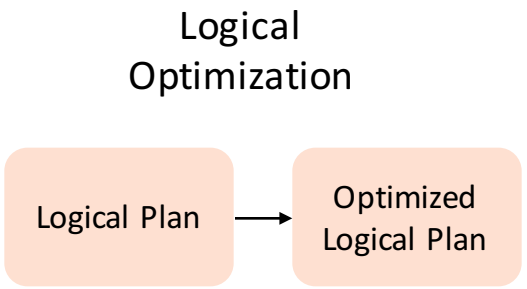
SELECT *col* FROM *sales*

Spark SQL – How It Works with Catalyst – Logical Optimization

- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc.)

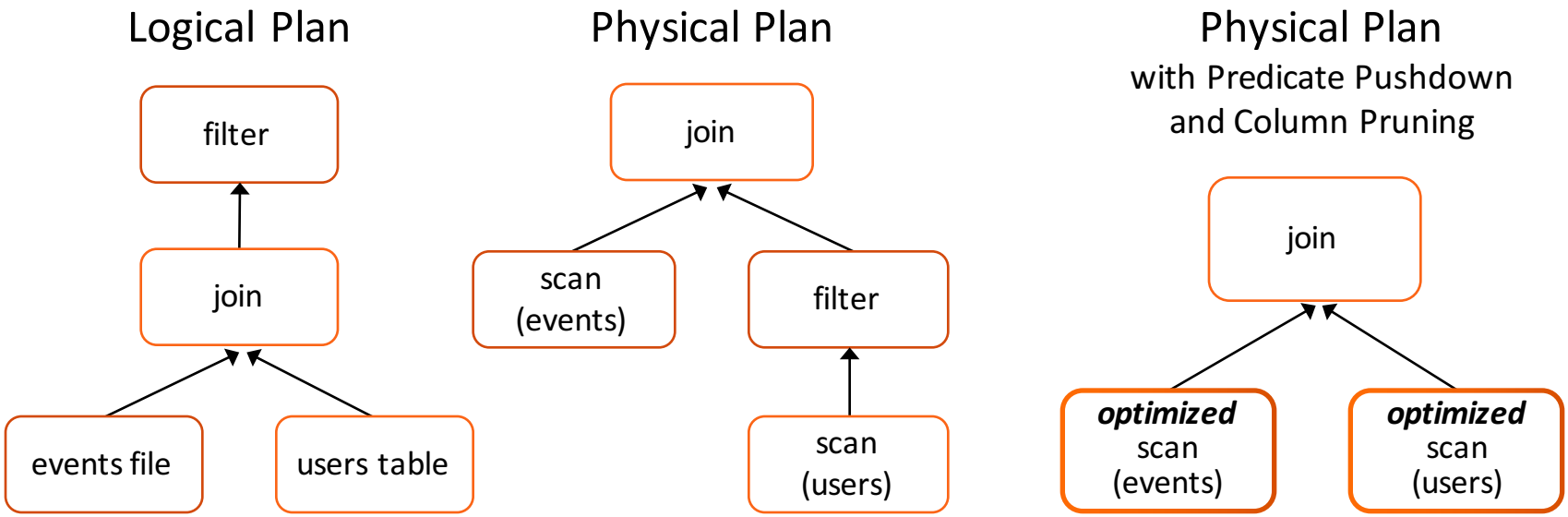
```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
              if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```



Spark SQL – How It Works with Catalyst – Physical Planning

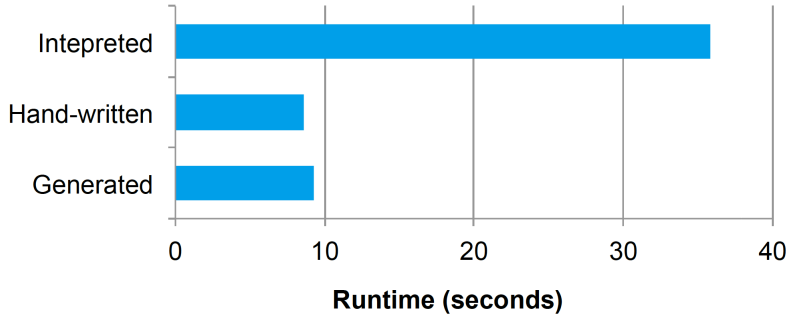
```
def add_demographics(events):
   u = sqlCtx.table("users")                    # Load partitioned Hive table
   events
     .join(u, events.user_id == u.user_id)      # Join on user_id
     .withColumn("city", zipToCity(df.zip))     # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```



Spark SQL – How It Works with Catalyst – Code Generation

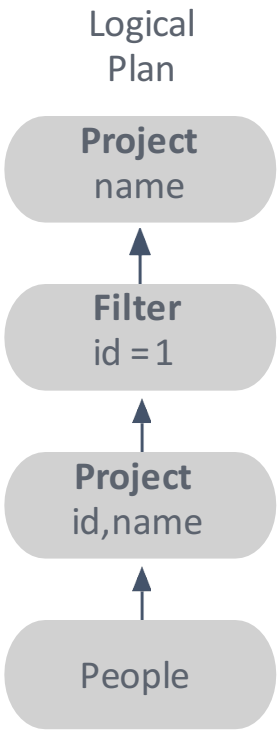
- Relies on Scala's *quasiquotes (notation to manipulate syntax tree)* to simplify code gen.
- Catalyst transforms a SQL tree into an abstract syntax tree (AST) for Scala code to evaluate expression and generate code
- 700LOC

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```



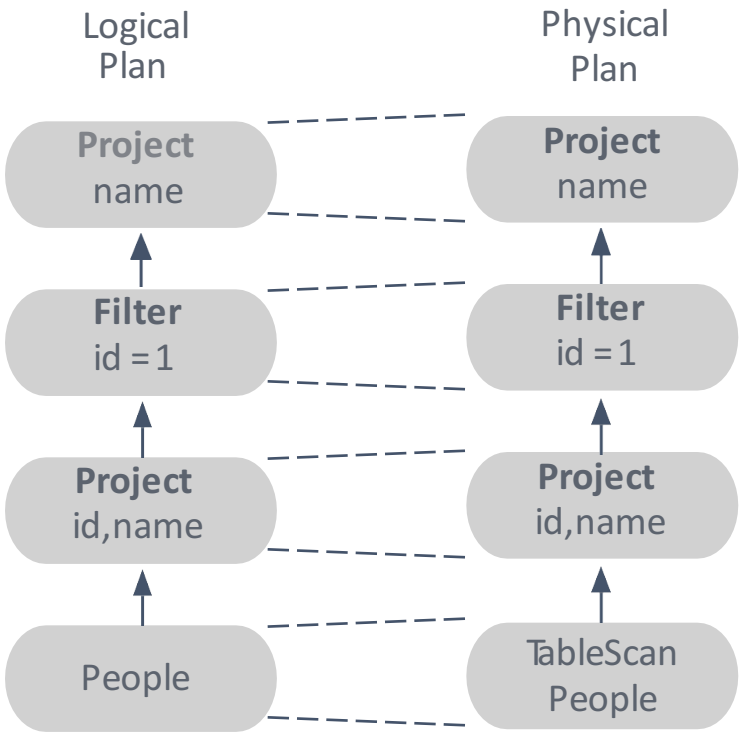
Side Note: Optimization Example – Start From an Example Query

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

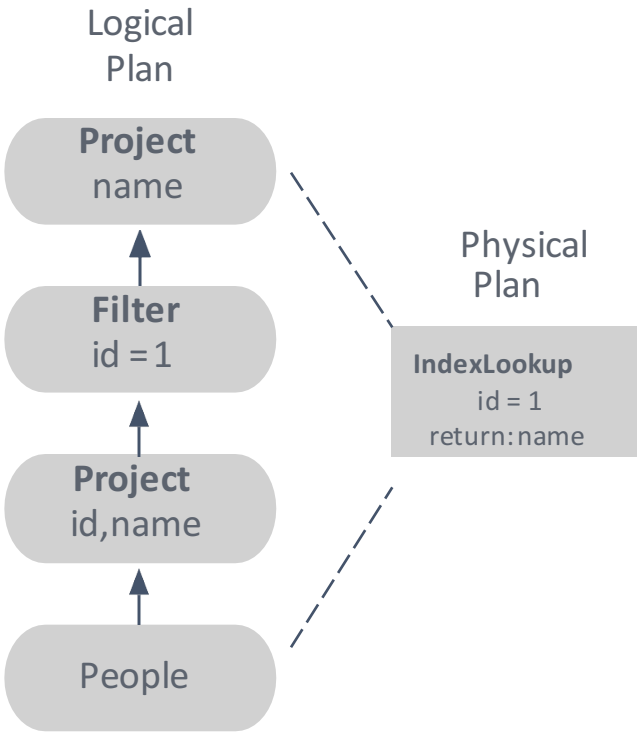


Optimization Example - Native Query Planning

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



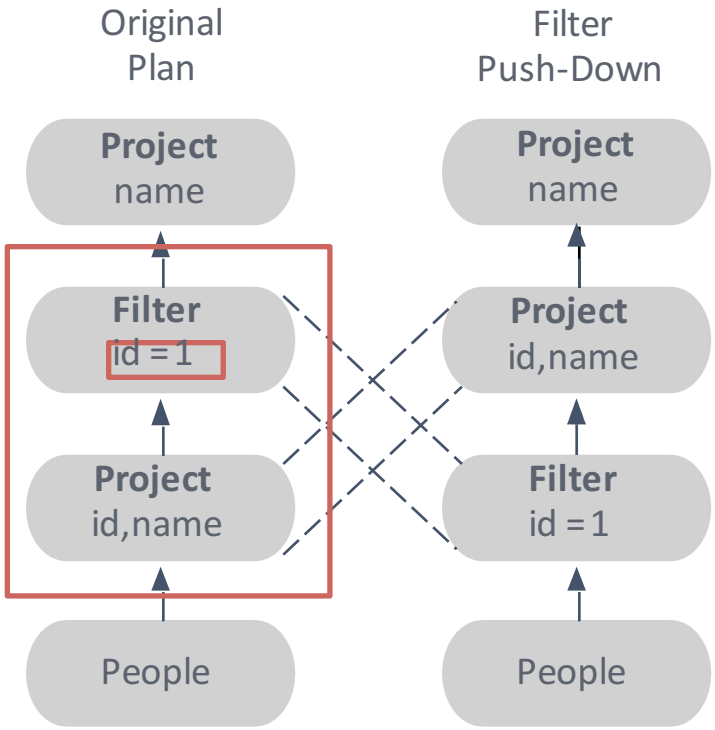
Optimization Example - Optimized Execution



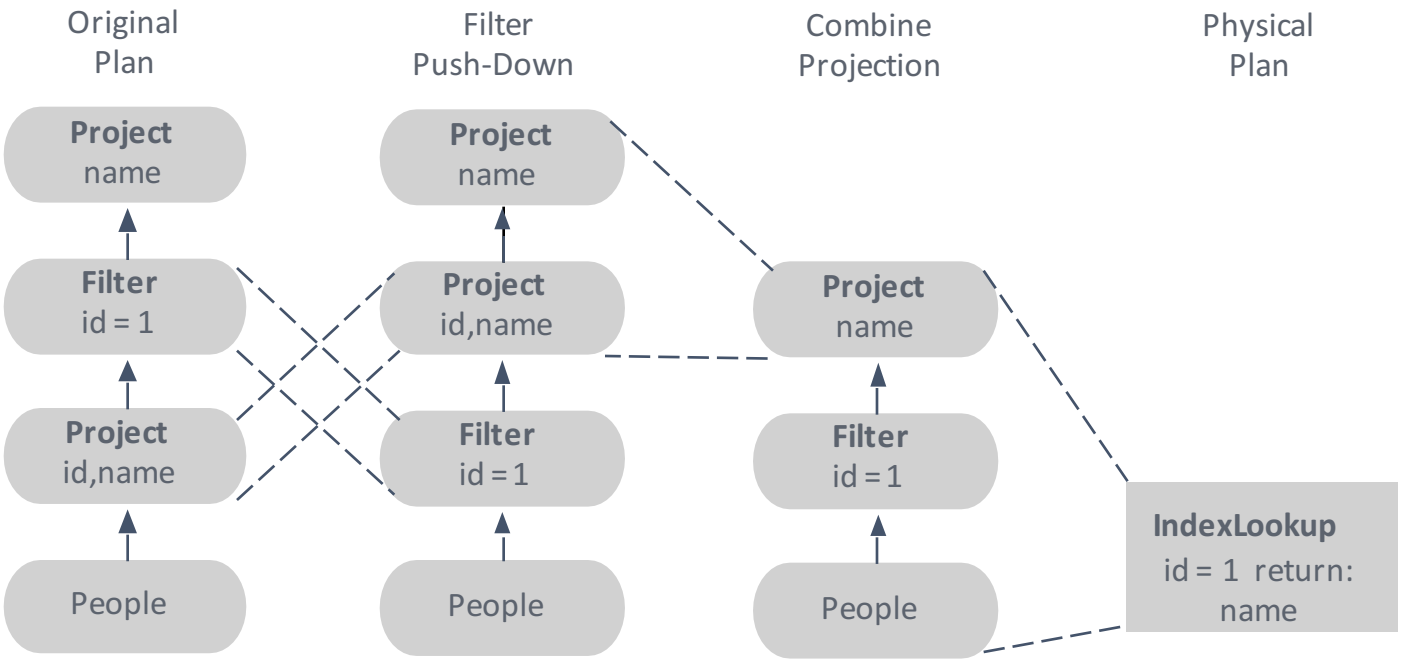
- Writing imperative code to optimize all possible patterns is hard.
- Instead write simple rules:
  - ✓ Each rule makes one change
  - ✓ Run many rules together to fixed point.

Writing Rules as Tree Transformation

- Find filters on top of projections.
- Check that the filter can be evaluated without the result of the project.
- If so, switch the operators.

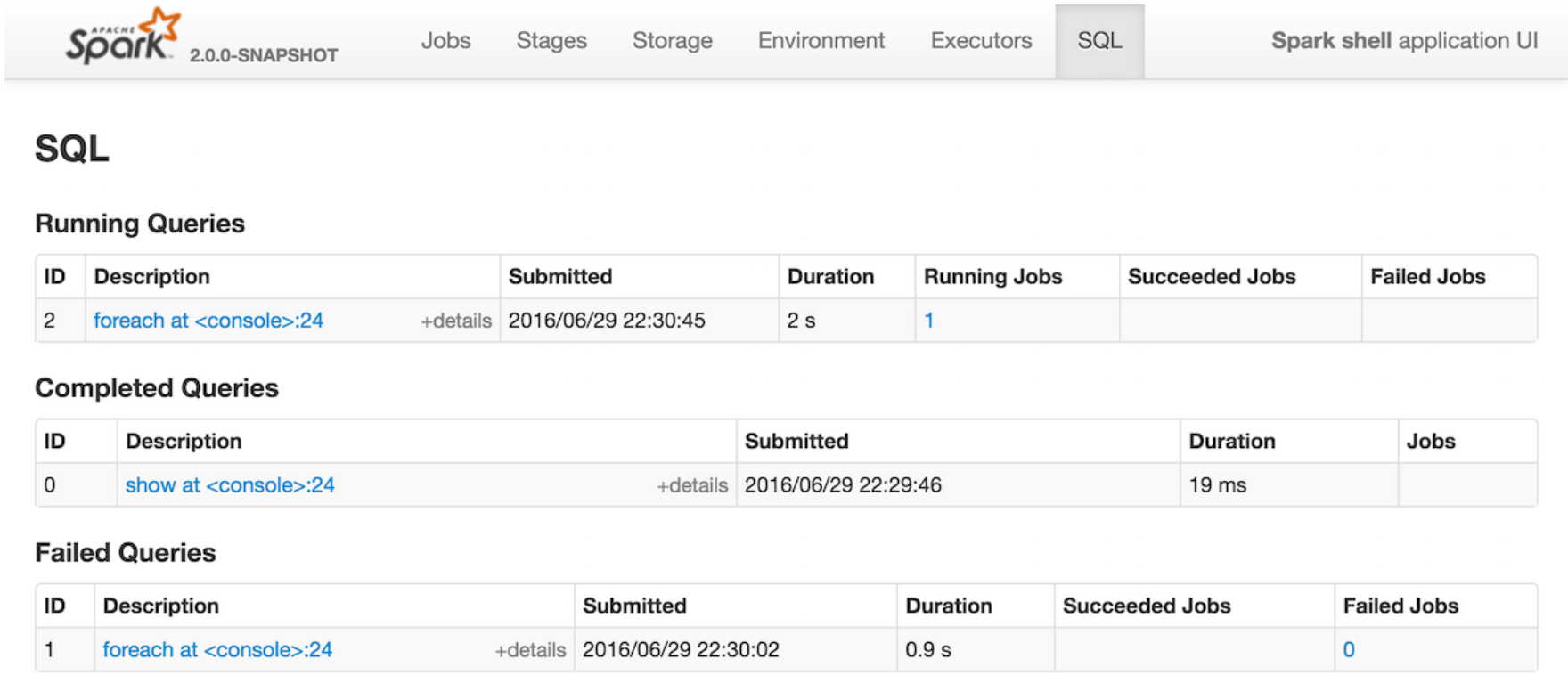


Optimization Example - Optimizing With Rules



Side Note: See the Physical Plan from SQL Tab

SQL tab in web UI shows SQLMetrics per physical operator in a structured query physical plan.
You can access the SQL tab under /SQL URL, e.g. http://localhost:4040/SQL/



Side Note: Spark Project Tungsten

Overcome JVM limitations:

- **Memory Management and Binary Processing**: leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection
- **Cache-aware computation**: algorithms and data structures to exploit memory hierarchy
- **Code generation**: using code generation to exploit modern compilers and CPUs

Spark SQL Core API

Data Source API and Dataset API

Before Start – Access/Setup Spark SQL Environment

Scala/Java IDE, such as IDEA/Eclipse Scala

- Add spark for Hadoop jar or use maven
- Run local or compile to jar and submit with spark-submit

Use command spark-shell

- Interactive command line application
- Get SparkSession or SQLContext.
- /bin/spark-shell for scala, ./bin/pyspark for python

Use command spark-sql

- Where spark using integrated hive metadata (does not work/configured in CDH some version)

Use other JDBC/ODBC client tool through thrift server

- Beeline
- Oracle SQL Developer

Side Notes: Setup Spark SQL Thrift Server

- Spark SQL thrift server is an interface for JDBC/ODBC client
- Spark thrift server is similar to the hive one but separate instance
- It is important for Spark SQL based ETL and Business Intelligence reporting
- Spark thrift server is not production GA, but some companies are still using it in production.

If it is not started in HDP, we'll can start it manually in following steps.

By default, HDB start spark2 thrift server at 10016

```
[root@sandbox spark-client]# cd /usr/hdp/current/spark-client
[root@sandbox spark-client]# ./sbin/start-thriftserver.sh --master yarn-client --executor-memory 512m --
hiveconf hive.server2.thrift.port=10016
[root@sandbox spark-client]# beeline
beeline>!connect jdbc:hive2://localhost:10016
beeline> user name:
beeline> password:
```

Extensible Input & Output Data Source

Spark's Data Source API allows optimizations like column pruning and filter pushdown into custom data sources.

Built-In

External

and more…

Spark SQL Data Source API – Files, such as Json/Parquet

- Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row]. This conversion can be done using *SparkSession.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.

- A more generic option of reading file is available using *load* with specified file format, such as json, parquet, jdbc, orc, csv, text, etc. Below is an example.

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

A Quick Demo and Exercise

- Download the data from https://raw.githubusercontent.com/datafibers/data_set/master/shopping_data_json.json
- Put it to a local Linux folder or HDFS folder or both
- Load the json file using SparkSession in spark-shell
- Explore the file using Spark SQL

Exercise Questions After Demo

1. Find out the lady who spent most money with bankcard card
2. Write the result into parquet file

Note:

- To write the file to specific format using dataset.write.format(”file_format").save("file_path")
- After write the file, using generic file load method to verify it

Side Note: Save Mode

df.write.mode(SaveMode.ErrorIfExists).save(...)

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists(default)	"error"(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.

Spark SQL Data Source API - Avro

Site

https://github.com/databricks/spark-avro

Scala API

```
val df = spark.read .format("com.databricks.spark.avro") .load("src/test/resources/episodes.avro")
```

Java API

```
Dataset<Row> df = spark.read().format("com.databricks.spark.avro")
.load("src/test/resources/episodes.avro");
```

SQL API

```
CREATE TEMPORARY TABLE episodes
USING com.databricks.spark.avro
OPTIONS (path "src/test/resources/episodes.avro")
```

Spark SQL Data Source API - JDBC

Include the lib

bin/spark-shell --driver-class-path postgresql-9.4.1207.jar --jars postgresql-9.4.1207.jar

Create context

```
scala> val dataframe_mysql = sqlContext.read.format("jdbc")
.option("url", "jdbc:mysql://localhost/sparksql")
.option("driver", "com.mysql.jdbc.Driver")
.option("dbtable", "baby_names")
.option("user", "root").option("password", "root")
.load()
```

Show the schema

```
scala> dataframe_mysql.show
```

Register table

```
scala> dataframe_mysql.registerTempTable("names")
```

Explore the data

```
scala> dataframe_mysql.sqlContext.sql("select * from names").collect.foreach(println)
```

Spark SQL Data Source API - Hive

- Spark SQL also supports reading and writing data (df.write.saveAsTable("test")) stored in Hive.
- To configure Spark working with hive, place your *hive-site.xml*, *core-site.xml* (for security configuration), and *hdfs-site.xml* (for HDFS configuration) file in *$SPARK_HOME/conf/.*
- When working with Hive in spark-shell, one must instantiate SparkSession with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions.
- Users who do not have an existing Hive deployment can still enable Hive support. When not configured by the *hive-site.xml*, the context automatically creates *metastore_db* in the current directory and creates a directory configured by *spark.sql.warehouse.dir*, which defaults to the directory spark-warehouse in the current directory that the Spark application is started.

Note: The *hive.metastore.warehouse.dir* property in hive-site.xml is deprecated since Spark 2.0.0. Instead, use *spark.sql.warehouse.dir* to specify the default location of database in warehouse.

Side Note: Quick Hive Overview

Data Structure	Logical	Physical
Database	A collection of tables	Folder with files
Table	A collection of rows of data	Folder with files
Partition	Columns to split data	Folder
Row	Line of records	Line in a file
Columns	Slice of records	Specified positions in each line

Spark SQL Data Source API – Memory Cache

- Spark SQL can cache tables using an in-memory columnar format by calling

spark.cacheTable("tableName") or ***dataFrame.cache()*** or "***CACHE table tableName***" in spark sql statement in spark-sql or .sql(statement)

- Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure.

- Do not forget to call ***spark.uncacheTable("tableName")*** or ***UNCACHE table tableName*** to remove the table from memory when necessary.

Spark SQL UDF and Demo

1. **Create a function (scala)**

```
val squared = (s: Int) => {
s * s
 }
```

2. **Register the function as a UDF**

```
spark.udf.register("square", squared)
```

3. **Call the UDF in Spark SQL**

```
spark.range(1, 20).registerTempTable("test")
spark.sql("select id, square(id) as id_squared from test").show
```

Spark SQL/Hive Exercise

https://github.com/datafibers/spark_training

Thank You

Questions?