Introduction of Apache Spark SQL

TANG Gen

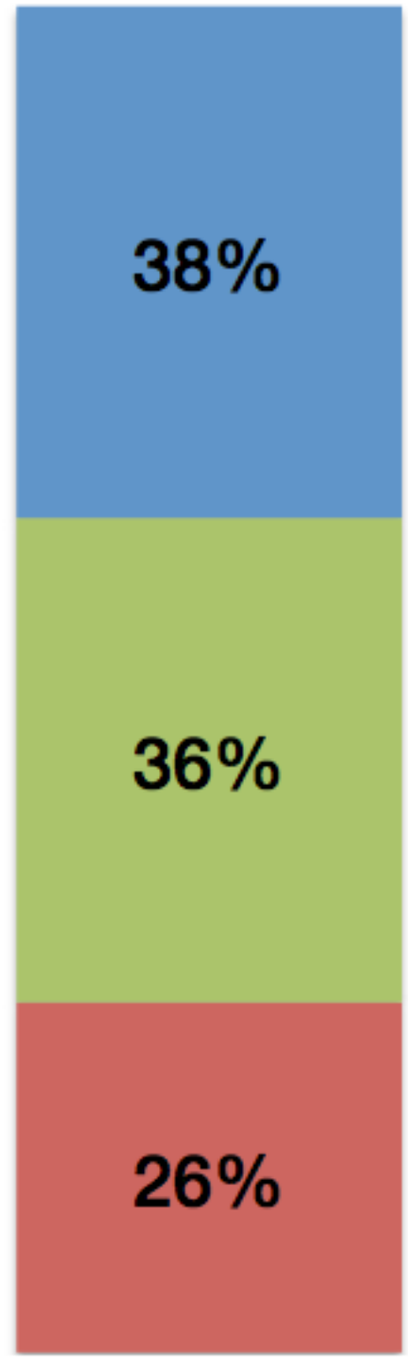
Outline

- Spark SQL overview
- Using Spark SQL and DataFrame
  - Linking with Spark SQL
  - Creating DataFrame
  - Querying
- JDBC/ODBC server
- User-defined functions
- Spark SQL vs. Trafodion
- Spark vs. HP DSM

Spark SQL overview

/Architecture

- Catalyst optimizer
  - Relational algebra + expressions
  - Query optimization
- Spark SQL Core
  - Execution of queries as RDDs
  - Reading in Parquet, JSON etc.
- Hive Support
  - HQL, MetaStore, SerDes, UDFs

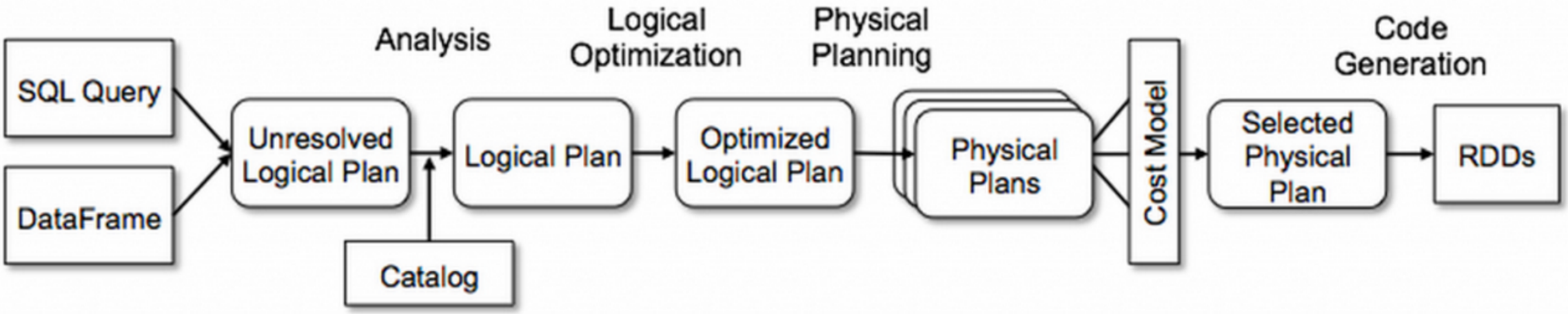


Spark SQL overview

/Catalyst

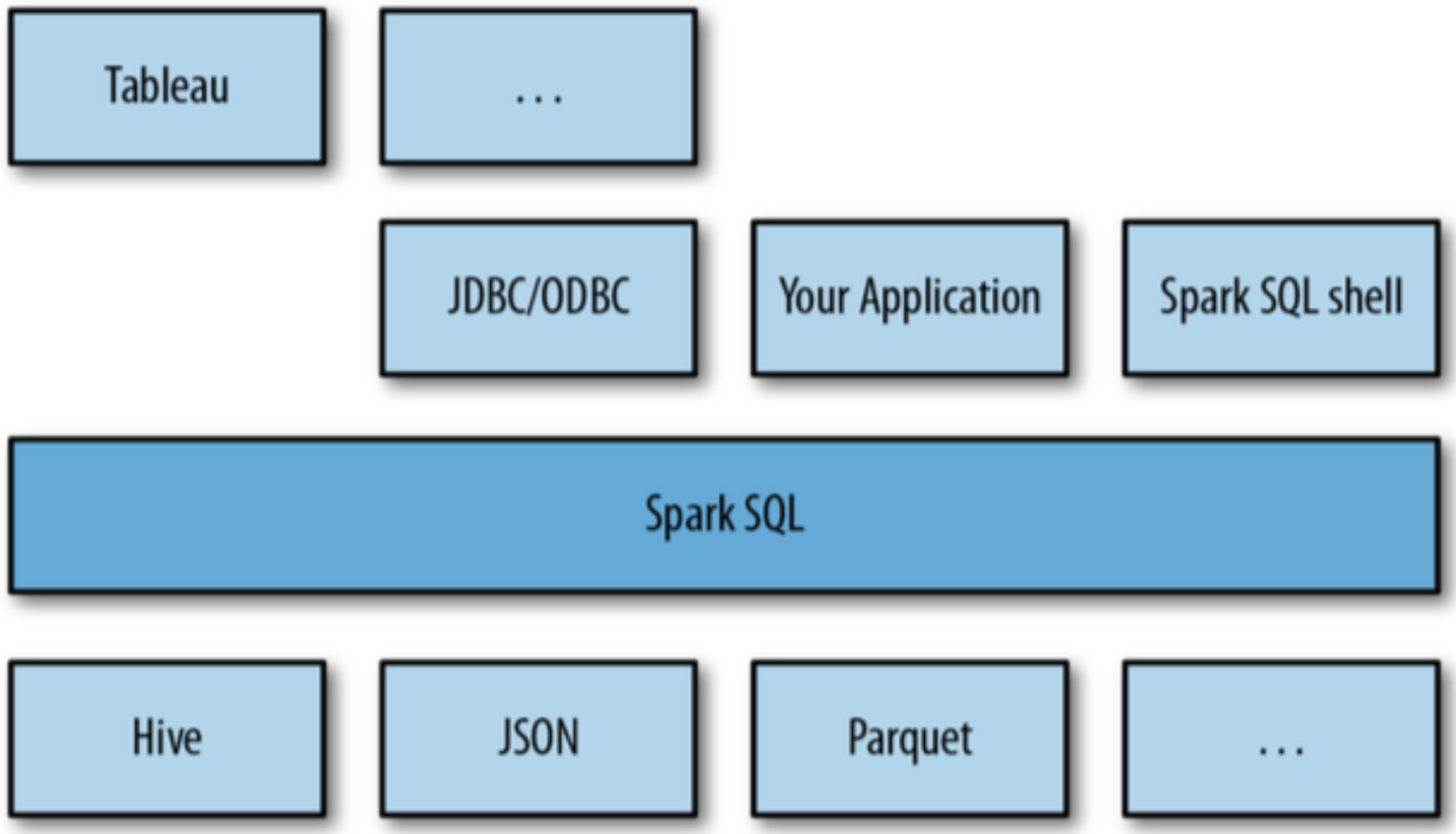
- Trees: The main data type in Catalyst is a tree composed of node objects. Each node has a node type and zero or more children

- Rules: Trees can be manipulated using rules, which are functions from a tree to another tree



Spark SQL overview

/Spark SQL usage



Using Spark SQL and DataFrame

/Linking with Spark SQL

- The entry point into all functionality of Spark SQL is the SQLcontext and in order to create a SQLcontext, all you need is a SparkContext

- In spark-shell, SQLcontext is already available as sqlContext. Otherwise, we can create SQLcontext as follows:

```
val sc: SparkContext
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

- In fact, you can also create a HiveContext to use HiveQL in Spark

```
val hiveContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

- Until now, hiveContext has more functionality than SQLcontext

Using Spark SQL and Dataframe

/Creating DataFrames

- To use sql query on RDDs, you need create a DataFrame which is a subclass of RDD (SchemaRDD, more precisely)
    - A DataFrame can be operated on as normal RDDs
    - It can also be registered as a temporary table
    - Registering a DataFrame as a table allows you to run SQL queries over its data
- Before Spark 1.3.0, we use SchemaRDD - partitioned collections of tuples

Using Spark SQL and Dataframe

/Creating DataFrames

- There are three ways to create a DataFrame

1. Using data source API

```
//Creating DataFrame via API
val df = sqlContext.jsonFile("/Users/tgbaggio/spark/spark-1.3.0-bin-hadoop2.4/examples/src/main/
resources/people.json")

val df = sqlContext.load("/Users/tgbaggio/spark/spark-1.3.0-bin-hadoop2.4/examples/src/main/reso
urces/people.json", "json")
```

2. Interoperating with RDDS

    - Inferring the schema using reflection

    - programmatically specifying the schema

- Spark DataFrame also support reading data directly from and to **Parquet** files

Using Spark SQL and DataFrame

/Creating DataFrame

- Inferring the schema using reflection: the Scala interface for Spark SQL supports automatically converting an RDD containing case classes to a DataFrame.

```
import sqlContext.implicits._
// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit,
// you can use custom classes that implement the Product interface.

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

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile(path + "people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.
toInt)).toDF()
```

Using Spark SQL and DataFrame

/Creating DataFrame

- A DataFrame can be created programmatically with three steps.
  1. Create an RDD of Rows from the original RDD;
  2. Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
  3. Apply the schema to the RDD of Rows via createDataFrame method provided by SQLContext.

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructType, StructField, StringType}
// Create an RDD of Rows
val people = sc.textFile(path + "people.txt")
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// Generate the schema based on the string of schema
val schemaString = "name,age"
val schema = StructType(
  schemaString.split(",").map(fieldName => StructField(fieldName, StringType, true))

val peopleDf = sqlContext.createDataFrame(rowRDD, schema)
```

Using Spark SQL and DataFrame

/Querying

- After creating the DataFrame, we can query the data either by DataFrame API or by SQL statements

```
// Querying with DataFrame API
df.show()

df.printSchema()

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

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

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

// Querying with SQL statements
peopleDf.registerTemple("people")
val results = sqlContext.sql("SELECT name FROM people")
// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
results.map(t => "Name: " + t(0)).collect().foreach(println)
```

Using Spark SQL and DataFrame

/Parquet and schema merging

- Parquet is a popular column-oriented storage format that can store records with nested fields efficiently

- Like ProtocolBuffer, Avro and Thrift, Parquet also supports schema evolution, which means that it can add column dynamically.

```
// Schema merging
val df1 = sc.makeRDD(1 to 5).map(i => (i, i * 2)).toDF("single", "double")
df1.saveAsParquetFile(path + "data/test_table/key=1")

val df2 = sc.makeRDD(6 to 10).map(i => (i, i * 3)).toDF("single", "triple")
df2.saveAsParquetFile(path + "data/test_table/key=2")

val df3 = sqlContext.parquetFile(path + "data/test_table")
df3.printSchema()
```

- The final schema consists of all 3 columns in the Parquet files together with the partitioning column appeared in the partition directory paths

Using Spark SQL and DataFrame

/Spark SQL and machine learning

- Neither SQLcontext nor HiveContext is complete. For example, it can handle nested query.

- It is not yet a replacement of Apache HIVE

- Right now, it is usually used in the pipeline of machine learning

```
training_data_table = sql("""
  SELECT e.action, u.age, u.latitude, u.logitude
    FROM Users  u
    JOIN Events e ON u.userId = e.userId""")

def featurize(u):
    LabeledPoint(u.action, [u.age, u.latitude, u.longitude])


// SQL results are RDDs so can be used directly in Mllib.
training_data = training_data_table.map(featurize)

model = new LogisticRegressionWithSGD.train(training_data)
```

JDBC/ODBC server

/JDBC to other databases

- Spark SQL can include a data source that can read data from other databases using JDBC

- To get started, we need to include the JDBC driver for our particular database on SPARK_CLASSPATH

- We can also use spark.executor.extraClassPath and --driver-class-path to include the JDBC driver

JDBC/ODBC server

/Using with beeline

- Spark SQL also provides JDBC connectivity, which is useful for connecting BI tools (business intelligence) to a Spark cluster

- Spark SQL's JDBC server corresponds to the HiveServer2 in Hive

- To start the JDBC server, run the following in the Spark directory

./sbin/start-thriftserver.sh

- The default port the server listens on is 10000. Now you can use beeline to test Thrift JDBC server

./bin/beeline

- Connect to the JDBC server in beeline with

beeline> !connect jdbc:hive2://localhost:10000

JDBC/ODBC server

/ODBC driver and BI tools

- The Spark ODBC driver is produced by Simba and can be download from various Spark vendors (e.g, Databricks)

- BI tools such as Microstrategy or Tableau can use this driver to connect Spark SQL

- In addition, most BI tools that have connectors to Hive can also connect to Spark SQL using their Hive connector

User-defined functions

/Spark SQL UDFs

- Spark SQL offers a built-in method to easily register UDFs by passing in a function in programming language

- In Scala and Python, we can use the native function and lambda syntax

- From Spark-1.3.0, in Scala we use udf object in sqlContext to register UDFs

sqlContext.udf.register(“functionName”, function)

- In Python, we use registerFunction method in sqlContext

sqlContext.registerFunction(“functionName”, function)

User-defined functions

/Hive UDFs

- Spark SQL can also use exsting Hive UDFs

- The standard Hive UDFs are already automatically included

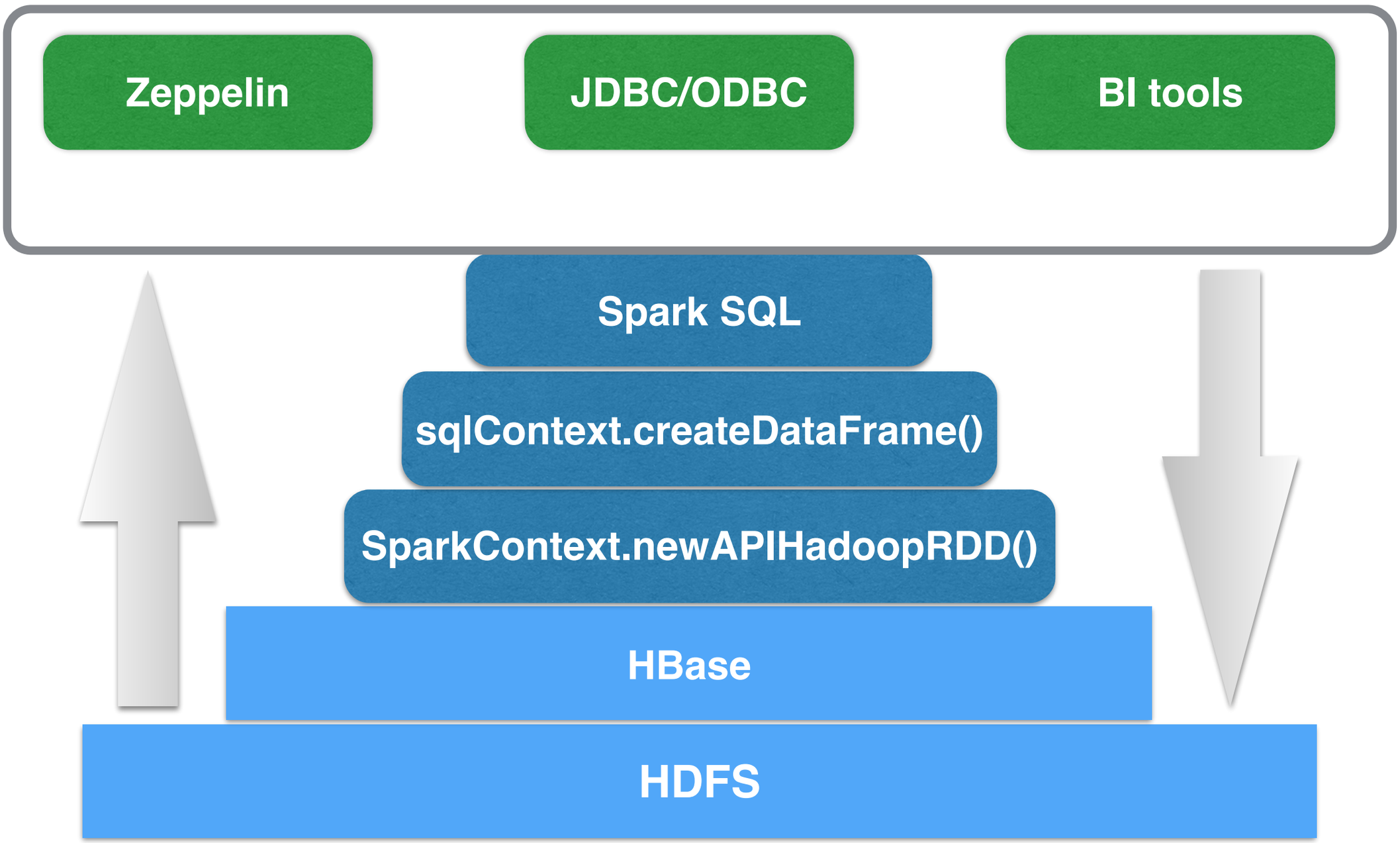
- To use customized UDF, we should include JARs with our application. If we run JDBC server, we can add this with --jars command-line flag

- Using a Hive UDF requires making it available

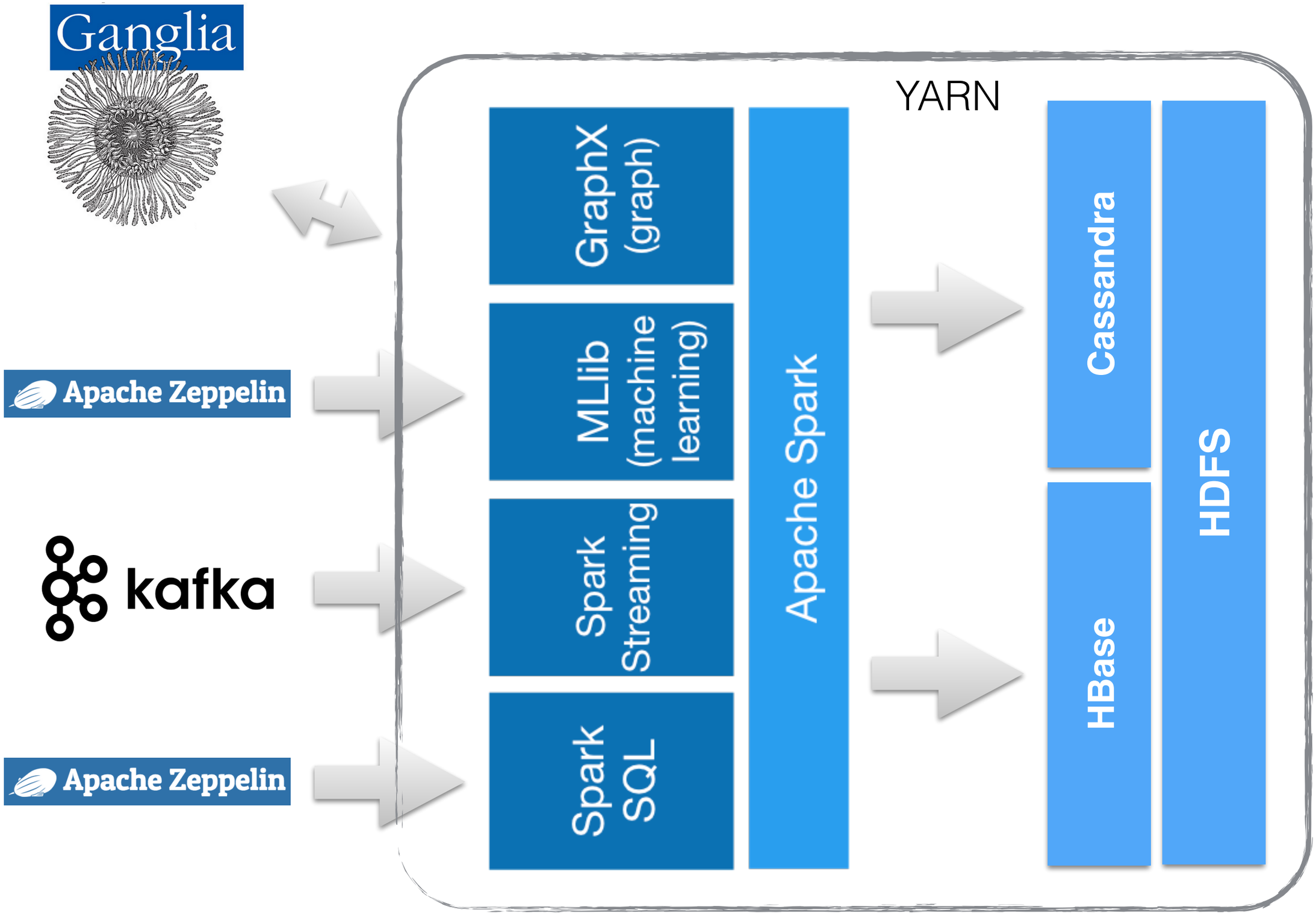
hiveCtx.sql(“CREATE TEMPORARY FUNCTION name AS class.function”)

Spark SQL vs. Trafodion

/SQL on Hbase with Spark SQL



Spark vs. HP DSM



Case studies

/Spark SQL at ebay

http://www.ebaytechblog.com/2014/05/28/using-spark-to-ignite-data-analytics/

- Several folks at eBay have begun using Spark with Shark (project merged into Spark SQL) to accelerate their Hadoop SQL performance.

- Many of these Shark queries are easily running 5X faster than their Hive counterparts.