Unified Data Access with Spark SQL

Michael Armbrust – Spark Summit 2014

Analytics the Old Way

Put the data into an RDBMS

- **Pros:** High level query language, optimized execution engine
- **Cons:** Have to ETL the data in, some analysis is hard to do in SQL, doesn't scale

Analytics the New Way

Map/Reduce, etc

- **Pros**: Full featured programming language, easy parallelism
- **Cons:** Difficult to do ad-hoc analysis, optimizations are left up to the developer.

SQL on HDFS

Put the data into a ~~RDBMS~~ HDFS

- **Pros:** High level query language, optimized execution engine
- **Cons:** ~~Have to ETL the data in~~, some analysis is hard to do in SQL, ~~doesn't scale~~

Spark SQL at Spark Summit 2013

SAN FRANCISCO | DECEMBER 2-3 2013

- 1 developer
- Able to run simple queries over data stored in Hive

Spark SQL at Spark Summit 2014

- 44 contributors
- Alpha release in Spark 1.0
- Support for Hive, Parquet, JSON
- Bindings in Scala, Java and Python
- More exciting features on the horizon!

Spark SQL Components

Share	Component
38%	Catalyst Optimizer
36%	Spark SQL Core
26%	Hive Support

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

Relationship to SHARK

Shark modified the Hive backend to run over Spark, but had two challenges:

» Limited integration with Spark programs
» Hive optimizer not designed for Spark

Spark SQL reuses the best parts of Shark:

Borrows

- Hive data loading
- In-memory column store

Adds

- RDD-aware optimizer
- Rich language interfaces

Migration from SHARK

Ending active development of Shark

Path forward for current users:

- Spark SQL to support CLI and JDBC/ODBC
- Preview release compatible with 1.0
- Full version to be included in 1.1

https://github.com/apache/spark/tree/branch-1.0-jdbc

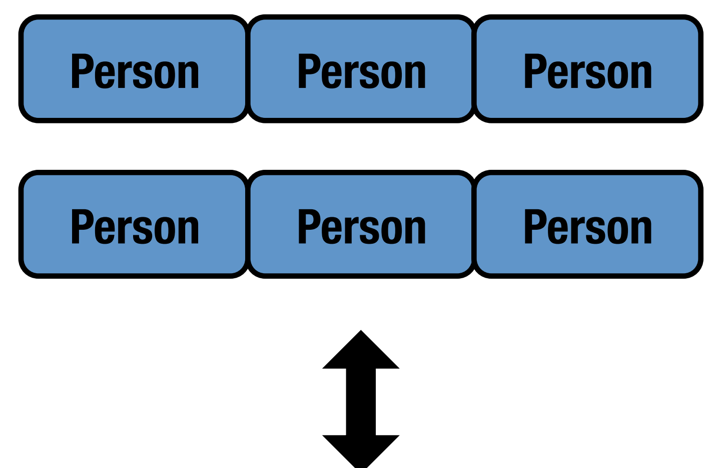
Adding Schema to RDDs

Spark + RDDs

Functional transformations on partitioned collections of **opaque** objects.

SQL + SchemaRDDs

Declarative transformations on partitioned collections of **tuples.**



Unified Data Abstraction

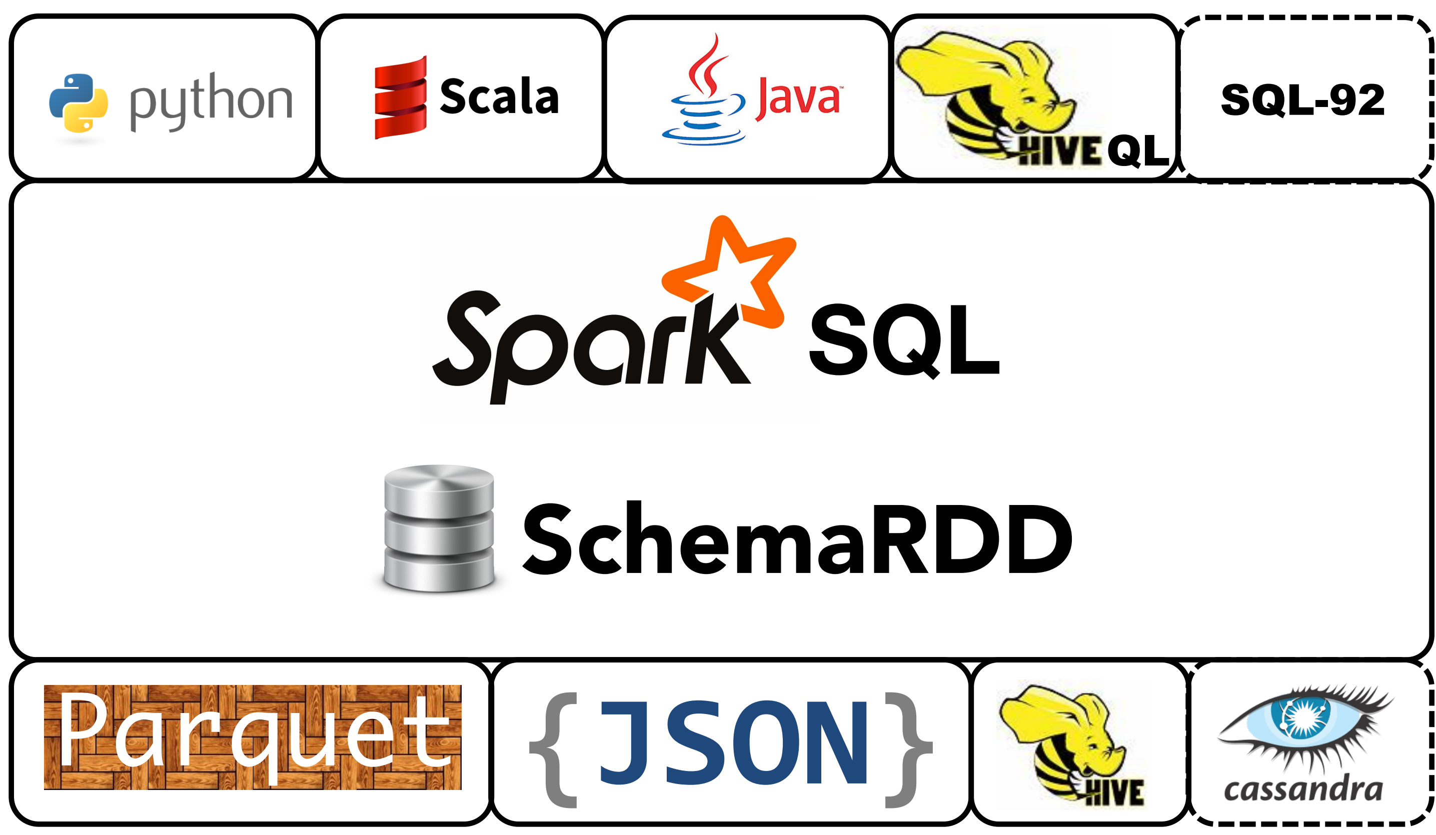


Image credit: http://barrymieny.deviantart.com/

RDDs into Relations (Python)

```
# Load a text file and convert each line to a dictionary.
lines = sc.textFile("examples/…/people.txt")

parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p:{"name": p[0],"age": int(p[1])})

# Infer the schema, and register the SchemaRDD as a table
peopleTable = sqlCtx.inferSchema(people)
peopleTable.registerAsTable("people")
```

RDDs into Relations (Scala)

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)

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

people.registerAsTable("people")
```

DATABRICKS

RDDs into Relations (Java)

```
public class Person implements Serializable {
  private String _name;
  private int _age;
  public String getName() { return _name;  }
  public void setName(String name) { _name = name; }
  public int getAge() { return _age; }
  public void setAge(int age) { _age = age; }
}

JavaSQLContext ctx = new org.apache.spark.sql.api.java.JavaSQLContext(sc)
JavaRDD<Person> people = ctx.textFile("examples/src/main/resources/
people.txt").map(
  new Function<String, Person>() {
    public Person call(String line) throws Exception {
      String[] parts = line.split(",");
      Person person = new Person();
      person.setName(parts[0]);
      person.setAge(Integer.parseInt(parts[1].trim()));
      return person;
    }
  });

JavaSchemaRDD schemaPeople = sqlCtx.applySchema(people, Person.class);
```

Language Integrated UDFs

```
registerFunction("countMatches",
  lambda (pattern, text):
    re.subn(pattern, '', text)[1])

sql("SELECT countMatches('a', text)…")
```

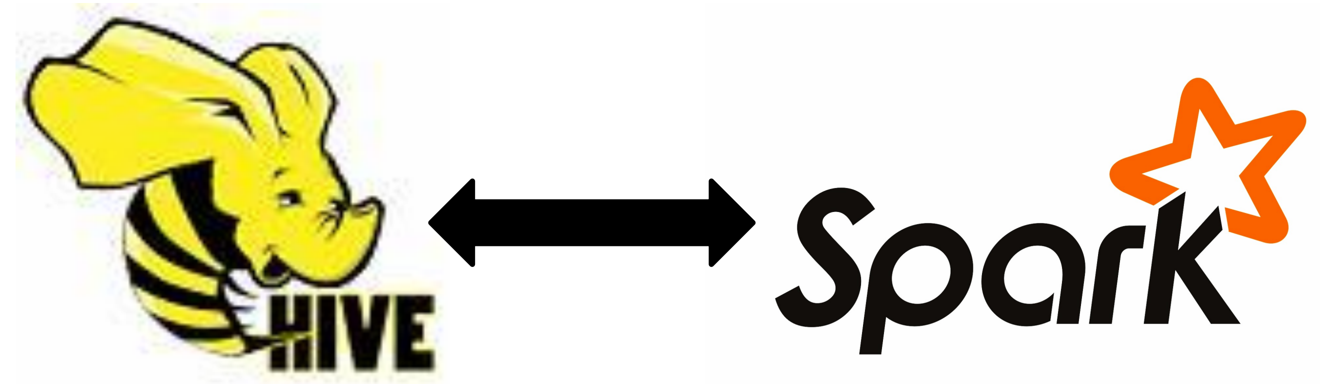
SQL and Machine Learning

```
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)
```

Hive Compatibility

Interfaces to access data and code in the Hive ecosystem:

- Support for writing queries in HQL
- Catalog info from Hive MetaStore
- Tablescan operator that uses Hive SerDes
- Wrappers for Hive UDFs, UDAFs, UDTFs



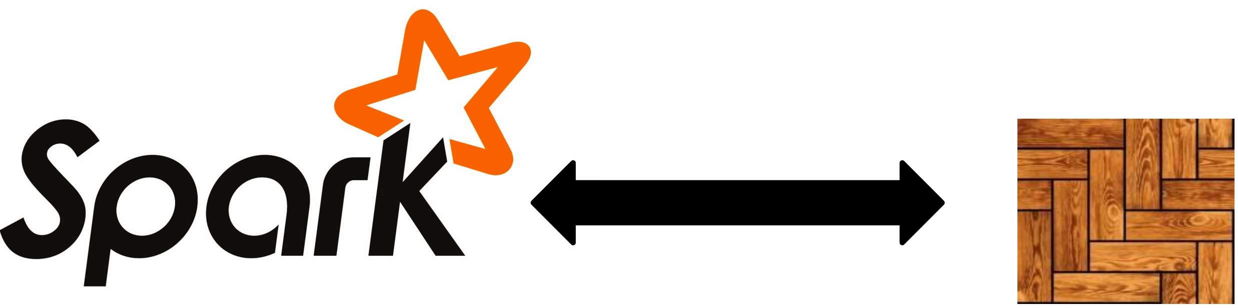
Reading Data Stored in Hive

```
from pyspark.sql import HiveContext
hiveCtx = HiveContext(sc)

hiveCtx.hql("""
  CREATE TABLE IF NOT EXISTS src (key INT, value STRING)""")

hiveCtx.hql("""
  LOAD DATA LOCAL INPATH 'examples/…/kv1.txt' INTO TABLE src""")

# Queries can be expressed in HiveQL.
results = hiveCtx.hql("FROM src SELECT key, value").collect()
```

Parquet Compatibility

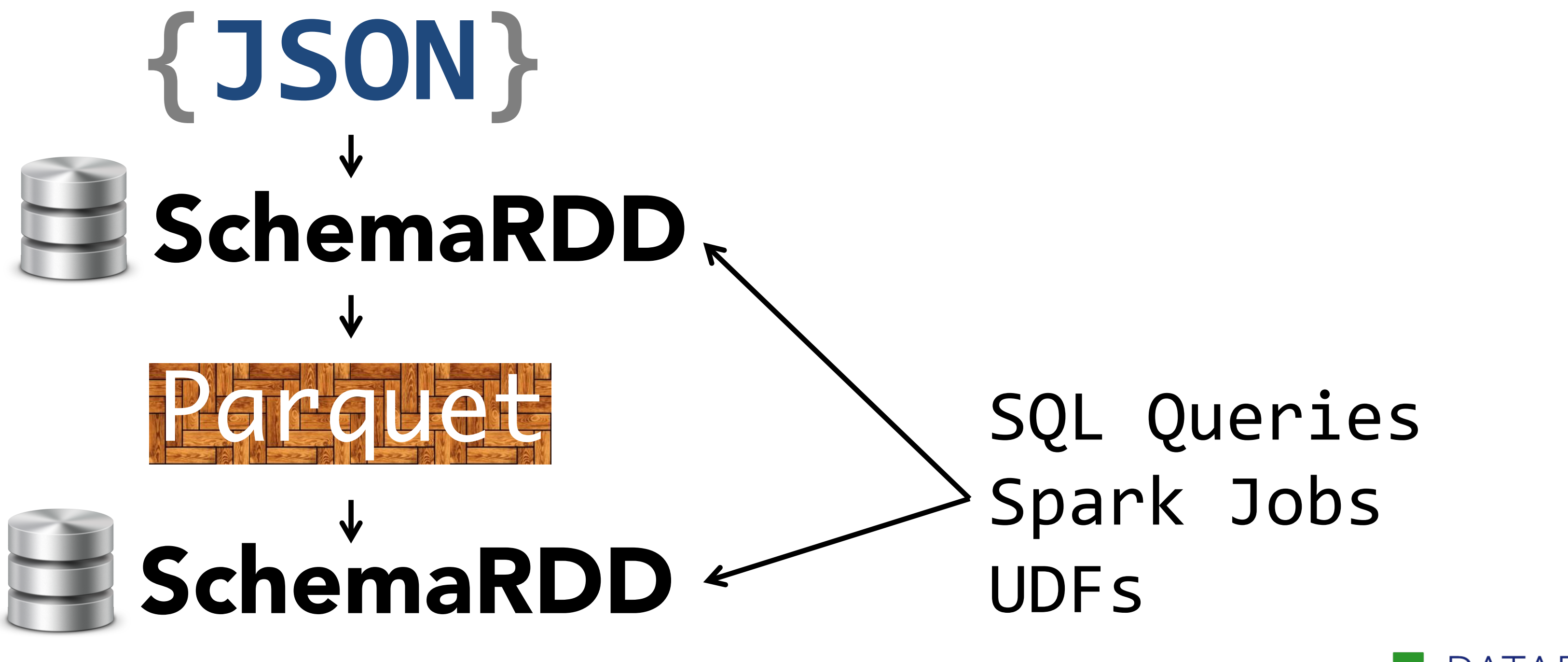
Native support for reading data in Parquet:

- Columnar storage avoids reading unneeded data.
- RDDs can be written to parquet files, preserving the schema.



DEMO

Use SchemaRDD as a bridge between data formats to make analysis much faster.



Spark SQL Performance

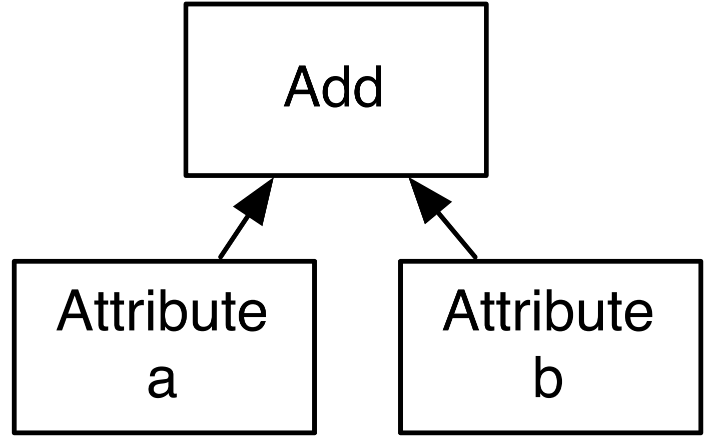
Efficient Expression Evaluation

Interpreting expressions (e.g., ‘a + b’) can very expensive on the JVM:

- Virtual function calls
- Branches based on expression type
- Object creation due to primitive boxing
- Memory consumption by boxed primitive objects

Interpreting “a+b”

1. Virtual call to Add.eval()
2. Virtual call to a.eval()
3. Return boxed Int
4. Virtual call to b.eval()
5. Return boxed Int
6. Integer addition
7. Return boxed result



Using Runtime Reflection

```
def generateCode(e: Expression): Tree = e match {
  case Attribute(ordinal) =>
    q"inputRow.getInt($ordinal)"
  case Add(left, right) =>
    q"""
      {
        val leftResult = ${generateCode(left)}
        val rightResult = ${generateCode(right)}
        leftResult + rightResult
      }
     """
}
```

Executing “a + b”

```
val left: Int = inputRow.getInt(0)
val right: Int = inputRow.getInt(1)
val result: Int = left + right
resultRow.setInt(0, result)
```

- Fewer function calls
- No boxing of primitives

Code Generation Made Simple

- Code generation is a well known trick for speeding up databases.
- **Scala Reflection + Quasiquotes** made our implementation an experiment done over a few weekends instead of a major system overhaul.

Initial Version ~1000 LOC

Performance Microbenchmark

Evaluating 'a+a+a' One Billion Times

Milliseconds

40
35
30
25
20
15
10
5
0

Intepreted Evaluation
Hand-written Code
Generated with Scala Reflection

Features Slated for 1.1

- Code generation
- Language integrated UDFs
- Auto-selection of Broadcast Join
- JSON and nested parquet support
- Many other performance / stability improvements

1.1 Preview: TPC-DS Results

Seconds

400
350
300
250
200
150
100
50
0

Query 19 | Query 53 | Query 34 | Query 59

Shark - 0.9.2 | SparkSQL + codegen

Questions?