Adding Native SQL Support to Spark with Catalyst

Michael Armbrust

DATABRICKS

Overview

- Catalyst is an optimizer framework for manipulating trees of relational operators.
- Catalyst enables native support for executing relational queries (SQL) in Spark.

Relationship to SHARK

SparkSQL is a nearly from scratch rewrite that leverages the best parts of Shark

Borrows

- Hive data loading code / in-memory columnar representation
- Hardened spark execution engine

Adds

- RDD-aware optimizer / query planner
- execution engine
- language interfaces.

Hive Compatibility

- Interfaces to access data and code in the Hive ecosystem:
  - Support for writing queries in HQL
  - Catalog for that interfaces with the Hive MetaStore
  - Tablescan operator that uses Hive SerDes
  - Wrappers for Hive UDFs, UDAFs, UDTFs

Implemented as an optional module.

Parquet Support

Native support for reading data stored in Parquet:

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

Currently only supports flat structures (nested data on short-term roadmap).

Using Spark SQL

SQLContext

- Entry point for all SQL functionality
- Wraps/extends existing spark context

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

// Importing the SQL context gives access to all the SQL functions and conversions.
import sqlContext._
```

SchemaRDDs

Resilient Distributed Datasets (RDDs) are Sparks core abstraction.

- Distributed coarse-grained transformations
- Opaque to Spark

RDDs + Schema

- Aware of the names and types of columnar data stored in RDDs

Turning an RDD into a Relation

```
// 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")
```

Querying Using SQL

```scala
// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

```scala
// The results of SQL queries are SchemaRDDs and support normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
val nameList = teenagers.map(t => "Name: " + t(0)).collect()
```

Querying Using the Scala DSL

Spark SQL also allows you to express queries using functions, instead of SQL strings.

```
// The following is the same as:
// 'SELECT name FROM people WHERE age >= 10 AND age <= 19'
val teenagers =
  people.where('age >= 10).where('age <= 19).select('name)
```

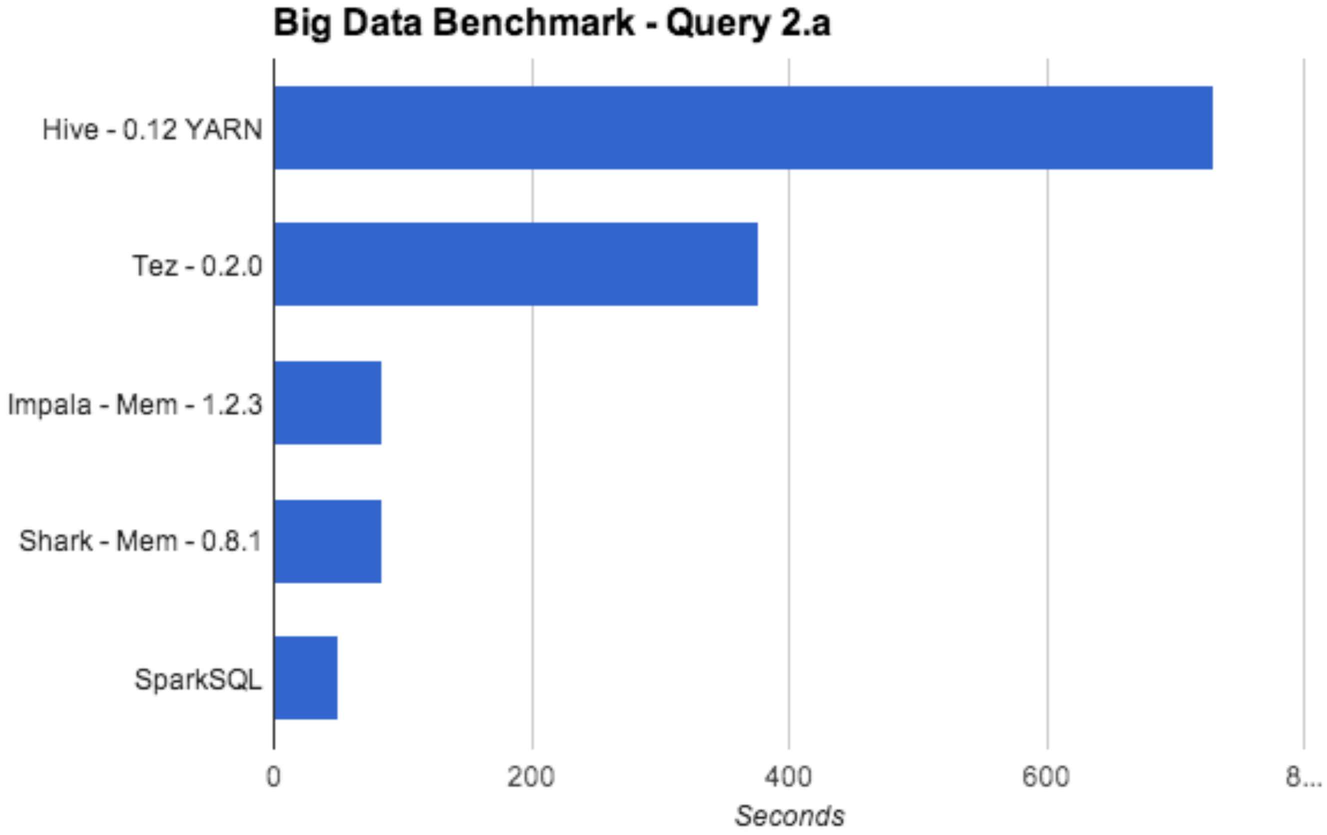
Caching Tables In-Memory

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

- Scan only required columns
- Fewer allocated objects (less GC)
- Automatically selects best compression

```
cacheTable("people")
```

In-Memory Performance



Using Parquet

```
// Any SchemaRDD can be stored as Parquet.
people.saveAsParquetFile("people.parquet")

// Parquet files are self-describing so the schema is preserved.
val parquetFile = sqlContext.parquetFile("people.parquet")

//Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerAsTable("parquetFile")
val teenagers = sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
```

Reading Data Stored In Hive

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

hql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH '.../kv1.txt' INTO TABLE src")

// Queries can be expressed in HiveQL.
hql("FROM src SELECT key, value")
```

Mixing SQL and Machine Learning

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

// Since `sql` returns an RDD, the results of can be easily used in MLlib
val trainingData = trainingDataTable.map { row =>
  val features = Array[Double](row(1), row(2), row(3))
  LabeledPoint(row(0), features)
}
val model = new LogisticRegressionWithSGD().run(trainingData)
```

Mix Data From Multiple Sources

```
val hiveContext = new HiveContext(sc)
import hiveContext._

// Data stored in Hive
hql("CREATE TABLE IF NOT EXISTS hiveTable (key INT, value STRING)")
hql("LOAD DATA LOCAL INPATH 'kv1.txt' INTO TABLE hiveTable")

// Data in existing RDDs
val rdd = sc.parallelize((1 to 100).map(i => Record(i, s"val_$i")))
rdd.registerAsTable("rddTable")

// Data stored in Parquet
hiveContext.loadParquetFile("parquet.file").registerAsTable("parquetTable")

// Query all sources at once!
sql("SELECT * FROM hiveTable JOIN rddTable JOIN parquetTable WHERE ...")
```

Supports Java Too!

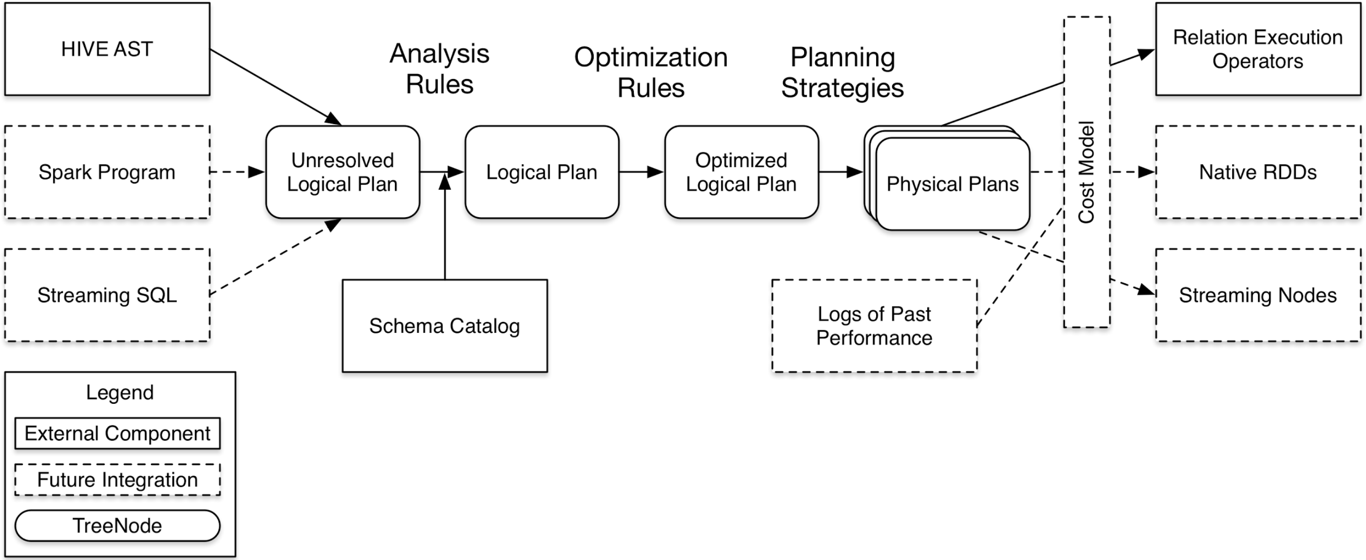
```
public class Person implements Serializable {
  private String _name;
  private int _age;
  String getName() { return _name;  }
  void setName(String name) { _name = name; }
  int getAge() { return _age; }
  void setAge(int age) { _age = age; }
}

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);
```

C🔥talyst Internals

Architecture

Phases of **rules** analyze, optimize and plan relational queries.



Tree Transformations

- Many concepts can be represented as trees:
  - Logical Plans, Expressions, Physical Operators
- Phases of transformations prepare trees for execution.
- Explicitly decoupling of phases:
  - Easier to expand.
  - Easier to add cost based optimization.

Example: Optimization with Rules

Rules are concise, modular specification of tree transformations.

```
object ConstantFolding extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transformAllExpression {
    case l: Literal => l     // Skip redundant folding of literals.
    case e if e.foldable => Literal(e.apply(EmptyRow), e.dataType)
  }
}

object CombineFilters extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case Filter(c1, Filter(c2, grandChild)) => Filter(And(c1,c2), grandChild)
  }
}
```

Rules can be run once or to fixed point.

Rules are used for:

- Analysis
- Providing Hive-specific semantics
- Optimization
- Query Planning

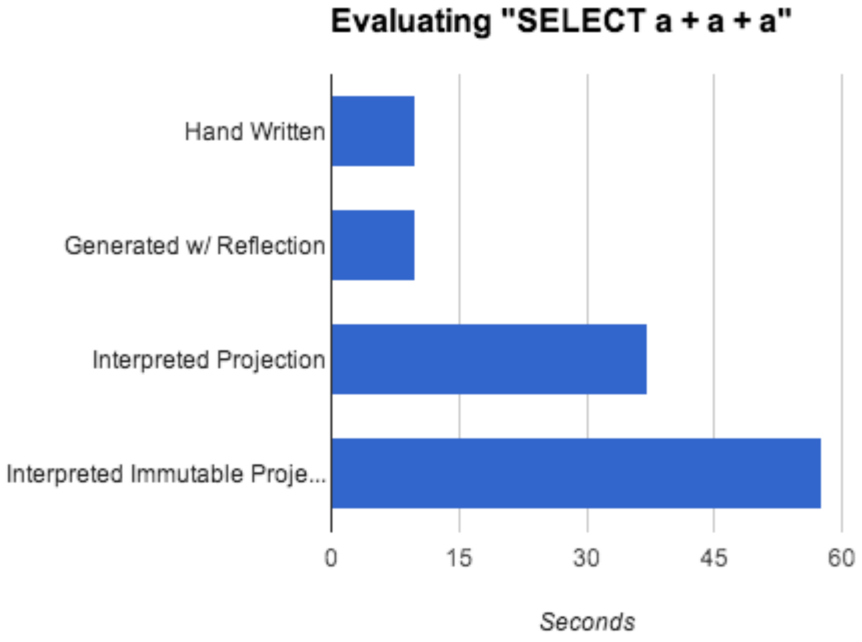
Example Collapse Filters

Collapse two adjacent filters into a single one with an AND

```
object CombineFilters extends Rule[LogicalPlan] {
  def apply(plan: LogicalPlan): LogicalPlan = plan transform {
    case Filter(c1, Filter(c2, grandChild)) =>
      Filter(And(c1,c2), grandChild)
  }
}
```

Performance: Code generation

- Generic evaluation of expression logic is 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
- Generating custom bytecode can eliminate these overheads



Easily extensible code-generation

Scala reflection (new in 2.10) makes it easy to extend code generation capabilities:

- Pattern matching expressions
- Quasi-quotes turn strings into splice-able Scala ASTs
- Scala does the hard work of generating efficient bytecode.
- Support for generating expression evaluation logic and custom data storage structures.

```
case Cast(e, StringType) =>
  val eval = expressionEvaluator(e)
  eval.code ++
  q"""
   val $nullTerm = ${eval.nullTerm}
   val $primitiveTerm =
     if($nullTerm)
       ${defaultPrimitive(StringType)}
     else
       ${eval.primitiveTerm}.toString
 """.children
```

Code Generation Example

```
case Cast(e, StringType) =>
  val eval = expressionEvaluator(e)
  eval.code ++
  q"""
   val $nullTerm = ${eval.nullTerm}
   val $primitiveTerm =
     if($nullTerm)
       ${defaultPrimitive(StringType)}
     else
       ${eval.primitiveTerm}.toString
 """.children
```

Road Map

Available now in Spark Master.

April - Alpha release with Spark 1.0

May - Preview Release of Shark on SparkSQL

Near term focus:

Stability, compatibility, performance, integration with spark ecosystem.

Questions?

Other resources:

- **Check it out:** https://github.com/apache/spark
- **Programming Guide:** http://tinyurl.com/sparkSql

\+