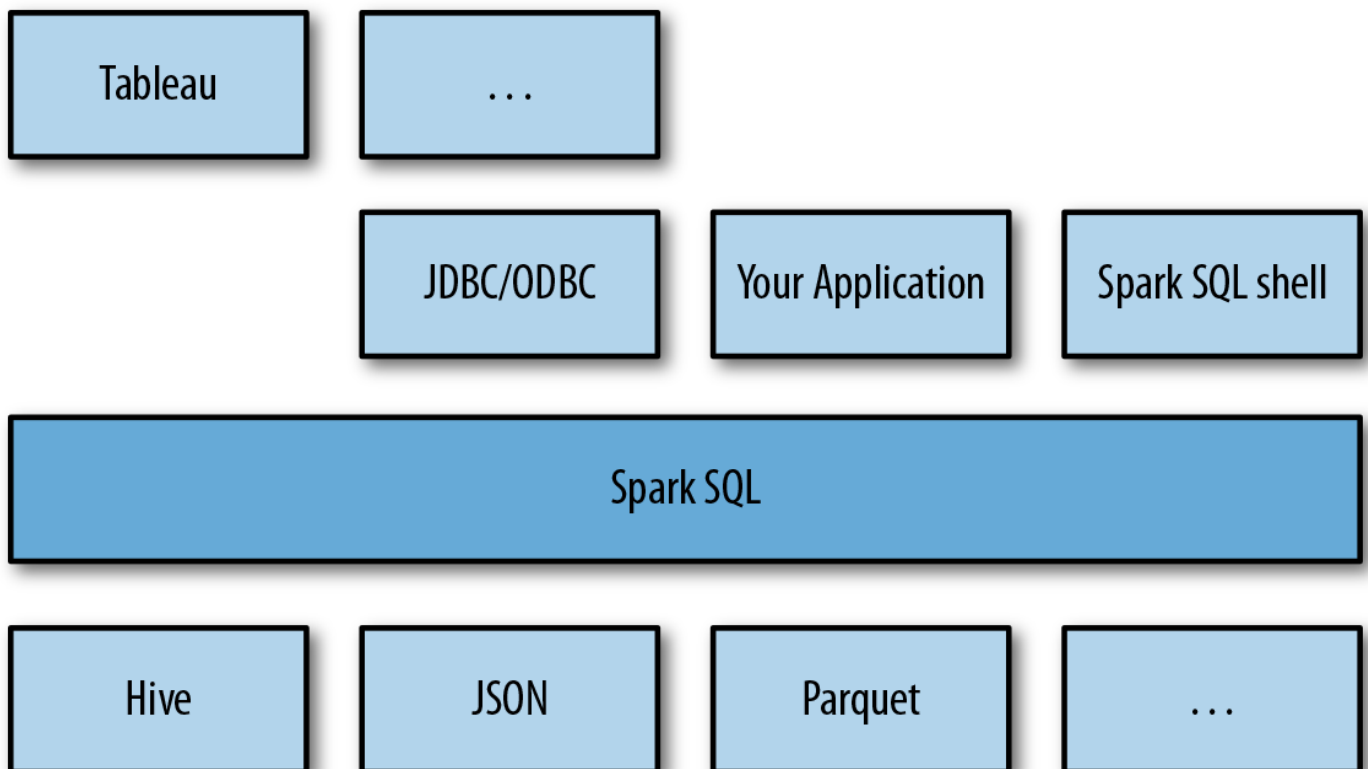


구조적 데이터(Structured Data)와 반구조적(SemiStructured Data) 를 다룰 수 있는 Spark SQL 과 Spark Interface 를 소개한다.

구조적 데이터란? Schema 를 갖고 있는 데이터를 의미한다. 만약 구조적 데이터를 다룰 때, Spark SQL 을 사용하면 쉽고, 효율적으로 다룰 수 있다.

- 다양한 데이터 유형 처리 가능
- SQL 을 사용하여 쿼리 가능
- RDD 와 SQL Table 을 Join 하는 기능을 포함하여 기존 코드(spark-core)와 통합이 가능

이런 기능들을 제공하기 위해 Spark SQL 은 SchemaRDD 를 사용한다. 이는 Row 객체의 RDD 이며, 각 아이템은 Record 를 의미한다. SchemaRDD 는 기존 RDD 와 유사해 보이지만 내부적으로는 좀 더 효율적인 방법으로 저장되고, Schema 의 이점 또한 활용한다. 그리고 SQL 수행 같은 기존 RDD 에서는 제공되지 않는 Operation 도 제공한다. 이런 SchemaRDD 는 외부 데이터 소스나 조회 결과 또는 일반적인 RDD 로부터 생성이 가능하다.



SchemaRDD 를 이용하여 Spark SQL 이 구조적인 데이터를 어떻게 읽고 조회하는지 알아보자.

Linking with Spark SQL

Spark SQL 은 두 가지 타입이 제공된다.

- Hive 포함
Hive Table, UDF(User-Defined Function), SerDes(Serialization&Deserialization), HiveQL 사용 가능
Apache Hive 가 꼭 설치되어 있을 필요는 없음.
- Hive 미포함

Spark Download 페이지에서 빌드된 바이너리를 다운로드 받는 경우, 이미 Hive 를 포함하고 있는 버전이다. 만약 source code 를 받아서 수동으로 빌드를 하게 된다면, sbt 빌드시 -Phive 옵션을 줘야 한다.

```
sbt/sbt -Phive assembly
```

Scala 나 Java 에서 사용하려면 Maven Lib 이 필요하다. 아래와 같이 dependency 설정을 하면 된다.

```
<dependency>
<groupId>org.apache.spark</groupId>
<artifactId>spark-hive_2.10</artifactId>
<version>1.3.0</version>
</dependency>
```

만약 Hive 를 추가할 수 없다면, spark-sql_2.10 을 추가하면 된다.

Spark SQL 을 사용할 때는, Hive 라이브러리를 사용할 수 있는지 없는지에 따라 두 가지 사용법이 있다.

- HiveContext - Hive 설치 필요없음
- SQLContext - Hive 라이브러리가 필요없음

Spark SQL 과 함께 HiveQL(이하 HQL) 을 사용하면 좋다.

만약 이미 설치된 Hive 가 있다면, Hive 설정파일인 hive-site.xml 을 Spark 설정 디렉토리 (\$SPARK_HOME/conf)로 복사해야 한다. Hive 가 설치되지 않았더라도 Spark 는 그대로 수행된다.

Using Spark SQL in Applications

Spark SQL 을 사용하려면 HiveContext 를 생성해야 한다. HiveContext 는 Spark SQL 데이터를 조회 또는 연산되기 위해서 몇 가지 편리한 함수들을 제공하고 있다.

HiveContext 를 사용하면 일반적인 RDD 에서 map() 함수처럼, 구조적 데이터인 SchemaRDD 를 생성하고 SQL Operation 을 수행할 수 있다.

Initializing Spark SQL

Spark SQL 을 시작하려면 몇 가지 Import 가 필요하다.

```
// Import Spark SQL
import org.apache.spark.sql.hive.HiveContext ;
// Or if you can't have the hive dependencies
import org.apache.spark.sql.SQLContext ;
// Import the JavaSchemaRDD
import org.apache.spark.sql.SchemaRDD ;
import org.apache.spark.sql.Row ;
```

Import 가 되었다면, HiveContext 를 생성해 보자.

```
JavaSparkContext ctx = new JavaSparkContext(...);
SQLContext sqlCtx = new HiveContext(ctx);
```

이렇게 하면 데이터를 읽거나 조회할 준비가 되었다.

Basic Query Example

HiveContext 나 SQLContext 에서 제공되는 sql() 함수를 통해서 쿼리를 수행해 볼 수 있다.

아래 예제는 JSON 파일을 읽어서 tweets 라는 임시 테이블에 저장한 뒤 쿼리를 수행하고 있다.

```
SchemaRDD input = hiveCtx.jsonFile(inputFile);
// Register the input schema RDD
input.registerTempTable("tweets");
// Select tweets based on the retweetCount
SchemaRDD topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
    tweets ORDER BY retweetCount LIMIT 10");
```

SchemaRDDs

데이터를 읽거나 쿼리를 수행한 뒤 결과는 SchemaRDD 로 반환되는데, SchemaRDD 는 기존 RDBMS 에서의 Table 과 유사하다.

SchemaRDD 는 컬럼 타입정보를 포함하여, Row 형태의 RDD 라고 할 수 있다. Row 객체는 Primitive Type 의 객체의 Wrapper Class 이다.

(Spark 상위버전에서는 SchemaRDD 가 DataFrame 으로 변경될 수 있다.)

SchemaRDD 는 일반적인 RDD 타입으로 기존 RDD 에 제공되는 map() 이나 filter() 같은 Operation 은 모두 사용할 수 있다. SchemaRDD 에서 가장 중요한 특징은 HiveContext.sql 이나 SparkContext.sql 을 통해서 임시 테이블로 등록이 가능하다는 점이다.(registerTempTable() 도 활용할 수 있다.)

SchemaRDD 에서 사용되는 자료유형은 아래와 같다.

Spark SQL/HiveQL type	Scala type	Java type	Python
TINYINT	Byte	Byte/byte	int/long (in range of -128 to 127)
SMALLINT	Short	Short/short	int/long (in range of -32768 to 32767)
INT	Int	Int/int	int or long

Spark SQL/HiveQL type	Scala type	Java type	Python
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	Java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.Timestamp	java.sql.Timestamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list, tuple, or array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1: COL1_TYPE, ...>	Row	Row	Row

마지막에 언급된 Structure 는 Spark SQL 의 다른 Row 가 될 수 있다. 모든 타입은 중첩해서 사용할 수 있는데, 예를 들면 Structure 배열이나, Structure 로 구성된 Map 을 들 수 있다.

Working with Row objects

SchemaRDD 내에 있는 Row 객체는 간단히 말하면 고정된 길이의 Field Arrays 라고 할 수 있다. Row 객체에는 각 field 를 접근하기 위해 getter 함수를 제공한다.

- `get()`
Object 타입으로 반환된다.
- `getType()`
각 자료형으로 반환된다.
ex) `getString(0)`

```
JavaRDD<String> topTweetText = topTweets.toJavaRDD().map(new Function<Row, String>
() {
    public String call(Row row) {
        return row.getString(0);
    }
});
```

Python에서는 타입변환이 필요하지 않다. 그냥 `row[i]` 나 `row.column_name` 방식으로 접근할 수 있다.

Caching

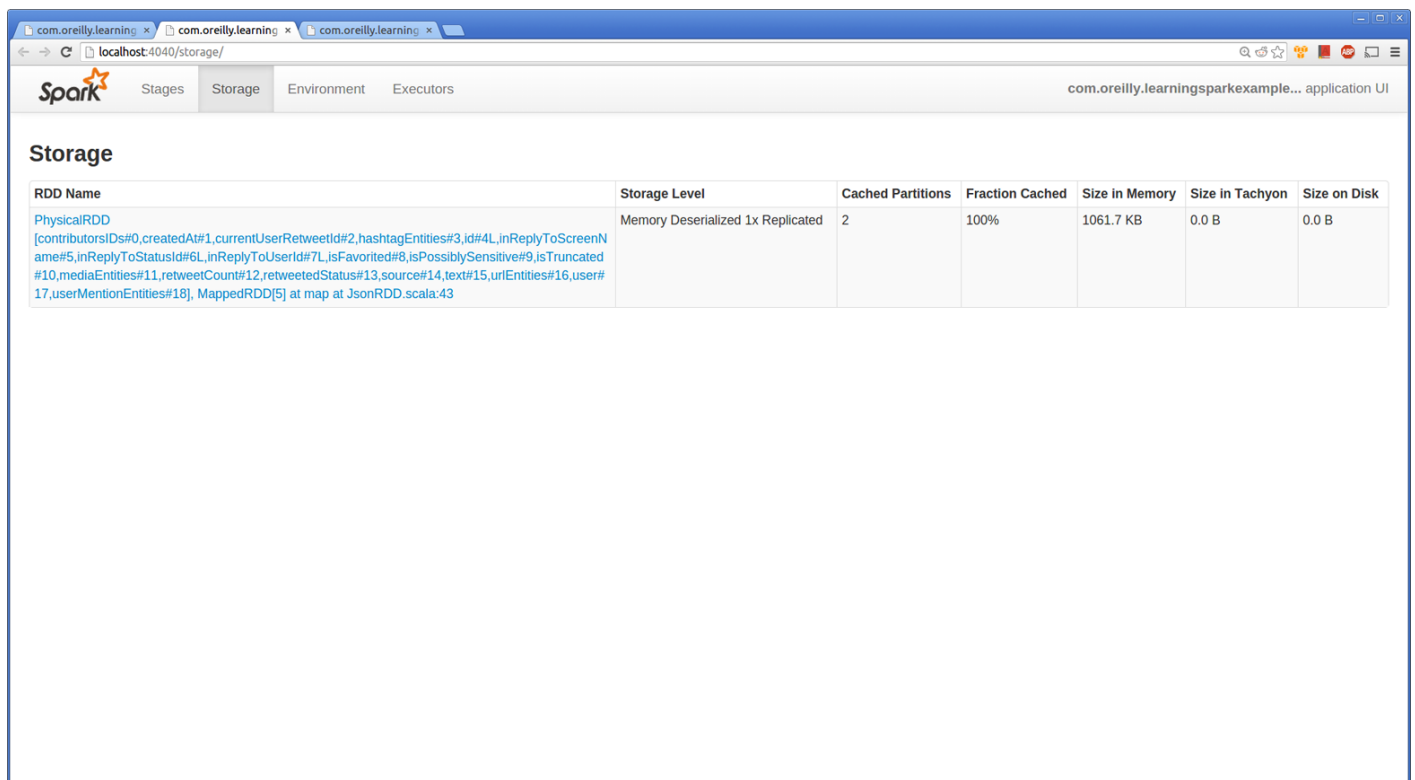
Spark SQL에서의 Caching은 조금 다르게 동작한다. 이미 각 컬럼의 데이터 유형을 알기 때문에, 좀 더 효율적으로 데이터를 저장할 수 있다.

`hiveCtx.cacheTable("tableName")` 메소드를 통해서 caching을 할 수 있다. Caching된 테이블은 Drive Program이 수행되는 동안에만 유지가 된다. 그래서 종료된 뒤에는 다시 사용하려면 다시 Caching을 해야한다. 따라서 동일한 데이터를 가지고 다수의 작업을 수행하거나 쿼리를 수행할 때 Caching을 사용한다.

또한 HiveQL/SQL을 사용할 수 있는데, Cache 또는 UnCache를 할 때는 아래와 같이 할 수 있다.

- `CACHE TABLE tableName`
- `UNCACHE TABLE tableName`

Cache된 SchemaRDD는 기존 RDD와 동일하게 Spark Application UI에서 확인할 수 있다.



RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
PhysicalRDD [contributorsIDs#0, createdAt#1, currentUserRetweetId#2, hashtagEntities#3, id#4L, inReplyToScreenName#5, inReplyToStatusId#6L, inReplyToUserId#7L, isFavorited#8, isPossiblySensitive#9, isTruncated#10, mediaEntities#11, retweetCount#12, retweetedStatus#13, source#14, text#15, uriEntities#16, user#17, userMentionEntities#18], MappedRDD[5] at map at JsonRDD.scala:43	Memory Deserialized 1x Replicated	2	100%	1061.7 KB	0.0 B	0.0 B

Loading and Saving Data

Spark SQL은 Hive Table, JSON, Parquet file 등을 지원한다. 쿼리를 수행하거나 일부 컬럼만 선택하고자 할 때, Spark SQL은 `SparkContext.hadoopFile` 처럼 모든 데이터를 스캔하는 것이 아니라 필요한 일부 컬럼만 스캔한다.

일반 RDD에서 Schema만 설정해 줌으로서 SchemaRDD로 변환할 수 있는데, 이는 Java나 Python 데이터이더라도 쉽게 SQL을 수행할 수 있게 한다. 그리고 많은 양의 데이터를 한 번에 계산할 수도 있고, 서로 다른 저장소에서 읽은 SchemaRDD를 Join할 수도 있다.

Apache Hive

SerDes, Text file, RCFile, ORC, Parquet, Avro 그리고 Protocol Buffer 같은 Hive 가 지원하는 데이터 유형은 모두 Spark가 지원한다.

이미 설치된 Hive 에 연결하기 위해서는 hive-site.xml 파일을 \$SPARK_HOME/conf/ 하위에 복사해 주어야 한다. 만약 단지 조회만 한다면 hive-site.xml 을 설정하지 않고 local Hive metastore 를 사용하게 된다.

```
import org.apache.spark.sql.hive.HiveContext ;
import org.apache.spark.sql.Row ;
import org.apache.spark.sql.SchemaRDD ;

HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT key, value FROM mytable");
JavaRDD<Integer> keys = rdd.toJavaRDD().map(new Function<Row, Integer>() {
    public Integer call(Row row) { return row.getInt(0); }
});
```

Parquet

Parquet 는 컬럼 기반의 데이터 저장 포맷이다. 이는 Spark SQL 의 모든 데이터 유형을 지원한다. Spark SQL 은 Parquet File 을 바로 읽을 수 있는 Method 를 지원한다.

```
# Load some data in from a Parquet file with field's name and favouriteAnimal
rows = hiveCtx.parquetFile(parquetFile)
names = rows.map(lambda row: row.name)
print "Everyone"
print names.collect()
```

Parquet file 을 Spark SQL 의 임시 테이블로도 등록할 수 있다.

```
# Find the panda lovers
tbl = rows.registerTempTable("people")
pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE favouriteAnimal = W'panda'")
print "Panda friends"
print pandaFriends.map(lambda row: row.name).collect()
```

저장도 가능하다.

```
pandaFriends.saveAsParquetFile("hdfs://...")
```

JSON

동일한 포맷을 갖는 JSON 파일이 있다면, Spark 에서 사용가능하다.

JSON 파일 내용을 읽으려면 `jsonFile()` 함수를 사용하면 된다. 읽은 JSON 파일의 `schema` 를 확인하려면 `SchemaRDD.printSchema()` 를 사용하면 확인 할 수 있다.

```
{ "name" : "Holden" }
{ "name" : "Sparky The Bear", "lovesPandas" : true, "knows" :
{ "friends" : [ "holden" ] } }
```

```
SchemaRDD input = hiveCtx.jsonFile(jsonFile);
```

Tweet 메시지의 Schema 를 확인해 보자.

```
root
|-- contributorsIDs: array (nullable = true)
|   |-- element: string (containsNull = false)
|-- createdAt: string (nullable = true)
|-- currentUserRetweetId: integer (nullable = true)
|-- hashtagEntities: array (nullable = true)
|   |-- element: struct (containsNull = false)
|   |   |-- end: integer (nullable = true)
|   |   |-- start: integer (nullable = true)
|   |   |-- text: string (nullable = true)
|-- id: long (nullable = true)
|-- inReplyToScreenName: string (nullable = true)
|-- inReplyToStatusId: long (nullable = true)
|-- inReplyToUserId: long (nullable = true)
|-- isFavorited: boolean (nullable = true)
|-- isPossiblySensitive: boolean (nullable = true)
|-- isTruncated: boolean (nullable = true)
|-- mediaEntities: array (nullable = true)
|   |-- element: struct (containsNull = false)
|   |   |-- displayURL: string (nullable = true)
|   |   |-- end: integer (nullable = true)
|   |   |-- expandedURL: string (nullable = true)
|   |   |-- id: long (nullable = true)
|   |   |-- mediaURL: string (nullable = true)
|   |   |-- mediaURLHttps: string (nullable = true)
|   |   |-- sizes: struct (nullable = true)
|   |   |   |-- 0: struct (nullable = true)
|   |   |   |   |-- height: integer (nullable = true)
|   |   |   |   |-- resize: integer (nullable = true)
|   |   |   |   |-- width: integer (nullable = true)
|   |   |   |-- 1: struct (nullable = true)
|   |   |   |   |-- height: integer (nullable = true)
|   |   |   |   |-- resize: integer (nullable = true)
|   |   |   |   |-- width: integer (nullable = true)
|   |   |   |-- 2: struct (nullable = true)
|   |   |   |   |-- height: integer (nullable = true)
|   |   |   |   |-- resize: integer (nullable = true)
|   |   |   |   |-- width: integer (nullable = true)
|   |   |   |-- 3: struct (nullable = true)
|   |   |   |   |-- height: integer (nullable = true)
|   |   |   |   |-- resize: integer (nullable = true)
```

```

|      |      |      |      |-- width: integer (nullable = true)
|      |      |-- start: integer (nullable = true)
|      |      |-- type: string (nullable = true)
|      |      |-- url: string (nullable = true)
|-- retweetCount: integer (nullable = true)
...

```

Spark SQL 을 사용할 때 중첩된 field 접근하려면 .(dot) 을 사용하면 된다.

ex) topLevel.nextLevel

From RDDs

SchemaRDD 는 기존 RDD 로 부터 생성할 수 있다.

Python 에서는 Row객체의 RDD 를 생성하고, inferSchema() 를 호출한다.

```

happyPeopleRDD = sc.parallelize([Row(name="holden", favouriteBeverage="coffee")])
happyPeopleSchemaRDD = hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleSchemaRDD.registerTempTable("happy_people")

```

Java 의 경우, getter/setter 를 각 속성별로 갖고 있고, 직렬화(Serializable) 이 구현되어 있는 클래스 RDD 에서 applySchema() 함수를 통해 SchemaRDD 로 변환할 수 있다.

```

class HappyPerson implements Serializable {
    private String name;
    private String favouriteBeverage;
    public HappyPerson() {}
    public HappyPerson(String n, String b) {
        name = n; favouriteBeverage = b;
    }
    public String getName() { return name; }
    public void setName(String n) { name = n; }
    public String getFavouriteBeverage() { return favouriteBeverage; }
    public void setFavouriteBeverage(String b) { favouriteBeverage = b; }
};

...
ArrayList<HappyPerson> peopleList = new ArrayList<HappyPerson>();
peopleList.add(new HappyPerson("holden", "coffee"));
JavaRDD<HappyPerson> happyPeopleRDD = sc.parallelize(peopleList);
SchemaRDD happyPeopleSchemaRDD = hiveCtx.applySchema(happyPeopleRDD,
    HappyPerson.class);
happyPeopleSchemaRDD.registerTempTable("happy_people");

```

JDBC/ODBC Server

Spark 는 BI(Business Intelligence) 사용할 때 유용한 JDBC connection 도 지원한다. JDBC Server 는 여러 Client 들이 사용할 수 있도록 Standalone Driver Program 처럼 수행된다. Spark SQL JDBC Server 는 Hive 에서 지원하는 HiveServer2 와 연동된다. 이는 Thrift Server 인데, Thrift protocol 을 사용한다.


```
sbin/start-thriftserver.sh
sbin/stop-thriftserver.sh
```

자세한 설정은 책 175 페이지를 참고하자.

많은 외부 툴들은 Spark SQL 을 사용할 때, ODBC Driver 를 사용한다. Spark SQL ODBC Driver 는 [Simba](#) 를 이용하여 만들어졌고, 여러 Spark vendor 에서 다운로드 받을 수 있다. 보통 Microstrategy 나 Tableau 같은 BI 툴에서 많이 사용된다.

Working with Beeline

Beeline client 를 이용하여 HiveQL 을 사용할 수 있다. HiveQL 에 대한 내용은 여기서 언급하지 않는다. 다만 몇가지 Operation 에 대해서 설명한다.

1. Data 를 읽어서 Table 을 생성

Hive 는 ,(comma) 로 구분된 CSV 같은 파일로 된 데이터를 로딩할 수 있도록 지원한다.

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'learning-spark-examples/files/int_string.csv'
  INTO TABLE mytable;
```

2. 테이블 목록 조회 및 Schema 확인

```
> SHOW TABLES;
mytable
Time taken: 0.052 seconds
```

만약 Table 을 Caching 하고자 한다면, CACHE TABLE tableName 을 사용하면 된다. 후에 Cache 에서 해제할 땐, UNCACHE TABLE tableName 을 사용하자.

3. 실행계획 확인

```
spark-sql> EXPLAIN SELECT * FROM mytable where key = 1;
== Physical Plan ==
Filter (key#16 = 1)
  HiveTableScan [key#16,value#17], (MetastoreRelation default, mytable, None), None
Time taken: 0.551 seconds
```

Long-Lived Tables and Queries

Spark SQL JDBC Server 를 사용하는 이점 중 하나는 table 을 caching 하여 여러 프로그램에서 공유할 수 있는 것이다. 이는 JDBC Thrift Server 가 단일 Driver Program 이기 때문이다. 이를 사용하기 위해서는 table 을 등록하고, CACHE 명령어를 사용하기만 하면 된다.

User-Defined Functions

SQL 에서 사용할 수 있는 사용자 정의 함수(이하 UDF)를 Python/Scala/Java 모두 지원한다. Spark SQL 은 UDF 와 Hive UDF 모두 지원한다.

Spark SQL UDFs

Spark SQL 은 UDF 를 쉽게 등록할 수 있도록 내장 함수를 제공한다. Scala 나 Python 에서는 native function 이나 lambda 표현을 사용할 수 있고, Java 에서는 UDF class 를 상속해서 구현해야 한다. UDF 는 다양한 타입을 갖고 있다.

Python 이나 Java 에서는 SchemaRDD 유형 중 하나를 반환한다. Java 의 경우는 이 타입을 org.apache.spark.sql.api.java.DataType 에서 확인할 수 있고, Python 은 DataType 을 import 해야 한다.

```
// Import UDF function class and DataTypes
// Note: these import paths may change in a future release
import org.apache.spark.sql.api.java.UDF1 ;
import org.apache.spark.sql.types.DataTypes ;
```

```
hiveCtx.udf().register("stringLengthJava", new UDF1<String, Integer>() {
    @Override
    public Integer call(String str) throws Exception {
        return str.length();
    }
}, DataTypes.IntegerType);
SchemaRDD tweetLength = hiveCtx.sql(
    "SELECT stringLengthJava('text') FROM tweets LIMIT 10");
List<Row> lengths = tweetLength.collect();
for (Row row : result) {
    System.out.println(row.get(0));
}
```

Hive UDFs

Spark SQL 은 표준 Hive UDF 를 사용할 수 있다. 만약 UDF 를 직접 만들고자 한다면, UDF 클래스가 포함된 Jar 를 Spark Application 수행시 포함될 수 있도록 --jar 로 추가해 주어야 한다. 또한 Hive UDF 를 사용하기 위해서는 SparkContext 가 아니라 HiveContext 를 사용해야 한다.

```
hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS class.function")
```

Spark SQL Performance

Spark SQL 에서는 Caching 된 데이터를 사용할 때, 메모리상의 컬럼형 데이터를 사용한다. 이는 Caching 할 때 적은 공간을 사용할 뿐만 아니라 데이터 일부만 읽고자 할 때, Spark SQL 은 일부의 데이터(field) 만 읽는다.

Spark SQL 은 Oracle 에서 지원되는 Predicate Push-Down 이 지원된다. Spark 에서 특정 row 만 읽고자 하더라도 기본적으로는 전체 데이터를 읽고 filter 를 수행한다. 하지만 Spark SQL 에서는 키 범위의 일부 데이터만 추출할 수 있고, 결과를 갖고 올 때 아주 작은 데이터만 읽어서 처리한다.

Performance Tuning Option

Spark SQL에서는 성능 개선을 위해 많은 Tuning Option 을 제공한다. (아래 표 참고)

Option	Default	Usage
<code>spark.sql.codegen</code>	<code>false</code>	When <code>true</code> , Spark SQL will compile each query to Java bytecode on the fly. This can improve performance for large queries, but <code>codegen</code> can slow down very short queries.
<code>spark.sql.inMemoryColumnarStorage.compressed</code>	<code>false</code>	Compress the in-memory columnar storage automatically.
<code>spark.sql.inMemoryColumnarStorage.batchSize</code>	<code>1000</code>	The batch size for columnar caching. Larger values may cause out-of-memory problems
<code>spark.sql.parquet.compression.codec</code>	<code>snappy</code>	Which compression codec to use. Possible options include <code>uncompressed</code> , <code>snappy</code> , <code>gzip</code> , and <code>lzo</code> .

JDBC Connector 나 Beeline shell 을 사용할 때, `set command` 를 통해서 옵션들을 설정한다.

```
beeline> set spark.sql.codegen=true;
SET spark.sql.codegen=true
spark.sql.codegen=true
Time taken: 1.196 seconds
```

기존에는 Spark SQL 에서 아래와 같이 `spark.sql.codegen` 옵션을 설정하기도 하였다.

```
conf.set("spark.sql.codegen", "true")
```

몇 가지 옵션들은 설정할 때 신중해야 하는데, 두 가지를 예를 들어 설명한다.

1. `spark.sql.codegen`

해당 속성이 `true` 라면 SQL 을 매번 Java Bytecode 로 변환하여 수행한다. 쿼리가 아주 길거나

자주 수행되는 쿼리에 대해서는 부분적으로 빠르지만 짧은 쿼리나 ad-hoc 쿼리 같은 경우에는 매번 SQL 을 변환해야 하기 때문에 overhead 가 있을 수 있다.

2. spark.sql.inMemoryColumnarStorage.batchSize

기본값은 1000으로 설정되어 있고, 각 Batch 마다 압축을 한다. Batch Size 가 작은 경우 압축할 용량이 작지만, 아주 큰 경우에는 메모리 상에서 압축을 하기에는 너무 클 수 있기 때문에 문제가 될 수 있다. 만약 Row 가 아주 크다면, batch size 를 줄여서 OOM 이 발생하지 않도록 해야한다. 보통은 기본 값이면 적당하다.