E6895 Advanced Big Data Analytics Lecture 4:

Data Store

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
Mgr., Dept. of Network Science and Big Data Analytics, IBM Watson Research Center
Reference

Learning Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia
Spark SQL

- Tableau
- ...
- JDBC/ODBC
- Your Application
- Spark SQL shell
- Spark SQL
- Hive
- JSON
- Parquet
- ...

E6895 Advanced Big Data Analytics – Lecture 4: Data Store
© 2015 CY Lin, Columbia University
Spark SQL can be built with or without Apache Hive, the Hadoop SQL engine. Spark SQL with Hive support allows us to access Hive tables, UDFs (user-defined functions), SerDes (serialization and deserialization formats), and the Hive query language (HiveQL). Hive query language (HQL) It is important to note that including the Hive libraries does not require an existing Hive installation. In general, it is best to build Spark SQL with Hive support to access these features. If you **download Spark in binary form**, it should already be built with Hive support. If you are building Spark from source, you should run `sbt/sbt -Phive assembly`. 
Apache Hive

- Administration and Server Coordination:
  - Hive Web Interface
  - Hive CLI
  - Hue

- Client Applications:
  - Squirrel SQL
  - Microsoft Excel
  - Hive Thrift Client
  - Hive JDBC Driver
  - Hive ODBC Driver

- Hive:
  - Hive Thrift Server
  - Hive Driver
  - Metastore
  - Apache Derby DB

- Processing Frameworks Resource Management:
  - MapReduce v2
  - Tez
  - MapReduce v1
  - YARN

- Distributed Storage:
  - HDFS
Using Hive to Create a Table

(A) $ $HIVE_HOME/bin hive --service cli
(B) hive> set hive.cli.print.current.db=true;
(C) hive (default)> CREATE DATABASE ourfirstdatabase;
OK
Time taken: 3.756 seconds
(D) hive (default)> USE ourfirstdatabase;
OK
Time taken: 0.039 seconds
(E) hive (ourfirstdatabase)> CREATE TABLE our_first_table (  
  > FirstName STRING,
  > LastName STRING,
  > EmployeeId INT);
OK
Time taken: 0.043 seconds
hive (ourfirstdatabase)> quit;
(F) $ ls /home/biadmin/Hive/warehouse/ourfirstdatabase.db
our_first_table
Creating, Dropping, and Altering DBs in Apache Hive

(1) $ $HIVE_HOME/bin hive --service cli
(2) hive> set hive.cli.print.current.db=true;
(3) hive (default)> USE ourfirstdatabase;
(4) hive (ourfirstdatabase)> ALTER DATABASE ourfirstdatabase SET DBPROPERTIES
('creator'='Bruce Brown', 'created_for'='Learning Hive DDL');
OK
Time taken: 0.138 seconds
(5) hive (ourfirstdatabase)> DESCRIBE DATABASE EXTENDED ourfirstdatabase;
OK
ourfirstdatabase file:/home/biad
min/Hive/warehouse/ourfirstdatabase.db {created_for=Learning Hive DDL, creator=Bruce Brown}
Time taken: 0.084 seconds, Fetched: 1 row(s)
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS]
database_name
(6) hive (ourfirstdatabase)> DROP DATABASE ourfirstdatabase CASCADE;
OK
Time taken: 0.132 seconds
Another Hive Example

(A) CREATE TABLE IF NOT EXISTS FlightInfo2007 (  
    Year SMALLINT, Month TINYINT, DayofMonth TINYINT, DayOfWeek TINYINT,  
    DepTime SMALLINT, CRSDepTime SMALLINT, ArrTime SMALLINT, CRSArrTime SMALLINT,  
    UniqueCarrier STRING, FlightNum STRING, TailNum STRING,  
    ActualElapsedTime SMALLINT, CRSElapsedTime SMALLINT,  
    AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,  
    Origin STRING, Dest STRING, Distance INT,  
    TaxiIn SMALLINT, TaxiOut SMALLINT, Cancelled SMALLINT,  
    CancellationCode STRING, Diverted SMALLINT,  
    CarrierDelay SMALLINT, WeatherDelay SMALLINT,  
    NASDelay SMALLINT, SecurityDelay SMALLINT, LateAircraftDelay SMALLINT)  
COMMENT 'Flight InfoTable'  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
LINES TERMINATED BY '\n'  
STORED AS TEXTFILE  
TBLOPTIONS ('creator'='Bruce Brown', 'created_at'='Thu Sep 19 10:58:00 EDT 2013');
Hive’s operation modes

[Diagram showing Hive’s operation modes with client interactions and data flow through Hive CLI and Hive Web Interface, leading to Apache Derby DB.]
Using HiveQL for Spark SQL

When programming against Spark SQL we have two entry points depending on whether we need Hive support. The recommended entry point is the HiveContext to provide access to HiveQL and other Hive-dependent functionality. The more basic SQLContext provides a subset of the Spark SQL support that does not depend on Hive. The separation exists for users who might have conflicts with including all of the Hive dependencies. Using a HiveContext does not require an existing Hive setup.

HiveQL is the recommended query language for working with Spark SQL. Many resources have been written on HiveQL, including Programming Hive and the online Hive Language Manual. In Spark 1.0 and 1.1, Spark SQL is based on Hive 0.12,
Hive Language Manual

This is the Hive Language Manual.

- Commands and CLIs
  - Commands
  - Hive CLI
  - Variable Substitution
  - Beeline CLI for HiveServer2
  - HCatalog CLI
- File Formats
  - Avro Files
  - ORC Files
  - Parquet
  - Compressed Data Storage
  - LZO Compression
- Data Types
Using Spark SQL — Steps and Example

Example 9-5. Python SQL imports

```python
# Import Spark SQL
from pyspark.sql import HiveContext, Row
```

Example 9-8. Constructing a SQL context in Python

```python
hiveCtx = HiveContext(sc)
```

Example 9-11. Loading and querying tweets in Python

```python
input = hiveCtx.jsonFile(inputFile)
# Register the input schema RDD
input.registerTempTable("tweets")
# Select tweets based on the retweetCount
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM tweets ORDER BY retweetCount LIMIT 10""")
```
Query testtweet.json

Get it from Learning Spark Github ==> https://github.com/databricks/learning-spark/tree/master/files

```json
{"createdAt":"Nov 4, 2014 4:56:59 PM","id":529799371026485248,"text":"Adventures With Coffee, Code, and Writing.","source":"\u003ca href=\u003d"http://twitter.com\u003eClient\u003aTwitter Web Client\u003c/a\u003e","isTruncated":false,"inReplyToStatusId":-1,"inReplyToUserId":-1,"isFavorited":false,"retweetCount":0,"isPossiblySensitive":false,"contributorsIDs":[],"userMentionEntities":[],"urlEntities":[],"hashtagEntities":[],"mediaEntities":[],"currentUserRetweetId":-1,"user":{"id":15594928,"name":"Holden Karau","screenName":"holdenkarau","location":"","description":"","descriptionURLEntities":[],"isContributorsEnabled":false,"profileImageURL":"http://pbs.twimg.com/profile_images/3005696115/2036374bbadbed85249cdd50aac6e170_normal.jpg","profileImageURLHttps":"https://pbs.twimg.com/profile_images/3005696115/2036374bbadbed85249cdd50aac6e170_normal.jpg","isProtected":false,"followersCount":1231,"profileBackgroundColor":"C0DEDE","profileTextColor":"333333","profileLinkColor":"004B44","profileSidebarFillColor":"DDE6F6","profileSidebarBorderColor":"FFFFFF","profileUseBackgroundImage":true,"showAllInlineMedia":false,"friendsCount":600,"createdAt":"Aug 5, 2011 9:42:44 AM","favouritesCount":1095,"utcOffset":-3,"profileBackgroundColorURL":"","profileBannerImageURL":"","profileBackgroundTiled":true,"lang":"en","statusesCount":6234,"isGeoEnabled":true,"isVerified":false,"translator":false,"listedCount":0,"isFollowRequestSent":false}

>>> print topTweets.collect()
[Row(text=u'Adventures With Coffee, Code, and Writing.', retweetCount=0)]
```
SchemaRDD

Both loading data and executing queries return SchemaRDDs. SchemaRDDs are similar to tables in a traditional database. Under the hood, a SchemaRDD is an RDD composed of Row objects with additional schema information of the types in each column. Row objects are just wrappers around arrays of basic types (e.g., integers and strings).
Row Objects

Row objects represent records inside SchemaRDDS, and are simply fixed-length arrays of fields.

Example 9-14. Accessing the text column in the topTweets SchemaRDD in Python

topTweetText = topTweets.map(lambda row: row.text)

<table>
<thead>
<tr>
<th>Spark SQL/HiveQL type</th>
<th>Scala type</th>
<th>Java type</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRUCT&lt;COL1: COL1_TYPE, ...&gt;</td>
<td>Row</td>
<td>Row</td>
<td>Row</td>
</tr>
</tbody>
</table>
### Types stored by Schema RDDs

<table>
<thead>
<tr>
<th>Spark SQL/HiveQL type</th>
<th>Scala type</th>
<th>Java type</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>Byte</td>
<td>Byte/byte</td>
<td>int/long (in range of −128 to 127)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>Short</td>
<td>Short/short</td>
<td>int/long (in range of −32768 to 32767)</td>
</tr>
<tr>
<td>INT</td>
<td>Int</td>
<td>Int/int</td>
<td>int or long</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Long</td>
<td>Long/long</td>
<td>long</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Float</td>
<td>Float/float</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Double</td>
<td>Double/Double/Double</td>
<td>float</td>
</tr>
<tr>
<td>STRING</td>
<td>String</td>
<td>String</td>
<td>string</td>
</tr>
<tr>
<td>BINARY</td>
<td>Array[Byte]</td>
<td>byte[]</td>
<td>bytearray</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>Boolean</td>
<td>Boolean/Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>java.sql.Timestamp</td>
<td>datetime.datetime</td>
</tr>
<tr>
<td>ARRAY&lt;DATA_TYPE&gt;</td>
<td>Seq</td>
<td>List</td>
<td>list, tuple, or array</td>
</tr>
<tr>
<td>MAP&lt;KEY_TYPE, VAL_TYPE&gt;</td>
<td>Map</td>
<td>Map</td>
<td>dict</td>
</tr>
</tbody>
</table>
Look at the Schema

```python
>>> input.printSchema()
root
|-- contributorsIDs: array (nullable = true)
  |-- element: string (containsNull = false)
|-- createdAt: string (nullable = true)
|-- currentUserId: integer (nullable = true)
|-- hashtagEntities: array (nullable = true)
  |-- element: string (containsNull = false)
|-- id: long (nullable = true)
|-- inReplyToStatusId: integer (nullable = true)
|-- inReplyToUserId: integer (nullable = true)
|-- isFavorited: boolean (nullable = true)
|-- isPossiblySensitive: boolean (nullable = true)
|-- isTruncated: boolean (nullable = true)
|-- mediaEntities: array (nullable = true)
  |-- element: string (containsNull = false)
|-- retweetCount: integer (nullable = true)
|-- source: string (nullable = true)
|-- text: string (nullable = true)
|-- urlEntities: array (nullable = true)
  |-- element: string (containsNull = false)
```

(not a complete screen shot)
Another way to create SchemaRDD

Example 9-28. Creating a SchemaRDD using Row and named tuple in Python

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

Spark SQL provides JDBC connectivity, which is useful for connecting business intelligence tools to a Spark cluster and for sharing a cluster across multiple users.

The server can be launched with `sbin/start-thriftserver.sh` in your Spark directory (Example 9-31). This script takes many of the same options as `spark-submit`. By default it listens on `localhost:10000`, but we can change these with either environment variables (HIVE_SERVER2_THRIFT_PORT and HIVE_SERVER2_THRIFT_BIND_HOST), or with Hive configuration properties (hive.server2.thrift.port and hive.server2.thrift.bind.host). You can also specify Hive properties on the command line with `--hiveconf property=value`.

**Example 9-31. Launching the JDBC server**

`.sbin/start-thriftserver.sh --master sparkMaster`

**Example 9-32. Connecting to the JDBC server with Beeline**

`holden@hmbp2:/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000`

Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
User-Defined Functions (UDF)

UDFs allow you to register custom functions in Python, Java, and Scala to call within SQL.

This is a very popular way to expose advanced functionality to SQL users in an organization, so that these users can call into it without writing code.

Example 9-36. Python string length UDF

```python
# Make a UDF to tell us how long some text is
hiveCtx.registerFunction("strLenPython", lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql("SELECT strLenPython('text') FROM tweets LIMIT 10")
```
RDF and SPARQL
Spark Streaming

Spark Streaming is Spark’s module for such applications. It lets users write streaming applications using a very similar API to batch jobs, and thus reuse a lot of the skills and even code they built for those.

Much like Spark is built on the concept of RDDs, Spark Streaming provides an abstraction called DStreams, or discretized streams. A DStream is a sequence of data arriving over time. Internally, each DStream is represented as a sequence of RDDs arriving at each time step (hence the name “discretized”).

In Spark 1.1, Spark Streaming is available only in Java and Scala. Spark 1.2 has limited Python support.
Spark Streaming architecture
Spark Streaming with Spark’s components
Try these examples

Databricks Reference Apps

At Databricks, we are developing a set of reference applications that demonstrate how to use Apache Spark. This book/repo contains the reference applications.

- View the code in the Github Repo here: https://github.com/databricks/reference-apps
- Read the documentation here: http://databricks.gitbooks.io/databricks-spark-reference-applications/
- Submit feedback or issues here: https://github.com/databricks/reference-apps/issues

The reference applications will appeal to those who want to learn Spark and learn better by example. Browse the applications, see what features of the reference applications are similar to the features you want to build, and refashion the code samples for your needs. Additionally, this is meant to be a practical guide for using Spark in your systems, so the applications mention other technologies that are compatible with Spark - such as what file systems to use for storing your massive data sets.

- **Log Analysis Application** - The log analysis reference application contains a series of tutorials for learning Spark by example as well as a final application that can be used to monitor Apache access logs. The examples use Spark in batch mode, cover Spark SQL, as well as Spark Streaming.

- **Twitter Streaming Language Classifier** - This application demonstrates how to fetch and train a language classifier for Tweets using Spark MLLib. Then Spark Streaming is used to call the trained classifier and filter out live tweets that match a specified cluster. To build this example go into the twitter_classifier/scala and follow the direction in the README.
RDF and SPARQL
WHAT DO RDF AND SPARQL BRING TO BIG DATA PROJECTS?

Bob DuCharme
TopQuadrant, Charlottesville, Virginia

Resource Description Format (RDF)

- A W3C standard since 1999
- Triples
- Example: A company has nine of part p1234 in stock, then a simplified triple representing this might be \{p1234 inStock 9\}.
- Instance Identifier, Property Name, Property Value.
- In a proper RDF version of this triple, the representation will be more formal. They require uniform resource identifiers (URIs).

```html
@prefix fbd:<http://foobarco.net/data/>.
@prefix fbv:<http://foobarco.net/vocab/>.

fbd:p1234 fbv:inStock "9".
fbd:p1234 fbv:supplier "Joe’s Part Company".
```
An example complete description

@prefix fbd:<http://foobarco.net/data/>.
@prefix fbv:<http://foobarco.net/vocab/>.
fbd:p1234 fbv:inStock "9".
fbd:p1234 fbv:name "Blue reverse flange".
fbd:p1234 fbv:supplier fbd:s9483.
fbd:s9483 fbv:name "Joe's Part Company".
fbd:s9483 fbv:homePage "http://www.joespartco.com".
fbd:s9483 fbv:contactName "Gina Smith".
fbd:s9483 fbv:contactEmail "gina.smith@joespartco.com".
Advantages of RDF

- Virtually any RDF software can parse the lines shown above as self-contained, working data file.
  - You can declare properties if you want.
  - The RDF Schema standard lets you declare classes and relationships between properties and classes.
  - The flexibility that the lack of dependence on schemas is the first key to RDF's value.

- Split trips into several lines that won't affect their collective meaning, which makes sharding of data collections easy.
  - Multiple datasets can be combined into a usable whole with simple concatenation.

- For the inventory dataset's property name URIs, sharing of vocabulary makes easy to aggregate.
The following SPQRL query asks for all property names and values associated with the fbd:s9483 resource:

```sparql
PREFIX fbd:<http://foobarco.net/data/>

SELECT ?property ?value
WHERE {fbd:s9483 ?property ?value.}
```

The heart of any SPARQL query is the WHERE clause, which specifies the triples to pull out of the dataset. Various options for the rest of the query tell the SPARQL processor what to do with those triples, such as sorting, creating, or deleting triples. The above query’s WHERE clause has a single triple pattern, which resembles a triple but may have variables substituted for any or all of the triple’s three parts. The triple pattern above says that we’re interested in triples that have fbd:s9483 as the subject and—because variables function as wildcards—anything at all in the triple’s second and third parts.
The SPAQRL Query Result from the previous example

<table>
<thead>
<tr>
<th>property</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://foobaro.net/vocab/contactEmail">http://foobaro.net/vocab/contactEmail</a></td>
<td>&quot;<a href="mailto:gina.smith@joespartco.com">gina.smith@joespartco.com</a>&quot;</td>
</tr>
<tr>
<td><a href="http://foobaro.net/vocab/contactName">http://foobaro.net/vocab/contactName</a></td>
<td>&quot;Gina Smith&quot;</td>
</tr>
<tr>
<td><a href="http://foobaro.net/vocab/name">http://foobaro.net/vocab/name</a></td>
<td>&quot;Joe's Part Company&quot;</td>
</tr>
</tbody>
</table>
Another SPARQL Example

What is this query for?

PREFIX fbd:<http://foobaro.net/data/>  
PREFIX fbv:<http://foobaro.net/vocab/>  

SELECT ?flangeContactEmail  
WHERE  
{  
  ?part fbv:name "Blue reverse flange".  
  ?supplier fbv:contactEmail ?flangeContactEmail.  
}

Data

@prefix fbd:<http://foobaro.net/data/>  
@prefix fbv:<http://foobaro.net/vocab/>  
  fbd:p1234 fbv:inStock "9".  
  fbd:p1234 fbv:name "Blue reverse flange".  
  fbd:p1234 fbv:supplier fbd:s9483.  
  fbd:s9483 fbv:name "Joe’s Part Company".  
  fbd:s9483 fbv:homePage "http://www.joespartco.com".  
  fbd:s9483 fbv:contactName "Gina Smith".  
  fbd:s9483 fbv:contactEmail "gina.smith@joespartco.com".  

Open Source Software – Apache Jena

Apache Jena

A free and open source Java framework for building Semantic Web and Linked Data applications.

Get started now! Download

RDF

RDF API
Interact with the core API to create and read Resource Description Framework (RDF) graphs. Sanitise your triples using popular formats such as RDF/XML or Turtle.

ARQ (SPARQL)
Query your RDF data using ARQ, a SPARQL 1.1 compliant engine. ARQ supports remote federated

Triple store

TDB
Persist your data using TDB, a native high performance triple store. TDB supports the full range of Jena APIs.

Fuseki
Expose your triples as a SPARQL end-point accessible over HTTP. Fuseki provides REST-style interaction with your RDF data.

OWL

Ontology API
Work with models, RDFS and the Web Ontology Language (OWL) to add extra semantics to your RDF data.

Inference API
Reason over your data to expand and check the content of your triple store. Configure your own inference rules or