Spark SQL

10 Things You Need to Know

@KrisDevelops

algebraix data

About Algebraix

- Located in Encinitas, CA & Austin, TX
- We work on a technology called Data Algebra
- We hold nine patents in this technology
- Create turnkey performance enhancement for db engines
- We’re working on a product called Algebraix Query Accelerator
- The first public release of the product focuses on Apache Spark
- The product will be available in Amazon Web Services and Azure

About Me

- I’m a long time developer turned product person
- I love music (playing music and listening to music)
- I grew up in Palm Springs, CA
- **Development:** Full stack developer, UX developer, API dev, etc
- **Product:** worked in UX Lead, Director Product Innovation, VP Product
- **Certs:** AWS Solutions Architect, AWS Developer
- **Education:** UCI - Polical Science, Chemistry

About You

- Does anyone here use:
    - Spark or Spark SQL? In Production?
    - Amazon Web Services?
    - SQL Databases?
    - NoSQL Databases?
    - Distributed Databases?
- What brings you in here?
    - Want to become a rockstar at work?
    - Trying to do analytics at a larger scale?
    - Want exposure to the new tools in the industry?

Goals of This Talk

For you to be able to get up and running quickly with real world data and produce applications, while avoiding major pitfalls that most newcomers to Spark SQL would encounter.

Spark SQL: 10 things you should know

1. Spark SQL use cases
2. Loading data: in the cloud vs locally, RDDs vs DataFrames
3. SQL vs the DataFrame API. What's the difference?
4. Schemas: implicit vs explicit schemas, data types
5. Loading & saving results
6. What SQL functionality works and what doesn't?
7. Using SQL for ETL
8. Working with JSON data
9. Reading and Writing to an external SQL databases
10. Testing your work in the real world

Notes

How to read what I'm writing

>>>

When I write ">>>" it means that I'm talking to the pyspark console and what follows immediately afterward is the output.

Notes

How to read what I’m writing

>>>

When I write >>> it means that I’m talking to the pyspark console and what follows immediately afterward is the output.

```
>>> aBunchOfData.count()
901150
```

Notes

How to read what I’m writing

>>>

When I write >>> it means that I’m talking to the pyspark console and what follows immediately afterward is the output.

```
>>> aBunchOfData.count()
901150
```

Otherwise, I’m just demonstrating pyspark code.

Notes

How to read what I'm writing

When I write >>> it means that I'm talking to the pyspark console and what follows immediately afterward is the output.

```
>>> aBunchOfData.count()
901150
```

Otherwise, I'm just demonstrating pyspark code.

@KrisDevelops

algebraixdata

afterward is the output.

```
>>> aBunchOfData.count()
901150
```

Otherwise, I'm just demonstrating pyspark code.

When I show you this guy ^^ it means we're done with this section, and feel free to ask questions before we move on. =D

Notes

How to read what I'm writing

1

Spark SQL Use Cases

What, when, why

1

Spark SQL Use Cases

What, when, why

Ad-hoc querying of data in files

1

Spark SQL Use Cases

What, when, why

Ad-hoc querying of data in files

ETL capabilities alongside familiar SQL

1

Spark SQL Use Cases

What, when, why

Ad-hoc querying of data in files

Live SQL analytics over streaming data

ETL capabilities alongside familiar SQL

1

Spark SQL Use Cases

What, when, why

- Ad-hoc querying of data in files
- ETL capabilities alongside familiar SQL
- Live SQL analytics over streaming data
- Interaction with external Databases

1

Spark SQL Use Cases

What, when, why

Ad-hoc querying of data in files

ETL capabilities alongside familiar SQL

Live SQL analytics over streaming data

Interaction with external Databases

Scalable query performance with larger clusters

2

Loading Data

Cloud vs local,
RDD vs DF

2

Loading Data

Cloud vs local,
RDD vs DF

You can load data directly into a DataFrame, and begin querying it relatively quickly. Otherwise you'll need to load data into an RDD and transform it first.

2

Loading Data

Cloud vs local,
RDD vs DF

You can load data directly into a DataFrame, and begin querying it relatively quickly. Otherwise you'll need to load data into an RDD and transform it first.

```
# loading data into an RDD in Spark 2.0
sc = spark.sparkContext
oneSysLog = sc.textFile("file:/var/log/system.log")
allSysLogs = sc.textFile("file:/var/log/system.log*")
allLogs = sc.textFile("file:/var/log/*.log")
```

2

Loading Data

Cloud vs local,
RDD vs DF

You can load data directly into a DataFrame, and begin querying it relatively quickly. Otherwise you'll need to load data into an RDD and transform it first.

```
# loading data into an RDD in Spark 2.0
sc = spark.sparkContext
oneSysLog = sc.textFile("file:/var/log/system.log")
allSysLogs = sc.textFile("file:/var/log/system.log*")
allLogs = sc.textFile("file:/var/log/*.log")

# lets count the lines in each RDD
>>> oneSysLog.count()
8339
>>> allSysLogs.count()
47916
>>> allLogs.count()
546254
```

2

Loading Data

Cloud vs local,
RDD vs DF

you'll need to load data into an RDD and transform it first.

```
# loading data into an RDD in Spark 2.0
sc = spark.sparkContext
oneSysLog = sc.textFile("file:/var/log/system.log")
allSysLogs = sc.textFile("file:/var/log/system.log*")
allLogs = sc.textFile("file:/var/log/*.log")

# lets count the lines in each RDD
>>> oneSysLog.count()
8339
>>> allSysLogs.count()
47916
>>> allLogs.count()
546254
```

That's great, but you can't query this. You'll need to convert the data to Rows, add a schema, and convert it to a dataframe.

2

Loading Data

Cloud vs local,
RDD vs DF

```
allLogs = sc.textFile("file:/var/log/*.log")

# lets count the lines in each RDD
>>> oneSysLog.count()
8339
>>> allSysLogs.count()
47916
>>> allLogs.count()
546254
```

That's great, but you can't query this. You'll need to convert the data to Rows, add a schema, and convert it to a dataframe.

```
# import Row, map the rdd, and create dataframe
from pyspark.sql import Row
sc = spark.sparkContext
allSysLogs = sc.textFile("file:/var/log/system.log*")
logsRDD = allSysLogs.map(lambda logRow: Row(log=logRow))
logsDF = spark.createDataFrame(logsRDD)
```

```
>>> allLogs.count()
546254
```

That's great, but you can't query this. You'll need to convert the data to Rows, add a schema, and convert it to a dataframe.

```
# import Row, map the rdd, and create dataframe
from pyspark.sql import Row
sc = spark.sparkContext
allSysLogs = sc.textFile("file:/var/log/system.log*")
logsRDD = allSysLogs.map(lambda logRow: Row(log=logRow))
logsDF = spark.createDataFrame(logsRDD)
```

Once the data is converted to *at least* a DataFrame with a schema, now you can talk SQL to the data.

2

Loading Data

Cloud vs local, RDD vs DF

2

Loading Data

Cloud vs local,
RDD vs DF

```
from pyspark.sql import Row
sc = spark.sparkContext
allSysLogs = sc.textFile("file:/var/log/system.log*")
logsRDD = allSysLogs.map(lambda logRow: Row(log=logRow))
logsDF = spark.createDataFrame(logsRDD)
```

Once the data is converted to *at least* a DataFrame with a schema, now you can talk SQL to the data.

```
# write some SQL
logsDF = spark.createDataFrame(logsRDD)
logsDF.createOrReplaceTempView("logs")
>>> spark.sql("SELECT * FROM logs LIMIT 1").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

2

Loading Data

Cloud vs local, RDD vs DF

Once the data is converted to *at least* a DataFrame with a schema, now you can talk SQL to the data.

```
# write some SQL
logsDF = spark.createDataFrame(logsRDD)
logsDF.createOrReplaceTempView("logs")
>>> spark.sql("SELECT * FROM logs LIMIT 1").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

But, you can also load certain types of data and store it directly as a DataFrame. This allows you to get to SQL quickly.

2

Loading Data

Cloud vs local,
RDD vs DF

```
# write some SQL
logsDF = spark.createDataFrame(logsRDD)
logsDF.createOrReplaceTempView("logs")
>>> spark.sql("SELECT * FROM logs LIMIT 1").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

But, you can also load certain types of data and store it directly as a DataFrame. This allows you to get to SQL quickly.

Both JSON and Parquet formats can be loaded as a DataFrame straightaway because they contain *enough* schema information to do so.

2

Loading Data

Cloud vs local,
RDD vs DF

But, you can also load certain types of data and store it directly as a DataFrame. This allows you to get to SQL quickly.

Both JSON and Parquet formats can be loaded as a DataFrame straightaway because they contain *enough* schema information to do so.

```
# load parquet straight into DF, and write some SQL
logsDF = spark.read.parquet(“file:/logs.parquet”)
logsDF.createOrReplaceTempView(“logs”)
>>> spark.sql(“SELECT * FROM logs LIMIT 1”).show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

2

Loading Data

Cloud vs local,
RDD vs DF

Both JSON and Parquet formats can be loaded as a DataFrame straightaway because they contain *enough* schema information to do so.

```
# load parquet straight into DF, and write some SQL
logsDF = spark.read.parquet("file:/logs.parquet")
logsDF.createOrReplaceTempView("logs")
>>> spark.sql("SELECT * FROM logs LIMIT 1").show()
+--------------------+
|                 row|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

In fact, now they even have support for querying parquet files directly! Easy peasy!

2

Loading Data

Cloud vs local,
RDD vs DF

```
>>> spark.sql("SELECT * FROM logs LIMIT 1").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

In fact, now they even have support for querying parquet files directly! Easy peasy!

```
# load parquet straight into DF, and write some SQL
>>> spark.sql("""
  SELECT * FROM parquet.`path/to/logs.parquet` LIMIT 1
""").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

2

Loading Data

Cloud vs local,
RDD vs DF

In fact, now they even have support for querying parquet files directly! Easy peasy!

```
# load parquet straight into DF, and write some SQL
>>> spark.sql("""
  SELECT * FROM parquet.`path/to/logs.parquet` LIMIT 1
""").show()
+--------------------+
|                 log|
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

That's one aspect of loading data. The other aspect is using the protocols for cloud storage (i.e. s3://). In some cloud ecosystems, support for their storage protocol comes installed already.

```
+--------------------+
|Jan  6 16:37:01 (...|
+--------------------+
```

That's one aspect of loading data. The other aspect is using the protocols for cloud storage (i.e. s3://). In some cloud ecosystems, support for their storage protocol comes installed already.

```
# i.e. on AWS EMR, s3:// is installed already.
sc = spark.sparkContext
decemberLogs = sc.textFile("s3://acme-co/logs/2016/12/")

# count the lines in all of the december 2016 logs in S3
>>> decemberLogs.count()
910125081250

# wow, such logs. Ur poplar.
```

2

Loading Data

Cloud vs local,
RDD vs DF

2

Loading Data

Cloud vs local, RDD vs DF

aspect is using the protocols for cloud storage (i.e. s3://). In some cloud ecosystems, support for their storage protocol comes installed already.

```
# i.e. on AWS EMR, s3:// is installed already.
sc = spark.sparkContext
decemberLogs = sc.textFile("s3://acme-co/logs/2016/12/")

# count the lines in all of the december 2016 logs in S3
>>> decemberLogs.count()
910125081250

# wow, such logs. Ur poplar.
```

Sometimes you actually need to provide support for those protocols if your VM's OS doesn't have it already.

2

Loading Data

Cloud vs local,
RDD vs DF

```
# count the lines in all of the december 2016 logs in S3
>>> decemberLogs.count()
910125081250

# wow, such logs. Ur poplar.
```

Sometimes you actually need to provide support for those protocols if your VM's OS doesn't have it already.

```
my-linux-shell$ pyspark --packages
com.amazonaws:aws-java-sdk-pom:1.10.34,com.amazonaws:aws-java-sdk:1.7.4,org.apache.hadoop:hadoop-aws:2.7.1 demo2.py


>>> rdd = sc.readText("s3a://acme-co/path/to/files")
rdd.count()

# note: "s3a" and not "s3" -- "s3" is specific to AWS EMR.
```

2

Loading Data

Cloud vs local,
RDD vs DF

Sometimes you actually need to provide support for those protocols if your VM's OS doesn't have it already.

```
my-linux-shell$ pyspark --packages
com.amazonaws:aws-java-sdk-pom:1.10.34,com.amazonaws:aws-jav
a-sdk:1.7.4,org.apache.hadoop:hadoop-aws:2.7.1


>>> rdd = sc.readText("s3a://acme-co/path/to/files")
rdd.count()

# note: "s3a" and not "s3" -- "s3" is specific to AWS EMR.
```

Now you should have several ways to load data to quickly start writing SQL with Apache Spark.

2

Loading Data

Cloud vs local,
RDD vs DF

```
my-linux-shell$ pyspark --packages
com.amazonaws:aws-java-sdk-pom:1.10.34,com.amazonaws:aws-jav
a-sdk:1.7.4,org.apache.hadoop:hadoop-aws:2.7.1


>>> rdd = sc.readText("s3a://acme-co/path/to/files")
rdd.count()

# note: "s3a" and not "s3" -- "s3" is specific to AWS EMR.
```

Now you should have several ways to load data to quickly start writing SQL with Apache Spark.

@KrisDevelops

algebraixdata

3

Dataframe Functions vs SQL

What is a Dataframe?

3

Dataframe Functions vs SQL

What is a Dataframe?

What is a DataFrame?

You can think of dataframes like RDDs with a schema.

3

Dataframe Functions vs SQL

What is a Dataframe?

What is a DataFrame?

Note: "DataFrame is just a type alias for Dataset of Row" -- Databricks

3

Dataframe Functions vs SQL

What is a Dataframe?

What is a DataFrame?

You can think of dataframes like RDDs with a schema.

Why DataFrame over RDD?

Catalyst optimization & schemas

3

Dataframe Functions vs SQL

What is a Dataframe?

What is a DataFrame?

You can think of dataframes like RDDs with a schema.

Why DataFrame over RDD?

Catalyst optimization & schemas

What kind of data can DataFrames handle?

Text, JSON, XML, Parquet, and more

3

Dataframe Functions vs SQL

What is a Dataframe?

What is a DataFrame?

You can think of dataframes like RDDs with a schema.

What can I do with a DataFrame?

Use SQL-like and actual SQL. Also, you can apply schemas to your data and benefit from the performance enhancements of the **Catalyst** optimizer.

Why DataFrame over RDD?

Catalyst optimization & schemas

What kind of data can DataFrames handle?

Text, JSON, XML, Parquet, and more

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

Still Catalyst Optimized

Both SQL and API Functions in DF's sit atop Catalyst

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

DataFrame Functions

Provides a bridge between to features of Spark APIs

Still Catalyst Optimized

Both SQL and API Functions in DF's sit atop Catalyst

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

SQL With DataFrames

Allows you a familiar way to interact with the data

DataFrame Functions

Provides a bridge between to features of Spark APIs

Still Catalyst Optimized

Both SQL and API Functions in DF's sit atop Catalyst

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

SQL With DataFrames

Allows you a familiar way to interact with the data

DataFrame Functions

Provides a bridge between to features of Spark APIs

SQL-Like Functions in DataFrame API

For many of the expected features of SQL, there are similar functions in the DF API that do practically the same thing, allowing for .functional().chaining()

Still Catalyst Optimized

Both SQL and API Functions in DF's sit atop Catalyst

3

Dataframe Functions vs SQL

SQL-Like functions in the Dataframe API

SQL With DataFrames

Allows you a familiar way to interact with the data

DataFrame Functions

Provides a bridge between to features of Spark APIs

SQL-Like Functions in DataFrame API

For many of the expected features of SQL, there are similar functions in the DF API that do practically the same thing, allowing for .functional().chaining()

Still Catalyst Optimized

Both SQL and API Functions in DF's sit atop Catalyst

4

Schemas

Inferred
vs
explicit

4

Schemas

Inferred
vs
explicit

Schemas can be inferred, i.e. guessed, by spark. With inferred schemas, you usually end up with a bunch of strings and ints. If you have more specific needs, supply your own schema.

4

Schemas

Inferred
vs
explicit

Schemas can be inferred, i.e. guessed, by spark. With inferred schemas, you usually end up with a bunch of strings and ints. If you have more specific needs, supply your own schema.

```
# sample data - “people.txt”
1|Kristian|Algebraix Data|San Diego|CA
2|Pat|Algebraix Data|San Diego|CA
3|Lebron|Cleveland Cavaliers|Cleveland|OH
4|Brad|Self Employed|Hollywood|CA
```

4

Schemas

Inferred
vs
explicit

With inferred schemas, you usually end up with a bunch of strings and ints. If you have more specific needs, supply your own schema.

```
# sample data - "people.txt"
1|Kristian|Algebraix Data|San Diego|CA
2|Pat|Algebraix Data|San Diego|CA
3|Lebron|Cleveland Cavaliers|Cleveland|OH
4|Brad|Self Employed|Hollywood|CA
```

```
# load as RDD and map it to a row with multiple fields
rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=s[0],name=s[1],company=s[2],state=s[4])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD)

# full syntax: .createDataFrame(peopleRDD, schema)
```

```
3|Lebron|Cleveland Cavaliers|Cleveland|OH
4|Brad|Self Employed|Hollywood|CA
```

4

Schemas

Inferred
vs
explicit

```
# load as RDD and map it to a row with multiple fields
rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=s[0],name=s[1],company=s[2],state=s[4])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD)

# full syntax: .createDataFrame(peopleRDD, schema)
```

```
# we didn't actually pass anything into that 2nd param.
# yet, behind the scenes, there's still a schema.
>>> peopleDF.printSchema()
Root
 |-- company: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- state: string (nullable = true)
```

4

Schemas

Inferred
vs
explicit

```
def mapper(line):
  s = line.split("|")
  return Row(id=s[0],name=s[1],company=s[2],state=s[4])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD)

# full syntax: .createDataFrame(peopleRDD, schema)
```

```
# we didn't actually pass anything into that 2nd param.
# yet, behind the scenes, there's still a schema.
>>> peopleDF.printSchema()
Root
 |-- company: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name: string (nullable = true)
 |-- state: string (nullable = true)
```

Spark SQL can certainly handle queries where `id` is a string, but it should be an int.

4

Schemas

Inferred
vs
explicit

Spark SQL can certainly handle queries where `id` is a string, but what if we don't want it to be?

```
# load as RDD and map it to a row with multiple fields
rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=int(s[0]),name=s[1],company=s[2],state=s[4])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD)
>>> peopleDF.printSchema()
Root
 |-- company: string (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- state: string (nullable = true)
```

4

Schemas

Inferred
vs
explicit

```
# load as RDD and map it to a row with multiple fields
rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=int(s[0]),name=s[1],company=s[2],state=s[4])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD)
>>> peopleDF.printSchema()
Root
 |-- company: string (nullable = true)
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- state: string (nullable = true)
```

You can actually provide a schema, too, which will be more authoritative.

4

Schemas

Inferred
vs
explicit

You can actually provide a schema, too, which will be more authoritative.

```
# load as RDD and map it to a row with multiple fields
import pyspark.sql.types as types

rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=s[0],name=s[1],company=s[2],state=s[4])

schema = types.StructType([
   types.StructField('id',types.IntegerType(), False)
  ,types.StructField('name',types.StringType())
  ,types.StructField('company',types.StringType())
  ,types.StructField('state',types.StringType())
])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD, schema)
```

4

Schemas

Inferred
vs
explicit

```
rdd = sc.textFile("file:/people.txt")
def mapper(line):
  s = line.split("|")
  return Row(id=s[0],name=s[1],company=s[2],state=s[4])

schema = types.StructType([
   types.StructField('id',types.IntegerType(), False)
  ,types.StructField('name',types.StringType())
  ,types.StructField('company',types.StringType())
  ,types.StructField('state',types.StringType())
])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD, schema)
```

```
>>> peopleDF.printSchema()
Root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- company: string (nullable = true)
 |-- state: string (nullable = true)
```

4

Schemas

Inferred
vs
explicit

```
    return Row(id=s[0],name=s[1],company=s[2],state=s[4])

schema = types.StructType([
    types.StructField('id',types.IntegerType(), False)
   ,types.StructField('name',types.StringType())
   ,types.StructField('company',types.StringType())
   ,types.StructField('state',types.StringType())
])

peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD, schema)
```

```
>>> peopleDF.printSchema()
Root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- company: string (nullable = true)
 |-- state: string (nullable = true)
```

And what are the available types?

4

Schemas

available types

```
peopleRDD = rdd.map(mapper)
peopleDF = spark.createDataFrame(peopleRDD, schema)
```

```
>>> peopleDF.printSchema()
Root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = true)
 |-- company: string (nullable = true)
 |-- state: string (nullable = true)
```

And what are the available types?

```
# http://spark.apache.org/docs/2.0.0/api/python/_modules/pyspark/sql/types.html
__all__ = ["DataType", "NullType", "StringType",
"BinaryType", "BooleanType", "DateType", "TimestampType",
"DecimalType", "DoubleType", "FloatType", "ByteType",
"IntegerType", "LongType", "ShortType", "ArrayType",
"MapType", "StructField", "StructType"]
```

4

Schemas

available types

And what are the available types?

```
# http://spark.apache.org/docs/2.0.0/api/python/_modules/pyspark/sql/types.html
__all__ = ["DataType", "NullType", "StringType",
"BinaryType", "BooleanType", "DateType", "TimestampType",
"DecimalType", "DoubleType", "FloatType", "ByteType",
"IntegerType", "LongType", "ShortType", "ArrayType",
"MapType", "StructField", "StructType"]
```

Gotcha alert Spark doesn't seem to care when you leave dates as strings.

```
# Spark SQL handles this just fine as if they were
# legit date objects.

spark.sql(“””
  SELECT * FROM NewHires n WHERE n.start_date > “2016-01-01”
“””).show()
```

@KrisDevelops

algebraixdata

4

Schemas

available types

```
"BinaryType", "BooleanType", "DateType", "TimestampType",
"DecimalType", "DoubleType", "FloatType", "ByteType",
"IntegerType", "LongType", "ShortType", "ArrayType",
"MapType", "StructField", "StructType"]
```

****Gotcha alert**** Spark doesn’t seem to care when you leave dates as strings.

```
# Spark SQL handles this just fine as if they were
# legit date objects.

spark.sql(“””
  SELECT * FROM NewHires n WHERE n.start_date > “2016-01-01”
“””).show()
```

Now you know about inferred and explicit schemas, and the available types you can use.

4

Schemas

available types

Gotcha alert Spark doesn't seem to care when you leave dates as strings.

```
# Spark SQL handles this just fine as if they were
# legit date objects.

spark.sql(“””
  SELECT * FROM NewHires n WHERE n.start_date > “2016-01-01”
“””).show()
```

Now you know about inferred and explicit schemas, and the available types you can use.

@KrisDevelops

algebraix data

5

Loading & Saving Results

Types,
performance,
& considerations

5

Loading & Saving Results

Types, performance, & considerations

Loading and Saving is fairly straight forward. Save your dataframes in your desired format.

5

Loading & Saving Results

Types, performance, & considerations

Loading and Saving is fairly straight forward. Save your dataframes in your desired format.

```
# picking up where we left off
peopleDF = spark.createDataFrame(peopleRDD, schema)

peopleDF.write.save("s3://acme-co/people.parquet",
  format="parquet") # format= defaults to parquet if omitted

# formats: json, parquet, jdbc, orc, libsvm, csv, text
```

5

Loading & Saving Results

Types, performance, & considerations

Loading and Saving is fairly straight forward. Save your dataframes in your desired format.

```
# picking up where we left off
peopleDF = spark.createDataFrame(peopleRDD, schema)

peopleDF.write.save("s3://acme-co/people.parquet",
  format="parquet") # format= defaults to parquet if omitted

# formats: json, parquet, jdbc, orc, libsvm, csv, text
```

When you read, some types preserve schema. Parquet keeps the full schema, JSON has inferrable schema, and JDBC pulls in schema.

5

Loading & Saving Results

Types, performance, & considerations

```
# picking up where we left off
peopleDF = spark.createDataFrame(peopleRDD, schema)

peopleDF.write.save("s3://acme-co/people.parquet",
  format="parquet") # format= defaults to parquet if omitted

# formats: json, parquet, jdbc, orc, libsvm, csv, text
```

When you read, some types preserve schema. Parquet keeps the full schema, JSON has inferrable schema, and JDBC pulls in schema.

```
# read from stored parquet
peopleDF = spark.read.parquet(“s3://acme-co/people.parquet”)

# read from stored JSON
peopleDF = spark.read.json(“s3://acme-co/people.json”)
```

```
  format="parquet") # format= defaults to parquet if omitted

# formats: json, parquet, jdbc, orc, libsvm, csv, text
```

5

Loading & Saving Results

Types, performance, & considerations

When you read, some types preserve schema. Parquet keeps the full schema, JSON has inferrable schema, and JDBC pulls in schema.

```
# read from stored parquet
peopleDF = spark.read.parquet("s3://acme-co/people.parquet")

# read from stored JSON
peopleDF = spark.read.json("s3://acme-co/people.json")
```

6

SQL Function Coverage

What works and what doesn't

6

SQL Function Coverage

Spark 1.6

Spark 1.6

- Limited support for subqueries and various other noticeable SQL functionalities
- Runs roughly half of the 99 TPC-DS benchmark queries
- More SQL support in HiveContext

http://www.slideshare.net/databricks/apache-spark-20-faster-easier-and-smarter

6

SQL Function Coverage

Spark 2.0

Spark 2.0

In DataBricks' Words

- SQL2003 support
- Runs all 99 of TPC-DS benchmark queries
- A native SQL parser that supports both ANSI-SQL as well as Hive QL
- Native DDL command implementations
- Subquery support, including
  - Uncorrelated Scalar Subqueries
  - Correlated Scalar Subqueries
  - NOT IN predicate Subqueries (in WHERE/HAVING clauses)
  - IN predicate subqueries (in WHERE/HAVING clauses)
  - (NOT) EXISTS predicate subqueries (in WHERE/HAVING clauses)
- View canonicalization support
- In addition, when building without Hive support, Spark SQL should have almost all the functionality as when building with Hive support, with the exception of Hive connectivity, Hive UDFs, and script transforms.

http://www.slideshare.net/databricks/apache-spark-20-faster-easier-and-smarter

6

SQL Function Coverage

Spark 2.0

Spark 2.0

In DataBricks' Words

- SQL2003 support
- Runs all 99 of TPC-DS benchmark queries
- A native SQL parser that supports both ANSI-SQL as well as Hive QL
- Native DDL command implementations
- Subquery support, including
  - Uncorrelated Scalar Subqueries
  - Correlated Scalar Subqueries
  - NOT IN predicate Subqueries (in WHERE/HAVING clauses)
  - IN predicate subqueries (in WHERE/HAVING clauses)
  - (NOT) EXISTS predicate subqueries (in WHERE/HAVING clauses)
- View canonicalization support
- In addition, when building without Hive support, Spark SQL should have almost all the functionality as when building with Hive support, with the exception of Hive connectivity, Hive UDFs, and script transforms.

7

Using Spark SQL for ETL

Tips and tricks

7

Using Spark SQL for ETL

Tips and tricks

These things are some of the things we learned to start doing after working with Spark a while.

7

Using Spark SQL for ETL

Tips and tricks

These things are some of the things we learned to start doing after working with Spark a while.

Tip 1: In production, break your applications into smaller apps as steps. I.e. “Pipeline pattern”

7

Using Spark SQL for ETL

Tips and tricks

These things are some of the things we learned to start doing after working with Spark a while.

Tip 1: In production, break your applications into smaller apps as steps. I.e. “Pipeline pattern”

Tip 2: When tinkering locally, save a small version of the dataset via spark and test against that.

7

Using Spark SQL for ETL

Tips and tricks

These things are some of the things we learned to start doing after working with Spark a while.

Tip 1: In production, break your applications into smaller apps as steps. I.e. “Pipeline pattern”

Tip 2: When tinkering locally, save a small version of the dataset via spark and test against that.

Tip 3: If using EMR, create a cluster with your desired steps, prove it works, then export a CLI command to reproduce it, and run it in Data Pipeline to start recurring pipelines / jobs.

7

Using Spark SQL for ETL

Tips and tricks

Tip 1: In production, break your applications into smaller apps as steps. I.e. “Pipeline pattern”

Tip 2: When tinkering locally, save a small version of the dataset via spark and test against that.

Tip 3: If using EMR, create a cluster with your desired steps, prove it works, then export a CLI command to reproduce it, and run it in Data Pipeline to start recurring pipelines / jobs.

8

Working with JSON

Quick and easy

8

Working with JSON

Quick and easy

JSON data is most easily read-in as line delimied json objects*

```
{“n”:”sarah”,”age”:29}
{“n”:”steve”,”age”:45}
```

8

Working with JSON

Quick and easy

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

8

Working with JSON

Quick and easy

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

```
# json explode example

>>> spark.read.json("file:/json.explode.json").createOrReplaceTempView("json")
>>> spark.sql("SELECT * FROM json").show()
+----+----------------+
|   x|               y|
+----+----------------+
|row1| [1, 2, 3, 4, 5]|
|row2|[6, 7, 8, 9, 10]|
+----+----------------+

>>> spark.sql("SELECT x, explode(y) FROM json").show()
+----+---+
|   x|col|
+----+---+
|row1|  1|
|row1|  2|
|row1|  3|
|row1|  4|
|row1|  5|
|row2|  6|
|row2|  7|
|row2|  8|
|row2|  9|
|row2| 10|
+----+---+
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

JSON data is most easily read-in as line delimied json objects*

```
{“n”:”sarah”,”age”:29}
{“n”:”steve”,”age”:45}
```

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Access nested-objects with dot syntax

```
SELECT
    field.subfield
FROM json
```

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Access nested-objects with dot syntax

```
SELECT
    field.subfield
FROM json
```

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

For multi-line JSON files,
you've got to do much more:

```
# a list of data from files.
files = sc.wholeTextFiles("data.json")

# each tuple is (path, jsonData)
rawJSON = files.map(lambda x: x[1])

# sanitize the data
cleanJSON = rawJSON.map(\
  lambda x: re.sub(r"\s+", "",x,flags=re.UNICODE)\
)

# finally, you can then read that in as "JSON"
spark.read.json( scrubbedJSON )

# PS -- the same goes for XML.
```

If you want
flatten
JSON
th

Schema is
ferred upon
d. Unlike
lazy
s, this
some
e

n
wit

```
SELECT
    field.s
FROM json
```

ays
array
ax

```
l[1], col[3]
ROM json
```

@KrisDevelops

algebraixdata

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Access nested-objects with dot syntax

```
SELECT
    field.subfield
FROM json
```

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

JSON data is most easily read-in as line delimied json objects*

```
{"n":"sarah","age":29}
{"n":"steve","age":45}
```

Access nested-objects with dot syntax

```
SELECT
    field.subfield
FROM json
```

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

8

Working with JSON

Quick and easy

If you want to flatten your JSON data, use the `explode` method

(works in both DF API and SQL)

JSON data is most easily read-in as line delimied json objects*

```
{“n”:”sarah”,”age”:29}
{“n”:”steve”,”age”:45}
```

Schema is inferred upon load. Unlike other lazy operations, this **will** cause some work to be done.

Access nested-objects with dot syntax

```
SELECT
    field.subfield
FROM json
```

Access arrays with inline array syntax

```
SELECT
  col[1], col[3]
FROM json
```

@KrisDevelops

algebraixdata

9

External Databases

Reading and Writing

9

External Databases

Reading and Writing

To read from an external database, you've got to have your JDBC connectors (jar) handy. In order to pass a jar package into spark, you'd use the --jars flag when starting pyspark or spark-submit.

9

External Databases

Reading and Writing

To read from an external database, you've got to have your JDBC connectors (jar) handy. In order to pass a jar package into spark, you'd use the --jars flag when starting pyspark or spark-submit.

```
# loading data into an RDD in Spark 2.0
my-linux-shell$ pyspark \
  --jars /path/to/mysql-jdbc.jar\
  --packages

# note: you can also add the path to your jar in the
spark.defaults config file to these settings:

  spark.driver.extraClassPath
  spark.executor.extraClassPath
```

have your JDBC connectors (jar) handy. In order to pass a jar package into spark, you'd use the --jars flag when starting pyspark or spark-submit.

```
# loading data into an RDD in Spark 2.0
my-linux-shell$ pyspark \
  --jars /path/to/mysql-jdbc.jar\
  --packages

# note: you can also add the path to your jar in the
spark.defaults config file to these settings:

  spark.driver.extraClassPath
  spark.executor.extraClassPath
```

Once you've got your connector jars successfully imported, now you can read an existing database into your spark application or spark shell as a dataframe.

9

External Databases

Reading and Writing

9

```
spark.driver.extraClassPath
spark.executor.extraClassPath
```

External Databases

Reading and Writing

Once you've got your connector jars successfully imported, now you can read an existing database into your spark application or spark shell as a dataframe.

```
# line broken for readibility
sqlURL = “jdbc:mysql://<db-host>:<port>
  ?user=<user>
  &password=<pass>
  &rewriteBatchedStatements=true
  &continueBatchOnError=true”

df = spark.read.jdbc(url=sqlURL, table="<db>.<table>")

df.createOrReplaceTempView(“myDB”)
spark.sql(“SELECT * FROM myDB”).show()
```

dataframe.

9

External Databases

Reading and Writing

```
# line broken for readibility
sqlURL = “jdbc:mysql://<db-host>:<port>
  ?user=<user>
  &password=<pass>
  &rewriteBatchedStatements=true   # omg use this.
  &continueBatchOnError=true”      # increases performance.

df = spark.read.jdbc(url=sqlURL, table="<db>.<table>")

df.createOrReplaceTempView(“myDB”)
spark.sql(“SELECT * FROM myDB”).show()
```

If you’ve done some work and built created or manipulated a dataframe, you can write it to a database by using the `spark.read.jdbc` method. Be prepared, it can a while.

9

External Databases

Reading and Writing

```
df = spark.read.jdbc(url=sqlURL, table="<db>.<table>")

df.createOrReplaceTempView(“myDB”)
spark.sql(“SELECT * FROM myDB”).show()
```

If you’ve done some work and built created or manipulated a dataframe, you can write it to a database by using the `spark.read.jdbc` method. Be prepared, it can a while.

Also, be warned, save modes in spark can be a bit destructive. “Overwrite” doesn’t just overwrite your data, it overwrites your schemas too.

Say goodbye to those precious indices.

9

External Databases

Reading and Writing

If you’ve done some work and built created or manipulated a dataframe, you can write it to a database by using the `spark.read.jdbc` method. Be prepared, it can a while.

Also, be warned, save modes in spark can be a bit destructive. “Overwrite” doesn’t just overwrite your data, it overwrites your schemas too.

Say goodbye to those precious indices.

Real World Testing

Things to consider

10

Real World Testing

Things to consider

If testing locally, do not load data from S3 or other similar types of cloud storage.

10

Real World Testing

Things to consider

Construct your applications as much as you can in advance. Cloudy clusters are expensive.

If testing locally, do not load data from S3 or other similar types of cloud storage.

10

Real World Testing

Things to consider

Construct your applications as much as you can in advance. Cloudy clusters are expensive.

If testing locally, do not load data from S3 or other similar types of cloud storage.

In the cloud, you can test a lot of your code reliably with a 1-node cluster.

10

Real World Testing

Things to consider

Construct your applications as much as you can in advance. Cloudy clusters are expensive.

Get really comfortable using .parallelize() to create dummy data.

If testing locally, do not load data from S3 or other similar types of cloud storage.

In the cloud, you can test a lot of your code reliably with a 1-node cluster.

10

Real World Testing

Things to consider

Construct your applications as much as you can in advance. Cloudy clusters are expensive.

Get really comfortable using .parallelize() to create dummy data.

If testing locally, do not load data from S3 or other similar types of cloud storage.

In the cloud, you can test a lot of your code reliably with a 1-node cluster.

If you're using big data, and many nodes, don't use .collect() unless you intend to

Final Questions?

AMA

Thank you!

Bonus

AQA Demo

Hiring

Big Data Engineer