Introduction to Apache Spark
This Lecture
Course Objectives and Prerequisites
What is Apache Spark?
Where Big Data Comes From?
The Structure Spectrum
Apache Spark and DataFrames
Transformations and Actions
Course Objectives

Experiment with use cases for **Apache Spark**
» Extract-Transform-Load operations, data analytics and visualization

Understand Apache Spark’s history and development

Understand the conceptual model: **DataFrames** & **SparkSQL**

Know Apache Spark essentials
» Transformations, actions, **pySpark**, **SparkSQL**
» Basic debugging of Apache Spark programs
» Where to find answers to Spark questions
Course Prerequisites

Basic programming skills and experience

Some experience with Python 2.7
» Take this Python mini-course to learn Python quickly

Google Chrome web browser
» Internet Explorer, Edge, Safari are not supported
What is Apache Spark?

Scalable, efficient analysis of Big Data
What is Apache Spark?

Scalable, efficient analysis of Big Data

This lecture
What is Apache Spark?

*Scalable, efficient analysis of Big Data*

Next lecture
What is Apache Spark?

Scalable, efficient analysis of Big Data
Where Does Big Data Come From?

It’s all happening online – could record every:

» Click
» Ad impression
» Billing event
» Fast Forward, pause,…
» Server request
» Transaction
» Network message
» Fault
» …
Where Does Big Data Come From?

User Generated Content (Web & Mobile)
» Facebook
» Instagram
» Yelp
» TripAdvisor
» Twitter
» YouTube
» …
Where Does Big Data Come From?

Health and Scientific Computing

Images:
- http://www.economist.com/node/16349358
Graph Data

Lots of interesting data has a graph structure:
• Social networks
• Telecommunication Networks
• Computer Networks
• Road networks
• Collaborations/Relationships
• …

Some of these graphs can get quite large (e.g., Facebook user graph)
Log Files – Apache Web Server Log

uplherc.upl.com - - [01/Aug/1995:00:00:07 +0400] "GET / HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 +0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 +0400] "GET /images/MOSAIC-logosmall.gif HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 +0400] "GET /images/USA-logosmall.gif HTTP/1.0" 304 0
ix-esc-ca2-07.ix.netcom.com - - [01/Aug/1995:00:00:09 +0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1713
uplherc.upl.com - - [01/Aug/1995:00:00:10 +0400] "GET /images/WORLD-logosmall.gif HTTP/1.0" 304 0
slppp6.intermind.net - - [01/Aug/1995:00:00:10 -0400] "GET /history/skylab/skylab.html HTTP/1.0" 200 1687
piweba4y.prodigy.com - - [01/Aug/1995:00:00:10 -0400] "GET /images/launchmedium.gif HTTP/1.0" 200 11853
dhcp-47-129:CS100_1> syslog -w 10
unexpected field ID 23 with type 8. Skipping.
unexpected field ID 17 with type 12. Skipping.
Feb 3 15:18:11 dhcp-47-129 Evernote[1140] <Warning>: -
[EDAMAuthenticationResult read]: unexpected field ID 6 with type 11. Skipping.
Feb 3 15:18:11 dhcp-47-129 Evernote[1140] <Warning>: -
[EDAMAuthenticationResult read]: unexpected field ID 7 with type 11. Skipping.
unexpected field ID 19 with type 8. Skipping.
unexpected field ID 23 with type 8. Skipping.
unexpected field ID 17 with type 12. Skipping.
unexpected field ID 5 with type 10. Skipping.
needed for volume Macintosh HD (/) with 18.9 <= 20.0 pct free space
Redwood tree humidity and temperature at various heights
Internet of Things: RFID tags

California FasTrak Electronic Toll Collection transponder

Used to pay tolls

Collected data also used for traffic reporting

» http://www.511.org/

http://en.wikipedia.org/wiki/FasTrak
Key Data Management Concepts

A *data model* is a collection of concepts for describing data.

A *schema* is a description of a particular collection of data, using a given data model.
The Structure Spectrum

Structured (schema-first)
- Relational Database
- Formatted Messages

Semi-Structured (schema-later)
- Documents
- XML
- Tagged Text/Media

Unstructured (schema-never)
- Plain Text
- Media
Semi-Structured Tabular Data

One of the most common data formats

A **table** is a collection of **rows** and **columns**

Each column has a **name**

Each cell may or may not have a **value**
Semi-Structured Data

Each column has a type (string, integer, …)
  » Together, the column types are the schema for the data

Two choices for how the schema is determined:
  » Spark dynamically infers the schema while reading each row
  » Programmer statically specifies the schema
# Tabular Data Example

Fortune 500 companies

» Top 500 US closely held and public corporations by gross revenue

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rank</td>
<td>company</td>
<td>cik</td>
<td>ticker</td>
<td>sic</td>
<td>state_location</td>
<td>state_of_incorporation</td>
<td>revenues</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Wal-Mart Stores</td>
<td>104169</td>
<td>WMT</td>
<td>5331</td>
<td>AR</td>
<td>DE</td>
<td>421849</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Exxon Mobil</td>
<td>34088</td>
<td>XOM</td>
<td>2811</td>
<td>TX</td>
<td>NJ</td>
<td>354674</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Chevron</td>
<td>93410</td>
<td>CVX</td>
<td>2911</td>
<td>CA</td>
<td>DE</td>
<td>196337</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>ConocoPhillips</td>
<td>116318</td>
<td>COP</td>
<td>2911</td>
<td>TX</td>
<td>DE</td>
<td>184986</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>Fannie Mae</td>
<td>310522</td>
<td>FNM</td>
<td>6111</td>
<td>DC</td>
<td>DC</td>
<td>153825</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>General Electric</td>
<td>40545</td>
<td>GE</td>
<td>3600</td>
<td>CT</td>
<td>NY</td>
<td>151628</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Berkshire Hathaway</td>
<td>1067983</td>
<td>BRKA</td>
<td>6331</td>
<td>NE</td>
<td>DE</td>
<td>136185</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>General Motors</td>
<td>1467858</td>
<td>GM</td>
<td>3711</td>
<td>MI</td>
<td>MI</td>
<td>135592</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>Bank of America Corp.</td>
<td>70858</td>
<td>BAC</td>
<td>6021</td>
<td>NC</td>
<td>DE</td>
<td>134194</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>Ford Motor</td>
<td>37996</td>
<td>F</td>
<td>3711</td>
<td>MI</td>
<td></td>
<td>128954</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>Hewlett-Packard</td>
<td>47217</td>
<td>HPQ</td>
<td>3670</td>
<td>CA</td>
<td>DE</td>
<td>126033</td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>AT&amp;T</td>
<td>732717</td>
<td>T</td>
<td>4813</td>
<td>TX</td>
<td>DE</td>
<td>124629</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>J.P. Morgan Chase &amp; Co.</td>
<td>19617</td>
<td>JPM</td>
<td>6021</td>
<td>NY</td>
<td>DE</td>
<td>115475</td>
</tr>
<tr>
<td>15</td>
<td>14</td>
<td>Citigroup</td>
<td>831001</td>
<td>C</td>
<td>6021</td>
<td>NY</td>
<td>DE</td>
<td>111055</td>
</tr>
<tr>
<td>16</td>
<td>15</td>
<td>McKesson</td>
<td>927653</td>
<td>MCK</td>
<td>5122</td>
<td>CA</td>
<td>DE</td>
<td>108702</td>
</tr>
<tr>
<td>17</td>
<td>16</td>
<td>Verizon Communications</td>
<td>732712</td>
<td>VZ</td>
<td>4813</td>
<td>NY</td>
<td>DE</td>
<td>108655</td>
</tr>
<tr>
<td>18</td>
<td>17</td>
<td>American International Group</td>
<td>52722</td>
<td>AIG</td>
<td>6331</td>
<td>NY</td>
<td>DE</td>
<td>104417</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>International Business Machines</td>
<td>51143</td>
<td>IBM</td>
<td>3570</td>
<td>NY</td>
<td>DE</td>
<td>99870</td>
</tr>
<tr>
<td>20</td>
<td>19</td>
<td>Cardinal Health</td>
<td>721371</td>
<td>CAH</td>
<td>5122</td>
<td>OH</td>
<td>CH</td>
<td>98931.9</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>Freddie Mac</td>
<td>37785</td>
<td>FMC</td>
<td>2800</td>
<td>PA</td>
<td>DE</td>
<td>98388</td>
</tr>
</tbody>
</table>

Protein Data Bank

HEADER  APOPTOSIS                               23-DEC-12   3J2T
TITLE     AN IMPROVED MODEL OF THE HUMAN APOPTOSOME
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: APOPTOTIC PROTEASE-ACTIVATING FACTOR 1;
COMPND   3 CHAIN: A, B, C, D, E, F, G;
COMPND   4 SYNONYM: APAF-1;
COMPND   5 ENGINEERED: YES;
COMPND   6 MOL_ID: 2;
COMPND   7 MOLECULE: CYTOCHROME C;
COMPND   8 CHAIN: H, I, J, K, L, M, N
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: HOMO SAPIENS;
SOURCE   3 ORGANISM_COMMON: HUMAN;
SOURCE   4 ORGANISM_TAXID: 9606;
SOURCE   5 GENE: APAF-1, APAF1, KIAA0413;
SOURCE   6 EXPRESSION_SYSTEM: SPODOPTERA FRUGIPERDA;
SOURCE   7 EXPRESSION_SYSTEM_COMMON: FALL ARMYWORM;
KEYWDS    APOPTOSIS PROTEASE ACTIVATING FACTOR-1, APAF-1, CYTOCHROME C,
KEYWDS   2 APOPTOSIS
EXPDTA    ELECTRON MICROSCOPY
AUTHOR    S.YUAN,M.TOPF,C.W.AKEY
REVDAT   2   17-APR-13 3J2T    1       JRNL
REVDAT   1   10-APR-13 3J2T    0

PDB Format:
Field #, Values
Field #, Values
Field #, Values
...

http://www.rcsb.org/pdb/files/3J2T.pdb
Structured Data

A *relational data model* is the most used data model

» *Relation*, a table with rows and columns

Every relation has a *schema* defining each columns’ *type*

The programmer must statically specify the *schema*
Element: Instance of Students Relation

Students(sid: string, name: string, login: string, age: integer, gpa: real)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@eecs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@statistics</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Whither Structured Data?

Conventional Wisdom:
» Only 20% of data is structured

Decreasing due to:
» Consumer applications
» Enterprise search
» Media applications
Unstructured Data

Only one column with string or binary type

Examples:
» Facebook post
» Instagram image
» Vine video
» Blog post
» News article
» User Generated Content
» …
The Structure Spectrum

Structured (schema-first)
- Relational Database
- Formatted Messages

Semi-Structured (schema-later)
- Documents XML
- Tagged Text/Media

Unstructured (schema-never)
- Plain Text
- Media

ETL
- Extract-Transform-Load
  - Impose structure on unstructured data
What is Apache Spark?

Scalable, efficient analysis of Big Data
Some Traditional Analysis Tools

Unix shell commands (grep, awk, sed), pandas, R

All run on a single machine!
What Can You do with Big Data?

Crowdsourcing + Physical modeling + Sensing + Data Assimilation

http://traffic.berkeley.edu
Real World Spark Analysis Use Cases

- Big Data Genomics using ADAM
- Conviva optimizing Internet video stream delivery
- Data processing for wearables and Internet of Things
- Personalized Yahoo! news pages
- Analytics for Yahoo! advertising
- Capital One product recommendations
The Big Data Problem

Data growing faster than computation speeds

Growing data sources
  » Web, mobile, scientific, …

Storage getting cheaper
  » Size doubling every 18 months

But, stalling CPU speeds and storage bottlenecks
Big Data Examples

Facebook’s daily logs: 60 TB

1,000 genomes project: 200 TB

Google web index: 10+ PB

Cost of 1 TB of disk: ~$35

Time to read 1 TB from disk: 3 hours (100 MB/s)
The Big Data Problem

One machine can not process or even store all the data!

Solution is to distribute data over cluster of machines

Lots of hard drives

... and CPUs

... and memory!
Big Data

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 1

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 2

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 3

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>am</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>14</td>
<td>1</td>
</tr>
<tr>
<td>Word</td>
<td>Index</td>
<td>Count</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

Spark DataFrames

Partition 1

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>am</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 2

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sam</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

Partition 3

<table>
<thead>
<tr>
<th>Word</th>
<th>Index</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>am</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>Sam</td>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

Big Data
The Spark Computing Framework

Provides programming abstraction and parallel runtime to hide complexities of fault-tolerance and slow machines

“Here’s an operation, run it on all of the data”
   » I don’t care where it runs (you schedule that)
   » In fact, feel free to run it twice on different nodes
Apache Spark Components

Spark SQL
Spark Streaming
MLlib & ML (machine learning)
GraphX (graph)

Apache Spark
Apache Spark Components

Spark SQL

Spark Streaming

MLlib & ML (machine learning)

GraphX (graph)
Apache Spark References

http://spark.apache.org/docs/latest/programming-guide.html

http://spark.apache.org/docs/latest/api/python/index.html

http://spark.apache.org/docs/latest/api/python/pyspark.sql.html
Python Spark (pySpark)

We are using the Python programming interface to Spark (pySpark)

pySpark provides an easy-to-use programming abstraction and parallel runtime:

» “Here’s an operation, run it on all of the data”

DataFrames are the key concept
Spark Driver and Workers

A Spark program is two programs:
  » A **driver program** and a **workers program**

Worker programs run on cluster nodes or in local threads

DataFrames are distributed across workers
Spark and SQL Contexts

A Spark program first creates a `SparkContext` object

» `SparkContext` tells Spark how and where to access a cluster
» pySpark shell, Databricks CE automatically create `SparkContext`
» `iPython` and programs must create a new `SparkContext`

The program next creates a `sqlContext` object

Use `sqlContext` to create DataFrames

In the labs, we create the `SparkContext` and `sqlContext` for you
Spark Essentials: Master

The **master** parameter for a **SparkContext** determines which type and size of cluster to use.

<table>
<thead>
<tr>
<th>Master Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>local</code></td>
<td>run Spark locally with one worker thread (no parallelism)</td>
</tr>
<tr>
<td><code>local[K]</code></td>
<td>run Spark locally with K worker threads (ideally set to number of cores)</td>
</tr>
<tr>
<td><code>spark://HOST:PORT</code></td>
<td>connect to a Spark standalone cluster; PORT depends on config (7077 by default)</td>
</tr>
<tr>
<td><code>mesos://HOST:PORT</code></td>
<td>connect to a Mesos cluster; PORT depends on config (5050 by default)</td>
</tr>
</tbody>
</table>

In the labs, we set the master parameter for you.
DataFrames

The primary abstraction in Spark
» Immutable once constructed
» Track lineage information to efficiently recompute lost data
» Enable operations on collection of elements in parallel

You construct DataFrames
» by parallelizing existing Python collections (lists)
» by transforming an existing Spark or pandas DFs
» from files in HDFS or any other storage system
DataFrames

Each row of a DataFrame is a Row object

The fields in a Row can be accessed like attributes

```python
>>> row = Row(name="Alice", age=11)
>>> row
Row(age=11, name='Alice')
>>> row['name'], row['age']
('Alice', 11)
>>> row.name, row.age
('Alice', 11)
```
DataFrames

Two types of operations: *transformations* and *actions*

Transformations are lazy (*not computed immediately*)

Transformed DF is executed when action runs on it

Persist (cache) DFs in memory or disk
Working with DataFrames

Create a DataFrame from a data source: 

Apply transformations to a DataFrame: select filter

Apply actions to a DataFrame: show count

show action causes createDataFrame, filter, and select transforms to be executed
Creating DataFrames

Create DataFrames from Python collections (lists)

```python
>>> data = [('Alice', 1), ('Bob', 2)]
>>> data
[('Alice', 1), ('Bob', 2)]
>>> df = sqlContext.createDataFrame(data)
[Row(_1=u'Alice', _2=1), Row(_1=u'Bob', _2=2)]
>>> sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```

No computation occurs with `sqlContext.createDataFrame()`
- Spark only records how to create the DataFrame
pandas: Python Data Analysis Library

Open source data analysis and modeling library
  » An alternative to using R

pandas **DataFrame**: a table with named columns
  » The most commonly used pandas object
  » Represented as a Python **Dict** (column_name → Series)
  » Each pandas **Series** object represents a column
    • 1-D labeled array capable of holding any data type
  » R has a similar **data frame** type
Creating DataFrames

Easy to create pySpark DataFrames from pandas (and R) DataFrames

```python
# Create a Spark DataFrame from Pandas
>>> spark_df = sqlContext.createDataFrame(pandas_df)
```
Creating DataFrames

From HDFS, text files, JSON files, Apache Parquet, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard: /data/201404*

>>> df = sqlContext.read.text("README.txt")

>>> df.collect()

[Row(value=u'hello'), Row(value=u'this')]
Creating a DataFrame from a File

```
distFile = sqlContext.read.text (...)  
```

Loads text file and returns a DataFrame with a single string column named "value"

Each line in text file is a row

*Lazy evaluation* means no execution happens now
Spark Transformations

Create new **DataFrame** from an existing one

Use *lazy evaluation*: results not computed right away – Spark remembers set of transformations applied to base **DataFrame**

» Spark uses **Catalyst** to optimize the required calculations
» Spark recovers from failures and slow workers

*Think of this as a recipe for creating result*
Column Transformations

The apply method creates a DataFrame from one column:

```python
>>> ageCol = people.age
```
Column Transformations

The apply method creates a DataFrame from one column:

```python
>>> ageCol = people.age
```

You can `select` one or more columns from a DataFrame:

```python
>>> df.select('**')
* selects all the columns
```
Column Transformations

The apply method creates a 

```
DataFrame
```

from one column:

```python
>>> ageCol = people.age
```

You can 

```
select
```

one or more columns from a 

```
DataFrame
```

```python
>>> df.select('*')
```

* selects all the columns

```python
>>> df.select('name', 'age')
```

* selects the 

```
name
```

and 

```
age
```

columns
Column Transformations

The apply method creates a DataFrame from one column:

```python
>>> ageCol = people.age
```

You can `select` one or more columns from a DataFrame:

```python
>>> df.select('*')
    * selects all the columns
>>> df.select('name', 'age')
    * selects the name and age columns
>>> df.select(df.name,
                       (df.age + 10).alias('age'))
    * selects the name and age columns,
        increments the values in the age column by 10, 
        and renames (alias) the age +10 column as age
More Column Transformations

The `drop` method returns a new `DataFrame` that drops the specified column:

```python
>>> df.drop(df.age)
[Row(name=u'Alice'), Row(name=u'Bob')]
```
Small anonymous functions (not bound to a name)  
`lambda a, b: a + b`  
» returns the sum of its two arguments

Can use lambda functions wherever function objects are required

Restricted to a single expression
User Defined Function Transformations

Transform a DataFrame using a User Defined Function

```python
>>> from pyspark.sql.types import IntegerType
>>> slen = udf(lambda s: len(s), IntegerType())
>>> df.select(slen(df.name).alias('slen'))

* Creates a DataFrame of [Row(slen=5), Row(slen=3)]
```

UDF takes named or lambda function and the return type of the function
## Other Useful Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filter(func)</code></td>
<td>returns a new DataFrame formed by selecting those rows of the source on which <code>func</code> returns true</td>
</tr>
<tr>
<td><code>where(func)</code></td>
<td><code>where</code> is an alias for <code>filter</code></td>
</tr>
<tr>
<td><code>distinct()</code></td>
<td>return a new DataFrame that contains the distinct rows of the source DataFrame</td>
</tr>
<tr>
<td><code>orderBy(*cols, **kw)</code></td>
<td>returns a new DataFrame sorted by the specified column(s) and in the sort order specified by <code>kw</code></td>
</tr>
<tr>
<td><code>sort(*cols, **kw)</code></td>
<td>Like <code>orderBy</code>, <code>sort</code> returns a new DataFrame sorted by the specified column(s) and in the sort order specified by <code>kw</code></td>
</tr>
<tr>
<td><code>explode(col)</code></td>
<td>returns a new row for each element in the given array or map</td>
</tr>
</tbody>
</table>

`func` is a Python named function or lambda function
Using Transformations (I)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```
Using Transformations (I)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> from pyspark.sql.types import IntegerType
>>> doubled = udf(lambda s: s * 2, IntegerType())
>>> df2 = df.select(df.name, doubled(df.age).alias('age'))
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=4)]

* selects the name and age columns, applies the UDF to age column and aliases resulting column to age
Using Transformations (l)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> from pyspark.sql.types import IntegerType
>>> doubled = udf(lambda s: s * 2, IntegerType())
>>> df2 = df.select(df.name, doubled(df.age).alias('age'))
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=4)]
* selects the name and age columns, applies the UDF to age column and aliases resulting column to age

>>> df3 = df2.filter(df2.age > 3)
[Row(name=u'Bob', age=4)]
* only keeps rows with age column greater than 3
```
Using Transformations (II)

```python
>>> data2 = [('Alice', 1), ('Bob', 2), ('Bob', 2)]
>>> df = sqlContext.createDataFrame(data2, ['name', 'age'])
[Row(name=u'Albert', age=1), Row(name=u'Bob', age=2), Row(name=u'Bob', age=2)]
>>> df2 = df.distinct()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
* only keeps rows that are distinct
```
Using Transformations (II)

```python
>>> data2 = [('Alice', 1), ('Bob', 2), ('Bob', 2)]
>>> df = sqlContext.createDataFrame(data2, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2), Row(name=u'Bob', age=2)]
>>> df2 = df.distinct()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
* only keeps rows that are distinct

>>> df3 = df2.sort("age", ascending=False)
[Row(name=u'Bob', age=2), Row(name=u'Alice', age=1)]
* sort ascending on the age column
```
Using Transformations (III)

```python
>>> data3 = [Row(a=1, intlist=[1, 2, 3])]
>>> df4 = sqlContext.createDataFrame(data3)
[Row(a=1, intlist=[1, 2, 3])]
>>> df4.select(explode(df4.intlist).alias("anInt"))
[Row(anInt=1), Row(anInt=2), Row(anInt=3)]

* turn each element of the intlist column into a Row, alias the resulting column to anInt, and select only that column
```
**GroupedData Transformations**

`groupBy(*cols)` groups the DataFrame using the specified columns, so we can run aggregation on them.

<table>
<thead>
<tr>
<th>GroupedData Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>agg(*exprs)</code></td>
<td>Compute aggregates (avg, max, min, sum, or count) and returns the result as a DataFrame</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>counts the number of records for each group</td>
</tr>
<tr>
<td><code>avg(*args)</code></td>
<td>computes average values for numeric columns for each group</td>
</tr>
</tbody>
</table>
Using GroupedData (I)

```python
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df1 = df.groupBy(df.name)
>>> df1.agg({'*': 'count'}).collect()
[Row(name=u'Alice', count(1)=2), Row(name=u'Bob', count(1)=2)]
```
Using GroupedData (I)

```python
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df1 = df.groupBy(df.name)
>>> df1.agg({'*': 'count'}).collect()
[Row(name=u'Alice', count(1)=2), Row(name=u'Bob', count(1)=2)]

>>> df.groupBy(df.name).count()
[Row(name=u'Alice', count=2), Row(name=u'Bob', count=2)]
```
Using GroupedData (II)

```python
>>> data = [('Alice',1,6), ('Bob',2,8), ('Alice',3,9), ('Bob',4,7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df.groupBy().avg().collect()
[Row(avg(age)=2.5, avg(grade)=7.5)]
```
Using GroupedData (II)

```python
>>> data = [('Alice', 1, 6), ('Bob', 2, 8), ('Alice', 3, 9), ('Bob', 4, 7)]
>>> df = sqlContext.createDataFrame(data, ['name', 'age', 'grade'])
>>> df.groupBy().avg().collect()
[Row(avg(age)=2.5, avg(grade)=7.5)]

>>> df.groupBy('name').avg('age', 'grade').collect()
[Row(name=u'Alice', avg(age)=2.0, avg(grade)=7.5),
  Row(name=u'Bob', avg(age)=3.0, avg(grade)=7.5)]
```
Transforming a DataFrame

```python
linesDF = sqlContext.read.text('...')

commentsDF = linesDF.filter(isComment)
```

Lazy evaluation means nothing executes – Spark saves recipe for transforming source
Spark Actions

Cause Spark to execute recipe to transform source

Mechanism for getting results out of Spark
### Some Useful Actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>show(n, truncate)</code></td>
<td>prints the first $n$ rows of the DataFrame</td>
</tr>
<tr>
<td><code>take(n)</code></td>
<td>returns the first $n$ rows as a list of Row</td>
</tr>
<tr>
<td><code>collect()</code></td>
<td>return all the records as a list of Row</td>
</tr>
<tr>
<td></td>
<td><strong>WARNING:</strong> make sure will fit in driver program</td>
</tr>
<tr>
<td><code>count()</code></td>
<td>returns the number of rows in this DataFrame</td>
</tr>
<tr>
<td><code>describe(*cols)</code></td>
<td>Exploratory Data Analysis function that computes statistics (count, mean, stddev, min, max) for numeric columns – if no columns are given, this function computes statistics for all numerical columns</td>
</tr>
</tbody>
</table>

+`count` for **DataFrames** is an action, while for **GroupedData** it is a transformation
Getting Data Out of DataFrames (I)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
```
Getting Data Out of DataFrames (I)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> df.show()
+-----+-----+
| name| age |
+-----+-----+
| Alice| 1   |
|   Bob| 2   |
+-----+-----+
```
Getting Data Out of DataFrames (I)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.collect()
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]

>>> df.show()
+-------+
| name | age |
+-------+
| Alice | 1   |
|   Bob | 2   |
+-------+

>>> df.count()
2
```
Getting Data Out of DataFrames (II)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.take(1)
[Row(name=u'Alice', age=1)]
```
Getting Data Out of DataFrames (II)

```python
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
>>> df.take(1)
[Row(name='u'Alice', age=1)]

>>> df.describe()

+---------+-----+
| summary | age |
|---------+-----|
| count   | 2   |
| mean    | 1.5 |
| stddev  | 0.7071067811865476 |
| min     | 1   |
| max     | 2   |
```
Spark Programming Model

```python
linesDF += sqlContext.read.text('...')

print linesDF.count()
```

`count()` causes Spark to:
- read data
- sum within partitions
- combine sums in driver
Spark Programming Model

```python
linesDF = sqlContext.read.text('...')
commentsDF = linesDF.filter(isComment)
print linesDF.count(), commentsDF.count()
```

Spark recomputes linesDF:
- read data (again)
- sum within partitions
- combine sums in driver
Caching DataFrames

```
linesDF = sqlContext.read.text('...')
linesDF.cache()  # save, don't recompute!
commentsDF = linesDF.filter(isComment)
print linesDF.count(), commentsDF.count()
```
Spark Program Lifecycle

1. Create DataFrames from external data or `createDataFrame` from a collection in driver program

2. Lazily `transform` them into new DataFrames

3. `cache()` some DataFrames for reuse

4. Perform `actions` to execute parallel computation and produce results
Local or Distributed?

Where does code run?
» Locally, in the driver
» Distributed at the executors
» Both at the driver and the executors

Very important question:
» Executors run in parallel
» Executors have much more memory
Where Code Runs

Most Python code runs in driver
  » Except for code passed to transformations

Transformations run at executors

Actions run at executors and driver
Examples

>>> a = a + 1

Your application (driver program)

Worker
Spark executor

Worker
Spark executor
Examples

```python
>>> a = a + 1
```

```python
>>> linesDF.filter(isComment)
```

Your application (driver program)

Worker Spark executor

Worker Spark executor
Examples

```python
>>> a = a + 1
>>> linesDF.filter(isComment)
>>> commentsDF.count()
```
How Not to Write Code

Let’s say you want to combine two DataFrames: `aDF, bDF`

You remember that `df.collect()` returns a list of `Row`, and in Python you can combine two lists with `+`

A naïve implementation would be:

```python
>>> a = aDF.collect()
>>> b = bDF.collect()
>>> cDF = sqlContext.createDataFrame(a + b)
```

Where does this code run?
\[ a + b \]

```python
>>> a = aDF.collect()
>>> b = bDF.collect()
```

* all distributed data for \( a \) and \( b \) is sent to driver

What if \( a \) and/or \( b \) is very large?

» Driver could run out of memory: Out Of Memory error (OOM)
» Also, takes a long time to send the data to the driver
What if the list \( a + b \) is very large?

- Driver could run out of memory: Out Of Memory error (OOM)
- Also, takes a long time to send the data to executors
The Best Implementation

```python
>>> cDF = aDF.unionAll(bDF)
```

Use the **DataFrame** reference API!

» `unionAll()`: “Return a new DataFrame containing union of rows in this frame and another frame”

» Runs **completely** at executors:
  - Very scalable and efficient
Some Programming Best Practices

Use Spark Transformations and Actions wherever possible
» Search DataFrame reference API

Never use `collect()` in production, instead use `take(n)`
cache() DataFrames that you reuse a lot