DataFrame abstraction
for distributed data processing

Pelle Jakovits

16 November, 2018, Tartu
Outline

• DataFrame abstraction
• Spark DataFrame API
  – Importing and Exporting data
  – DataFrame and column transformations
  – Advanced DataFrame features
  – User Defined Functions
• Advantages & Disadvantages
DataFrame abstraction

• DataFrame is a tabular format of data
  – Data objects are divided into rows and labelled columns
  – Column data types are fixed

• Simplifies working with tabular datasets
  – Restructuring and manipulating tables
  – Applying user defined functions to a set of columns

• DataFrame implementations
  – Pandas DataFrame in Python
  – DataFrames in R
Spark DataFrames

- Spark DataFrame is a collection of data organized into labelled columns
  - Stored in Resilient Distributed Datasets (RDD)
- Equivalent to a table in a relational DB or DataFrame in R or Python
- Shares built-in & UDF functions with HiveQL and Spark SQL

- Different API from Spark RDD
  - DataFrame API is more column focused
  - Functions are applied on columns rather than row tuples
    - `map(fun) -> select(cols), withColumn(col, fun(col))`
    - `reduceByKey(fun) -> agg(fun(col)), sum(col), count(col)"
Spark DataFrames

- Operations on Spark DataFrames are inherently parallel
  - DataFrame is split by rows into RDD partitions

- Optimized under-the-hood
  - Logical execution plan optimizations
  - Physical code generation and deployment optimizations

- Can be constructed from a wide array of sources
  - Structured data files (json, csv, ...)
  - Tables in Hive
  - Existing Spark RDDs
  - Python Pandas or R DataFrames
  - External relational and non-relational databases
Using Spark DataFrame API

```python
# Load in data as a DataFrame
bank_accounts = spark.read.option("header", True) \
    .option("inferSchema", True) \
    .csv("bank_folder")

# Execute DataFrame operations, result is a DataFrame
result = bank_accounts.select("Balance", "City") \
    .groupBy("City") \
    .sum("Balance")

# Show results
result.show(5, False)

# Store results
result.write.format("json").save("output_folder")
```
Loading DataFrames from files

- DataFrame schema can be generated automatically
- Reading data From JSON file example:

```python
df = spark.read.option("inferSchema", True) \
    .json("/data/people.json")
```

- Reading data From CSV file:

```python
df = spark.read.option("header","true") \
    .option("inferSchema", True) \
    .option("delimiter", ":") \
    .csv("/data/Top_1000_Songs.csv")
```
Creating DataFrame from RDD

• When loading from an existing RDD, we must specify schema separately
• Example: RDD `people`, which contains tuples of (name, age)

```python
schema = StructType([  
    StructField("name", StringType(), True),  
    StructField("age", StringType(), True)])

peopleDF = spark.createDataFrame(people, schema)
```
import numpy as np
import pandas as pd

matrix = np.random.rand(6, 6)
dataframe = pd.DataFrame(matrix)

sparkDF = spark.createDataFrame(dataframe)
Saving DataFrames

• Can save DF's in csv, json, text, binary, etc. format
• You can control how many files are created using:
  – `df.coalesce(N)`
  – It re-structures DF into N partitions
  – Be careful, each DF partition should fit into memory!

```python
df.write
  .format("csv")
  .option("header", True)
  .option("compression", "gzip")
  .save("output_folder")

df.coalesce(1).write
  .format("json")
  .save("output_folder")
```
Save modes

- **Error** - Default option: Throw error if output folder exist
- **Ignore** - Silent ignore if output folder exist
- **Append** - Add new files into output folder
- **Overwrite** - Replace output folder

```python
df.write.mode("append") \ 
  .format("json") \ 
  .save("output_folder")
```
Spark DataFrame DB connectors

• Load DataFrame from PostgreSQL table

```scala
jdbcDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .load()
```

• Store Dataframe into PostgreSQL table

```scala
jdbcDF.write
  .format("jdbc")
  .option("url", "jdbc:postgresql:dbserver")
  .option("dbtable", "schema.tablename")
  .option("user", "username")
  .option("password", "password")
  .save()
```
Manipulating DataFrames

• DataFrame operations
  – Provide information about DataFrame content and structure
  – Transform DataFrame structure
  – Group, select, add, modify columns

• Column Functions
  – Generate or change the content of columns
  – Shares the same column functions with SQL
  – Can add UDF's as new Column functions
Structure of the DataFrame

```scala
bank_accounts.printSchema()

root
|-- Last_Name: string (nullable = true)
|-- First_Name: string (nullable = true)
|-- Balance: double (nullable = true)
|-- Address: string (nullable = true)
|-- City: string (nullable = true)
|-- Last_Trans: string (nullable = true)
|-- bank_name: string (nullable = true)
```
Show / Transform table contents

```sql
def bank_accounts.show()

<table>
<thead>
<tr>
<th>Last_Name</th>
<th>First_Name</th>
<th>Balance</th>
<th>Address</th>
<th>City</th>
<th>Last_Trans</th>
<th>bank_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>KELLY</td>
<td>JUSTIN R</td>
<td>74.5</td>
<td>UNKNOWN,UNKNOWN</td>
<td>02/26/1983</td>
<td>BANK OF NOVA SCOTIA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.88</td>
<td></td>
<td></td>
<td>06/04/1993</td>
<td>TORONTO-DOMINION BANK</td>
</tr>
<tr>
<td>NEED NEWS</td>
<td></td>
<td>787.51</td>
<td>12055 - 95 ST.</td>
<td>Edmonton</td>
<td>04/02/1980</td>
<td>HSBC BANK CANADA</td>
</tr>
<tr>
<td>BIANCHI</td>
<td>BERNARD</td>
<td>357.98</td>
<td></td>
<td></td>
<td>03/29/1995</td>
<td>HSBC BANK CANADA</td>
</tr>
<tr>
<td>CHAN</td>
<td>SUI PANG</td>
<td>102.34</td>
<td></td>
<td></td>
<td>04/17/1990</td>
<td>BANK OF MONTREAL</td>
</tr>
</tbody>
</table>
```

```sql
def bank_accounts.select("Balance", "City")

<table>
<thead>
<tr>
<th>City</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANMORE ALTA</td>
<td>ROYAL BANK OF CANADA</td>
</tr>
<tr>
<td>CHIPMAN</td>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
</tr>
<tr>
<td>EDMONTON, ALBERTA T5</td>
<td>HSBC BANK CANADA</td>
</tr>
<tr>
<td>Edmonton</td>
<td>ING BANK OF CANADA</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
</tr>
</tbody>
</table>
```

DataFrame Example - WordCount

# Load the dataframe content from a text file, Lines DataFrame contains a single column: value - a single line from the text file.
lines = spark.read.text(input_folder)

#Split the value column into words and explode the resulting list into multiple records, Explode and split are column functions
words = lines.select(explode(split(lines.value, " ")).alias("word"))

#group by Word and apply count function
wordCounts = words.groupBy("word").count()

#print out the results
wordCounts.show(10)

+-------------------+-------+
| word              | count |
+-------------------+-------+
| online            | 4     |
| By                | 9     |
| Text-Book         | 1     |
| hope              | 8     |
| some              | 75    |
+-------------------+-------+
Working with columns

• Addressing columns:
  – df.column
  – df['column']
  – F.col("column")
  – "column"

accounts.select( "Balance",
                  accounts.Balance,
                  accounts['Balance'],
                  F.col("Balance") )
Modifying columns

- Rename column
  - `df.col.alias("new_label")`

- Cast column into another type
  - `df.col.cast("string")`
  - `df.col("Balance").cast(StringType())`

```python
accounts.select(accounts.balance.cast("double").alias("bal"))
```
Adding columns

• Add a new column
  – `df2 = df.withColumn('age2', df.age + 2)`
  – If new column label already exists, it is replaced/overwritten

• Rename a column:
  – `df2 = df.withColumnRenamed('age', 'age2')`
Filtering rows

```python
bank_accounts.filter("Last_Trans LIKE '%1980' ")
bank_accounts.filter(bank_accounts.Last_Trans.contains("1980"))
```

<table>
<thead>
<tr>
<th>Last_Name</th>
<th>First_Name</th>
<th>Balance</th>
<th>Address</th>
<th>City</th>
<th>Last_Trans</th>
<th>bank_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEED NEWS</td>
<td></td>
<td>787.51</td>
<td>12055 - 95 ST.</td>
<td>Edmonton</td>
<td>04/02/1980</td>
<td>HSBC BANK CANADA</td>
</tr>
<tr>
<td>BAKER</td>
<td>DAPHNE</td>
<td>93.85</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AKIYAMA</td>
<td>M</td>
<td>5646.64</td>
<td>RC 2-4</td>
<td>UTSUNOMIYA</td>
<td>02/02/1980</td>
<td>BANK OF MONTREAL</td>
</tr>
<tr>
<td>WATSON</td>
<td>RONALD</td>
<td>5199.89</td>
<td>PO STN C</td>
<td>Edmonton</td>
<td>01/09/1980</td>
<td>ROYAL BANK OF CANADA</td>
</tr>
<tr>
<td>LO</td>
<td>ANNIE</td>
<td>4256.07</td>
<td>14208 96 AVENUE</td>
<td>Edmonton</td>
<td>04/18/1980</td>
<td>ROYAL BANK OF CANADA</td>
</tr>
</tbody>
</table>
Grouping DataFrames

```python
class bank_accounts.
    groupBy("City", "bank_name").sum("Balance")

class bank_accounts.
    groupBy("City", "bank_name").agg(F.sum("Balance"))
```

```
<table>
<thead>
<tr>
<th></th>
<th>bank_name</th>
<th>sum(Balance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>YELLOWKNIFE NT</td>
<td>BANK OF MONTREAL</td>
<td>1790.68</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>751.94</td>
</tr>
<tr>
<td>EDMONTON, ALBERTA T5</td>
<td>HSBC BANK CANADA</td>
<td>528.28</td>
</tr>
<tr>
<td>Edmonton</td>
<td>ING BANK OF CANADA</td>
<td>636.42</td>
</tr>
<tr>
<td>CANMORE ALTA</td>
<td>ROYAL BANK OF CAN...</td>
<td>51.37</td>
</tr>
<tr>
<td>CHIPMAN</td>
<td>CANADIAN IMPERIAL...</td>
<td>20.59</td>
</tr>
<tr>
<td>ST. ALBERT AB</td>
<td>HSBC BANK CANADA</td>
<td>83.57</td>
</tr>
</tbody>
</table>
```
Joining DataFrames

- DataFrames can be joined by defining the join expression or join key
- Supports broadcast join
  - One DataFrame is fully read into memory and In-Memory join is performed
  - Wrap one of the tables with `broadcast(df)`
  - When both joined tables are marked, Spark broadcasts smaller table.

```python
df = business.join(review, business.business_id == review.business_id)
df = business.join(review, "business_id")
df = broadcast(business).join(review, "business_id")
```
Window functions

• Allows to modify how aggregation functions are applied inside DataFrames
• Compute nested aggregations without changing the original DataFrame structure
• Process rows in groups while still returning a single value for every input row
• Supports sliding windows and cumulative aggregations
Over(Window)

```sql
bankWind = Window.partitionBy("bank_name")
cityWind = Window.partitionBy("City")
bank_a.select("City", "bank_name", "Balance") \
    .withColumn("bank_sums", F.sum("Balance").over(bankWind)) \
    .withColumn("city_sums", F.sum("Balance").over(cityWind))
```

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>Balance</th>
<th>bank_sums</th>
<th>city_sums</th>
</tr>
</thead>
<tbody>
<tr>
<td>HONG KONG</td>
<td>HSBC BANK CANADA</td>
<td>82.67</td>
<td>477164.0</td>
<td>1147.0</td>
</tr>
<tr>
<td>HONG KONG</td>
<td>ROYAL BANK OF CANADA</td>
<td>1064.79</td>
<td>1341940.0</td>
<td>1147.0</td>
</tr>
<tr>
<td>THORSBY ALTA</td>
<td>ROYAL BANK OF CANADA</td>
<td>177.39</td>
<td>1341940.0</td>
<td>177.0</td>
</tr>
<tr>
<td>IRMA AB</td>
<td>BANK OF MONTREAL</td>
<td>2264.51</td>
<td>1476425.0</td>
<td>2265.0</td>
</tr>
<tr>
<td>RADWAY AB</td>
<td>BANK OF MONTREAL</td>
<td>182.04</td>
<td>1476425.0</td>
<td>182.0</td>
</tr>
<tr>
<td>AIRDRIE AB</td>
<td>BANK OF MONTREAL</td>
<td>397.79</td>
<td>1476425.0</td>
<td>432.0</td>
</tr>
<tr>
<td>AIRDRIE AB</td>
<td>TORONTO-DOMINION BANK</td>
<td>34.35</td>
<td>1154282.0</td>
<td>432.0</td>
</tr>
<tr>
<td>STAR CAN</td>
<td>TORONTO-DOMINION BANK</td>
<td>45.11</td>
<td>1154282.0</td>
<td>45.0</td>
</tr>
</tbody>
</table>
Cumulative aggregation

```scala
bankWind = Window.partitionBy("bank_name").orderBy("year")
bank_a.select("bank_name", "Balance", "year")
.withColumn("cumul_sum", F.sum("Balance").over(bankWin))
```

<table>
<thead>
<tr>
<th>bank_name</th>
<th>Balance</th>
<th>year</th>
<th>cumul_sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>821.07</td>
<td>1935</td>
<td>821.07</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>2572.61</td>
<td>1939</td>
<td>3393.68</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1974.39</td>
<td>1948</td>
<td>5368.07</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1732.65</td>
<td>1960</td>
<td>7100.72</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1954.07</td>
<td>1961</td>
<td>11791.81</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1706.68</td>
<td>1961</td>
<td>11791.81</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1030.34</td>
<td>1961</td>
<td>11791.81</td>
</tr>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1799.0</td>
<td>1965</td>
<td>13590.81</td>
</tr>
</tbody>
</table>
Sliding Window

- **RowsBetween** – Window size based on fixed number of rows

```java
Window.partitionBy("bank_name")
  .orderBy("year")
  .rowsBetween(-2, 2)
```

- **RangeBetween** - Window size based on column values

```java
Window.partitionBy("bank_name")
  .orderBy("year")
  .rangeBetween(-10, 10)
```
TF-IDF with DataFrames

words = lines.select(
    F.explode(F.split("value", "[^a-zA-Z]+")) .alias("word"),
    F.substring_index("file", '/', -1).alias("file")
)

#Extract document name and split lines into words

counts = words.groupBy("word", "file") \
    .agg(F.count("*").alias("n"))

#First WordCount

fileWind = Window.partitionBy("file")
wordWind = Window.partitionBy("word")

withN = counts.withColumn("bigN", F.sum("n").over(fileWind)) \
    .withColumn("m", F.count("*").over(wordWind))

#Compute N and m as new columns

tfidf = withN.withColumn("tfidf", 
    withN['n']/withN['bigN'] * F.log2(D/withN['m'])
)

#Finally compute TF-IDF value
Load Input Documents

```python
lines = spark.read.text("in").withColumn("file", F.input_file_name())
lines.show(10, False)
```

<table>
<thead>
<tr>
<th>value</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Project Gutenberg EBook of Frank Merriwell at Yale, by Burt L. Standish</td>
<td>file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt</td>
</tr>
<tr>
<td>This eBook is for the use of anyone anywhere at no cost and with almost no restrictions whatsoever. You may copy it, give it away or re-use it under the terms of the Project Gutenberg License included with this eBook or online at <a href="http://www.gutenberg.net">www.gutenberg.net</a></td>
<td>file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt</td>
</tr>
<tr>
<td>Title: Frank Merriwell at Yale</td>
<td>file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt</td>
</tr>
</tbody>
</table>
Extract document name and split lines into words

```python
words = lines.select(
    F.explode(F.split("value", "[^a-zA-Z]+")) alias("word"),
    F.substring_index("file", '/', -1) alias("file")
)
```

<table>
<thead>
<tr>
<th>file</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>11115.txt</td>
<td>The</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Project</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Gutenberg</td>
</tr>
<tr>
<td>11115.txt</td>
<td>EBook</td>
</tr>
<tr>
<td>11115.txt</td>
<td>of</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Frank</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Merriwell</td>
</tr>
<tr>
<td>11115.txt</td>
<td>at</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Yale</td>
</tr>
<tr>
<td>11115.txt</td>
<td>by</td>
</tr>
</tbody>
</table>
```
First WordCount

counts = words.groupBy("word", "file")
  .agg(F.count("*").alias("n"))

+-----------------+-----------------+----------+
<table>
<thead>
<tr>
<th>file</th>
<th>word</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>11115.txt</td>
<td>accomplish</td>
<td>4</td>
</tr>
<tr>
<td>11115.txt</td>
<td>will</td>
<td>244</td>
</tr>
<tr>
<td>11115.txt</td>
<td>white</td>
<td>24</td>
</tr>
<tr>
<td>11115.txt</td>
<td>midst</td>
<td>3</td>
</tr>
<tr>
<td>11115.txt</td>
<td>resumed</td>
<td>2</td>
</tr>
<tr>
<td>11115.txt</td>
<td>rubbing</td>
<td>4</td>
</tr>
<tr>
<td>11115.txt</td>
<td>powwow</td>
<td>1</td>
</tr>
<tr>
<td>11115.txt</td>
<td>people</td>
<td>9</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Our</td>
<td>3</td>
</tr>
<tr>
<td>11115.txt</td>
<td>familiar</td>
<td>8</td>
</tr>
</tbody>
</table>

+-----------------+-----------------+----------+
Compute \( N \) and \( m \) as new columns

```scala
fileWind = Window.partitionBy("file")
wordWind = Window.partitionBy("word")

withN = counts.withColumn("bigN", F.sum("n").over(fileWind)) \
    .withColumn("m", F.count("*").over(wordWind))
```

<table>
<thead>
<tr>
<th>file</th>
<th>word</th>
<th>n</th>
<th>bigN</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>11115.txt</td>
<td>By</td>
<td>26</td>
<td>90089</td>
<td>2</td>
</tr>
<tr>
<td>11102.txt</td>
<td>By</td>
<td>12</td>
<td>47979</td>
<td>2</td>
</tr>
<tr>
<td>11102.txt</td>
<td>Cannot</td>
<td>1</td>
<td>47979</td>
<td>1</td>
</tr>
<tr>
<td>11115.txt</td>
<td>Drink</td>
<td>4</td>
<td>90089</td>
<td>1</td>
</tr>
<tr>
<td>11102.txt</td>
<td>Easter</td>
<td>2</td>
<td>47979</td>
<td>1</td>
</tr>
<tr>
<td>11102.txt</td>
<td>Heaven</td>
<td>1</td>
<td>47979</td>
<td>1</td>
</tr>
<tr>
<td>11102.txt</td>
<td>JOHNSON</td>
<td>4</td>
<td>47979</td>
<td>1</td>
</tr>
<tr>
<td>11102.txt</td>
<td>July</td>
<td>25</td>
<td>47979</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Finally compute TF-IDF

tfidf = withN.withColumn(
    "tfidf",
    withN['n']/withN['bigN'] * F.log2(D(withN['m'])))

<table>
<thead>
<tr>
<th>word</th>
<th>file</th>
<th>n</th>
<th>bigN</th>
<th>m</th>
<th>tfidf</th>
</tr>
</thead>
<tbody>
<tr>
<td>By</td>
<td>11115.txt</td>
<td>26</td>
<td>90089</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>By</td>
<td>11102.txt</td>
<td>12</td>
<td>47979</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>Cannot</td>
<td>11102.txt</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084245190604222...</td>
</tr>
<tr>
<td>Drink</td>
<td>11115.txt</td>
<td>4</td>
<td>90089</td>
<td>1</td>
<td>4.440053724650068E-5</td>
</tr>
<tr>
<td>Easter</td>
<td>11102.txt</td>
<td>2</td>
<td>47979</td>
<td>1</td>
<td>4.168490381208445...</td>
</tr>
<tr>
<td>Heaven</td>
<td>11102.txt</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084245190604222...</td>
</tr>
<tr>
<td>July</td>
<td>11102.txt</td>
<td>25</td>
<td>47979</td>
<td>1</td>
<td>5.210612976510557E-4</td>
</tr>
</tbody>
</table>
**Crosstab**

- Crosstab operation creates a frequency table between two DataFrame columns

```python
bank_accounts.crosstab("City", "bank_name")
```

<table>
<thead>
<tr>
<th>City_bank_name</th>
<th>BANK OF MONTREAL</th>
<th>BANK OF NOVA SCOTIA</th>
<th>CITIBANK CANADA</th>
<th>HSBC BANK CANADA</th>
</tr>
</thead>
<tbody>
<tr>
<td>URANIUM CITY SASK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUNDRE ALTA</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GRIMSHAW,AB</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NANAIMO BC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ARLINGTON USA</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MESA,USA</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TOFIELD AB</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TETTENHALL, WOLVE...</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Pivot

- **pivot(col, [fields])** DF into a crosstable with a chosen aggregation function
- Takes an optional list of **fields** to transform into columns, otherwise all possible values of pivot column are transformed into columns

```r
bank_accounts.groupBy("City") \n  .pivot("bank_name", ["BANK OF MONTREAL ", "BANK OF NOVA SCOTIA ", "CITIBANK CANADA "] ) \n  .sum("Balance")
```

<table>
<thead>
<tr>
<th>City</th>
<th>BANK OF MONTREAL</th>
<th>BANK OF NOVA SCOTIA</th>
<th>CITIBANK CANADA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edmonton</td>
<td>775441.37</td>
<td>10147.86</td>
<td>3825.5</td>
</tr>
<tr>
<td>St. Albert</td>
<td>36592.55</td>
<td>1065.36</td>
<td>6.75</td>
</tr>
<tr>
<td>Sherwood Park</td>
<td>29561.52</td>
<td>374.14</td>
<td>6.72</td>
</tr>
<tr>
<td>Stony Plain</td>
<td>20848.49</td>
<td>109.8</td>
<td>null</td>
</tr>
<tr>
<td>Leduc</td>
<td>9509.77</td>
<td>5.57</td>
<td>8.82</td>
</tr>
<tr>
<td>EDMONTON</td>
<td>8515.96</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

34/45
Other functions

- **collect_list(col)**
  - Aggregation function to collect all fields from a column into a list

- **sort_array(col)**
  - Sort array or list inside a column

- **histogram(col, bins)**
  - Computes a histogram of a column using non-uniformly spaced bins.

- **sentences(string str, string lang, string locale)**
  - Tokenizes a string of natural language text into sentences

- **ngrams(sentences, int N, int K, int pf)**
  - Returns the top-k N-grams from a set of tokenized sentences

- **corr(col1, col2)**
  - Returns the Pearson coefficient of correlation of a pair of two numeric columns
User Defined Functions

• Java, Scala, Python, R functions can be used as UDF
• Python functions can be used directly, but must specify their output schema and data types
• Special Pandas DataFrame UDFs

• In SQL:
  – `spark.udf.register("tfidf_udf", tfidf, DoubleType())`
• In DataFrame API:
  – `tfidf_udf = F.udf(tfidf, DoubleType())`
Spark SQL UDF example

```python
# Define Python function

def tfidf(n, bigN, m, D):
    return (float(n)/bigN * math.log(float(D)/m, 2))

# Register function as UDF

tfidf_udf = F.udf(tfidf, DoubleType())

# Call UDF from SQL

tfidf = withN.withColumn(
    "tfidf",
    tfidf_udf(withN["n"], withN["bigN"], withN["m"], D)
)
```
Spark UDF example II

def low3(balances):
    sorted(balances)
    low2 = balances[1] if len(balances) > 1 else None
    low3 = balances[2] if len(balances) > 2 else None
    return (balances[0], low2, low3)

schema = StructType([
    StructField("low1", DoubleType(), True),
    StructField("low2", DoubleType(), True),
    StructField("low3", DoubleType(), True),
])

low3_udf = F.udf(low3, schema)
Spark UDF example II

```python
lows = bank_accounts.groupBy("City", "bank_name")
lows.agg(low3_udf(collect_list("Balance")).alias("balances"))
```

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>balances</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANMORE ALTA</td>
<td>ROYAL BANK OF CANADA</td>
<td>[51.37,,]</td>
</tr>
<tr>
<td>CHIPMAN</td>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>[20.59,,]</td>
</tr>
<tr>
<td>EDMONTON, ALBERTA T5</td>
<td>HSBC BANK CANADA</td>
<td>[528.28,,]</td>
</tr>
<tr>
<td>Edmonton</td>
<td>ING BANK OF CANADA</td>
<td>[291.26, 155.53, 136.17]</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>[751.94,,]</td>
</tr>
</tbody>
</table>

root
|-- City: string (nullable = true)
|-- bank_name: string (nullable = true)
|-- balances: struct (nullable = true)
|  |-- low1: double (nullable = true)
|  |-- low2: double (nullable = true)
|  |-- low3: double (nullable = true)
Selecting nested columns

```r
lows.select("City", "bank_name", "balances.low1", "balances.low2", "balances.low3")
```

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>low1</th>
<th>low2</th>
<th>low3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANMORE ALTA</td>
<td>ROYAL BANK OF CAN...</td>
<td>51.37</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>CHIPMAN</td>
<td>CANADIAN IMPERIAL...</td>
<td>20.59</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>EDMONTON, ALBERTA T5</td>
<td>HSBC BANK CANADA</td>
<td>528.28</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Edmonton</td>
<td>ING BANK OF CANADA</td>
<td>291.26</td>
<td>155.53</td>
<td>136.17</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>751.94</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>YELLOWKNIFE NT</td>
<td>BANK OF MONTREAL</td>
<td>1790.68</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>SHERWOOD PARK,AB</td>
<td>BANK OF NOVA SCOTIA</td>
<td>144.3</td>
<td>130.28</td>
<td>113.27</td>
</tr>
</tbody>
</table>
Performance considerations

- Spark can also cache DataFrames into memory using dataframe.cache()
- Use broadcast(df) for smaller DataFrames
- Avoid nested structures with lots of small objects and pointers
- Instead of using strings for keys, use numeric values as keys
DataFrame vs SQL

• Complex transformations may require very large pure-SQL statements.
  – It is much more step-by-step process with DataFrames
• Internally, Spark uses the same data structures, functions and optimizations for both
• Both can be used interchangeably
• It is up to the user preference, which interface is more convenient
RDD vs DataFrames

• **RDD**
  – When dealing with Raw unstructured data
  – When dealing with tuples of variable length and types
  – Need to apply lower-level transformations
  – Want to optimize on the lower-level

• **DataFrames**
  – When data is structured in a (nested) tabular format
  – Fixed number of columns and fixed column types
  – General data transformation operations (groupBy, withColumn, agg) are enough
  – More information about the data structure/schema gives more opportunity for automatic optimization
Thats All

• Following practice session is
  – Processing data with **Spark DataFrames**

• Next week lecture is
  – Stream Data Processing
  – Real-time vs Batch streaming
  – Spark Streaming (Python)
  – Spark Structured Streaming (DataFrames)