Parallel DataFrame's & Machine Learning

Pelle Jakovits

29 November, 2021, Tartu
Outline

• Recollecting how Spark RDD's work.
• DataFrame abstraction
• Spark DataFrame API
• Parallel Machine Learning with Spark DataFrames
Spark Resilient Distributed Datasets

- Collections of data objects
- Distributed across cluster
- Stored in RAM or Disk
- Immutable/Read-only
- Built through parallel transformations
- Automatically rebuilt on failures

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
DataFrame abstraction

Source: https://www.geeksforgeeks.org/python-pandas-dataframe/
Spark DataFrames

- Stored in Resilient Distributed Datasets (RDD)
  - Operations on Spark DataFrames are inherently parallel
- 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

• 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, ...)
  – Existing Spark RDDs
  – Python Pandas or R DataFrames
  – External relational and non-relational databases
Spark DataFrame partitions

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)
```
From Pandas DataFrame

```python
import numpy as np
import pandas as pd

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")
```
```python
df.coalesce(1).write \
  .format("json") \
  .save("output_folder")
```
Spark DataFrame DB connectors

- Load DataFrame from PostgreSQL table
  
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

  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

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

**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></td>
<td>UNKNOWN,UNKNOWN</td>
<td>02/26/1983</td>
<td>BANK OF NOVA SCOTIA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.88</td>
<td></td>
<td></td>
<td></td>
<td>06/04/1993</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>UNKNOWN AB</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></td>
<td>04/17/1990</td>
</tr>
</tbody>
</table>

**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())`

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

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>sum(Balance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>YELLLOWKNIFE 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>HSBC BANK 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)
```

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

```python
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)

`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))`
Cumulative aggregation

```
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

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

• RangeBetween - Window size based on column values

Window.partitionBy("bank_name")
 .orderBy("year")
 .rangeBetween(-10, 10)
Example: Term Weighting

• Term weights consist of two components
  – Local: how important is the term in this document?
  – Global: how important is the term in the collection?

• Here’s the intuition:
  – Terms that appear often in a document should get high weights
  – Terms that appear in many documents should get low weights

• How do we capture this mathematically?
  – Term frequency (local)
  – Inverse document frequency (global)
\[ w_{i,j} = \text{tf}_{i,j} \cdot \log \frac{N}{n_i} \]

- \( w_{i,j} \): weight assigned to term \( i \) in document \( j \)
- \( \text{tf}_{i,j} \): number of occurrence of term \( i \) in document \( j \)
- \( N \): number of documents in entire collection
- \( n_i \): number of documents with term \( i \)

**TF-IDF**: Term frequency – Inverse Document Frequency
TF-IDF with DataFrames

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

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

fileWind = Window.partitionBy("file")  # Compute N and m as new columns
wordWind = Window.partitionBy("word")
withN = counts.withColumn("bigN", F.sum("n").over(fileWind)) \
    .withColumn("m", F.count("*").over(wordWind))

# Finally compute TF-IDF value
tfidf = withN.withColumn(  "tfidf",
    withN['n']/withN['bigN'] * F.log2(D/withN['m'])
)
```
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/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/pellesparkone/in/11115.txt</td>
</tr>
<tr>
<td>Title: Frank Merriwell at Yale</td>
<td>file:///home/pelle/pellesparkone/in/11115.txt</td>
</tr>
</tbody>
</table>
Extract document name and split lines into words

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"))

+-----------------+-----------------+---+
| file            | word            | n  |
+-----------------+-----------------+---+
| 11115.txt       | accomplish      | 4  |
| 11115.txt       | will            | 244|
| 11115.txt       | white           | 24 |
| 11115.txt       | midst           | 3  |
| 11115.txt       | resumed         | 2  |
| 11115.txt       | rubbing         | 4  |
| 11115.txt       | powwow          | 1  |
| 11115.txt       | people          | 9  |
| 11115.txt       | Our             | 3  |
| 11115.txt       | familiar        | 8  |
+-----------------+-----------------+---+
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']))

+-------------------+-------------------+---+---+-------------------+
| word   | file     | n  | bigN | m     | tfidf             |
+-------------------+-------------------+---+---+-------------------+
| By      | 11115.txt | 26 | 90089| 2     | 0.0               |
| By      | 11102.txt | 12 | 47979| 2     | 0.0               |
| Cannot  | 11102.txt | 1  | 47979| 1     | 2.084245190604222... |
| Drink   | 11115.txt | 4  | 90089| 1     | 4.440053724650068E-5 |
| Easter  | 11102.txt | 2  | 47979| 1     | 4.168490381208445... |
| Heaven  | 11102.txt | 1  | 47979| 1     | 2.084245190604222... |
| July    | 11102.txt | 25 | 47979| 1     | 5.210612976510557E-4 |
+-------------------+-------------------+---+---+-------------------+
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

```python
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>

37
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

```python
tfidf_udf = F.udf(tfidf, DoubleType())
```
Spark UDF example

```python
def top3(balances):
    sorted(balances)
    top2 = balances[1] if len(balances) > 1 else None
    top3 = balances[2] if len(balances) > 2 else None
    return (balances[0], top2, top3)
```

```python
schema = StructType([
    StructField("top1", DoubleType(), True),
    StructField("top2", DoubleType(), True),
    StructField("top3", DoubleType(), True),
])
```

```python
top3_udf = F.udf(top3, schema)
```

#Define Python function

#Define function output data structure

#Register function as UDF
Spark UDF example II

tops = bank_accounts.groupBy("City", "bank_name")
tops.agg(top3_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)
  |-- top1: double (nullable = true)
  |-- top2: double (nullable = true)
  |-- top3: double (nullable = true)
Selecting nested columns

tops.select("City", "bank_name", "balances.top1", "balances.top2", "balances.top3")

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>top1</th>
<th>top2</th>
<th>top3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANMORE ALTA</td>
<td>ROYAL BANK OF CANADA</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>YELLOWSKIFE 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>
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
Spark Machine learning library

- A set of scalable machine learning methods implemented in Spark
- Accessible through both RDD and DataFrame interface
  - DataFrame based ML API `spark.ml` is considered primary
  - RDD-based APIs in the `spark.mllib` package is in maintenance mode
Spark ML-Lib functionality

- **Feature manipulation**
  - feature extraction, transformation, selection, dimensionality reduction

- **Machine learning methods**
  - Regression, classification, clustering, etc.

- **Pipelines**

- **Persistence**
  - Models and Pipelines

- **Utilities**
  - Linear algebra, data formats, UDF’s, etc.
  - Statistical Summarizer, Hypothesis testing, Correlation matrices
Feature manipulation

• **Feature Extractors**
  – TF-IDF, Word2Vec, CountVectorizer, FeatureHasher (hash trick)

• **Feature Selectors**
  – VectorSlicer, Rformula
  – ChiSqSelector (Pick top features according to a chi-squared test)

• **Feature Transformers**
  – Tokenizer, n-gram, Normalizer, VectorAssembler
Classification & Regression

Classification
• Decision tree classifier
• Random forest classifier
• Linear Support Vector Machine
• Naive Bayes
• Logistic regression
• Binomial logistic regression
• Multinomial logistic regression
• Gradient-boosted tree classifier
• Multilayer perceptron classifier
• One-vs-Rest classifier

Regression
• Linear regression
• Generalized linear regression
• Available families
• Decision tree regression
• Random forest regression
• Gradient-boosted tree regression
• Survival regression
• Isotonic regression
Clustering and recommendation

• K-means
• Latent Dirichlet allocation (LDA)
  – Document topic modelling
  – Topics are cluster centers
• Bisecting k-means - Hierarchical clustering
• Gaussian Mixture Model (GMM)
  – Probabilistic: avoid fixing data objects into specific cluster
  – Better when clusters have different „shapes“ and sizes

• Collaborative filtering
  – Recommender systems
  – Predict what users like based on what similar users have liked
K-means example (Iris dataset)

dataset = spark.read.option("inferSchema", True).csv(input_file) \
          .toDF("slen", "swidth", "plen", "pwidth", "class")

<table>
<thead>
<tr>
<th>slen</th>
<th>swidth</th>
<th>plen</th>
<th>pwidth</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>Iris-setosa</td>
</tr>
</tbody>
</table>

dataset.select("class").distinct().show()

+-------------+      
| class       |      
+-------------+      
| Iris-virginica |      
| Iris-setosa  |      
| Iris-versicolor |
Feature selection using VectorAssembler

```python
assembler = VectorAssembler(
    inputCols=["slen", "swidth", "plen", "pwidth"],
    outputCol="features")
featured = assembler.transform(dataset)
```

<table>
<thead>
<tr>
<th>slen</th>
<th>swidth</th>
<th>plen</th>
<th>pwidth</th>
<th>class</th>
<th>features</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.1, 3.5, 1.4, 0.2]</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.9, 3.0, 1.4, 0.2]</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.7, 3.2, 1.3, 0.2]</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.6, 3.1, 1.5, 0.2]</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.0, 3.6, 1.4, 0.2]</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>Iris-setosa</td>
<td>[5.4, 3.9, 1.7, 0.4]</td>
</tr>
</tbody>
</table>
Building and using the K-Means model

```python
kmeans = KMeans().setK(3).setSeed(1)
model = kmeans.fit(featured)
predictions = model.transform(featured)
predictions.show(10, False)
```

<table>
<thead>
<tr>
<th>slen</th>
<th>swidth</th>
<th>plen</th>
<th>pwidth</th>
<th>class</th>
<th>features</th>
<th>prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.1, 3.5, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.9, 3.0, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.7, 3.2, 1.3, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.6, 3.1, 1.5, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.0, 3.6, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>5.4</td>
<td>3.9</td>
<td>1.7</td>
<td>0.4</td>
<td>Iris-setosa</td>
<td>[5.4, 3.9, 1.7, 0.4]</td>
<td>2</td>
</tr>
<tr>
<td>4.6</td>
<td>3.4</td>
<td>1.4</td>
<td>0.3</td>
<td>Iris-setosa</td>
<td>[4.6, 3.4, 1.4, 0.3]</td>
<td>2</td>
</tr>
<tr>
<td>5.0</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[5.0, 3.4, 1.5, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.4</td>
<td>2.9</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-setosa</td>
<td>[4.4, 2.9, 1.4, 0.2]</td>
<td>2</td>
</tr>
<tr>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>Iris-setosa</td>
<td>[4.9, 3.1, 1.5, 0.1]</td>
<td>2</td>
</tr>
</tbody>
</table>
### Clustering results

- Let's evaluate how well the classes were distributed among the clusters.
- We can apply the Spark DataFrame crosstab operation.

```python
predictions.crosstab("class", "prediction").show()
```

<table>
<thead>
<tr>
<th>class_prediction</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iris-virginica</td>
<td>14</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>48</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

Cluster ID: 52
ML Pipelines

• Spark ML operations can be chained into pipelines
• Join VectorAssembler and Kmeans into a single Pipeline

```python
assembler = VectorAssembler(
    inputCols = ["slen", "swidth", "plen", "pwidth"],
    outputCol = "features")

kmeans = KMeans().setK(3).setSeed(1)

pipeline = Pipeline(stages = [assembler, kmeans])
model = pipeline.fit(dataset)
```
Model persistence

- Models and pipelines can be saved to the filesystem (HDFS) and later loaded from there
- Allows to build models/pipelines ahead of time
- Built models can be migrated to other servers/clusters
- Saving models:

```python
model.save("myKmeansPipeline")
```

- Loading saved models:

```python
saved_model = KMeansModel.load("myKmeansPipeline")
```
Why use DataFrames for ML?

- Provide a uniform API across multiple languages
  - Java, Scala, Python,
  - R not so similar API
- More user-friendly than RDDs.
  - Convenient for users who have used dataframes in other languages
- Tabular format is natural for feature selection and manipulation
- Spark Data sources streamline data importing (CSV, JSON, Kafka streams)
- Tungsten and Catalyst optimizers can take advantage of stricter tabular data schemas
  - **Catalyst**: SQL query-to-code optimizer (Logical and Physical plans)
  - **Tungsten**: Off-Heap Memory Management, cache-aware computations, whole-stage code generation/merging