UNIVERSITY of TARTU

INSTITUTE OF COMPUTER SCIENCE

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

- Ddifferent 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

```
# 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:

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

- Reading data From CSV file:

```
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**)

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

peopleDF = spark.createDataFrame(people, schema)
```

From Pandas DataFrame

```
import numpy as np
import pandas as pd
matrix = np.random.rand(6, 6)
dataframe = pd.DataFrame(matrix)

sparkDF = spark.createDataFrame(dataframe)
```

```
+----------+----------+----------+----------+----------+----------+
|         0|         1|         2|         3|         4|         5|
+----------+----------+----------+----------+----------+----------+
|      0.81|      0.86|      0.11|      0.73|      0.43|      0.14|
|       0.5|      0.27|      0.72|      0.22|      0.64|      0.91|
|      0.78|      0.01|       0.5|      0.11|      0.31|       0.8|
|      0.42|      0.13|      0.66|      0.45|      0.72|      0.36|
|      0.24|      0.96|      0.83|      0.65|      0.19|      0.96|
|      0.08|      0.53|      0.44|      0.62|      0.45|      0.92|
+----------+----------+----------+----------+----------+----------+
```

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!

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

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

Save modes

- Save operations have multiple 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

```
df.write.mode("append") \
        .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()
```

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

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

City	Bank
CANMORE ALTA	ROYAL BANK OF CANADA
CHIPMAN	CANADIAN IMPERIAL BANK OF COMMERCE
EDMONTON, ALBERTA T5	HSBC BANK CANADA
Edmonton	ING BANK OF CANADA
TOKYO JAPAN	BANK OF MONTREAL

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

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

```
+----------+----------+-------+--------------+---------+----------+--------------------+
|Last_Name |First_Name |Balance|Address        |City      |Last_Trans|bank_name           |
+----------+----------+-------+--------------+---------+----------+--------------------+
|NEED NEWS |           |787.51 |12055 - 95 ST. |Edmonton  |04/02/1980|HSBC BANK CANADA    |
|BAKER     |DAPHNE     |93.85  |               |          |11/13/1980|BANK OF MONTREAL    |
|AKIYAMA   |M          |5646.64|RC 2-4         |UTSUNOMIYA|02/02/1980|ROYAL BANK OF CANADA |
|WATSON    |RONALD     |5199.89|PO STN C       |Edmonton  |01/09/1980|ROYAL BANK OF CANADA |
|LO        |ANNIE      |4256.07|14208 96 AVENUE|Edmonton  |04/18/1980|ROYAL BANK OF CANADA |
+----------+----------+-------+--------------+---------+----------+--------------------+
```

Grouping DataFrames

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

```
+--------------------+--------------------+------------------+
|                City|           bank_name|      sum(Balance)|
+--------------------+--------------------+------------------+
|     YELLOWKNIFE NT |   BANK OF MONTREAL |           1790.68|
|         TOKYO JAPAN |   BANK OF MONTREAL |            751.94|
|EDMONTON, ALBERTA T5|   HSBC BANK CANADA |            528.28|
|            Edmonton| ING BANK OF CANADA |            636.42|
|        CANMORE ALTA |ROYAL BANK OF CAN...|             51.37|
|             CHIPMAN |CANADIAN IMPERIAL...|             20.59|
|        ST. ALBERT AB|   HSBC BANK CANADA |             83.57|
+--------------------+--------------------+------------------+
```

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.

```
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)

```
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))
```

City	bank_name	Balance	bank_sums	city_sums
HONG KONG	HSBC BANK CANADA	82.67	477164.0	1147.0
HONG KONG	ROYAL BANK OF CANADA	1064.79	1341940.0	1147.0
THORSBY ALTA	ROYAL BANK OF CANADA	177.39	1341940.0	177.0
IRMA AB	BANK OF MONTREAL	2264.51	1476425.0	2265.0
RADWAY AB	BANK OF MONTREAL	182.04	1476425.0	182.0
AIRDRIE AB	BANK OF MONTREAL	397.79	1476425.0	432.0
AIRDRIE AB	TORONTO-DOMINION BANK	34.35	1154282.0	432.0
STAR CAN	TORONTO-DOMINION BANK	45.11	1154282.0	45.0

Cumulative aggregation

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

```
+-------------------------------------+-------+----+----------------+
|bank_name                            |Balance|year|cumul_sum       |
+-------------------------------------+-------+----+----------------+
|CANADIAN IMPERIAL BANK OF COMMERCE   |821.07 |1935|821.07          |
|CANADIAN IMPERIAL BANK OF COMMERCE   |2572.61|1939|3393.68         |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1974.39|1948|5368.07         |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1732.65|1960|7100.72         |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1954.07|1961|11791.81        |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1706.68|1961|11791.81        |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1030.34|1961|11791.81        |
|CANADIAN IMPERIAL BANK OF COMMERCE   |1799.0 |1965|13590.81        |
+-------------------------------------+-------+----+----------------+
```

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)
```

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")
)

                                                #First WordCount
counts = words.groupBy("word", "file") \
              .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

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

```
+-----------------------------------------------------------------+------------------------------------------------------+
|value                                                            |file                                                  |
+-----------------------------------------------------------------+------------------------------------------------------+
|The Project Gutenberg EBook of Frank Merriwell at Yale, by Burt L. Standish|file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|                                                                 |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|This eBook is for the use of anyone anywhere at no cost and with |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|almost no restrictions whatsoever.  You may copy it, give it away or|file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|re-use it under the terms of the Project Gutenberg License included|file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|with this eBook or online at www.gutenberg.net                   |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|                                                                 |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|                                                                 |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|Title: Frank Merriwell at Yale                                   |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
|                                                                 |file:///home/pelle/PycharmProjects/pellesparkone/in/11115.txt|
+-----------------------------------------------------------------+------------------------------------------------------+
```

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")
)
```

```
+---------+---------+
|file     |word     |
+---------+---------+
|11115.txt|The      |
|11115.txt|Project  |
|11115.txt|Gutenberg|
|11115.txt|EBook    |
|11115.txt|of       |
|11115.txt|Frank    |
|11115.txt|Merriwell|
|11115.txt|at       |
|11115.txt|Yale     |
|11115.txt|by       |
+---------+---------+
```

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

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

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

```
+---------+---------+---+-----+---+
|file     |word     |n  |bigN |m  |
+---------+---------+---+-----+---+
|11115.txt|By       |26 |90089|2  |
|11102.txt|By       |12 |47979|2  |
|11102.txt|Cannot   |1  |47979|1  |
|11115.txt|Drink    |4  |90089|1  |
|11102.txt|Easter   |2  |47979|1  |
|11102.txt|Heaven   |1  |47979|1  |
|11102.txt|JOHNSON  |4  |47979|1  |
|11102.txt|July     |25 |47979|1  |
+---------+---------+---+-----+---+
```

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

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

```
+------------------+---------------+------------------+--------------+----------------+
|     City_bank_name|BANK OF MONTREAL |BANK OF NOVA SCOTIA |CITIBANK CANADA |HSBC BANK CANADA |
+------------------+---------------+------------------+--------------+----------------+
|  URANIUM CITY SASK |              0|                 0|             0|               0|
|        SUNDRE ALTA |              1|                 0|             0|               0|
|        GRIMSHAW,AB |              0|                 3|             0|               0|
|         NANAIMO BC |              0|                 0|             0|               0|
|      ARLINGTON USA |              1|                 0|             0|               0|
|           MESA,USA |              0|                 0|             0|               0|
|         TOFIELD AB |              2|                 0|             0|               0|
|TETTENHALL, WOLVE...|              0|                 0|             0|               0|
+------------------+---------------+------------------+--------------+----------------+
```

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

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

```
+--------------------+------------------+--------------------+------------------+
|                City| BANK OF MONTREAL |BANK OF NOVA SCOTIA |  CITIBANK CANADA |
+--------------------+------------------+--------------------+------------------+
|            Edmonton|         775441.37|            10147.86|            3825.5|
|          St. Albert|          36592.55|             1065.36|              6.75|
|       Sherwood Park|          29561.52|              374.14|              6.72|
|         Stony Plain|          20848.49|               109.8|              null|
|               Leduc|           9509.77|                5.57|              8.82|
|           EDMONTON |           8515.96|                null|              null|
+--------------------+------------------+--------------------+------------------+
```

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 **b** 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

```
                                    #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):                                    #Define Python function
    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([                                  #Define function output data structure
   StructField("low1", DoubleType(), True),
   StructField("low2", DoubleType(), True),
   StructField("low3", DoubleType(), True),
])
                                                       #Register function as UDF
low3_udf = F.udf(low3, schema)
```

Spark UDF example II

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

```
+--------------------+------------------------------------+------------------------+
|City                |bank_name                           |balances                |
+--------------------+------------------------------------+------------------------+
|CANMORE ALTA        |ROYAL BANK OF CANADA                |[51.37,,]               |
|CHIPMAN             |CANADIAN IMPERIAL BANK OF COMMERCE|[20.59,,]               |
|EDMONTON, ALBERTA T5|HSBC BANK CANADA                    |[528.28,,]              |
|Edmonton            |ING BANK OF CANADA                  |[291.26, 155.53, 136.17]|
|TOKYO JAPAN         |BANK OF MONTREAL                    |[751.94,,]              |
+--------------------+------------------------------------+------------------------+
```

```
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

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

```
+-------------------+-------------------+-------+------+------+
|               City|          bank_name|   low1|  low2|  low3|
+-------------------+-------------------+-------+------+------+
|       CANMORE ALTA |ROYAL BANK OF CAN...|  51.37|  null|  null|
|            CHIPMAN |CANADIAN IMPERIAL...|  20.59|  null|  null|
|EDMONTON, ALBERTA T5|   HSBC BANK CANADA | 528.28|  null|  null|
|           Edmonton| ING BANK OF CANADA | 291.26|155.53|136.17|
|        TOKYO JAPAN |   BANK OF MONTREAL | 751.94|  null|  null|
|     YELLOWKNIFE NT |   BANK OF MONTREAL |1790.68|  null|  null|
|   SHERWOOD PARK,AB |BANK OF NOVA SCOTIA |  144.3|130.28|113.27|
+-------------------+-------------------+-------+------+------+
```

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)