SQL abstraction
for distributed data processing

Pelle Jakovits

9 November 2018, Tartu
Outline

• Role of SQL in distributed data processing (DDP)
• Overview of SQL DDP frameworks
  – Apache HiveQL
  – Apache Spark SQL
• Using SQL language for data processing
• User Defined Functions
• Advantages and disadvantages
Role of SQL In BigData

• SQL standard is well known and widely used
• Has traditionally been used in database systems for data querying and manipulation
  – Oracle DB, Postgre, Cassandra
• Data analytics platforms often also support SQL
  – SAP, SiSense
• More approachable for data analysts who are not professional programmers
• Bridge between existing analytics applications and distributed data processing platforms
SQL DDP frameworks

• **Apache Hive**
  – SQL on-top of Hadoop MapReduce

• **Spark SQL**
  – SQL interface in Apache Spark

• **Other SQL frameworks**
  – **Apache Impala** - SQL over HDFS
  – **Apache Phoenix** - SQL over Hbase
  – **Apache Kylin** - OLAP on Hadoop from eBay
  – **Apache Drill** - Open source Google BigQuery 'clone'
Apache Hive

- Data warehousing framework on-top of Hadoop
- Designed for:
  1. Simple data summarization and ad-hoc querying
  2. Analysis of large volumes of data.
- Uses SQL-based HiveQL language
- Input and output data is stored in HDFS
- Uses partitioning and bucketing to systematically segregate data which can significantly speed up data scanning
- Supports Java Database Connectivity (JDBC) drivers for importing (and exporting) from common DB systems
HiveQL

- HiveQL language follows most of the SQL standard
- HiveQL statements are automatically translated into Hadoop MapReduce jobs
- Hive Tables are "external" schemas that describe the internal structure and schema of HDFS files
- Supports Map and Reduce-like UDF functions using custom scripts
Data structures

• **Databases**
  – Namespaces that separate tables and other data units from naming conflicts

• **Tables**
  – Homogeneous units of data which have the same schema
  – Consists of specified columns accordingly to table schema

• **Partitions**
  – Each Table can have one or more partition Keys which determines how data is stored
  – Partitions allow the user to efficiently identify the rows that satisfy a certain criteria
  – It is the user's job to guarantee the relationship between partition name and data!

• **Buckets (or Clusters)**
  – Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table.
  – For example, the page_views table may be bucketed by userid to sample data by users
SELECT count(*) FROM sales WHERE YEAR = 2018, username = 'PelleJ';
hash('PelleJ') = 1
Importing Data into Hive

- There are multiple ways to load data into Hive tables.
- The user can create an external table that points to a specified location within HDFS.
- The user can copy a file into the specified location in HDFS and create a table pointing to this location with all the relevant row format information.
- Once this is done, the user can transform the data using HiveQL (SQL) and insert results into other Hive tables.
Create SQL Tables

- HiveQL Table describes the schema of a dataset.
- Data itself is stored in HDFS

```sql
CREATE TABLE page_view(
    viewTime INT,
    userid BIGINT,
    page_url STRING,
    referrer_url STRING,
    ip STRING);
```
Describing HDFS data sets

• Hive Tables can also be used to describe existing HDFS datasets
• Specify file type, line and field delimiters and location.

CREATE EXTERNAL TABLE page_view_stg(
    viewTime INT,
    userid BIGINT,
    page_url STRING,
    referrer_url STRING,
    ip STRING,
    country STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\n'
LINES TERMINATED BY '\t'
STORED AS TEXTFILE LOCATION '/data/page_view';
Spark SQL

• Implementation of HiveQL in Spark
  – But can avoid constantly having to define SQL schemas
  – Most user defined Hive functions are directly available

• SQL statements are compiled into Spark code

• Can use interchangeably with other Spark interfaces
  – Spark Java, Scala, Python, R
  – DataFrames, MLlib

• Uses Java Database Connectivity (JDBC) drivers to enable loading (and storing) data from most common DB systems (Oracle, MySQL, Postgre, etc.)
Using Spark SQL

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

#Register dataframe as an SQL table
bank_accounts.registerTempTable("bank_tab")

#Execute SQL query to create a new DataFrame
#Result is a DataFrame!
a = spark.sql("SELECT SUM(Balance) FROM bank_tab GROUP BY City")

#Show results
a.show(5, False)
Loading SQL tables

- Table schema is generated automatically when loading datasets through DataFrame API
- Reading data from JSON file example:

```python
df = spark.read.json("/data/people.json")
df.registerTempTable("people")
```

- Reading data from CSV file:

```python
df = spark.read.format("csv").option("header","true")
    .load("/data/Top_1000_Songs.csv")
df.registerTempTable("songs")
```
Loading SQL tables

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

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

```python
schemaPeople = spark.createDataFrame(people, schema)  
schemaPeople.createOrReplaceTempView("people_table")
```
USING SQL FOR DATA PROCESSING

In Apache Spark
Display table structure

```sql
DESCRIBE bank_accounts
```

<table>
<thead>
<tr>
<th>col_name</th>
<th>data_type</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last_Name</td>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>First_Name</td>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>Balance</td>
<td>double</td>
<td>null</td>
</tr>
<tr>
<td>Address</td>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>City</td>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>Last_Trans</td>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>bank_name</td>
<td>string</td>
<td>null</td>
</tr>
</tbody>
</table>
### SELECT * FROM bank_accounts

<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></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>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>

### SELECT City, Bank_name as Bank, Balance FROM bank_accounts

<table>
<thead>
<tr>
<th>City</th>
<th>Bank</th>
<th>Balance</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>19.35</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>751.94</td>
</tr>
</tbody>
</table>
Filtering rows

```sql
SELECT * FROM bank_accounts WHERE Last_Trans LIKE '%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>11/13/1980</td>
<td>BANK OF MONTREAL</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>ROYAL BANK OF CANADA</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 and Aggregating

```sql
SELECT City, bank_name, SUM(Balance) as bal_sum
FROM bank_accounts
GROUP BY City, bank_name
```

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>bal_sum</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 CANADA</td>
<td>51.37</td>
</tr>
</tbody>
</table>
WordCount using SQL

```sql
SELECT 
    explode( split( line, '[^a-zA-Z]+' ) ) as word
FROM lines

SELECT word, COUNT(*) as count 
FROM words 
GROUP BY word
```

<table>
<thead>
<tr>
<th>word</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>online</td>
<td>4</td>
</tr>
<tr>
<td>By</td>
<td>9</td>
</tr>
<tr>
<td>Text-Book</td>
<td>1</td>
</tr>
<tr>
<td>hope</td>
<td>8</td>
</tr>
<tr>
<td>some</td>
<td>75</td>
</tr>
</tbody>
</table>
SQL Joins

- LEFT OUTER, RIGHT OUTER or FULL OUTER joins are supported
- Can join more than 2 tables at once
- Spark SQL supports broadcast join
  - One table is fully read into memory and In-Memory join is performed
  - Mark one of the tables with SQL /*+ BROADCAST(r) */ comment
  - When both joined tables are marked, Spark broadcasts smaller table.

```sql
SELECT /*+ BROADCAST(r) */ r.key, s.key
FROM records r
JOIN src s
ON r.key = s.key
```

```sql
SELECT /*+ BROADCAST(r) */ r.key, s.key
FROM records r
JOIN src s
ON r.key = s.key
```
Over ... Partition ...

• Compute **In-place** aggregations without changing the original structure of the table
• Can utilize multiple different Grouping/Partitioning keys at the same time
• Supports sliding windows and cumulative aggregations
Over ... Partition ... Example

```sql
SELECT city, bank_name, bal_sum,
    SUM(bal_sum) OVER (PARTITION by bank_name) as bank_sums,
    SUM(bal_sum) OVER (PARTITION by City) as city_sums
FROM bank_accounts
```

<table>
<thead>
<tr>
<th>City</th>
<th>bank_name</th>
<th>bal_sum</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

SELECT bank_name, Balance, year,
    SUM(Balance) over (PARTITION BY bank_name ORDER BY year)
as cumulative_sum FROM bank_accounts2

<table>
<thead>
<tr>
<th>bank_name</th>
<th>Balance</th>
<th>year</th>
<th>cumulative_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>
<tr>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</td>
<td>1799.0</td>
<td>1965</td>
<td>13590.81</td>
</tr>
</tbody>
</table>
**collect_list(column)**

- Aggregation function to collect all fields from a column into a list

```sql
SELECT City, bank_name, collect_list(Balance) as balances
FROM bank_accounts
GROUP BY City, bank_name
```

<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, 123.3]</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, 34.11, 19.35]</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>[751.94, 11.43, 77.02]</td>
</tr>
</tbody>
</table>
**sort_array()**

```sql
SELECT City, bank_name, sort_array(collect_list(Balance)) as balances FROM bank_accounts GROUP BY City, bank_name
```

<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, 123.3]</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>[19.35, 34.11, 136.17, 155.53, 291.26]</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>[11.43, 77.02, 751.94]</td>
</tr>
</tbody>
</table>
Addressing nested array elements

```sql
SELECT
  City, bank_name,
  balances[0] as low1,
  balances[1] as low2,
  balances[2] as low3
FROM balances
```

<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 CANADA</td>
<td>51.37</td>
<td>123.3</td>
<td>null</td>
</tr>
<tr>
<td>CHIPMAN</td>
<td>CANADIAN IMPERIAL BANK OF COMMERCE</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>19.35</td>
<td>34.11</td>
<td>136.17</td>
</tr>
<tr>
<td>TOKYO JAPAN</td>
<td>BANK OF MONTREAL</td>
<td>11.43</td>
<td>77.02</td>
<td>751.94</td>
</tr>
</tbody>
</table>
Other functions

• `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
Histogram

- histogram_numeric(column, b)
- Computes a histogram of a column using b non-uniformly spaced bins.

SELECT
  Theme, 
  histogram_numeric(YEAR, 4) as histogram 
FROM songs GROUP BY THEME 

<table>
<thead>
<tr>
<th>Theme</th>
<th>histogram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Politics and protest</td>
<td>[[1935.16,6.0], [1969.3,61.0], [1984.4,52.0], [2004.6,22.0]]</td>
</tr>
<tr>
<td>Love</td>
<td>[[1933.5,2.0], [1964.4,73.0], [1978.9,48.0], [1999.8,16.0]]</td>
</tr>
<tr>
<td>Life and death</td>
<td>[[1933.0,4.0], [1949.5,2.0], [1968.6,63.0], [1993.3,62.0]]</td>
</tr>
<tr>
<td>People and places</td>
<td>[[1933.7,12.0], [1963.9,74.0], [1979.8,44.0], [2000.9,15.0]]</td>
</tr>
<tr>
<td>Heartbreak</td>
<td>[[1931.0,1.0], [1951.0,1.0], [1972.6,98.0], [1999.6,45.0]]</td>
</tr>
</tbody>
</table>
TF-IDF with SQL

SELECT  
    #Extract document name and split lines into words
    substring_index(file, '/', -1) as file,
    explode(split(line, '^[a-zA-Z]+$')) as word
FROM lines

#First WordCount
SELECT file, word, count(*) as n FROM words group by file, word

SELECT  
    #Compute N and m as new columns
    file, word, n,
    SUM(n) OVER (PARTITION by file) as bigN,
    COUNT(file) OVER (PARTITION by word) as m
FROM counts

SELECT  
    #Finally compute TF-IDF value
    file, word, n, bigN, m,
    (n/bigN * log2(2/m)) as tfidf
FROM withN ORDER BY word
Load Input Documents

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

```sql
SELECT
    substring_index(file, '/', -1) as file,
    explode(split(line, '[^a-zA-Z]+')) as word
FROM lines
```

<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

```
SELECT file, word, count(*) as n FROM words GROUP BY file, word

+-----------------+-----------+-----+
|     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

SELECT
    file, word, n,
    SUM(n) OVER (PARTITION by file) as bigN,
    COUNT(file) OVER (PARTITION by word) as m
FROM counts

<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>11115.txt</td>
<td>Fairbanks</td>
<td>1</td>
<td>90089</td>
<td>2</td>
</tr>
<tr>
<td>11102.txt</td>
<td>Fairbanks</td>
<td>1</td>
<td>47979</td>
<td>2</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

```sql
SELECT
    file, word, n, bigN, m,
    (n/bigN * log2(2/m)) as tfidf
FROM withN
ORDER BY word
```

<table>
<thead>
<tr>
<th>file</th>
<th>word</th>
<th>n</th>
<th>bigN</th>
<th>m</th>
<th>tfidf</th>
</tr>
</thead>
<tbody>
<tr>
<td>11102.txt</td>
<td>A</td>
<td>97</td>
<td>47979</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>11115.txt</td>
<td>A</td>
<td>85</td>
<td>90089</td>
<td>2</td>
<td>0.0</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ABOUT</td>
<td>2</td>
<td>47979</td>
<td>1</td>
<td>4.168E-5</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ACCREDITING</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084E-5</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ACCURACY</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084E-5</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ACHIEVEMENTS</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084E-5</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ACQUISITIONS</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084E-5</td>
</tr>
<tr>
<td>11102.txt</td>
<td>ACTIVE</td>
<td>1</td>
<td>47979</td>
<td>1</td>
<td>2.084E-5</td>
</tr>
</tbody>
</table>
```
User Defined Functions

• **HiveQL**
  – External Java functions can be registered as UDFs
    • Must implement Hive UDF interface
    • Must deployed as .jar files into the cluster HDFS
  – Possibility to use any command line command through **MapReduce streaming** interface as UDF

• **Spark SQL**
  – Java, Scala, Python functions can be used as UDF
  – Java functions must implement specific interfaces
  – Python functions can be used directly, but must specify their output schema and data types
Hive Java UDF example
	package com.example.hive.udf;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Lower extends UDF {
    public Text evaluate(Text s) {
        if (s == null) {
            return null;
        }
        return new Text(s.toString().toLowerCase());
    }
}

Using UDF in Hive

ADD JAR /tmp/my_jar.jar; /* HDFS PATH */

CREATE TEMPORARY FUNCTION my_lower
as 'com.example.hive.udf.Lower';

SELECT
    my_lower(word), count(*)
FROM words GROUP BY my_lower(word);
Spark SQL UDF example

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

spark.udf.register("tfidf_udf", tfidf, DoubleType())

SELECT
    file, word, n, bigN, m,
    tfidf_udf(n, bigN, m, D) as tfidf
FROM withN;
```

# Define Python function

# Register function as UDF

# Call UDF from SQL
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([
    StructField("low1", DoubleType(), True),
    StructField("low2", DoubleType(), True),
    StructField("low3", DoubleType(), True),
])

# Define function output data structure

# Register function as UDF
spark.udf.register("low3_udf", low3, schema)
Spark UDF example II

SELECT
    City, bank_name, low3_udf(collect_list(Balance)) as balances
FROM bank_accounts
GROUP BY City, bank_name

+-------------------+-----------------------------+-----------------+
<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>
+-------------------+-----------------------------+-----------------+

#Call UDF from SQL

#Table/DataFrame schema

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

```sql
SELECT
    City, bank_name,
    balances.low1 as L1,
    balances.low2 as L2
FROM balance_table
```
Advantages of SQL DDP frameworks

• Higher level data processing language
  – Simplifies working with large amounts of data
• Lower learning curve than Pig Latin or MapReduce
  – HiveQL is basically extended SQL
  – Less trial-and-error than Pig Latin
• SQL statements are automatically parallelized and scaled up using Hadoop MapReduce or Apache Spark
• Can depend on the data processing engine optimizations
• Can re-use existing SQL queries
• Can utilize partitioning and bucketing
SQL disadvantages

• Share some of the disadvantages of underlying frameworks
  – HiveSQL is slow start-up and clean-up of MapReduce jobs.
  – Not suitable for interactive OLAP Analytics (< 1 sec results)
  – No real time access to data (must scan whole buckets)

• Less suitable for data transformation
  – Limitations of the SQL language
  – Tasks like co-grouping can get complicated

• Updating data is complicated
  – Can add records or overwrite whole buckets
Pig Latin or SQL

• Pig Latin
  – Processing unstructured data
  – Data parsing and transformation

• SQL
  – When data is already structured
  – Data warehousing and analytics
  – Scaling up existing SQL-based applications
That's all for this week

• Next practice session is:
  – Processing data with Spark SQL

• Next lecture is:
  – DataFrame abstraction for distributed data processing
Spark DataFrame DB connectors

- Load data from PostgreSQL

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

1. Store Dataframe into PostgreSQL

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