Big Data Engineering

Laboratories Report

Table of contents:

Big Data Engineering

Laboratories 1 Report

Group B04 members:

1. Jan Radzimiński
2. Szymon Gałecki
3. René Pichler

Task 1

1. Data modelling for Cassandra

0. Introduction

- While modelling data for the following queries, we have to first decide which columns will have to appear in primary key (either as clustering or partition key) in every table, so that the table is structured for an appropriate query. For complete uniqueness of every row the 3 columns would have to be used: 'video_id', 'trending_date' and 'region'. However, since in Cassandra each table is structured for a given query, such a primary key **was not** needed in every case (uniqueness of all imported rows was not always necessary).

  The combination of these 3 columns would guarantee a unique primary key for a given entry, since no 2 videos can be trending in the same region at the same date unless they are the same entries. Such entries (with all these 3 values being identical) actually existed in the original csv files, although in each case if these 3 values were identical, the whole rows were identical as well, therefore they can or even should be omitted.

1.1. Get all trending videos for a given channel_title

- For that query, we basically want to filter our data by values from column "channel_title". To achieve that, the records with the same "channel_title" value should be stored in the same partition on our cluster. Therefore, the "channel_title" should be the partition key. Moreover, since we want to get all videos, the video_id should also appear in the primary key. Therefore, we used "channel_title" as partition key and "video_id" as clustering key (not as partition key since we do not need to

filter data by them).

Finally, we created the table with the CQLSH command:

- ```
  CREATE TABLE videos_by_channel_title(video_id text,
  trending_date date, title text, channel_title text,
  category_id text, publish_time timestamp, tags list<text>,
  views int, likes int, dislikes int, comment_count int,
  thumbnail_link text, comments_disabled boolean,
  ratings_disabled boolean, video_error_or_removed boolean,
  description text, region text, PRIMARY KEY((channel_title),
  video_id));
  ```

- Then we imported our data using the command:
- ```
  COPY
  videos_by_channel_title(video_id,trending_date,title,channel_
  title,category_id,publish_time,tags,views,likes,dislikes,comm
  ent_count,thumbnail_link,comments_disabled,ratings_disabled,v
  ideo_error_or_removed,description,region)
  FROM './data/*.csv' WITH DELIMITER=',' AND HEADER=TRUE AND
  QUOTE='"'
  ```

- Thanks to this data model, we can make the queries depending on “channel_title”, for example:
- ```
  SELECT COUNT(*) FROM videos_by_channel_title WHERE
  channel_title = ‘EminemVEVO’;
  ```

- Which would return number of trending videos for a channel title ‘EminemVEVO’ - which was the initial goal of this task:
- ```
  cqlsh:groupb04> SELECT COUNT(*) FROM videos_by_channel_title WHERE channel_title = 'EminemVEVO';

   count
  -------
       9

  (1 rows)
  ```

1.2. Get all trending videos for a given region and trending-date (should allow range queries (from-to) for the trending-date) ordered by likes

- To query the data with these conditions, we needed a more complex Primary Key defined. First of all, similarly as in the previous table, the records with the same region value should be stored on the one partition. Moreover, to perform range

queries based on the trending date, the data on a given partition should be sorted depending on that column. Also it should be sorted by likes. Therefore, we need the "region" to be the partition key. Then, the trending_date and likes should be our clustering keys, along with the 'video_id' to assure the uniqueness of the key (region and trending_date are already a part of it). Also, we needed to set the clustering order to these values. Finally, the command used for table creation looked like this:

- ```
  CREATE TABLE videos_by_region_and_date(video_id text,
  trending_date date, title text,channel_title text,
  category_id text, publish_time timestamp, tags list<text>,
  views int, likes int, dislikes int, comment_count int,
  thumbnail_link text, comments_disabled boolean,
  ratings_disabled boolean, video_error_or_removed boolean,
  description text, region text,  PRIMARY KEY((region),
  trending_date, likes, video_id))      WITH CLUSTERING ORDER BY
  (trending_date ASC, likes ASC, video_id ASC)
  ```

- Then we imported our data using the command:
- ```
  COPY
  videos_by_region_and_date(video_id,trending_date,title,channe
  l_title,category_id,publish_time,tags,views,likes,dislikes,co
  mment_count,thumbnail_link,comments_disabled,ratings_disabled
  ,video_error_or_removed,description,region)
  FROM  './data/*.csv'  WITH  DELIMITER=','  AND  HEADER=TRUE  AND
  QUOTE='"'
  ```

- Thanks to this data model, we can make the queries depending on "region", between given "trending_date" and sorted by likes:

- ```
  SELECT video_id, trending_date, likes FROM
  videos_by_region_and_date WHERE region = 'CA' AND
  trending_date > '2017-11-15' AND trending_date < '2017-11-17'
  LIMIT 10;
  ```

- Which would return the first 10 videos only from Canada, from 16 of November 2017, ordered by number of likes.
  ```

```
cqlsh:groupb04> SELECT video_id, trending_date, likes FROM videos_by_region
_and_date WHERE region = 'CA' AND trending_date > '2017-11-15' AND trending
_date < '2017-11-17' LIMIT 10;

 video_id    | trending_date | likes
-------------+---------------+-------
 0ayARJdf7I4 |    2017-11-16 |     0
 5khTN_EnAlY |    2017-11-16 |     0
 HnCU2OCu0fs |    2017-11-16 |     0
 IkQhv4qzbc4 |    2017-11-16 |     2
 lSozbwZmq74 |    2017-11-16 |    28
 Ay3gshiguqc |    2017-11-16 |    29
 lTshJ-2GVKM |    2017-11-16 |    31
 q0nZcgvDouc |    2017-11-16 |    48
 HIz1H_rkrl0 |    2017-11-16 |    74
 AT_XoYrpcDU |    2017-11-16 |    79

(10 rows)
```

1.3. Get all trending videos for a given region and category_id containing the tag "comedy"

- To achieve this type of query, similarly to the previous tasks, the records with the same "region" and "category_id" should be stored on a given partition. To achieve that, both of them should become the partition key in this table, along with our must-have column (video_id) as clustering key:

- ```
  CREATE TABLE videos_by_region_and_category(video_id text,
  trending_date date, title text,channel_title text,
  category_id text, publish_time timestamp, tags list<text>,
  views int, likes int, dislikes int, comment_count int,
  thumbnail_link text, comments_disabled boolean,
  ratings_disabled boolean, video_error_or_removed boolean,
  description text, region text,   PRIMARY KEY((region,
  category_id), video_id));
  ```

- Then we imported our data using the command:
- ```
  COPY
  videos_by_region_and_category(video_id,trending_date,title,ch
  annel_title,category_id,publish_time,tags,views,likes,dislike
  s,comment_count,thumbnail_link,comments_disabled,ratings_disa
  bled,video_error_or_removed,description,region)
  FROM  './data/*.csv'  WITH  DELIMITER=','  AND  HEADER=TRUE  AND
  QUOTE='"'
  ```

- The more tricky part is filtering by a specific tag. To achieve that, we needed to set the indexes on the tags found in the column with the command:

- `CREATE INDEX tag_id ON videos_by_region_and_category(tags);`

- With that set, we could perform the queries like:

- `SELECT video_id, region, trending_date, category_id FROM videos_by_region_and_category WHERE region = 'US' AND category_id = '1' AND tags CONTAINS 'comedy';`

- Which would return 12 videos from US, with category id 1 and containing tag comedy, which was the goal of this task

```
cqlsh:groupb04> SELECT video_id, region, trending_date, category_id FROM videos_by_region_and_category WHERE re
gion = 'US' AND category_id = '1' AND tags CONTAINS 'comedy';

 video_id    | region | trending_date | category_id
-------------+--------+---------------+-------------
 6tAB3YWWkxY |     US |    2018-03-18 |           1
 9H2SSvQ8ihA |     US |    2018-05-03 |           1
 D8PXRnNOKPU |     US |    2017-12-08 |           1
 Ept29ceiVfk |     US |    2018-02-23 |           1
 NBSAQenU2Bk |     US |    2018-06-11 |           1
 TjXQzRWmb_I |     US |    2018-06-12 |           1
 dRvKm__GrqU |     US |    2017-12-21 |           1
 drg74wOy8z8 |     US |    2018-01-03 |           1
 euaGS3tPRBw |     US |    2017-11-27 |           1
 oVOzkMe1K8s |     US |    2018-06-12 |           1
 pWqRw4NfFDA |     US |    2018-01-18 |           1
 wrpeEitIEpA |     US |    2018-01-29 |           1

(12 rows)
```

-

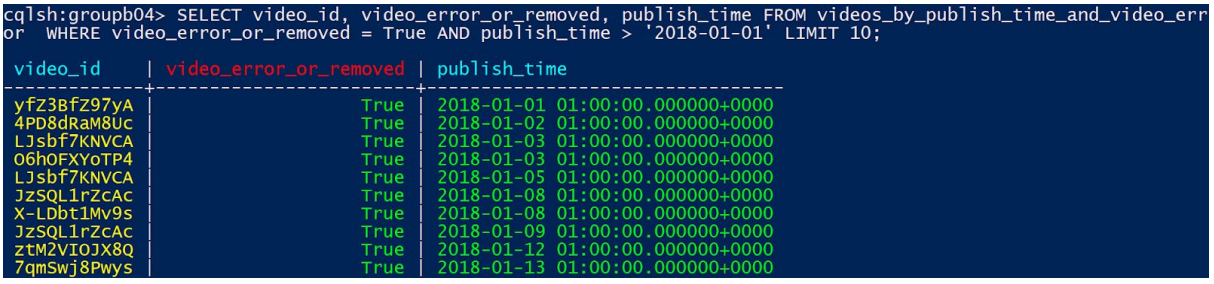
1.4. Try some other queries like you would in an RDBMS

1.4.1. Get all videos which were removed (or with error) for given publish time

- Similarly as in previous task, to create table for this query we needed to set the "video_error_or_removed" as partition key and "publish_time" as clustering key, along with necessary ones:

- `CREATE TABLE videos_by_publish_time_and_video_error(video_id text,trending_date date,title text,channel_title text,category_id text,publish_time timestamp,tags list<text>,views int,likes int,dislikes int,comment_count int,thumbnail_link text,comments_disabled boolean,ratings_disabled boolean,video_error_or_removed boolean,description text,region text,`
  `PRIMARY KEY((video_error_or_removed), publish_time, video_id))`

```
WITH CLUSTERING ORDER BY (publish_time ASC, video_id ASC)
```

- Then we imported our data using the command:
- ```
  COPY videos_by_publish_time_and_video_error(video_id,trending_date,title,channel_title,category_id,publish_time,tags,views,likes,dislikes,comment_count,thumbnail_link,comments_disabled,ratings_disabled,video_error_or_removed,description,region) FROM './data/*.csv' WITH DELIMITER=',' AND HEADER=TRUE AND QUOTE='"'
  ```
- Which would allow to get desired results by executing:
- ```
  SELECT video_id, video_error_or_removed, publish_time FROM videos_by_publish_time_and_video_error  WHERE video_error_or_removed = True AND publish_time > '2018-01-01' LIMIT 10;
  ```
- 

```
cqlsh:groupb04> SELECT video_id, video_error_or_removed, publish_time FROM videos_by_publish_time_and_video_err
or  WHERE video_error_or_removed = True AND publish_time > '2018-01-01' LIMIT 10;

 video_id    | video_error_or_removed | publish_time
-------------+------------------------+---------------------------------
 yfZ3BfZ97yA |                   True | 2018-01-01 01:00:00.000000+0000
 4PD8dRaM8Uc |                   True | 2018-01-02 01:00:00.000000+0000
 LJsbf7KNVCA |                   True | 2018-01-03 01:00:00.000000+0000
 O6hOFXYoTP4 |                   True | 2018-01-03 01:00:00.000000+0000
 LJsbf7KNVCA |                   True | 2018-01-05 01:00:00.000000+0000
 JzSQL1rZcAc |                   True | 2018-01-08 01:00:00.000000+0000
 X-LDbt1Mv9s |                   True | 2018-01-08 01:00:00.000000+0000
 JzSQL1rZcAc |                   True | 2018-01-09 01:00:00.000000+0000
 ztM2VIOJX8Q |                   True | 2018-01-12 01:00:00.000000+0000
 7qmSwj8Pwys |                   True | 2018-01-13 01:00:00.000000+0000
```

1.4.2. Get all trending videos with disabled comments ordered by dislikes descending

- Analogically, in that table the “comments_disabled” should be the partition key, while dislikes - clustering key along with necessary ones:

- CREATE TABLE videos_by_comments_order_by_dislikes(video_id text, trending_date date, title text,channel_title text, category_id text, publish_time timestamp, tags list<text>, views int, likes int, dislikes int, comment_count int, thumbnail_link text, comments_disabled boolean, ratings_disabled boolean, video_error_or_removed boolean, description text, region text, **PRIMARY KEY((comments_disabled), dislikes, video_id))**
  WITH CLUSTERING ORDER BY (dislikes DESC, video_id ASC)

- Then we imported our data using the command:
- COPY videos_by_comments_order_by_dislikes(video_id,trending_date,title,channel_title,category_id,publish_time,tags,views,likes,dislikes,comment_count,thumbnail_link,comments_disabled,ratings_disabled,video_error_or_removed,description,region) FROM './data/*.csv' WITH DELIMITER=',' AND HEADER=TRUE AND QUOTE='"'

- Which would allow to get desired results by executing:

- SELECT title, dislikes FROM videos_by_comments_order_by_dislikes WHERE comments_disabled = 'True' LIMIT 10;

-
```
cqlsh:groupb04> SELECT title, dislikes FROM videos_by_comments_order_by_dislikes WHERE comments_disabled = True
 LIMIT 10;

 title                                | dislikes
--------------------------------------+----------
      This Is America: Women's Edit |    75955
 Judge Roy Moore Campaign Statement |    59691
 Judge Roy Moore Campaign Statement |    59687
 Judge Roy Moore Campaign Statement |    59684
 Judge Roy Moore Campaign Statement |    59680
 Judge Roy Moore Campaign Statement |    59671
 Judge Roy Moore Campaign Statement |    59607
 Judge Roy Moore Campaign Statement |    59597
 Judge Roy Moore Campaign Statement |    59583
 Judge Roy Moore Campaign Statement |    59573
```

2. Files processing

Before importing data into our tables, we needed to process it to match cassandra requirements. For that we used a python script (attached as first attachment at the end of document). In that script we formatted tags to Cassandra's format: "{'tag1', 'tag2'}". We also formatted dates to a proper Cassandra format, removed empty values and not complete rows. Finally to each row we attached a value corresponding to a given region.

Thanks to our processing and correct data modelling, we managed to import > 340 000 out of 356 448 (correct) entries into each table (for primary key containing columns guaranteeing uniqueness of each column - the number of imported rows differs for specific queries), which gives the success rate of around 95%.

```
cqlsh:groupb04> SELECT COUNT(*) FROM videos_by_channel_title;

 count
--------
 343995

(1 rows)
```

3. Questions

3.1 What problems did you encounter with the given dataset?

- Data importing

We had a lot of problems as well while importing the data into the cassandra. Countless errors crashed our cassandra container and only hard restart worked, which meant we had to start our modelling from scratch. Our main problem was that the failure of importing a proper number of entries did not result from bad file processing (as we thought at the beginning) but from incorrect Cassandra data modelling (not ensuring the uniqueness of each primary key).

3.2 What queries are not possible?

Joins are not possible in Cassandra, that is to be expected since we are not working on a relational database. For each query we have to come up with another table and not worry about the data duplication.

3.3 Can you find out on which nodes your data is stored? (Hint: token)

Yes, you can find out this by querying the table with token() function and inputting partition key as its parameter. This will result in a list of hashed tokens containing the partition key. Then the range for each node has to be determined. The following is an example query.

```
select token(channel_title), channel_title from videos_by_channel_title;
```

Resulting in:

```
cqlsh:groupb04> select token(channel_title), channel_title from videos_by_channel_title;

 system.token(channel_title) | channel_title
-----------------------------+---------------------------
        -9222800850958668105 |               Karina Nigay
        -9222800850958668105 |               Karina Nigay
        -9222800850958668105 |               Karina Nigay
```

Then we check ranges of tokens for each node by running `nodetool ring`:

```
groupB04@node1: $ nodetool ring
Datacenter: Analytics
==========
Address      Rack        Status State   Load            Owns                Token
                                                                            6173272606662181104
10.40.53.11  rack1       Up     Normal  8,79 GiB        ?                   -3050099430192594702
10.40.53.14  rack1       Up     Normal  4,99 GiB        ?                   -2207534357470784567
10.40.53.12  rack1       Up     Normal  6,59 GiB        ?                   1561586588234793201
10.40.53.13  rack1       Up     Normal  7,26 GiB        ?                   6173272606662181104
```

So we know that this part of our data is stored on Node 1 (range up to tokens starting with -3….)..

Task 2

1. **Data choice & description**

   Streaming services have been familiar to us for quite a long time now. Easy access, wide offer and small price is a very tempting combination. Public places of culture such as cinemas, theaters, concert halls, nowadays come with a significant amount of risk. Apart from that we have to go through repertoire and find something suitable for ourselves. Not to mention the lack of freedom in choosing the desired time and place. Streaming services seem to know us better the more we use them. It is obvious that they process ridiculous amounts of data just to offer us the 'right' series or the 'similar' song. Although we won't go into the great details of those processes we decided to take a closer look at what attributes are being used for tracks on music streaming service - Spotify.

   Set contains data of more than 160,000 tracks available on Spotify, from years 1921-2020. Data was collected with the use of Spotipy which is a Python library for the Spotify Web API. Most of the variables are numerical, making this big set - lightweight.

   Primary:

   - id (Id of track generated by Spotify)

   Numerical:

   - acousticness (Ranges from 0 to 1)
   - danceability (Ranges from 0 to 1)
   - energy (Ranges from 0 to 1)

- duration_ms (Integer typically ranging from 200k to 300k)
- instrumentalness (Ranges from 0 to 1)
- valence (Ranges from 0 to 1)
- popularity (Ranges from 0 to 100)
- tempo (Float typically ranging from 50 to 150)
- liveness (Ranges from 0 to 1)
- loudness (Float typically ranging from -60 to 0)
- speechiness (Ranges from 0 to 1)
- year (Ranges from 1921 to 2020)

Dummy:

- mode (0 = Minor, 1 = Major)
- explicit (0 = No explicit content, 1 = Explicit content)

Categorical:

- key (All keys on octave encoded as values ranging from 0 to 11, starting on C as 0, C# as 1 and so on…)
- artists (List of artists mentioned)
- release_date (Date of release mostly in yyyy-mm-dd format, however precision of date may vary)
- name (Name of the song)

Link to data source: https://www.kaggle.com/yamaerenay/spotify-dataset-19212020-160k-tracks

2. Data modelling for Cassandra

We decided to implement tables for following queries, based on our data. In this case (unlike as in task 1) each entry has a unique value of 'id' which makes it necessary only for the 'id' field to appear in the primary key (as partition or clustering key). Since all of them were made analogously to the queries from task one, we decided to skip the detailed descriptions:

1. Get a spotify track by track name

- Partition keys: **name**
- Clustering keys: **id**
- Final command: `CREATE TABLE spotify_tracks_by_name(acousticness float, artists list<text>, danceability float, duration_ms`

int, energy float, explicit boolean, id text, instrumentalness float, key int, liveness float, loudness float, mode int, name text, popularity int, release_date date, speechiness float, tempo float, valence float, year int, **PRIMARY KEY((name), id))**

- Importing command: `COPY spotify_tracks_by_name(acousticness,artists,danceability,duration_ms,energy,explicit,id,instrumentalness,key,liveness,loudness,mode,name,popularity,release_date,speechiness,tempo,valence,year) FROM './spotify_tracks.csv' WITH DELIMITER=',' AND HEADER=TRUE AND QUOTE='"'`
- Example query: `SELECT id, name, artists, year FROM spotify_tracks_by_name WHERE name = 'Time';`

```
cqlsh:groupb04> SELECT id, name, artists, year FROM spotify_tracks_by_name WHERE name = 'Time';

 id                     | name | artists                                                | year
------------------------+------+--------------------------------------------------------+------
 05v5mPkLAOoa7M5tnXXOwq | Time | ['Free Nationals', 'Mac Miller', 'Kali Uchis'] | 2019
 1N0UflnZhn7xOkrN2l8zxx | Time |                          ['Hootie & The Blowfish'] | 1994
 1lOHEUOFlISG3skIhTxfiH | Time |                            ['Yusuf / Cat Stevens'] | 1970
 1rHagM3ZYdJbOKgY3pLrIc | Time |                                  ['Nancy Sinatra'] | 1966
 248duoFk1LhS9s1R8Foahi | Time |                        ['Sly & The Family Stone'] | 1971
 2F8fU263coUQ6SFK9dIjWW | Time |                  ['Clifford Brown', 'Max Roach'] | 1956
 2Mx2IvmTfOj5xnWBtx7ZbY | Time |                                        ['Jungle'] | 2014
 2RAYBLhifSaJJZRnxuHh8w | Time |                                      ['Anthrax'] | 1990
 3TO7bbrUKrOSPGRTB5MeCz | Time |                                    ['Pink Floyd'] | 1973
 48yJZwYYDZX5GKFND7wDfC | Time |                ['The Alan Parsons Project'] | 1980
 4HXJQwFzUnUaSWJsbiBH8G | Time |                             ['Dennis Wilson'] | 1977
 65TIlSzHRK25EE5P972aYG | Time |                               ['Tom Waits'] | 1985
 6MxMqCgqmekcrOVwSU7AVH | Time |                                       ['NF'] | 2019
 6ZFbXIJkuI1dVNWvzJzown | Time |                              ['Hans Zimmer'] | 2010
 6uOEOFItvADPCgiAKGPr4s | Time |                               ['Edwin Starr'] | 1970
 7duFyBg8iBNpZQ2bFjvdUk | Time |                          ['Pozo Seco Singers'] | 1965
```

- Use case - This query can be used for:
  - Finding a song with a given name and reading data about it (artists, energy etc.)
  - Checking how many songs with a given name are there
  - Getting all artists that created a song with a given name

2. Get all spotify tracks from a given year sorted by acousticness descending

- Partition keys: year
- Clustering keys: acousticness
- Final command: `CREATE TABLE spotify_tracks_by_year(acousticness float, artists list<text>, danceability float, duration_ms int, energy float, explicit boolean, id text, instrumentalness float, key int, liveness float, loudness float, mode int, name text, popularity int, release_date`

```
date, speechiness float, tempo float, valence float, year
int,
PRIMARY KEY((year), acousticness, id))
WITH CLUSTERING ORDER BY (acousticness DESC, id ASC)
```

- Importing command: `COPY spotify_tracks_by_year(acousticness,artists,danceability,duration_ms,energy,explicit,id,instrumentalness,key,liveness,loudness,mode,name,popularity,release_date,speechiness,tempo,valence,year) FROM './spotify_tracks.csv' WITH DELIMITER=',' AND HEADER=TRUE AND QUOTE='"'`
- Example query: `SELECT name, year, acousticness FROM songs_by_year WHERE year = 2010 LIMIT 10;`

```
cqlsh:groupb04> SELECT year, name, acousticness FROM spotify_tracks_by_year WHERE year = 2012 LIMIT 10;

 year | name                                                    | acousticness
------+---------------------------------------------------------+--------------
 2012 |                                            Heart Chakra |        0.995
 2012 |                                             Root Chakra |        0.995
 2012 |                                            Annie's Song |        0.995
 2012 | 2 Pieces, Op. posth., B. 188: No. 1. Lullaby in G Major |        0.994
 2012 |                                  Rêverie, L. 68: Rêverie |        0.994
 2012 |                                                      Re |        0.993
 2012 |                                      Tracking Aeroplanes |        0.993
 2012 |                             Suite Bergamasque: Clair de Lune |        0.992
 2012 |                  Colors of the Wind (From Pocahontas)""" |        0.992
 2012 |                      Waltz in C-Sharp Minor, Op. 64 No. 2 |        0.991
```

- Use case - This query can be used for:
  - Finding a all songs from a given year
  - Counting how many songs were released in a given year
  - Seeing how causticness of the songs differ along the years

3. Get all spotify tracks by key and danceability (range) sorted by energy descending

- Partition keys: key
- Clustering keys: danceability, energy, id
- Final command: `CREATE TABLE spotify_tracks_by_key_and_danceability(acousticness float, artists list<text>, danceability float, duration_ms int, energy float, explicit boolean, id text, instrumentalness float, key int, liveness float, loudness float, mode int, name text, popularity int, release_date date, speechiness float, tempo float, valence float, year int,`

```
PRIMARY KEY((key), danceability, energy, id))
```

WITH CLUSTERING ORDER BY (danceability ASC, energy DESC, id ASC)

- Importing command: `COPY spotify_tracks_by_key_and_danceability(acousticness,artists,danceability,duration_ms,energy,explicit,id,instrumentalness,key,liveness,loudness,mode,name,popularity,release_date,speechiness,tempo,valence,year) FROM './spotify_tracks.csv' WITH DELIMITER=',' AND HEADER=TRUE AND QUOTE='"'`
- Example query: `SELECT name, key, danceability, energy FROM spotify_tracks_by_key_and_danceability WHERE key=10 AND danceability > 0.902 AND danceability < 0.904;`

```
cqlsh:groupb04> SELECT name, key, danceability, energy FROM spotify_tracks_by_key_and_danceability WHERE
key=10 AND danceability > 0.902 AND danceability < 0.904;

 name                                | key | danceability | energy
-------------------------------------+-----+--------------+--------
                         Putrid Pride |  10 |        0.903 |  0.643
                                Hello |  10 |        0.903 |   0.61
 Red Bentley (feat. Young Thug) |  10 |        0.903 |  0.609
                    Twist My Fingaz |  10 |        0.903 |  0.602
                          Long Hair |  10 |        0.903 |  0.529
       Well I Guess I Told You Off |  10 |        0.903 |  0.393
        Life Upon The Wicked Stage |  10 |        0.903 | 0.0697
```

- Use case: This query can be used for:
  - Djs finding tracks for harmonic mixing (switching to a song with a proper key and with good energy and danceability)
  - Finding a song to fit other song by key and its properties
  - Finding out the key of a song for making cover or remix

4. Get all spotify tracks from Swedish House Mafia (artist)

- Partition keys: id
- Clustering keys: -
- Final command: `CREATE TABLE spotify_tracks_by_artists(acousticness float, artists list<text>, danceability float, duration_ms int, energy float, explicit boolean, id text, instrumentalness float, key int, liveness float, loudness float, mode int, name text, popularity int, release_date date, speechiness float, tempo float, valence float, year int,`

  PRIMARY KEY(id)
- Importing command: `COPY spotify_tracks_by_artists(acousticness,artists,danceability,duration_ms,energy,explicit,id,instrumentalness,key,liveness,l`

```
oudness,mode,name,popularity,release_date,speechiness,tempo,v
alence,year) FROM './spotify_tracks.csv' WITH DELIMITER=','
AND HEADER=TRUE AND QUOTE='"'
```

- **`CREATE INDEX artist_id ON spotify_tracks_by_artists(artists);`**
- Example query: `SELECT name, artists FROM spotify_tracks_by_artists WHERE artists CONTAINS 'Swedish House Mafia';`

```
 name                                                | artists
-----------------------------------------------------+-----------------------------------------
 Miami 2 Ibiza - Swedish House Mafia vs. Tinie Tempah | ['Swedish House Mafia', 'Tinie Tempah']
                                Antidote - Radio Edit |  ['Swedish House Mafia', 'Knife Party']
                                     One - Radio Edit |                 ['Swedish House Mafia']
                              Don't You Worry Child - |  ['Swedish House Mafia', 'John Martin']
                                        Save The World |                 ['Swedish House Mafia']
                                            Greyhound |                 ['Swedish House Mafia']
                          Don't You Worry Child - Radio Edit |  ['Swedish House Mafia', 'John Martin']

(7 rows)
```

- Use case: This query can be used for:
  - Finding all songs from a given artist(s)
  - Counting all songs from a given artist(s)
  - Analysing the properties of songs from a given artist(s)

3. **File processing**

Similarly as in the first task, we processed data with the python script (attached as second attachment at the end of document), where we formatted the single year values into format year-month-date so that it fits cassandra format.

<u>Thanks to our processing and correct data modelling, we managed to import 167 812 out of 169 902 (correct) entries into each table (for each query), which gives the success rate of around 99%.</u>

```
cqlsh:groupb04> SELECT COUNT(*) FROM spotify_tracks_by_name;

 count
--------
 167812

(1 rows)
```

Attachment 1 - Python script used for formating the trending videos files:

```
import csv
import re
import sys
from datetime import datetime

FILE_NAME = 'CAvideos.csv'
# Regex for matching any '|' character outside of the quotes, used for proper
tags formating
TAGS_REPLACE_REGEX = re.compile('\|(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)')
OUTPUT_FOLDER = 'output'
INPUT_FOLDER = 'data'

def format_tags(tags_string):
    temp = re.sub(TAGS_REPLACE_REGEX, ',', tags_string).split(
        ',')
    temp[0] = '"' + temp[0] + '"'
    return '{' + ','.join(temp).replace('"', "'").replace(',,',
',').replace(', ,', ',') + '}'


def format_date(date_string, input_pattern, output_pattern):
    return datetime.strptime(date_string,
input_pattern).strftime(output_pattern)
def process_file(language):
    ENCODING = 'utf8'
    with open(f'./{INPUT_FOLDER}/{language}videos.csv', newline='\n',
encoding=ENCODING, errors='ignore') as csvfile:
        file = csv.reader(csvfile, delimiter=',', quotechar='"')
        row_number = 0
        empty_cells = 0
        with open(f'./{OUTPUT_FOLDER}/{language}formatted.csv', 'w',
newline='', encoding=ENCODING) as csvfile:
            outputCsv = csv.writer(csvfile, delimiter=',',
                                   quotechar='"',
quoting=csv.QUOTE_NONNUMERIC)
            try:
                for row in file:
                    try:
                        if row_number == 0:
                            row.append('region')
```

```
                        row_number += 1
                        outputCsv.writerow(row)
                        continue
                    # Handling incorrect rows
                    if len(row) != 16:
                        print('wrong row')
                        continue

                    # Formating the date
                    row[1] = format_date(row[1], '%y.%d.%m', '%Y-%m-%d')

                    # Formating the tags
                    row[6] = format_tags(row[6])

                    # Add region to row
                    row.append(language)

                    # Handling empty cells
                    wrong_cell = False
                    for cell in row:
                        test_cell = cell
                        test_cell = test_cell.replace(
                            ' ', '').replace('\n', '')
                        if not test_cell:
                            wrong_cell = True

                    if wrong_cell:
                        continue

                    outputCsv.writerow(row)

                    row_number += 1
                except:
                    continue
            print('processed: ' + language)
        except:
            print(sys.exc_info()[0])

languages = ['CA', 'DE', 'FR', 'GB', 'IN', 'JP', 'KR', 'MX', 'RU', 'US']
for language in languages:
    process_file(language)
```

Attachment 2 - Python script used for formating the spotify file:

```
import csv
import re
import sys
from datetime import datetime

def process_file():
    ENCODING = 'utf8'
    with open(f'data.csv', newline='\n', encoding=ENCODING) as csvfile:
        file = csv.reader(csvfile, delimiter=',', quotechar='"')
        row_number = 0
        empty_cells = 0
        with open(f'spotify_tracks.csv', 'w', newline='', encoding=ENCODING)
as csvfile:
            outputCsv = csv.writer(csvfile, delimiter=',',
                                   quotechar='"', quoting=csv.QUOTE_MINIMAL)

            for row in file:
                if row_number == 0:
                    row_number += 1
                    outputCsv.writerow(row)
                    continue

                # Formating the date
                if len(row[14]) < 5:
                    row[14] = f'{row[14]}-01-01'
                elif len(row[14]) < 8:
                    row[14] = f'{row[14]}-01'

                outputCsv.writerow(row)

                row_number += 1

            print(row_number)


process_file()
```

Big Data Engineering

Laboratories 2 Report

Group B04 members:

1. Jan Radzimiński
2. René Pichler

Introduction

The task in this laboratory was to implement two MapReduce jobs. In the first MapReduce job we looked at a number of songs for a given artist. The second use case was about calculating the average tempo for songs from each decade. For purposes of these use cases we created three Java files (Mapper, Reducer and Main Function) which were then compiled with maven. Then the created jar files were loaded with scp to the area reserved for our group on the server. Then we connected to the server with ssh and executed the created jar file. We have specified our data set "spotify_tracks.csv" as the data set to be used. After the execution was finished, we were able to look at the results, which can be seen in the results section for each use case.

Use Case 1

1. Use case description

As the first use case we decided to implement a MapReduce job which results in the number of songs for each artist.

2. Exercise implementation

a. Implementation of number of songs per main artist

i. Mapper

Our code for the Mapper class can be found below. The Frequency Mapper class extends the predefined Mapper class and overwrites the map function. And also includes the logic to process the names of the artists correctly.

```java
public class FrequencyMapper
  extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);

  @Override
  public void map(LongWritable offset, Text lineText, Context context)
      throws IOException, InterruptedException {

    String line = lineText.toString();
    String artists = line.split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1)[1];
    artists = artists.replaceAll("\"", "");
    if (artists.length() < 2) return;
    artists = artists.substring(1,artists.length()-1);
    String[] artistsArr = artists.split(",");

    for (String artist : artistsArr) {
      if (artist.length() < 2) continue;
      String artistSub=artist.trim();
      artistSub = artistSub.replaceAll("'", "");
      context.write(new Text(artistSub.trim()), one);
    }


  }
}
```

ii. Reducer

The FrequencyReducer class extends the predefined Reducer class and overwrites the reduce function, counting every song of a given artist.

```java
package fh.its.bde.wordcount;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class FrequencyReducer extends Reducer<Text , IntWritable , Text , IntWritable > {
    @Override public void reduce( Text eventID, Iterable<IntWritable> counts, Context context)
        throws IOException, InterruptedException {

      int sum = 0;
      for ( IntWritable count : counts) {
        sum += count.get();
      }
      context.write(eventID, new IntWritable(sum));
    }
}
```

iii. Main function

```java
package fh.its.bde.wordcount;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Frequency {

    Run | Debug
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: Frequency <input path> <output path>");
            System.exit(-1);
        }

        // create a Hadoop job and set the main class
        Job job = Job.getInstance();
        job.setJarByClass(Frequency.class);
        job.setJobName("Frequency");

        // set the input and output path
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        // set the Mapper and Reducer class
        job.setMapperClass(FrequencyMapper.class);
        job.setReducerClass(FrequencyReducer.class);

        // specify the type of the output
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // run the job
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

3. Results

The following is an extract from the results obtained. If you check this with the entries in the used database, for example for Alan Walker, the number of 20 songs is correct.

```
579   Alan Parker 1
580   Alan Parsons    3
581   Alan Rickman    1
582   Alan Silvestri  8
583   Alan Tam    9
584   Alan Titus  1
585   Alan Umstead    1
586   Alan Walker 20
587   Alan Watts  1
588   Alan Wayne  1
589   Alan Wilder 7
590   Alanalda    1
591   Alanis Morissette   43
592   Alannah Myles   2
593   Alasdair Fraser 2
```

Or, as can be seen in the solution to the previous laboratory, Swedish House Mafia has 7 tracks in total.

Use Case 2

1. **Use case description**

For the second task, we decided to implement MapReducer, which gets the average tempo (BPM) of songs for each decade - from 1920s, up to 2020s. To maximize the accuracy of results, we created 3 chained MapReducers, working one after each other.

2. **Exercise implementation**

    a. Firstly, we get the value for the single year - we decided to use the median, since there are a lot of side values which may decrease accuracy of the average. We also converted each tempo into one from interval 80 - 160 (since tempo 60 is basically the same as 120 and the given interval is most commonly used).

i. Mapper

```java
public class YearMapper extends Mapper<LongWritable, Text, Text, LongWritable> {
    private static final Logger LOG = Logger.getLogger(TempoAverage.class);

    private Text year = new Text();
    private final static LongWritable number = new LongWritable();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        LOG.info("Mapping started....");
        final int upperBpmBound = 160;
        final int lowerBpmBound = 80;
        final int songBpmBound = 30;

        LOG.info(key);
        int rowLength = 19;

        String[] parts = value.toString().split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1);
        if (parts.length != rowLength)
            return;

        try {
            long tempo = Math.round(Double.parseDouble(parts[16]));
            if (tempo < songBpmBound)
                return;
            while (tempo >= upperBpmBound) {
                tempo = (long) tempo / 2;
            }
            while (tempo < lowerBpmBound) {
                tempo = (long) tempo * 2;
            }
            year.set(parts[18]);
            number.set(tempo);
            LOG.info(year);
            context.write(year, number);
        } catch (Exception e) {
            System.out.println("error");
            return;
        }

    }
}
```

ii. Reducer

```java
public class YearReducer extends Reducer<Text, LongWritable, Text, LongWritable> {
  private LongWritable result = new LongWritable();

  private static final Logger LOG = Logger.getLogger(YearReducer.class);

  public void reduce(Text key, Iterable<LongWritable> values, Context context)
      throws IOException, InterruptedException {
    LOG.info("Reducing...");
    List<Long> list = new ArrayList<Long>();

    // Converting values to list
    for (LongWritable val : values) {
      list.add(val.get());
    }

    if (list.size() < 1)
      return;

    // Sorting the values
    Collections.sort(list);

    // Getting median
    long res = list.get((int) list.size() / 2);

    result.set(res);
    LOG.info(result.toString());
    // LOG.info("Key: " + key + ", result: " + result + ", avg: " + avg);
    context.write(key, result);
  }
}
```

b. Then, for years from each decade, we calculate average tempo by simple arithmetic average.

i. Mapper

```
public class YearsPeriodMapper extends Mapper<Text, LongWritable, Text, LongWritable> {
    private static final Logger LOG = Logger.getLogger(TempoAverage.class);

    private Text years = new Text();
    private final static LongWritable number = new LongWritable();

    public void map(Text key, LongWritable value, Context context) throws IOException, InterruptedException {
        LOG.info("Mapping started....");
        String k = key.toString().substring(0, 3).toString() + "0";

        years.set(k);
        LOG.info(years);
        number.set(value.get());

        context.write(years, number);
        // context.write(key, value);

    }
}
```

ii. Reducer

```
public class YearsPeriodReducer extends Reducer<Text, LongWritable, Text, LongWritable> {
  private LongWritable result = new LongWritable();

  private static final Logger LOG = Logger.getLogger(YearReducer.class);

  public void reduce(Text key, Iterable<LongWritable> values, Context context)
      throws IOException, InterruptedException {
    LOG.info("Reducing...");
    long sum = 0;
    int count = 0;
    for (LongWritable val : values) {
      LOG.info("val...");
      count++;
      sum += val.get();
    }
    if (count == 0)
      return;
    long val = Double.valueOf(sum / count).longValue();
    result.set(val);
    context.write(key, result);
  }
}
```

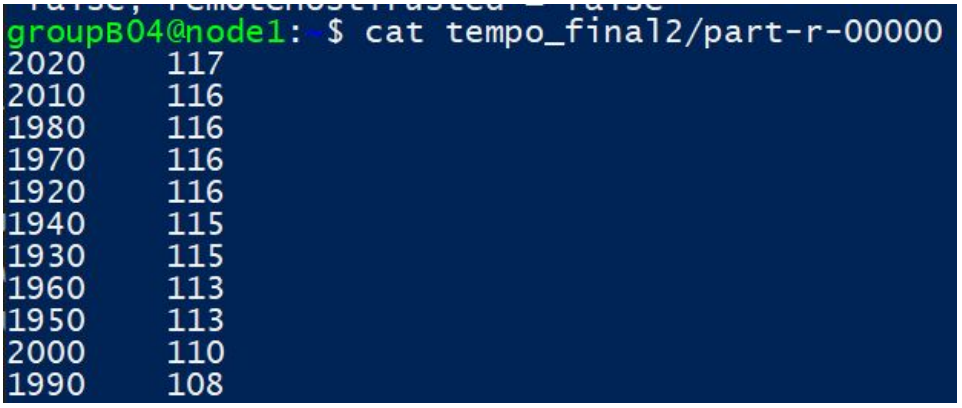
c. Finally, we use the last MapReducer just for sorting our results, by the tempo, descending.

i. Comparator

```
public class LongInverseComparator extends WritableComparator {

    private static final Log LOG = LogFactory.getLog(LongInverseComparator.class);

    public LongInverseComparator() {
        super(LongWritable.class, true);
    }

    @SuppressWarnings("rawtypes")    "rawtypes": Unknown word.
    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        LOG.info("-------------Comparing");
        LOG.info(a);
        LOG.info(b);
        return -((LongWritable) a).compareTo((LongWritable) b);

    }
}
```

3. Results

As the result, we achieve exactly what we needed:



We can observe that the fastest tempo appeared in the recent years (2010s, 2020s) and the slower one was between 1990s - 2000s.

4. References

In our solutions we used some code snippets from the course repository:

https://its-git.fh-salzburg.ac.at/bde/students-mapreduce-demos

Big Data Engineering

Laboratories 3 Report

Group B04 members:

1. Jan Radzimiński
2. René Pichler
3. Michael Ramsteck

Use Case 2 - Average Tempos

1. Topic Structure

As presented during first laboratories, our use case is data about tracks from Spotify streaming platform. The task was to structure our topic (with its partitions and messages), so that it suits the algorithm we implemented in previous laboratories - that is calculating average tempo of a song during a given decade.

In this use case, we are taking into the calculations all of the past songs for calculating the average tempo. Therefore, there is no reason to store given messages on a specific partition for balancing load of various workers, since the calculations are always about all of the songs together. Thus, there is no reason for defining the message key for the messages in this topic (because it does not matter on which partition the given song is stored). Instead, we decided to use 3 partitions where the load will be evenly splitted.

Finally the command to create the topic was as follows:

```
kafka-topics.sh --create --zookeeper node1:2181 \
--replication-factor 3 --partitions 3 --topic
groupB04.spotifytempos
```

2. Consumer groups

2.1 Define how you would arrange consumers and consumer groups

As previously stated, all of the songs messages sent to the topic do not have any message key, so they are spread evenly across the 3 partitions in the topic. Therefore, there is no reason for

any specific consumers arrangement in a group, since the given worker needs to take all of the songs to calculate data anyway.

Theoretically, we could use consumer groups with up to 3 consumers, where each consumer would process messages from a given partition and then they could use the average of results from each consumer for the final result, although it would not make much sense and would probably not result in any performance improvement.

3. Implementation

3.1. Java-Producer

Our algorithm is used for calculating the average tempo of songs from a given decade (while taking the median of songs tempo from each year). Therefore, the only significant data for each of the songs in our use case is song release year, tempo and song id (for not counting the same song twice). Therefore, we made a producer which takes our input csv file (minimized version of our original file with only around 1600 rows with songs from different years) and for each row it sends a message with proper data. Since we have only three significant values in each message we decided to create string values of them in the format of comma separated values with the following structure: “[id],[year],[tempo]”. For example, for a song with id 1, from 2020 and with 128 bpm tempo our producer would send the following message: “1, 2020, 128”.

For our implementation we used the classes from the demo project and changed them to meet our needs. Therefore, the producer was implemented as runnable java class, which has “run” method which looks as follows:

```java
@Override
public void run() {
    System.out.println("=== SPOTIFY PRODUCER ===");
    try {
        FileReader fr = new FileReader(INPUT_FILE);
        BufferedReader br = new BufferedReader(fr);

        String line;
        ProducerRecord<Long, String> record;
        long key;
        String value;
        int skipLines = 1;
        int sentMessages = 0;

        while ((line = br.readLine()) != null) {
            // Skipping column names row
            if (skipLines-- > 0) continue;

            // Exporting id, tempo and year from the row
            String[] parts = line.toString().split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1);
            if (parts.length != ROW_LENGTH) continue;
            String tempo = parts[TEMPO_COLUMN];
            String year = parts[YEAR_COLUMN];
            String id = parts[ID_COLUMN];

            // Setting message key
            try {
                key = Long.parseLong(year);
            } catch(NumberFormatException error) {
                System.out.println("Wrong year - cant convert to long\n");
                continue;
            }

            value = getRecordValue(id, year, tempo);
            record = new ProducerRecord(TOPIC, value);
            producer.send(record);
            System.out.printf("[Sent to %s] Value = %s\n",
                    TOPIC,
                    value);

            // Used only for testing
            if (messageCount != -1) {
                if (++sentMessages == messageCount) break;
            }
```

```java
            // Delay between records sending
            try {
                Thread.sleep(DELAY_BASE);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Closing ...");
        producer.flush();
        producer.close();

        System.out.println("Done.");
    } catch(FileNotFoundException e) {
        System.out.println("File not found");
        return;
    } catch(IOException e) {
        System.out.println("IO exception");
        return;
    }
}
```

And its main function:

```
public static void main(String[] args) {
    try {
        new Thread(new SpotifyProducer(Helper.createProducer(CLIENT_ID))).start();
    } catch (NumberFormatException e) {
        System.err.println("Usage: SpotifyProducer");
        System.exit(2);
    }
}
```

3.2. Java-Consumer

Then, we implemented a java consumer, which takes a record produced by our java producer, and it just gets the proper values from the string array, and prints the results to the console. Additionally, it should print out the partition and offset of a given message.

Similarly as with producer, for our implementation we were heavily inspired by the demo project, so our consumer was implemented as runnable as well. Its “run” method looks like this:

```
public void run() {
    System.out.println("=== SPOTIFY CONSUMER ===");
    try {
        System.out.println("=> Subscribed to " + SpotifyTemposProducer.TOPIC);
        System.out.println("==> Waiting for messages...");
        while (!closed.get()) {
            ConsumerRecords<Long, String> records = consumer.poll(Duration.ofMillis(POLL_DURATION_MS));
            System.out.println("==> After poll (enter to exit); records: " + records.count());
            for (ConsumerRecord<Long, String> record : records) {
                String[] values;
                values = getValuesFromArray(record.value());

                // Wrong record format
                if (values.length != 3) continue;

                System.out.printf(
                        "[Partition: %d, Offset: %d] Id: %s, Year: %s, Tempo: %s;\n",
                        record.partition(),
                        record.offset(),
                        values[0],
                        values[1],
                        values[2]);
            }
        }
    } catch (WakeupException e) {
        System.out.println(e.toString());
        /* Ignore exception if closing */
        if (!closed.get())
            throw e;
    } finally {
        consumer.close();
    }
```

And its main function:

```java
public static void main(String[] args) throws InterruptedException, IOException {

    SpotifyConsumer r = new SpotifyConsumer(Helper.createConsumer(CLIENTID, GROUPID));
    Thread t = new Thread(r);
    t.start();

    System.out.println("Press any key to exit");
    System.in.read();

    System.out.println("Shutting down ...");
    r.shutdown();
    t.join();
    System.out.println("Done.");
}
```

4. Testing

For testing our implemented functions we used the testing class with dummy producers and consumers that were presented in the demo repository - they only simulate sending or consuming messages of a given topic to check if the number of results are as expected.

Therefore, in the producer case, our testing class just creates the new producer which simulates sending the first 10 records with usage of dummy producer and then we check if the message count is correct. Similarly, in consumer we wait for 4 iterations of our consumer and check the number of read messages.

The program compiles correctly, with all tests succeeding which confirms correct execution of our producers and consumers:

```
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ spotifykafkaproducerconsumer ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running at.ac.fhsalzburg.bde.app.SpotifyKafkaTest
=== SPOTIFY PRODUCER ===
[Sent to groupB04.spotifytempos] Value = 6KbQ3uYMLKb5jDxLF7wYDD,1928,118.469
[Sent to groupB04.spotifytempos] Value = 3yZj8i9TwsWAVmBWKdMbaa,1930,185.062
[Sent to groupB04.spotifytempos] Value = 4h8gqHQ6pdgXWqElxBSGeo,1935,126.048
[Sent to groupB04.spotifytempos] Value = 4EafdWyeDb7Xlvhs7Xgca8,1936,100.55
[Sent to groupB04.spotifytempos] Value = 2TCHu1TbS9GNE3ULxJWqwa,1940,80.211
[Sent to groupB04.spotifytempos] Value = 528sYMqo03AtUAQcj1wlHS,1942,96.78
[Sent to groupB04.spotifytempos] Value = 2q5FlK299A35aDN4ufo6XT,1945,127.172
[Sent to groupB04.spotifytempos] Value = 50fu6sdcaguNf0Xm5cPOH9,1946,102.343
[Sent to groupB04.spotifytempos] Value = 56Q2aNM1WWQhXwcWQYbuHk,1947,123.382
[Sent to groupB04.spotifytempos] Value = 48U33WgDS5EEUKEhMT9Zwt,1948,85.126
Closing ...
Done.
=== SPOTIFY CONSUMER ===
=> Subscribed to groupB04.spotifytempos
==> Waiting for messages...
==> After poll (enter to exit); records: 10
==> After poll (enter to exit); records: 10
==> After poll (enter to exit); records: 10
shutting down ...
==> After poll (enter to exit); records: 10
done.
```

5. Results

5.1. Producer

After running the implemented java producer with command:

```
mvn exec:java@temposproducer
```

we can observe that the producer sends messages to topic groupB04.spotifytempos as expected - songs with different years with properly formatted values. The sample of console output look like this:

```
[INFO] --- exec-maven-plugin:1.2.1:java (temposproducer) @ spotifykafkaproducerconsumer ---
Using brokers: node1:9092, node2:9092, node3:9092, node4:9092
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=== SPOTIFY PRODUCER ===
[Sent to groupB04.spotifytempos] Value = 6KbQ3uYMLKb5jDxLF7wYDD,1928,118.469
[Sent to groupB04.spotifytempos] Value = 3yZj8i9TwsWAVmBWKdMbaa,1930,185.062
[Sent to groupB04.spotifytempos] Value = 4h8gqHQ6pdgXWqElxBSGeo,1935,126.048
[Sent to groupB04.spotifytempos] Value = 4EafdWyeDb7Xlvhs7Xgca8,1936,100.55
[Sent to groupB04.spotifytempos] Value = 2TCHu1TbS9GNE3ULxJWqwa,1940,80.211
[Sent to groupB04.spotifytempos] Value = 528sYMqo03AtUAQcj1wlHS,1942,96.78
[Sent to groupB04.spotifytempos] Value = 2q5FlK299A35aDN4ufo6XT,1945,127.172
[Sent to groupB04.spotifytempos] Value = 50fu6sdcaguNf0Xm5cPOH9,1946,102.343
[Sent to groupB04.spotifytempos] Value = 56Q2aNM1WWQhXwcWQYbuHk,1947,123.382
[Sent to groupB04.spotifytempos] Value = 48U33WgDS5EEUKEhMT9Zwt,1948,85.126
[Sent to groupB04.spotifytempos] Value = 1qBI0F4qSYG31GdVw5vIu4,1949,116.155
[Sent to groupB04.spotifytempos] Value = 0vXEDaNqu50bi9Va572lQ9,1950,82.146
[Sent to groupB04.spotifytempos] Value = 1KIOppMpY7eSPzhG7cob7n,1951,108.563
[Sent to groupB04.spotifytempos] Value = 1EUH4OBQ5Eac5mxyHASc4A,1952,84.569
[Sent to groupB04.spotifytempos] Value = 12CWFCFAWkkc3vPOG19dVK,1953,63.502
[Sent to groupB04.spotifytempos] Value = 6Limxo2wJuYL2WrtdewfKg,1954,73.579
[Sent to groupB04.spotifytempos] Value = 6UkJIDGuGv1CyJeAakb3Jj,1955,62.195
[Sent to groupB04.spotifytempos] Value = 0irnJklasJ1rtQorlM4f20,1956,92.163
[Sent to groupB04.spotifytempos] Value = 69RoHsh1PqtqKnrWy36MZl,1957,119.566
[Sent to groupB04.spotifytempos] Value = 38YV43s75rETyFK70bQKPw,1958,129.561
[Sent to groupB04.spotifytempos] Value = 4grD2NkyBWSjo15u0muqA2,1959,82.88799999999998
[Sent to groupB04.spotifytempos] Value = 7BjfhLDrFRE6omVIMzY8Fq,1960,87.333
[Sent to groupB04.spotifytempos] Value = 2Vwxs0FYxqBOGIzzXoZRq8,1961,88.64399999999998
[Sent to groupB04.spotifytempos] Value = 6KMZLiIL7aocI79d5BW2Rt,1962,104.0
[Sent to groupB04.spotifytempos] Value = 2Rj2FPiSEN8yw6Gi0iqKcK,1963,127.707
[Sent to groupB04.spotifytempos] Value = 5W6NMQ8j1cdzjDQTm4kvJi,1964,152.007
[Sent to groupB04.spotifytempos] Value = 6WGgyJqASKK3cQlu11pFYC,1965,86.85600000000002
[Sent to groupB04.spotifytempos] Value = 2XClue0OiR8tBFuObNCvZ0,1967,96.397
[Sent to groupB04.spotifytempos] Value = 5HB3pcue65MRxXbIDSXpTA,1968,135.289
[Sent to groupB04.spotifytempos] Value = 3rG2OYVaUNbWMKmONwXf4H,1969,94.084
[Sent to groupB04.spotifytempos] Value = 1vGqPTdYS0G7Bs1Ltru7Na,1970,113.464
[Sent to groupB04.spotifytempos] Value = 7LTf7FTOixNwfYK8gmig6j,1971,75.845
```

5.2. Consumer

After sending messages to Kafka with the producer, we can use our consumer to read those messages. It can be done with the command:

```
mvn exec:java@temposconsumer
```

Then, we can see that our consumer is reading the messages as expected. Moreover, the messages with the given year are stored on the separate partitions. Sample of the console output looked like this:

```
[INFO] --- exec-maven-plugin:1.2.1:java (temposconsumer) @ spotifykafkaproducerconsumer ---
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Press any key to exit
=== SPOTIFY CONSUMER ===
=> Subscribed to groupB04.spotifytempos
==> Waiting for messages...
==> After poll (enter to exit); records: 0
==> After poll (enter to exit); records: 0
==> After poll (enter to exit); records: 35
[Partition: 0, Offset: 14, MsgKey: null] Id: 3yZj8i9TwsWAVmBWKdMbaa, Year: 1930, Tempo: 185.062;
[Partition: 0, Offset: 15, MsgKey: null] Id: 4h8gqHQ6pdgXWqElxBSGeo, Year: 1935, Tempo: 126.048;
[Partition: 0, Offset: 16, MsgKey: null] Id: 2TCHu1TbS9GNE3ULxJWqwa, Year: 1940, Tempo: 80.211;
[Partition: 0, Offset: 17, MsgKey: null] Id: 2q5FlK299A35aDN4ufo6XT, Year: 1945, Tempo: 127.172;
[Partition: 0, Offset: 18, MsgKey: null] Id: 56Q2aNM1WWQhXwcWQYbuHk, Year: 1947, Tempo: 123.382;
[Partition: 0, Offset: 19, MsgKey: null] Id: 0vXEDaNqu50bi9Va572lQ9, Year: 1950, Tempo: 82.146;
[Partition: 0, Offset: 20, MsgKey: null] Id: 0irnJklasJ1rtQorlM4f20, Year: 1956, Tempo: 92.163;
[Partition: 0, Offset: 21, MsgKey: null] Id: 38YV43s75rETyFK70bQKPw, Year: 1958, Tempo: 129.561;
[Partition: 0, Offset: 22, MsgKey: null] Id: 7BjfhLDrFRE6omVIMzY8Fq, Year: 1960, Tempo: 87.333;
[Partition: 0, Offset: 23, MsgKey: null] Id: 6KMZLiIL7aocI79d5BW2Rt, Year: 1962, Tempo: 104.0;
[Partition: 0, Offset: 24, MsgKey: null] Id: 6WGgyJqASKK3cQlu11pFYC, Year: 1965, Tempo: 86.85600000000002;
[Partition: 0, Offset: 25, MsgKey: null] Id: 5HB3pcue65MRxXbIDSXpTA, Year: 1968, Tempo: 135.289;
[Partition: 0, Offset: 26, MsgKey: null] Id: 1vGqPTdYS0G7Bs1Ltru7Na, Year: 1970, Tempo: 113.464;
[Partition: 0, Offset: 27, MsgKey: null] Id: 3cmb7mFWij3I2hKfGSRE3z, Year: 1974, Tempo: 170.85;
[Partition: 2, Offset: 16, MsgKey: null] Id: 6KbQ3uYMLKb5jDxLF7wYDD, Year: 1928, Tempo: 118.469;
[Partition: 2, Offset: 17, MsgKey: null] Id: 4EafdWyeDb7Xlvhs7Xgca8, Year: 1936, Tempo: 100.55;
[Partition: 2, Offset: 18, MsgKey: null] Id: 50fu6sdcaguNf0Xm5cPOH9, Year: 1946, Tempo: 102.343;
[Partition: 2, Offset: 19, MsgKey: null] Id: 48U33WgDS5EEUKEhMT9Zwt, Year: 1948, Tempo: 85.126;
[Partition: 2, Offset: 20, MsgKey: null] Id: 1EUH4OBQ5Eac5mxyHASc4A, Year: 1952, Tempo: 84.569;
[Partition: 2, Offset: 21, MsgKey: null] Id: 6Limxo2wJuYL2WrtdewfKg, Year: 1954, Tempo: 73.579;
[Partition: 2, Offset: 22, MsgKey: null] Id: 69RoHsh1PqtqKnrWy36MZl, Year: 1957, Tempo: 119.566;
[Partition: 2, Offset: 23, MsgKey: null] Id: 2Vwxs0FYxqBOGIzzXoZRq8, Year: 1961, Tempo: 88.64399999999998;
[Partition: 2, Offset: 24, MsgKey: null] Id: 5W6NMQ8j1cdzjDQTm4kvJi, Year: 1964, Tempo: 152.007;
[Partition: 2, Offset: 25, MsgKey: null] Id: 2XClue0OiR8tBFuObNCvZ0, Year: 1967, Tempo: 96.397;
[Partition: 2, Offset: 26, MsgKey: null] Id: 3rG2OYVaUNbWMKmONwXf4H, Year: 1969, Tempo: 94.084;
[Partition: 2, Offset: 27, MsgKey: null] Id: 7LTf7FTOixNwfYK8gmig6j, Year: 1971, Tempo: 75.845;
[Partition: 2, Offset: 28, MsgKey: null] Id: 5UH9Nzv4AxSEmzcCOxpOzn, Year: 1973, Tempo: 77.751;
[Partition: 1, Offset: 15, MsgKey: null] Id: 528sYMqo03AtUAQcj1wlHS, Year: 1942, Tempo: 96.78;
[Partition: 1, Offset: 16, MsgKey: null] Id: 1qBI0F4qSYG31GdVw5vIu4, Year: 1949, Tempo: 116.155;
[Partition: 1, Offset: 17, MsgKey: null] Id: 1KIOppMpY7eSPzhG7cob7n, Year: 1951, Tempo: 108.563;
[Partition: 1, Offset: 18, MsgKey: null] Id: 12CWFCFAWkkc3vPOG19dVK, Year: 1953, Tempo: 63.502;
[Partition: 1, Offset: 19, MsgKey: null] Id: 6UkJIDGuGv1CyJeAakb3Jj, Year: 1955, Tempo: 62.195;
[Partition: 1, Offset: 20, MsgKey: null] Id: 4grD2NkyBWSjo15u0muqA2, Year: 1959, Tempo: 82.88799999999998;
[Partition: 1, Offset: 21, MsgKey: null] Id: 2Rj2FPiSEN8yw6Gi0iqKcK, Year: 1963, Tempo: 127.707;
[Partition: 1, Offset: 22, MsgKey: null] Id: 30t7Yif8FyeCANkW2Beii3, Year: 1972, Tempo: 104.411;
==> After poll (enter to exit); records: 0
```

Moreover, we can observe that since we don't have any message key in our messages (all are null), they are spread (more or less) evenly on our 3 partitions.

Use Case 1 - Songs per main artist

1. Topic Structure

Our second use case was about counting the number of songs of a given artist. However, to make this example more suitable for real time processing, we decided to make a producer

which fakes a stream of currently played songs - and count which artists were most popular across a given time window.

Similarly as before, the specific message keys are not suitable for that use case - it does not matter on which partition the given song is stored, we only want to know the artists of currently played songs.

Finally the command to create the topic was as follows:

```
kafka-topics.sh --create --zookeeper node1:2181 \
--replication-factor 3 --partitions 3 --topic
groupB04.spotifyartists
```

2. Consumer groups

2.1 Define how you would arrange consumers and consumer groups

Once again, some specific arrangement of consumers in groups is not suitable for this use case, one consumer is enough for taking songs and counting the artists from a given period.

3. Implementation

3.1. Java-Producer

In this use case, the relevant data are only the artists that are the authors of a given song. Therefore, similarly as before, we sent the messages in the form of comma separated values with the structure “[id],[artist1],[artist2],[...]”.

As before, for our implementation we used the classes from the demo project and changed them to meet our needs. Therefore, the “run” method of this producer looks as follows:

```java
@Override
public void run() {
    System.out.println("=== SPOTIFY PRODUCER ===");
    try {
        FileReader fr = new FileReader(INPUT_FILE);
        BufferedReader br = new BufferedReader(fr);

        String line;
        ProducerRecord<Long, String> record;
        long key;
        String value;
        int skipLines = 1;
        int sentMessages = 0;

        while ((line = br.readLine()) != null) {
            // Skipping column names row
            if (skipLines-- > 0) continue;

            // Exporting id, and artists from the row
            String[] parts = line.toString().split(",(?=(?:[^\"]*\"[^\"]*\")*[^\"]*$)", -1);
            if (parts.length != ROW_LENGTH) continue;
            String artists = parts[ARTISTS_COLUMN];
            String id = parts[ID_COLUMN];

            artists = artists.replaceAll("\"", "");
            artists = artists.replaceAll("'", "");
            artists = artists.replaceAll(", ", ",");
            artists = artists.replaceAll(" ,", ",");

            if (artists.length() < 2) continue;
            artists = artists.substring(1, artists.length() - 1);

            value = getRecordValue(id, artists);
            record = new ProducerRecord<>(TOPIC, value);
            producer.send(record);
            System.out.printf("[Sent to %s] Value = %s\n",
                    TOPIC,
                    value);

            // Used only for testing
            if (messageCount != -1) {
                if (++sentMessages == messageCount) break;
            }

            // Delay between records sending
            try {
                Thread.sleep(DELAY_BASE);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Closing ...");
        producer.flush();
        producer.close();

        System.out.println("Done.");
    } catch(FileNotFoundException e) {
        System.out.println("File not found");
        return;
    } catch(IOException e) {
        System.out.println("IO exception");
        return;
    }
```

And its main function:

```java
public static void main(String[] args) {
    try {
        new Thread(new SpotifyArtistsProducer(Helper.createProducer(CLIENT_ID))).start();
    } catch (NumberFormatException e) {
        System.err.println("Usage: SpotifyArtistsProducer");
        System.exit(2);
    }
}
```

3.2. Java-Consumer

Once again we implemented a java consumer, which takes a record produced by our java producer and prints them to the console with some other relevant data. Its “run” method looks like this:

```
public void run() {
    System.out.println("=== SPOTIFY ARTISTS CONSUMER ===");
    try {
        System.out.println("=> Subscribed to " + SpotifyArtistsProducer.TOPIC);
        System.out.println("==> Waiting for messages...");
        while (!closed.get()) {
            ConsumerRecords<Long, String> records = consumer.poll(Duration.ofMillis(POLL_DURATION_MS));
            System.out.println("==> After poll (enter to exit); records: " + records.count());
            for (ConsumerRecord<Long, String> record : records) {
                String[] values;
                values = getValuesFromArray(record.value());
                String id = values[0];
                String[] artists = Arrays.copyOfRange(values, 1, values.length);

                // Wrong record format
                if (values.length != 3) continue;

                System.out.printf(
                        "[Partition: %d, Offset: %d] Id: %s, Artists: %s;\n",
                        record.partition(),
                        record.offset(),
                        values[0],
                        String.join(",", artists));
            }
        }
    } catch (WakeupException e) {
        System.out.println(e.toString());
        /* Ignore exception if closing */
        if (!closed.get())
            throw e;
    } finally {
        consumer.close();
    }
}
```

And its main function:

```
public static void main(String[] args) throws InterruptedException, IOException {

    SpotifyArtistsConsumer r = new SpotifyArtistsConsumer(Helper.createConsumer(CLIENTID, GROUPID));
    Thread t = new Thread(r);
    t.start();

    System.out.println("Press any key to exit");
    System.in.read();

    System.out.println("Shutting down ...");
    r.shutdown();
    t.join();
    System.out.println("Done.");
}
```

4. Testing

For this use case we made identical tests as for the previous one.

The program compiles correctly, with all tests succeeding which confirms correct execution of our producers and consumers:

```
[INFO] --- maven-surefire-plugin:2.22.1:test (default-test) @ spotifykafkaproducerconsumer ---
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running at.ac.fhsalzburg.bde.app.SpotifyKafkaTest
=== SPOTIFY PRODUCER ===
[Sent to groupB04.spotifyartists] Value = 6KbQ3uYMLKb5jDxLF7wYDD,Carl Woitschach
[Sent to groupB04.spotifyartists] Value = 3yZj8i9TwsWAVmBWKdMbaa,Ignacio Corsini
[Sent to groupB04.spotifyartists] Value = 4h8gqHQ6pdgXWqElxBSGeo,Sinclair Lewis,Frank Arnold
[Sent to groupB04.spotifyartists] Value = 4EafdWyeDb7Xlvhs7Xgca8,Ludwig van Beethoven,Wiener Philharmoniker,Herbert von Karaja
[Sent to groupB04.spotifyartists] Value = 2TCHu1TbS9GNE3ULxJWqwa,Ying Yin Wu
[Sent to groupB04.spotifyartists] Value = 528sYMqo03AtUAQcj1wlHS,Charles Koechlin,Bidu Sayão
[Sent to groupB04.spotifyartists] Value = 2q5FlK299A35aDN4ufo6XT,Эрих Мария Ремарк
[Sent to groupB04.spotifyartists] Value = 50fu6sdcaguNf0Xm5cPOH9,Robert Schumann,Maryla Jonas
[Sent to groupB04.spotifyartists] Value = 56Q2aNM1WWQhXwcWQYbuHk,Zafar Khurshid
[Sent to groupB04.spotifyartists] Value = 48U33WgDS5EEUKEhMT9Zwt,John Stafford Smith,Igor Stravinsky,CBC Symphony Orchestra
Closing ...
Done.
=== SPOTIFY CONSUMER ===
=> Subscribed to groupB04.spotifytempos
==> Waiting for messages...
==> After poll (enter to exit); records: 10
==> After poll (enter to exit); records: 10
==> After poll (enter to exit); records: 10
shutting down ...
==> After poll (enter to exit); records: 10
done.
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.576 s - in at.ac.fhsalzburg.bde.app.SpotifyKafkaTest
```

5. Results

5.1. Producer

After running the implemented java producer with command:

```
mvn exec:java@artistsproducer
```

```
[INFO] --- exec-maven-plugin:1.2.1:java (artistsproducer) @ spotifykafkaproducerconsumer ---
Using brokers: node1:9092, node2:9092, node3:9092, node4:9092
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
=== SPOTIFY PRODUCER ===
[Sent to groupB04.spotifyartists] Value = 6KbQ3uYMLKb5jDxLF7wYDD,Carl Woitschach
[Sent to groupB04.spotifyartists] Value = 3yZj8i9TwsWAVmBWKdMbaa,Ignacio Corsini
[Sent to groupB04.spotifyartists] Value = 4h8gqHQ6pdgXWqElxBSGeo,Sinclair Lewis,Frank Arnold
[Sent to groupB04.spotifyartists] Value = 4EafdWyeDb7Xlvhs7Xgca8,Ludwig van Beethoven,Wiener Philharmoniker,Herbert von Karajan
[Sent to groupB04.spotifyartists] Value = 2TCHu1TbS9GNE3ULxJWqwa,Ying Yin Wu
[Sent to groupB04.spotifyartists] Value = 528sYMqo03AtUAQcj1wlHS,Charles Koechlin,Bidu Sayão
[Sent to groupB04.spotifyartists] Value = 2q5FlK299A35aDN4ufo6XT,Эрих Мария Ремарк
[Sent to groupB04.spotifyartists] Value = 50fu6sdcaguNf0Xm5cPOH9,Robert Schumann,Maryla Jonas
[Sent to groupB04.spotifyartists] Value = 56Q2aNM1WWQhXwcWQYbuHk,Zafar Khurshid
[Sent to groupB04.spotifyartists] Value = 48U33WgDS5EEUKEhMT9Zwt,John Stafford Smith,Igor Stravinsky,CBC Symphony Orchestra
[Sent to groupB04.spotifyartists] Value = 1qBI0F4qSYG31GdVw5vIu4,Sergei Prokofiev,Isaac Stern
[Sent to groupB04.spotifyartists] Value = 0vXEDaNqu50bi9Va572lQ9,Νίκος Τζουγάνος,Αλέκος Καραβίτης
[Sent to groupB04.spotifyartists] Value = 1KIOppMpY7eSPzhG7cob7n,Johannes Brahms,Pierre Monteux
[Sent to groupB04.spotifyartists] Value = 1EUH4OBQ5Eac5mxyHASc4A,Richard Wagner,Inge Borkh,GÜnther Treptow,Hans Hotter,Ira Malaniuk,
[Sent to groupB04.spotifyartists] Value = 12CWFCFAWkkc3vPOG19dVK,Jairam Shiledar,Lata Mangeshkar
[Sent to groupB04.spotifyartists] Value = 6Limxo2wJuYL2WrtdewfKg,Oscar Peterson
[Sent to groupB04.spotifyartists] Value = 6UkJIDGuGv1CyJeAakb3Jj,Dave Brubeck
[Sent to groupB04.spotifyartists] Value = 0irnJklasJ1rtQorlM4f20,Lord Cristo
[Sent to groupB04.spotifyartists] Value = 69RoHsh1PqtqKnrWy36MZl,Johann Sebastian Bach,Glenn Gould
[Sent to groupB04.spotifyartists] Value = 38YV43s75rETyFK70bQKPw,Billie Holiday,Ray Ellis & His Orchestra
[Sent to groupB04.spotifyartists] Value = 4grD2NkyBWSjo15u0muqA2,Ray Charles
[Sent to groupB04.spotifyartists] Value = 7BjfhLDrFRE6omVIMzY8Fq,Bobby Vee
[Sent to groupB04.spotifyartists] Value = 2Vwxs0FYxqBOGIzzXoZRq8,Little Richard
[Sent to groupB04.spotifyartists] Value = 6KMZLiIL7aocI79d5BW2Rt,Johnny Cash
[Sent to groupB04.spotifyartists] Value = 2Rj2FPiSEN8yw6Gi0iqKcK,Los Rurales De Gilberto Parra
[Sent to groupB04.spotifyartists] Value = 5W6NMQ8j1cdzjDQTm4kvJi,The Beach Boys
[Sent to groupB04.spotifyartists] Value = 6WGgyJqASKK3cQlu11pFYC,Astrud Gilberto
```

5.2. Consumer

After sending messages to Kafka with the producer, we can use our consumer to read those messages. It can be done with the command:

```
mvn exec:java@artistsconsumer
```

With the result:

```
[INFO] --- exec-maven-plugin:1.2.1:java (artistsconsumer) @ spotifykafkaproducerconsumer ---
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Press any key to exit
=== SPOTIFY ARTISTS CONSUMER ===
=> Subscribed to groupB04.spotifyartists
==> Waiting for messages...
==> After poll (enter to exit); records: 0
==> After poll (enter to exit); records: 47
[Partition: 2, Offset: 2] Id: 4h8gqHQ6pdgXWqElxBSGeo, Artists: Sinclair Lewis,Frank Arnold;
[Partition: 2, Offset: 4] Id: 12CWFCFAWkkc3vPOG19dVK, Artists: Jairam Shiledar,Lata Mangeshkar;
[Partition: 2, Offset: 8] Id: 3rG2OYVaUNbWMKmONwXf4H, Artists: Louis Hardin,Moondog;
[Partition: 2, Offset: 27] Id: 5ge5CjaeVPd4grpjRaAEFr, Artists: Franz Schubert,Isaac Stern;
[Partition: 2, Offset: 29] Id: 1bT5BmOrLruX7W3GAmpTIO, Artists: Georges Bizet,Fritz Reiner;
[Partition: 2, Offset: 30] Id: 04GWIVIjkinCpTbK8tsGmb, Artists: Franz Joseph Haydn,Fritz Reiner;
==> After poll (enter to exit); records: 94
[Partition: 1, Offset: 0] Id: 1qBI0F4qSYG31GdVw5vIu4, Artists: Sergei Prokofiev,Isaac Stern;
[Partition: 1, Offset: 1] Id: 0vXEDaNqu50bi9Va572lQ9, Artists: Νίκος Τζουγάνος,Αλέκος Καραβίτης;
[Partition: 1, Offset: 2] Id: 1KIOppMpY7eSPzhG7cob7n, Artists: Johannes Brahms,Pierre Monteux;
[Partition: 1, Offset: 5] Id: 69RoHsh1PqtqKnrWy36MZl, Artists: Johann Sebastian Bach,Glenn Gould;
[Partition: 1, Offset: 13] Id: 7bZGwyS6YQ5vE9DpdE1AYo, Artists: The Blues Brothers,Joe Gastwirt;
[Partition: 1, Offset: 23] Id: 7pda6TLAbVGxUjIUtbiNt0, Artists: ScHoolboy Q,E-40;
[Partition: 1, Offset: 26] Id: 3Bn5kqOOAIOBSrSk5qysTY, Artists: Franz Schubert,Yehudi Menuhin;
[Partition: 1, Offset: 28] Id: 0ycjDiexIJFW35vhsmphXm, Artists: Frédéric Chopin,Samson François;
[Partition: 0, Offset: 2] Id: 528sYMqo03AtUAQcj1wlHS, Artists: Charles Koechlin,Bidu Sayão;
[Partition: 0, Offset: 4] Id: 50fu6sdcaguNf0Xm5cPOH9, Artists: Robert Schumann,Maryla Jonas;
[Partition: 0, Offset: 8] Id: 38YV43s75rETyFK70bQKPw, Artists: Billie Holiday,Ray Ellis & His Orchestra;
[Partition: 0, Offset: 19] Id: 1by8KkLC21uRKPipwj3vjd, Artists: Newsies Ensemble,Newsies Additional Singing Cast;
[Partition: 0, Offset: 29] Id: 7E91ECLi2J4L21tEX0RBHn, Artists: Delacey,G-Eazy;
[Partition: 0, Offset: 30] Id: 5Y9SjlnRc7qWJVzVsXTwYR, Artists: Sinclair Lewis,Frank Arnold;
[Partition: 0, Offset: 31] Id: 5XjxV67r0A11lqEBP57Nhj, Artists: Parul Ghosh,H Khan Mastana;
[Partition: 0, Offset: 32] Id: 5ZCJs0I9tGlaDtPth4Gr3G, Artists: Frédéric Chopin,Maryla Jonas;
[Partition: 0, Offset: 34] Id: 15astVTsfxePM5eKl2K6rD, Artists: Mubarak Begum,Sulochana Kadam;
[Partition: 0, Offset: 41] Id: 4CRvbQ6GdqGB8QZXGisiYV, Artists: The Blues Brothers,Joe Gastwirt;
==> After poll (enter to exit); records: 0
```

References

In our solutions we used some code snippets from lecture slides and the course demo repository:

https://its-git.fh-salzburg.ac.at/bde/KafkaProducerConsumer

Big Data Engineering

Laboratories 4 Report

Group B04 members:

1. Jan Radzimiński
2. René Pichler
3. Michael Ramsteck

---

1. Introduction

The main purpose of this task was to implement Spark programs for processing the data streams coming for kafka. The goal was to implement the same algorithms as in Laboratories 2 - same calculations as in MapReduce functions. The input data for these algorithms was supposed to be “consumed” from the given kafka topic, which was created in Laboratories 3.

In our case, firstly, we used data about artists of a given song, produced to the topic “groupB04.spotifyartists” by Kafka producer in previous laboratories. These messages were to simulate real time songs played in a given time and their main artist. The goal of this task was to show most popular artists played in a given window. For simplicity of testing and results presentation we chose a window of 1 minute, although in the real-world case a longer period would be more reasonable. The result was supposed to be a bar plot of most listened to artists at the last minute (and how many times were they played).

Our second use case was calculation of average tempo of the tracks for a given decade - for example for “1980s” the calculation would result in average tempo of songs released in years 1980-1989. In our original MapReduce algorithm, this calculation was done by firstly calculating the median of tempos from a given year and then, in the second step, the average of tempos from a given decade (based on the results from the first step). In this task, we decided to omit the first step and directly calculate the average for all tempos from a given decade. This is, because currently multiple aggregations are not supported in Spark Structured Streaming. Anyway, the result will still be very similar to one obtained in laboratories 2.

2. Python Implementation

2.1. Use Case 1: Songs number for main artist

As described previously, the purpose of our first algorithm was to calculate most listened to artists in the past minute, and displaying results in the form of bar plot and table.

To do that, we firstly started a spark session and defined a data frame for reading the stream of messages from kafka topic “groupB04.spotifyartists”:

```
[20]: from pyspark.sql import SparkSession

      # Initialising Spark Session
      spark = SparkSession \
          .builder \
          .appName("Python Spark - Number of Spotify Songs for each Main Artist") \
          .getOrCreate()
```

```
[12]: # Used for stopping spark session
      #spark.stop()
```

```
[21]: # Constants defining streaming parammeters
      TOPIC = 'groupB04.spotifyartists'
      STARTING_OFFSETS = "latest"
```

```
[22]: # Starting spark data frame
      df = spark \
        .readStream \
        .format("kafka") \
        .option("kafka.bootstrap.servers", "node1:9092") \
        .option("subscribe", TOPIC) \
        .option("startingOffsets",STARTING_OFFSETS) \
        .option("failOnDataLoss", "false")\
        .option("checkpointLocation", "/tmp/b6") \
        .load()
```

For this use case we defined a windowing function for grouping data from a given minute. We set `STARTING_OFFSET` to `latest` so that only the most recent songs are taken into the calculations.

Then, we check that everything worked by printing the data frame schema:

```
[4]: # Checking if everything works by printing the schema of the data
     df.printSchema()
```

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Next part is about transforming the comma separated message into meaningful data. Firstly we create new columns by splitting the value of our message by ‘,’. Then we add two new columns to our data, where the first one is the id of the song and the second one is the main artists of a given song. Finally we create a new data frame “converted” where we cast our

columns into proper types and aliases, used by our aggregation algorithm. In this data frame, the timestamp is included for proper windowing of the data.

```
[5]: from pyspark.sql.functions import get_json_object, concat, lit, desc, window
from pyspark.sql.types import *
from pyspark.sql import functions as f

# Since our data is in CSV format, we get each variable by spliting our message by ','
split_col = f.split(df['value'], ',')

# Adding new columns to our data frame
df = df \
        .withColumn('UUID', split_col.getItem(0)) \
        .withColumn('artist', split_col.getItem(1))

# Converting our dataframe to values in correct types
converted = df \
    .selectExpr("UUID", "artist", "timestamp")\
    .select([\
            df["UUID"].cast(StringType()), \
            df["artist"].cast(StringType()).alias("Artist"), \
            df["timestamp"].cast(TimestampType()) \
            ])
```

Then we move to the main purpose of our algorithm, which is counting the songs. We grouped our data, firstly by window - in this case 1 minute (60 seconds) time period. We also decided to use a watermark of 30 seconds - the data arriving less than 30 seconds late will be still included in a proper calculation, and not if it arrived later. As the second group by argument we use column “Artists”. Then we count the number of our songs for each artist. We used funcion “count”. Finally we sort our data firstly by the window descending, so that most recent data is displayed on the graph and secondly by count number so that most popular artists are visible. Note that there might be situation where less than the 10 artists were played in a given frame, which would result in artists from the previous window showing up, however, we decided to leave it as it is since it would be highly unlikely for less than 10 artists being played even in such a short time as 1 minute.

```
[6]: window_duration = "60 seconds"
watermark_duration = "30 seconds"

groupdf = converted \
    .select("UUID", "artist", "timestamp") \
    .withWatermark("timestamp", watermark_duration) \
    .groupby(window("timestamp", window_duration), "Artist") \
    .count().alias("count_result") \
    .sort(desc("window"), desc("count")) \
    .limit(10)
```

Then we defined the function for printing data and displaying the plot for each epoch of data that comes into our stream. If there are no messages, the appropriate message is displayed. We plot the bar plot by using matplotlib python library. We also print the current data frame and display in which epoch are we:

```
[7]: %matplotlib inline
from pyspark.sql import SparkSession
from IPython import display
import matplotlib.pyplot as plt

# Defining the function that will be used for representing our data in real time (incrementaly)
def show_plot_and_df(epoch_frame, epoch_id):
    if (epoch_frame.count()==0):
        print("Waiting for new messages...")
        return
    # Printing plot with approprate labels
    epoch_frame.toPandas().plot(kind='bar', x='Artist', y='count')
    display.clear_output(wait=True)
    plt.title("Number of Spotify Songs for a given Artist")
    plt.ylabel("Number of Songs")
    plt.xlabel("Artist")
    plt.show()
    # Printing dataframe
    epoch_frame.show()
    # Printing current iteration
    print("Current Epoch: " + str(epoch_id))
```

Finally we run our function for the stream, for given processing time - 5 seconds. It means that every 5 second we will read the new data from the topic and display it to the user. Since we are using stream it is done incrementally - the results are visible in real time.

```
[8]: # Defining some parammeters
processingTime = "5 Seconds"
outputMode = "complete"

plt.rcParams['font.family'] = ['DejaVu Sans']
plt.rcParams['font.sans-serif'] = ['SimHei'] # for Chinese font (https://github.com/matplotlib/matplotlib/issues/15062)

# Finally, showing the data which updates every processingTime seconds
df = groupdf \
    .writeStream \
    .outputMode(outputMode) \
    .foreachBatch(show_plot_and_df) \
    .trigger(processingTime=processingTime) \
    .start() \
    .awaitTermination()
```

2.2. Use Case 2: Average tempos for decade

Our second algorithm was about calculating average tempo for a given decade. As mentioned earlier, in our original MapReduce function, it was firstly calculating the median of tempo for a given year, and then the average for a given decade. Here, we are calculating the average directly for each decade, since multiple aggregations are not currently supported in Spark Streams. This can be omitted by (for example) sending intermediate results to a new kafka topic and aggregating them once again in a different worker, although we decided to stick to the simpler case since the results will be similar anyway.

Similarly as in previous use case we start with creating Spark Session and our data frame which reads messages from topic “groupB04.spotifytempos” and checking correctness of reading by printing the schema:

[1]:
```
from pyspark.sql import SparkSession

# Initialising Spark Session
spark = SparkSession \
    .builder \
    .appName("Python Spark - Average Tempo of Spotify Songs for each Decade") \
    .getOrCreate()
```

[9]:
```
# Used for stopping spark session
#spark.stop()
```

[2]:
```
# Constants defining streaming parammeters
TOPIC = 'groupB04.spotifytempos'
STARTING_OFFSETS = "latest"
```

[3]:
```
# Starting spark data frame
df = spark \
  .readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", "node1:9092") \
  .option("subscribe", TOPIC) \
  .option("startingOffsets",STARTING_OFFSETS) \
  .option("failOnDataLoss", "false")\
  .option("checkpointLocation", "/tmp/b6") \
  .load()
```

[4]:
```
# Checking if everything works by printing the schema of the data
df.printSchema()
```

```
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)
```

Then, as in the previous use case, we split our comma separated value of our messages by ‘,’ and create appropriate new columns. In this use case, the message value was produced in the following form: `song_id,year,tempo`. Additionally, we create and add the new column which represents the decade for a given year. For example, for a song released in 1985, it will have the corresponding decade in form: “1980s”. Then we cast all relevant values into appropriate types.

[15]:
```
from pyspark.sql.functions import get_json_object, concat, lit
from pyspark.sql.types import *
from pyspark.sql import functions as f

# Since our data is in CSV format, we get each variable by spliting our message by ','
split_col = f.split(df['value'], ',')

# Constructing the decade string for each song which will be used later in aggregation
decade = concat(split_col.getItem(1)[0:3], lit('0s'))

# Adding new columns to our data frame
df = df \
        .withColumn('UUID', split_col.getItem(0)) \
        .withColumn('year', split_col.getItem(1)) \
        .withColumn('tempo', split_col.getItem(2)) \
        .withColumn('decade', decade) \

# Converting our dataframe to values in correct types
converted = df \
    .selectExpr("UUID", "year", "tempo", "offset", "decade")\
    .select([\
            df["UUID"].cast(StringType()), \
            df["year"].cast(IntegerType()), \
            df["tempo"].cast(DoubleType()), \
            df["decade"].cast(StringType()).alias("Decade")\
            ])
```

Then, we run our aggregation function, where we group by the decade and calculate average tempo for each of those. Additionally, we added a column with counted years, to see how many years from a given decade had songs released and a column with counted songs, to see how many songs were used for calculating a given value.

```
[6]: # Doing the calculations - grouping by the decade and calculating the tempo average
groupdf = converted \
    .select("UUID", "year", "Decade", "tempo") \
    .groupby("Decade") \
    .agg( \
         # Calculating average tempo of decade rounded to 2 decimal places
         f.round(f.avg("tempo"), 2).alias("Average Tempo"), \
         # Adding to df some additional data about number of songs used for calculation
         f.approxCountDistinct("UUID").alias("Songs Count"), \
         # Adding to df some additional data about number of years from given decade used for calculation
         f.approxCountDistinct("year").alias("Years Count")) \
    .sort("Decade")
```

Finally, as in previous use case, we define a function for displaying our data from each epoch in a form of a plot and a table:

```
[9]: %matplotlib inline
from pyspark.sql import SparkSession
from IPython import display
import matplotlib.pyplot as plt

# Defining the function that will be used for representing our data in real time (incrementaly)
def show_plot_and_df(epoch_frame, epoch_id):
    if (epoch_frame.count()==0):
        print("Waiting for new messages...")
        return
    # Printing plot with approprate labels
    epoch_frame.toPandas().plot(kind='bar', x='Decade', y='Average Tempo')
    display.clear_output(wait=True)
    plt.title("Average Tempo of Spotify Songs across Decades")
    plt.ylabel("Tempo in BPM")
    plt.xlabel("Decade")
    plt.gca().set_ylim([70,160])
    plt.show()
    # Printing dataframe
    epoch_frame.show()
    # Printing current iteration
    print("Current Epoch: " + str(epoch_id))
```

And we run it on our stream with 5 seconds processing time:

```
[ ]: # Defining some parammeters
processingTime = "5 Seconds"
outputMode = "complete"

# Finally, showing the data which updates every processingTime seconds
df = groupdf \
    .writeStream \
    .outputMode(outputMode) \
    .foreachBatch(show_plot_and_df) \
    .trigger(processingTime=processingTime) \
    .start() \
    .awaitTermination()
```

3. Results

3.1. Use Case 1: Songs number for main artist

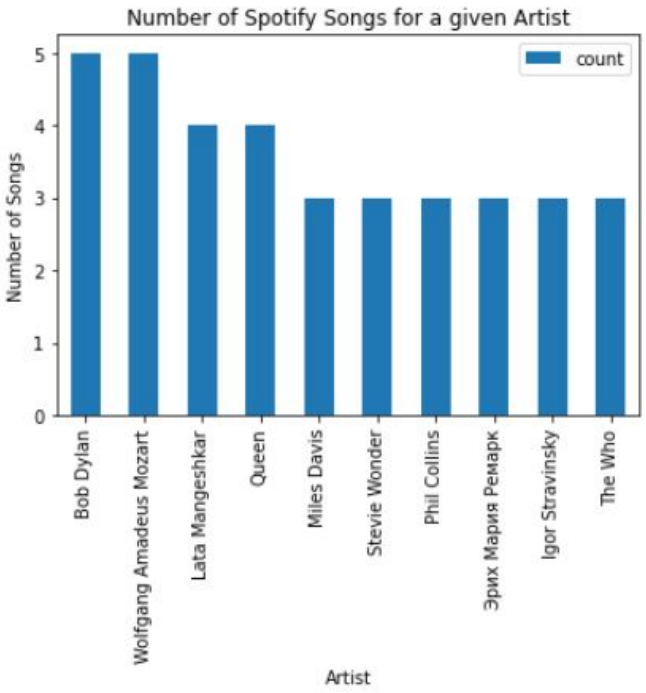
After starting our program we can see that it is waiting for new messages to be added to the stream.

```
[*]: # Defining some parammeters
processingTime = "5 Seconds"
outputMode = "complete"

plt.rcParams['font.family'] = ['DejaVu Sans']
plt.rcParams['font.sans-serif'] = ['SimHei'] # for Chinese font (https://github.com/matplotlib/matplotlib/issues/15062)

# Finally, showing the data which updates every processingTime seconds
df = groupdf \
    .writeStream \
    .outputMode(outputMode) \
    .foreachBatch(show_plot_and_df) \
    .trigger(processingTime=processingTime) \
    .start() \
    .awaitTermination()

Waiting for new messages...
```

Then, we run our Java Kafka Producer to simulates a real time songs plays which artists are sent as messages into the topic:

```
[Sent to groupB04.spotifyartists] Value = 6KbQ3uYMLKb5jDxLF7wYDD,Carl Woitschach
[Sent to groupB04.spotifyartists] Value = 3yZj8i9TwsWAVmBWKdMbaa,Ignacio Corsini
[Sent to groupB04.spotifyartists] Value = 4h8gqHQ6pdgXWqElxBSGeo,Sinclair Lewis,Frank Arnold
[Sent to groupB04.spotifyartists] Value = 4EafdWyeDb7Xlvhs7Xgca8,Ludwig van Beethoven,Wiener Philharmonik
er,Herbert von Karajan
[Sent to groupB04.spotifyartists] Value = 2TCHu1TbS9GNE3ULxJWqwa,Ying Yin Wu
[Sent to groupB04.spotifyartists] Value = 528sYMqoO3AtUAQcj1wlHS,Charles Koechlin,Bidu SayÃŁo
[Sent to groupB04.spotifyartists] Value = 2q5FlK299A35aDN4ufo6XT,Ð-Ñ?Ð¸Ñ? ÐšÐ°Ñ?Ð¸ÑZ Ð Ð?ÐŁÐ°Ñ?Ðş
[Sent to groupB04.spotifyartists] Value = 5Ofu6sdcaguNfOXm5cPOH9,Robert Schumann,Maryla Jonas
[Sent to groupB04.spotifyartists] Value = 56Q2aNM1WWQhXwcWQYbuHk,Zafar Khurshid
[Sent to groupB04.spotifyartists] Value = 48U33WgDS5EEUKEhMT9Zwt,John Stafford Smith,Igor Stravinsky,CBC
Symphony Orchestra
[Sent to groupB04.spotifyartists] Value = 1qBIOF4qSYG31GdVw5vIu4,Sergei Prokofiev,Isaac Stern
[Sent to groupB04.spotifyartists] Value = OvXEDaNqu5Obi9Va572lQ9,ÎŧÎŽÎşÎžÐ? Î¤Î?ÎžÐ?Î¹Î¬Î ÎžÐ?,Î?Î»Î-ÎşÎž
Ð? ÎšÎ?Ð?Î?Î¸ÎžÐ?Î?Ð?
[Sent to groupB04.spotifyartists] Value = 1KIOppMpY7eSPzhG7cob7n,Johannes Brahms,Pierre Monteux
[Sent to groupB04.spotifyartists] Value = 1EUH4OBQ5Eac5mxyHASc4A,Richard Wagner,Inge Borkh,GÄĽnther Trept
ow,Hans Hotter,Ira Malaniuk,Josef Greindl And Astrid Varnay
[Sent to groupB04.spotifyartists] Value = 12CWFCFAWkkc3vPOG19dVK,Jairam Shiledar,Lata Mangeshkar
[Sent to groupB04.spotifyartists] Value = 6Limxo2wJuYL2WrtdewfKg,Oscar Peterson
[Sent to groupB04.spotifyartists] Value = 6UkJIDGuGv1CyJeAakb3Jj,Dave Brubeck
[Sent to groupB04.spotifyartists] Value = OirnJklasJ1rtQorlM4f2O,Lord Cristo
[Sent to groupB04.spotifyartists] Value = 69RoHsh1PqtqKnrWy36MZl,Johann Sebastian Bach,Glenn Gould
[Sent to groupB04.spotifyartists] Value = 38YV43s75rETyFK7ObQKPw,Billie Holiday,Ray Ellis & His Orchestra
```

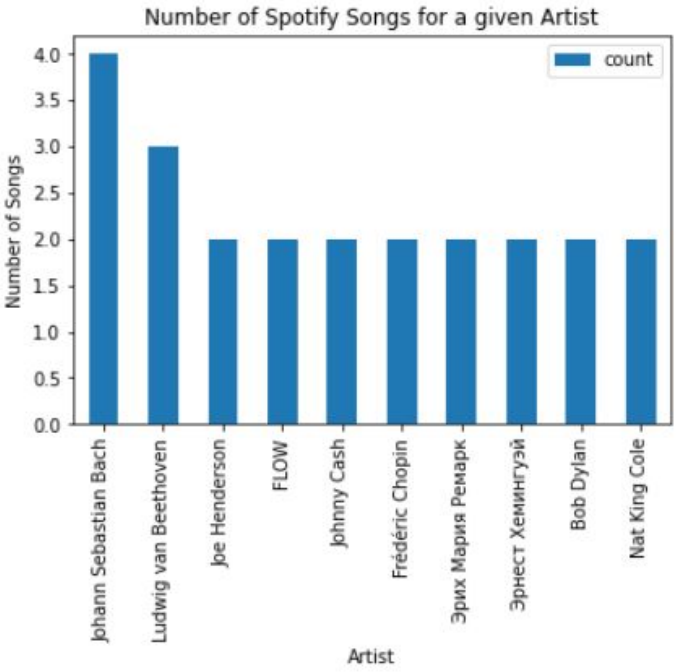
It results in our function updating and displaying appropriate graph and data frame:




```
+--------------------+--------------------+-----+
|              window|              Artist|count|
+--------------------+--------------------+-----+
|[2020-12-29 20:46...|           Bob Dylan|    5|
|[2020-12-29 20:46...|Wolfgang Amadeus ...|    5|
|[2020-12-29 20:46...|               Queen|    4|
|[2020-12-29 20:46...|     Lata Mangeshkar|    4|
|[2020-12-29 20:46...|   Эрих Мария Ремарк|    3|
|[2020-12-29 20:46...|     Igor Stravinsky|    3|
|[2020-12-29 20:46...|             The Who|    3|
|[2020-12-29 20:46...|        Phil Collins|    3|
|[2020-12-29 20:46...|         Miles Davis|    3|
|[2020-12-29 20:46...|       Stevie Wonder|    3|
+--------------------+--------------------+-----+

Current Epoch: 8
```

We can observe that our data is updating every 5 seconds, as expected.

Also after waiting 1 minute, the window changes and we can see the new updated data about the artists from the next minute:




```
+--------------------+--------------------+-----+
|              window|              Artist|count|
+--------------------+--------------------+-----+
|[2020-12-29 20:47...|Johann Sebastian ...|    4|
|[2020-12-29 20:47...|Ludwig van Beethoven|    3|
|[2020-12-29 20:47...|       Joe Henderson|    2|
|[2020-12-29 20:47...|                FLOW|    2|
|[2020-12-29 20:47...|         Johnny Cash|    2|
|[2020-12-29 20:47...|     Frédéric Chopin|    2|
|[2020-12-29 20:47...|   Эрих Мария Ремарк|    2|
|[2020-12-29 20:47...|    Эрнест Хемингуэй|    2|
|[2020-12-29 20:47...|           Bob Dylan|    2|
|[2020-12-29 20:47...|       Nat King Cole|    2|
+--------------------+--------------------+-----+

Current Epoch: 17
```

We can see that the artists with the highest number of plays were Bob Dylan in the first minute and Johann Sebastian Bach in the second one - everything is working as expected.

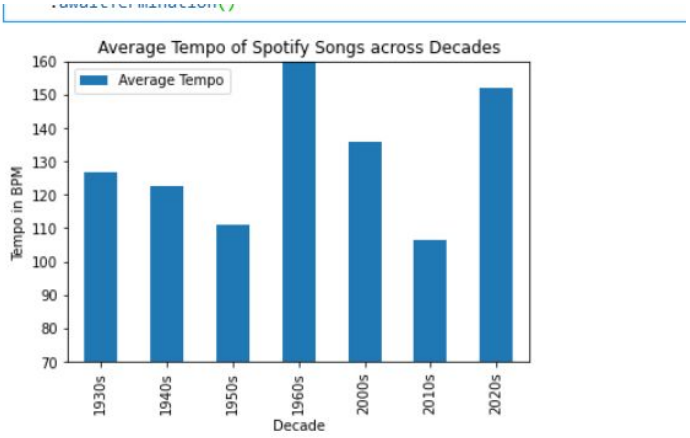
3.2. Use Case 2: Average tempos for decade

As in the previous case, our program starts with a message informing the user about waiting for new messages. Similarly, the program might take all of the previous messages into calculations if variable `STARTING_OFFSETS` is set to `earliest` while defining the initial dataframe, although we stick to the `latest` setting for simplicity of testing and results presentation.

After producing messages with our Kafka Producer:

```
[Sent to groupB04.spotifytempos] MsgKey = 1935, Value = 4h8gqHQ6pdgXWqElxBSGeo,1935,126.048
[Sent to groupB04.spotifytempos] MsgKey = 1936, Value = 4EafdWyeDb7Xlvhs7Xgca8,1936,100.55
[Sent to groupB04.spotifytempos] MsgKey = 1940, Value = 2TCHu1TbS9GNE3ULxJWqwa,1940,80.211
[Sent to groupB04.spotifytempos] MsgKey = 1942, Value = 528sYMqoO3AtUAQcj1wlHS,1942,96.78
[Sent to groupB04.spotifytempos] MsgKey = 1945, Value = 2q5FlK299A35aDN4ufo6XT,1945,127.172
[Sent to groupB04.spotifytempos] MsgKey = 1946, Value = 5Ofu6sdcaguNfOXm5cPOH9,1946,102.343
[Sent to groupB04.spotifytempos] MsgKey = 1947, Value = 56Q2aNM1WWQhXwcWQYbuHk,1947,123.382
[Sent to groupB04.spotifytempos] MsgKey = 1948, Value = 48U33WgDS5EEUKEhMT9Zwt,1948,85.126
[Sent to groupB04.spotifytempos] MsgKey = 1949, Value = 1qBIOF4qSYG31GdVw5vIu4,1949,116.155
[Sent to groupB04.spotifytempos] MsgKey = 1950, Value = OvXEDaNqu5Obi9Va572lQ9,1950,82.146
[Sent to groupB04.spotifytempos] MsgKey = 1951, Value = 1KIOppMpY7eSPzhG7cob7n,1951,108.563
[Sent to groupB04.spotifytempos] MsgKey = 1952, Value = 1EUH4OBQ5Eac5mxyHASc4A,1952,84.569
[Sent to groupB04.spotifytempos] MsgKey = 1953, Value = 12CWFCFAWkkc3vPOG19dVK,1953,63.502
[Sent to groupB04.spotifytempos] MsgKey = 1954, Value = 6Limxo2wJuYL2WrtdewfKg,1954,73.579
[Sent to groupB04.spotifytempos] MsgKey = 1955, Value = 6UkJIDGuGv1CyJeAakb3Jj,1955,62.195
[Sent to groupB04.spotifytempos] MsgKey = 1956, Value = OirnJklasJ1rtQorlM4f2O,1956,92.163
[Sent to groupB04.spotifytempos] MsgKey = 1957, Value = 69RoHsh1PqtqKnrWy36MZl,1957,119.566
[Sent to groupB04.spotifytempos] MsgKey = 1958, Value = 38YV43s75rETyFK7ObQKPw,1958,129.561
[Sent to groupB04.spotifytempos] MsgKey = 1959, Value = 4grD2NkyBWSjo15uOmuqA2,1959,82.88799999999998
[Sent to groupB04.spotifytempos] MsgKey = 1960, Value = 7BjfhLDrFRE6omVIMzY8Fq,1960,87.333
```

We can see that the results are changing every 5 seconds as expected:

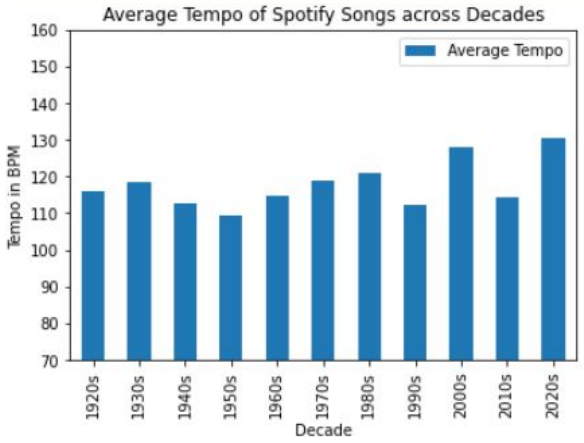


Decade	Average Tempo	Songs Count	Years Count	Current Offset
1930s	126.8	2	2	189
1940s	122.58	7	7	197
1950s	111.18	10	10	286
1960s	167.17	1	1	187
2000s	135.93	8	8	271
2010s	106.38	10	10	237
2020s	151.81	1	1	77

```
Current Epoch: 1
```

And every 5 seconds we can see our data updating in real time - incrementally. After programs running for some time we cans observe following result:

```
.awaitTermination()
```



Decade	Average Tempo	Songs Count	Years Count	Current Offset
1920s	115.97	2	1	171
1930s	118.56	12	3	203
1940s	112.73	46	7	211
1950s	109.38	64	10	307
1960s	114.59	68	10	308
1970s	118.94	73	10	282
1980s	120.8	72	10	283
1990s	112.07	65	10	309
2000s	127.88	57	10	290
2010s	114.22	59	10	255
2020s	130.57	5	1	82

```
Current Epoch: 24
```

We can see that the years with highest tempos were years from the 2020s and 2000s. Although, the data taken into consideration by this stream is only a small part of our 160 000 spotify data set, so the results are a little different from our results from MapReduce functions.

4. References

- In our solutions we used code snippets from course lecture slides and the course demo jupyter notebooks provided in the laboratories.
- https://databricks.com/blog/2017/05/08/event-time-aggregation-watermarking-apache-sparks-structured-streaming.html