VE472 Methods and Tools for Big Data
&
Million-Song-Dataset Project

pgroup 1

UM-SJTU Joint Institute

Lin Hao 517370910082
Lu Mingxuan 518021911166
Shen Shiyu 517370910132
Yang Ziqi 518370910075
Sun Yan 517370910147

August 6, 2021

Overview

1. Database Queries
   - Data Processing
   - Drill Queries
2. Advanced Data Analysis
   - Graph Generation
   - MapReduce
   - Spark
   - Further Analysis

HDF5 to Avro

Here is the schema in **song.avsc**.

```
{
    "namespace": "song.avro",
    "type": "record",
    "name": "SongSchema",
    "fields": [
        {"name": "songID", "type": "string"},
        {"name": "title", "type": ["string", "null"]},
        {"name": "artistID", "type": "string"},
        {"name": "artistName", "type": ["string", "null"]},
        {"name": "similar_artists", "type": "string"},
        {"name": "albumName", "type": ["string", "null"]},
        {"name": "duration",  "type": ["float", "null"]},
        {"name": "energy",  "type": ["float", "null"]},
        {"name": "hotttnesss",  "type": ["float", "null"]},
        {"name": "tempo",  "type": ["float", "null"]},
        {"name": "year",  "type": ["int", "null"]}
    ]
}
```

Data Processing

HDF5 to Avro

Functions in **hdf5_getters.py** (fetched from the official website of Millong Song Dataset) to read HDF5 files.

```python
def open_h5_file_read(h5filename):
    """
    Open an existing H5 in read mode.
    Same function as in hdf5_utils, here so we avoid one import
    """
    return tables.open_file(h5filename, mode='r')

def get_num_songs(h5):
    """
    Return the number of songs contained in this h5 file, i.e. the number of rows
    for all basic informations like name, artist, ...
    """
    return h5.root.metadata.songs.nrows

def get_artist_familiarity(h5,songidx=0):
    """
    Get artist familiarity from a HDF5 song file, by default the first song in it
    """
    return h5.root.metadata.songs.cols.artist_familiarity[songidx]
```

HDF5 to Avro

Functions to swritch HDF5 files into avro. Here are the headers in **hdf5ToAvro.py**.

```
import os
import glob
import hdf5_getters
import avro.schema
from avro.datafile import DataFileWriter
from avro.io import DatumWriter

basedir = "/home/data/dataset_2"
ext = ".h5"

schema = avro.schema.parse(open("song.avsc").read())
writer = DataFileWriter(open("song.avro", "wb"), DatumWriter(), schema)
```

Write into avro file by **writer.append()**.

Data Processing

Avro on HDFS

- Generate **song.avro**

  python hdf5ToAvro.py

- Put **song.avro** HDFS.

  hdfs dfs –put song.avro p1

- Start Drill on cluster.

  drill –conf

Drill Queries

The range of dates

```
apache drill> select max(Year) from hdfs.`/user/pgroup1/p1/song.avro` where Year > 0;
+--------+
| EXPR$0 |
+--------+
| 2010   |
+--------+
1 row selected (0.175 seconds)
apache drill> select min(Year) from hdfs.`/user/pgroup1/p1/song.avro` where Year > 0;
+--------+
| EXPR$0 |
+--------+
| 1939   |
+--------+
1 row selected (0.163 seconds)
```

Therefore, the range of dates covered by the songs in the dataset is

$$2010 - 1939 = 71 \ \textit{years}.$$

Drill Queries

The hottest song that is the shortest

```
apache drill> select max(cast(hotttnesss as float))
2..semicolon>   from hdfs.`/user/pgroup1/p1/song.avro`
3..semicolon>   where cast(hotttnesss as float) <= 1;
+---------+
| EXPR$0  |
+---------+
| 1.0     |
+---------+
1 row selected (0.143 seconds)
apache drill> select title, cast(duration as float), cast(hotttnesss as float)
2..semicolon>   from hdfs.`/user/pgroup1/p1/song.avro`
3..semicolon>   where cast(duration as float) > 0 and cast(hotttnesss as float) = 1
4..semicolon>   order by cast(duration as float) limit 1;
+--------------------------+-----------+---------+
|          title           |  EXPR$1   | EXPR$2  |
+--------------------------+-----------+---------+
| b'Something Good Can Work' | 164.17914 | 1.0     |
+--------------------------+-----------+---------+
1 row selected (0.163 seconds)
```

Drill Queries

The highest energy with lowest tempo

```
apache drill> select max(cast(energy as float)) from hdfs.`/user/pgroup1/p1/song.avro`;
+--------+
| EXPR$0 |
+--------+
| 0.0    |
+--------+
1 row selected (0.14 seconds)
apache drill> select title, cast(Tempo as float)
2..semicolon>   from hdfs.`/user/pgroup1/p1/song.avro`
3..semicolon>   where cast(Tempo as float) > 0
4..semicolon>   order by cast(Tempo as float) limit 1;
+-----------------------+--------+
|         title         | EXPR$1 |
+-----------------------+--------+
| b'Death In The Subway' | 14.937 |
+-----------------------+--------+
1 row selected (0.14 seconds)
```

Drill Queries

The name of the album with the most tracks

```
apache drill> select albumName, count(albumName) as NumOfTracks
2..semicolon>   from hdfs.`/user/pgroup1/p1/song.avro`
3..semicolon>   group by albumName order by NumOfTracks desc limit 1;
+-----------------+--------------+
|    albumName    | NumOfTracks  |
+-----------------+--------------+
| b'Greatest Hits' | 25           |
+-----------------+--------------+
1 row selected (0.239 seconds)
```

Drill Queries

The name of the band who recorded the longest song

```
apache drill> select artistName, duration
2..semicolon>   from hdfs.`/user/pgroup1/p1/song.avro`
3..semicolon>   order by CAST(duration as float) desc limit 5;
+-----------------------------------------------+------------+
|                  artistName                   |  duration  |
+-----------------------------------------------+------------+
| b'Liu Baorui'                                 | 3002.8274  |
| b'SUTCLIFFE J\xc3\x9cGEND'                    | 1713.3971  |
| b'Herbie Mann Quintet with Symphonic Orchestra' | 1695.1113  |
| b'Globe Unity Orchestra'                      | 1590.2036  |
| b'The Tangent'                                | 1514.7881  |
+-----------------------------------------------+------------+
5 rows selected (0.149 seconds)
```

Graph Generation

Indicate source node at the beginning of graph construction.

- Node represents each song
- Edge (without weight) indicates similarity of connected songs
- Criteria for edge connection: with similar hotness
  sorting requires $O(n \log n)$ for sequential implementation.

We do not use MapReduce to construct graph, though it works well with the provided subset, it cannot scale to big data which cannot fit into memory. (However, we can fix it using parallel version sorting)

MapReduce

Each node has three states, colored in white, gray and black.
Mapper function: for each node, emit its neighbours as well as itself while their distance information and color (state) is also transferred.
Reducer function: for each node, combine its information and return the most updated.

Perform this iteration several times, implemented by either submitting multiple jobs (slower but more flexible) or using multi-step for a single job (faster but hard to find iteration steps dynamically).

Spark

Apache Spark is an open-source unified analytics engine for large-scale data processing.

Characteristic

- MapReduce Substitute
- Heavy RAM Using
- Advanced Data Analysis

Spark

The main difference with MapReduce is the **reduce** function.
Instead of taking a key and a value as the input, the reduce function used by **reduceByKey()** function in pyspark takes two **values** or two songs, in this case, as the input. We then compare them and reserve the one with the smaller distance to reduce results.

MapReduce vs. Spark

We tested both MapReduce and Spark on a 10GB subset of MSD on the cluster. The time costed is as follows:

Time

Here, we fix BFS iteration steps to 5.

- MapReduce: 2 minutes 12 seconds
- Spark: 43 seconds

We guess Spark far prevails MapReduce in time due to its heavy use of memory and its repetitive transfer of data to each node.

Further Analysis

To find similar songs with the sources song, we pick all song/node with the color of **GRAY** and list their song IDs because based on our BFS setting, after n times' of iteration, the gray nodes should have the same distance to the source song, aka their similarity with the source is expected to be the same.

Further Analysis

For now, we have got the list of similar node songs for one source song, which is flexible for further data access based on various requirements; and we can use SQL quires to meet them.
Here are some cases.

- When one prefers a single piece of Rock music, we can go through its node songs, and the ones with close tempo or beats might be what he likes.
- For an art history learner, he may want to know the representative songs of certain age. When he finds a song meeting his needs, other songs can be chosen from the node song list with close year of distribution.

References

- Thierry Bertin-Mahieux et al. “The Million Song Dataset”. In: Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011). 2011
- http://millionsongdataset.com/pages/field-list/
- https://github.com/shriyamishra1/BFS-in-Pyspark
- https://github.com/AGeoCoder/Million-Song-Dataset-HDF5-to-CSV/blob/master/msdHDF5toCSV.py

Thanks for listening!

Q&A