Big Data Analysis on Million Song Dataset (MSD)

ECE4721J: Methods and Tools for Big Data

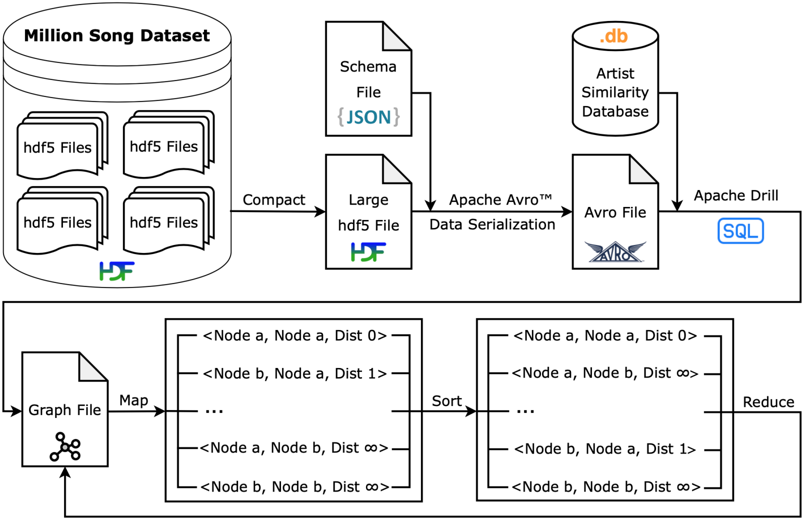
Yiding Chang Yifan Shen Kexuan Huang Qinhang Wu

July 29, 2022

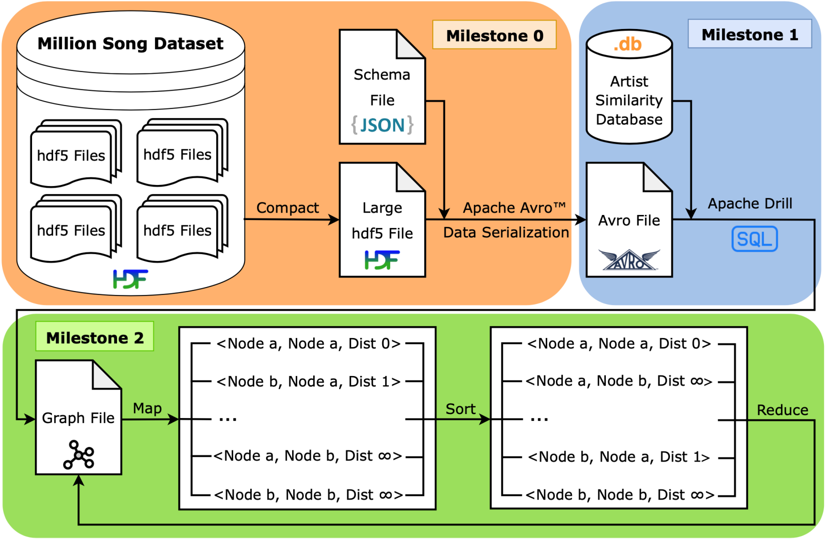
Overview

- Milestone 0: HDF5 Data Pre-process
- Milestone 1: Drill Database Query
- Milestone 2: Advanced Data Analysis

Workflow


Million Song Dataset
hdf5 Files
hdf5 Files
hdf5 Files
hdf5 Files
Compact
Schema File
{JSON}
Large hdf5 File
Apache Avro™
Data Serialization
.db
Artist Similarity Database
Avro File
Apache Drill
SQL
Graph File
Map
<Node a, Node a, Dist 0>
<Node b, Node a, Dist 1>
...
<Node a, Node b, Dist ∞>
<Node b, Node b, Dist ∞>
Sort
<Node a, Node a, Dist 0>
<Node a, Node b, Dist ∞>
...
<Node b, Node a, Dist 1>
<Node b, Node b, Dist ∞>
Reduce

Milestones



Million Song Dataset
hdf5 Files
hdf5 Files
hdf5 Files
hdf5 Files
Compact
Schema File
{JSON}
Large hdf5 File
Milestone 0
Apache Avro™ Data Serialization
.db
Artist Similarity Database
Milestone 1
Avro File
Apache Drill
SQL
Milestone 2
Graph File
Map
<Node a, Node a, Dist 0>
<Node b, Node a, Dist 1>
...
<Node a, Node b, Dist ∞>
<Node b, Node b, Dist ∞>
Sort
<Node a, Node a, Dist 0>
<Node a, Node b, Dist ∞>
...
<Node b, Node a, Dist 1>
<Node b, Node b, Dist ∞>
Reduce

Section 1

Milestone 0: HDF5 Data Pre-process

Goals

1. Compact small `hdf5` files into larger one
2. Read `hdf5` file and extract the information
3. Convert `hdf5` to `Avro` with `Apache Avro`

1. Compact small `hdf5` files into larger one

```
$ python3 create_aggregate_file.py <IN> <OUT>
```

- Input: a directory contains `hdf5` song files
- Output: an aggregate `hdf5` song file
- Example:

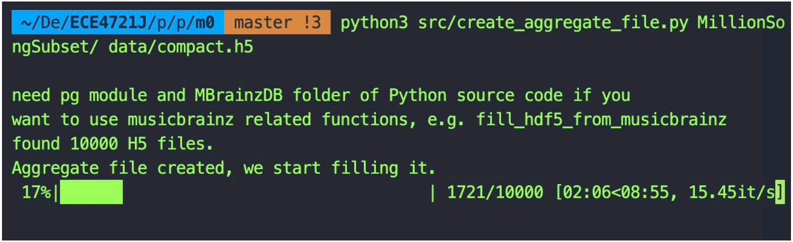


Figure 1: Compact 10000 `hdf5` files into larger one

2. Read hdf5 files and extract the information

```
$ python3 display_song.py [FLAGS] <HDF5> <idx> <field>
```

- Input: an `hdf5` song file
- Output: specified field content
- Example:

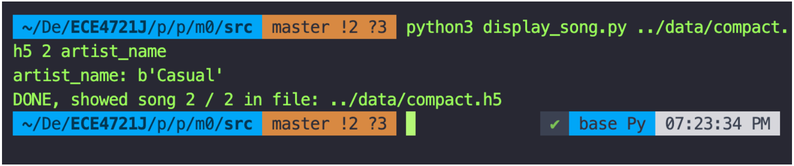


Figure 2: Get artist name of the second song in compacted `hdf5` file

3. Convert `hdf5` to `Avro` with `Apache Avro`

```
$ hdf5_to_avro.py [-h] -s <SCHEMA> -i <HDF5> -o <AVRO>
```

- Input:
    - an `Avro` schema file
    - an `hdf5` song file to be converted
- Output: an `Avro` song file

Sample schema file in `json` format:

```json
{
  "namespace": "song.avro",
  "type": "record",
  "name": "Song",
  "fields": [
    {
      "name": "artist_name",
      "type": ["string", "null"]
    },
    {
      "name": "title",
      "type": ["string", "null"]
    }
  ]
}
```

```
root@hadoop-master:/home/s/pj1/m0# python3 src/hdf5_to_avro.py -s schema/songs
.avsc -i data/compact.h5 -o data/output.avro
21:18:10 [Info] Convert a song file from hdf5 to Avro...
21:18:10 [Info] Avro schema path: schema/songs.avsc
21:18:10 [Info] hdf5 input path: data/compact.h5
21:18:10 [Info] Avro output path: data/output.avro
21:18:10 [Info] Avro schema file and hdf5 file exist
21:18:10 [Warning] Avro output file data/output.avro already exists
21:18:10 [Info] Parsing the Avro schema file...
21:18:10 [Info] Get the following fields:
                artist_hotttnesss    ["float", "null"]
                artist_id            ["string", "null"]
                artist_name          ["string", "null"]
                duration             ["float", "null"]
                energy               ["float", "null"]
                release              ["string", "null"]
                song_hotttnesss      ["float", "null"]
                song_id              "string"
                tempo                ["float", "null"]
                title                ["string", "null"]
                track_id             ["string", "null"]
                year                 ["int", "null"]
21:18:10 [Info] Found 10000 song(s)
21:18:10 [Info] Start converting hdf5 to Avro
21:18:10 Converting: 100%|████████| 10000/10000 [00:22<00:00, 436.24it/s]
```

Figure 3: Convert compacted `hdf5` file to `Avro`

Section 2

Milestone 1: Drill Database Query

Goals

Query Million Song Dataset (MSD) with `Drill`:

1. Find the range of dates covered by the songs in the dataset
2. Find the hottest song that is the shortest and shows highest energy with lowest tempo
3. Find the name of the album with the most tracks
4. Find the name of the band who recorded the longest song

1. The range of dates covered by the songs

- SQL:

```sql
-- Age of the oldest songs
SELECT 2022 - MAX(year) AS Age
FROM hdfs.`/pj/m0/output.avro`;

-- Age of the youngest songs
SELECT 2022 - MIN(year) AS Age
FROM hdfs.`/pj/m0/output.avro`
WHERE year > 0;
```

1. The range of dates covered by the songs

- Results:

```
+---------+        +---------+
|  Age    |        |  Age    |
+---------+        +---------+
| 12      |        | 96      |
+---------+        +---------+
1 row selected     1 row selected
```

The oldest song's age is **96** and the youngest is **12**. As a result, the range of dates covered by the songs is **84** years.

2. The hottest song that is the shortest and shows highest energy with lowest tempo

- SQL:

```
SELECT title
FROM hdfs.`/pj/m0/output.avro`
WHERE song_hotttnesss <> 'NaN'
ORDER BY song_hotttnesss DESC,
    duration ASC,
    energy DESC,
    tempo ASC
LIMIT 10;
```

- Remarks: This query returns **5648** results, but we only display the first **10** records.

2. The hottest song that is the shortest and shows highest energy with lowest tempo

```
+------------------------------------------------------------+
|                           title                            |
+------------------------------------------------------------+
| b'Immigrant Song (Album Version)'                          |
| b"Nothin' On You [feat. Bruno Mars] (Album Version)"       |
| b'This Christmas (LP Version)'                             |
| b'If Today Was Your Last Day (Album Version)'              |
| b'Harder To Breathe'                                       |
| b'Blue Orchid'                                             |
| b'Just Say Yes'                                            |
| b'They Reminisce Over You (Single Version)'                |
| b'Exogenesis: Symphony Part 1 [Overture]'                  |
| b'Inertiatic Esp'                                          |
+------------------------------------------------------------+
10 rows selected (0.471 seconds)
```

3. The name of the album with the most tracks

- SQL:

```
SELECT release, COUNT(release) AS NumTrack
FROM hdfs.`/pj/m0/output.avro`
GROUP BY release
ORDER BY NumTrack desc
LIMIT 1;
```

- Results:

```
+-------------------+----------+
|      release      | NumTrack |
+-------------------+----------+
| b'Greatest Hits' | 21       |
+-------------------+----------+
1 row selected (0.695 seconds)
```

4. The name of the band who recorded the longest song

- SQL:

```
SELECT artist_name, duration
FROM hdfs.`/pj/m0/output.avro`
ORDER BY duration DESC
LIMIT 1;
```

- Results:

```
+-------------+-----------+
| artist_name | duration  |
+-------------+-----------+
| b'UFO'      | 1819.7677 |
+-------------+-----------+
1 row selected (0.27 seconds)
```

Section 3

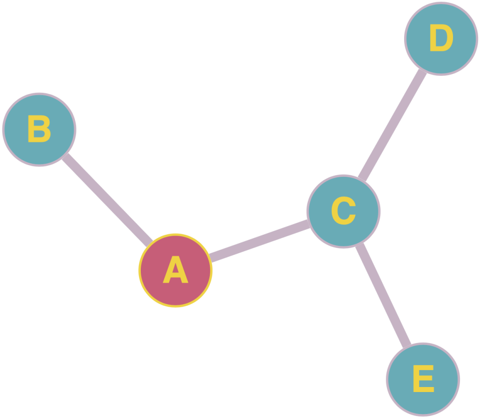
Milestone 2: Advanced Data Analysis

Goals

1. Determine distance between artists with adjacency matrix, using parallelized BFS
2. Propose similar songs with distance and "provide more diverse recommendations"
3. Implement the above algorithm in both Mapreduce and Spark
4. Compare the performance of Mapreduce and Spark

BFS with MapReduce - A Simple Example

Let's say we want to find artists similar to **A** with distance **3**, and we have following relationships (each edge has distance **1**):



Step 1: Initialize Graph File with Target Artist

Format: each line contains **Node** | **Distance** | **Neighbours**

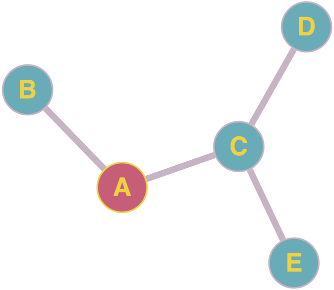


Figure 5: Initialize Graph File

```
A | 0 | A,B,C
B | ∞ | A,B
C | ∞ | A,C,D,E
D | ∞ | C,D
E | ∞ | C,E
```

Step 2: Generate Distance Pairs in Mapper

Mapper: Generate **Neighbours, Node, Distance+1** if not itself

```
A|0|A,B,C
B|∞|A,B
C|∞|A,C,D,E
D|∞|C,D
E|∞|C,E
```

$\xrightarrow{\textbf{Map}}$

```
A, A, 0
B, A, 1
C, A, 1
A, B, ∞
B, B, ∞
A, C, ∞
C, C, ∞
D, C, ∞
E, C, ∞
C, D, ∞
D, D, ∞
C, E, ∞
E, E, ∞
```

$\xrightarrow{\textbf{Sort}}$

```
A, A, 0
A, B, ∞
A, C, ∞
B, A, 1
B, B, ∞
C, A, 1
C, C, ∞
C, D, ∞
C, E, ∞
D, C, ∞
D, D, ∞
E, C, ∞
E, E, ∞
```

Step 3: Merge Distance Pairs in Reducer

Reducer: Merge the same **Neighbour**, keep distance **minimum**

```
A, A, 0
A, B, ∞
A, C, ∞
B, A, 1
B, B, ∞
C, A, 1
C, C, ∞
C, D, ∞
C, E, ∞
D, C, ∞
D, D, ∞
E, C, ∞
E, E, ∞
```

Reduce ⟶

```
A|0|A,B,C

B|1|A,B

C|1|A,C,D,E

D|∞|C,D

E|∞|C,E
```

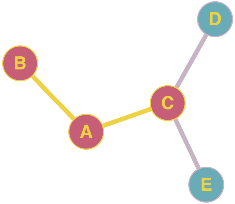


Figure 6: Graph after 1 MapReduce Iteration

Iteraiton 2: Mapper

```
A|0|A,B,C
B|1|A,B
C|1|A,C,D,E
D|∞|C,D
E|∞|C,E
```

$\xrightarrow{\textbf{Map}}$

```
A, A, 0
B, A, 1
C, A, 1
A, B, 2
B, B, 1
A, C, 2
C, C, 1
D, C, 2
E, C, 2
C, D, ∞
D, D, ∞
C, E, ∞
E, E, ∞
```

$\xrightarrow{\textbf{Sort}}$

```
A, A, 0
A, C, 2
A, B, 2
B, A, 1
B, B, 1
C, A, 1
C, C, 1
C, D, ∞
C, E, ∞
D, C, 2
D, D, ∞
E, C, 2
E, E, ∞
```

Iteration 2: Reducer

```
A, A, 0
A, C, 2
A, B, 2
B, A, 1
B, B, 1
C, A, 1
C, C, 1
C, D, ∞
C, E, ∞
D, C, 2
D, D, ∞
E, C, 2
E, E, ∞
```

Reduce ⟶

```
A|0|A,B,C

B|1|A,B

C|1|A,C,D,E

D|2|C,D

E|2|C,E
```

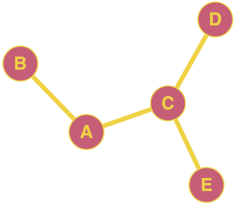


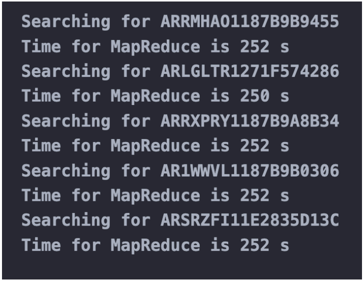
Figure 7: Graph after 2 MapReduce Iterations

BFS with Spark

- Same algorithm as is proposed before
- Implemented using Python with `PySpark`
- For the implementation, we:
  1. Convert data into RDD map
  2. Using Spark `sortByKey()` to sort RDD aggregated by node index and then combining the neighbours
  3. Using Spark `reduce()` to pick the minimum distance of different neighbours towards the central node
- Spark is assumed to be faster since subsequent steps are retained in memory with a trade-off of much more memory consumption

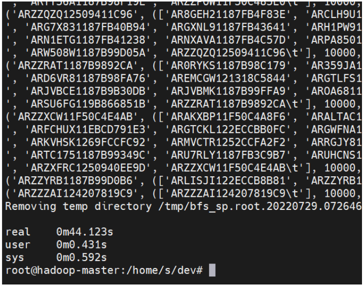
Benchmark

- Data size: around **200GB**
- Server: SJTU cluster with three machines
  - CPU: Dualcore Intel Xeon Processor (Skylake, IBRS)
  - Memory: 4GB



```
Searching for ARRMHAO1187B9B9455
Time for MapReduce is 252 s
Searching for ARLGLTR1271F574286
Time for MapReduce is 250 s
Searching for ARRXPRY1187B9A8B34
Time for MapReduce is 252 s
Searching for AR1WWVL1187B9B0306
Time for MapReduce is 252 s
Searching for ARSRZFI11E2835D13C
Time for MapReduce is 252 s
```

Figure 8: MapReduce: around **250s**



```
('ARZZQZQ12509411C96', (['AR8GEH21187FB4F83E', 'ARCLH9U1
', 'ARG7X831187FB40B94', 'ARGXNL91187FB43641', 'ARH1PW91
', 'ARN1ETG1187FB41238', 'ARNXAVA1187FB4C57D', 'ARPA8501
', 'ARW508W1187B99D05A', 'ARZZQZQ12509411C96\t'], 10000,
('ARZZRAT1187B9892CA', (['AR0RYKS1187B98C179', 'AR359JA1
', 'ARD6VR81187B98FA76', 'AREMCGW121318C5844', 'ARGTLFS1
', 'ARJVBCE1187B9B30DB', 'ARJVBMK1187B99FFA9', 'AROA6811
', 'ARSU6FG119B866851B', 'ARZZRAT1187B9892CA\t'], 10000,
('ARZZXCW11F50C4E4AB', (['ARAKXBP11F50C4A8F6', 'ARALTAC1
', 'ARFCHUX11EBCD791E3', 'ARGTCKL122ECCBB0FC', 'ARGWFNA1
', 'ARKVHSK1269FCCFC92', 'ARMVCTR1252CCFA2F2', 'ARRGJY81
', 'ARTC1751187B99349C', 'ARU7RLY1187FB3C9B7', 'ARUHCNS1
', 'ARZXFRC1250940EE9D', 'ARZZXCW11F50C4E4AB\t'], 10000,
('ARZZYRB1187B99D0B6', (['ARLISJI122ECCB8B81', 'ARZZYRB1
('ARZZZAI124207819C9', (['ARZZZAI124207819C9\t'], 10000,
Removing temp directory /tmp/bfs_sp.root.20220729.072646

real    0m44.123s
user    0m0.431s
sys     0m0.592s
root@hadoop-master:/home/s/dev#
```

Figure 9: Spark: around **45s**

Reference

1. Million Song Dataset

   `http://millionsongdataset.com`

2. Apache Avro Documentation

   `https://avro.apache.org/docs/current/index.html`

3. Apache Spark Documentation

   `https://spark.apache.org/docs/latest/`

Thanks for your attention!

Figure 10: Never Gonna Give You Up (by Rick Astley)