

Simple Search Engine design to locate a Musical group based on their ID

(By Shreyas Nagaraj)

Contents:

- Design considerations & assumptions
- DEMO snapshots (using Eclipse IDE)
- Snapshot of one Junit test run

A. Design considerations & assumptions:

- Since, the problem is more about how we can scale the solution to work with larger datasets – I was inclined to go with the following strategies.

1. Emulating a database:

- I have emulated a database, to store the records from the input file. Ideally you would use - a NoSQL database (key-value store type) like DynamoDB, Voldemort etc.
- Since, I shouldn't be using any external tools/frameworks/libraries in this exercise – I have emulated the database on memory. It is done by simply reading of the file onto the memory. Of-course, we won't be able to use this strategy to load a 1TB sized dataset.

2. Dividing the dataset:

- I have divided the dataset to store & process the dataset in parallel. Generally, we will be dealing with records of huge magnitude & we need to divide the dataset into chunks which can be processed in parallel. The division can be horizontal (rows) or vertical (columns). However, in this case let's consider *horizontal data shards* which are basically horizontal splits of the database. To complement this, we would also ideally go with distributed file system (like HDFS) running on cluster of nodes in-order to store the shards.
- In the real world environment, these things are generally solved by using MapReduce based approaches (E.g. Derivatives of Apache Hadoop, Amazon's Elastic MapReduce), where a divided dataset ("*Shards*") are stored in a distributed file system and then later are mapped to a set of distributed servers for processing.
- ***In my solution***, I won't be using any real distributed file system. Instead, I would be dividing the emulated database and assigning them to "*pseudo/virtual cluster nodes*".

3. Indexing & sorting:

- Regardless of what kind-of database is used, we will need indexing to improve the *search performance*. The likes of Apache Solr, Cloudera search & similar tools would be helpful to index large datasets in-order to assist the query/search operations.
- Indexing would let us search for a particular row/entry in "*logarithmic time*". The logarithmic time is because the searching can be done *efficiently* using the some variant of "*binary search algorithm*". Generally, variants of "*B trees*" are used sort and store the indices. However, the index would not contain the entire row and would need a pointer to point to the actual database row.

- The indexes with the pointers are also generally stored across a distributed file system (like HDFS) if we are talking about *massive datasets with billions of rows*. So, we may need shards for indexes also, not just the actual dataset.
- **In my solution**, since we just have two columns – I have used the *hashCode()* (java string hash for the “Musical group” name string) as the index-key which can be sorted and stored for quick loop-ups later on. **Note:** For large datasets the trivial *hashCode()* won’t give unique indices which we want. So, we will need a better way to choose one column or combination of column to generate the unique cluster indices for our massive dataset.
- In my solution, the trivial *hashCode()* way of indexing might generate duplicate indices since *hashCode()* doesn’t guarantee uniqueness. So, we might have one or more rows associated with the same index. Hence, when I get hold of the queried index – I also extract the right row by comparing the element which is used for the lookup when there are duplicate indices.
- Indices data structure will be a combination of index value and a pointer to the actual row. So, to sort these indices which will help the look-up process, we might need a good external sort algorithm (*like Hadoop based TeraSort*) if we are sorting millions of indices.
- Like, external sort **in my solution**, I don’t divide the dataset for sorting. External sorts operate upon divided datasets which will be sorted on various cluster nodes and then be stored on the local node after the sorting is done. Then, they will be merged together using techniques like k-way merge sorting or a similar sorting technique.
- **In my solution**, I have used an “*In-place quicksort*” technique to sort the indices. This would NOT be comparable to the *real external sort*, however by doing the *sorting in-place* we at least can be memory efficient while sorting our dataset.
- **Note:** Since, we only have 3 columns for our dataset including the calculated index – I don’t save the indices in a separate storage. Based on the index which is being looked-up, I can also retrieve the entry dataset entry for that particular index. And, several such dataset entries referred by their indices would be put into each of the virtual shard of the emulated database.

4. LRU Caching:

- We need to implement a caching mechanism to cache all the recently searched musical groups. If there is a miss in the cache – I take the bigger route to search the index in the virtual shards.
- **In my solution**, I have implemented a **configurable LRU** (least recently used) cache. The size of the cache can be configured to hold a fixed number of cache entries. And, the cache entry would contain the *index(HashCode)* along with the corresponding *Music Band IDs (result from a search)*. So, whenever a Music band is queried by the searcher and a corresponding ID is found in the database – then the band index/hash along with the retrieved band IDs makes it into the cache. The least recently used cache entry is always replaced with the new ones.
- **Concurrency in cache:** caches need to be concurrent, since they are constantly being updated. I had to draw some ideas from some online material for coding this up with few of the Java language utilities. (Ref: <https://code.google.com/p/concurrentlinkedhashmap/>)
- To further optimize caching, modern implementations use distributed cache techniques like *Memcached*, *Redis* etc. A distributed cache may span over multiple servers so that it can grow in size and in transactional capacity. However, I have not implemented any distributed caching in my solution. And, my

LRU cache SIZE can only be *statically* configured. But, ideally cache sizes could be dynamically configured too.

5. Partial Lookups (additional feature – just to enlist band names which user may be is trying to “search”):

- To lookup musical bands based on partial strings, I decided to go with a implementation of a prefix Tree (also called as “trie”) data structure. Such a structure would retrieve all possibilities of a valid musical group in the dataset based on a prefix string.
- One application of this would be to recommend close matches for the lookup – where the search for particular musical band **FAILS** to find a matching band on the dataset.
- ***In my solution***, I build a Trie when I initially parse through the dataset and store it separately. A trie though an additional data structure, would not consume too much memory based on the way the prefix tree is constructed. The partial string lookup will result in a bunch of qualified musical bands having the same prefix. This can be listed upon a failure of a band lookup after the query has been made.
- For the sake of simplicity, I do the partial loop-up only if the query music band string is of length “2” or more. And, then I list out the all the valid musical bands having the prefix as the query string. This is then displayed on the console for the user.

6. Fault Tolerance

- Distributed approach to solve this problem would mean that we need to maintain replicas of data shards. So, when on the node contains a few shards goes down – then the datasets should not be lost. We could hence distribute the same data shard replicas across other nodes in the cluster.
- Since, I am just emulating the distributed environment by having a bunch of virtual nodes containing shards, I won’t be handling any fault tolerance as a part of this solution. However, it is possible to extend the solution to create replicas of the shards and maintain them.

<DEMO Snapshots – NEXT

B. DEMO snapshots (using Eclipse IDE):

1. *Initialization followed by searching bands with → English/Non-English names; long names; names with spaces; names with many IDs*

```
@ Javadoc Declaration Console
Searcher [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jun 10, 2015, 10:11:37 PM)
Initializing the Emulated distributed database
And, building a Prefix Tree for Partial Searches...
Done!
Indexing dataset and sorting...
No of dataset entries(rows): 156246   No of Horizontal Shards: 2000   Cache Size: 20
Done!

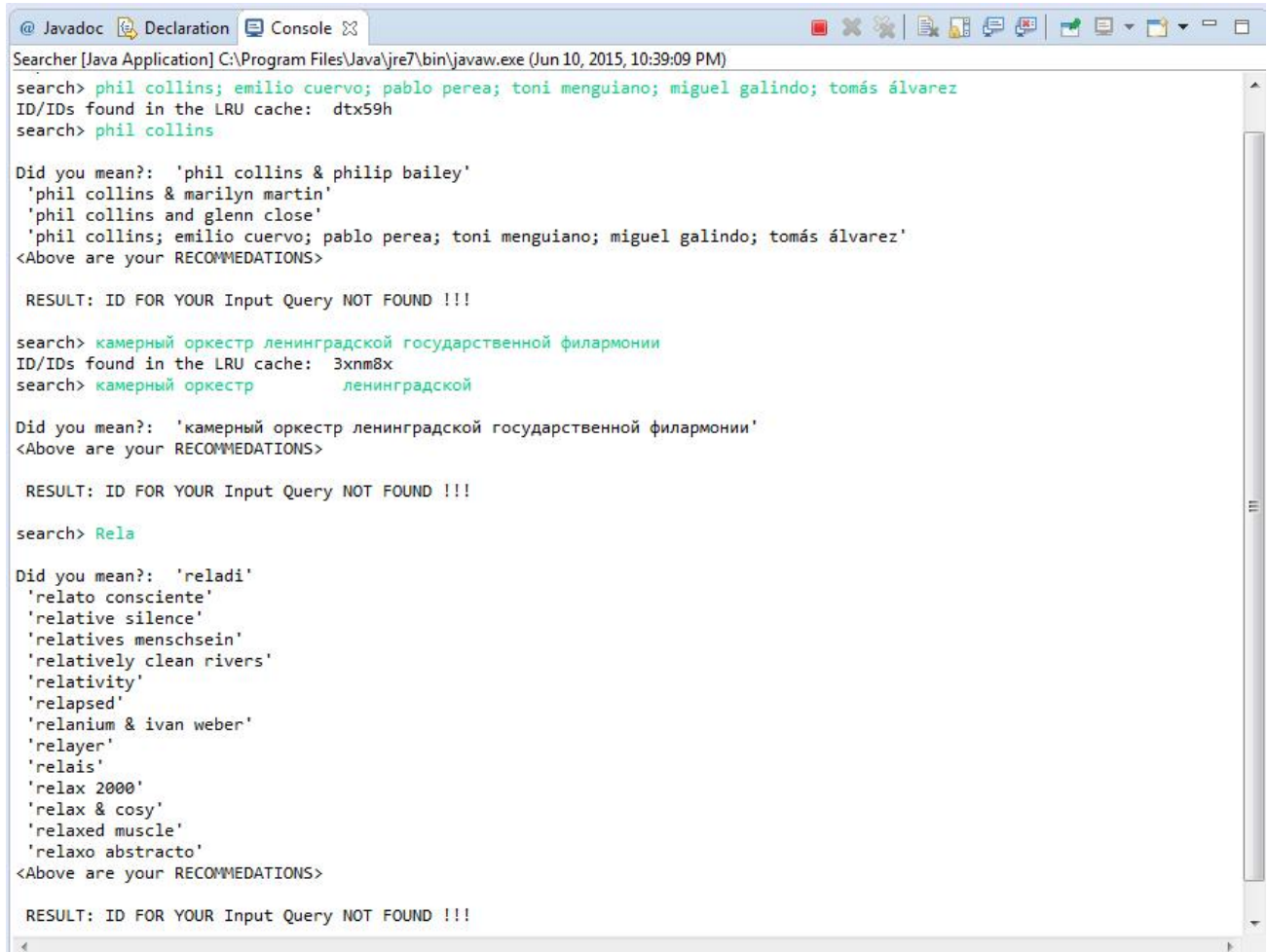
search> Yearbook Committee
ID/IDs found in the database: dykyh3
search> Overload
ID/IDs found in the database: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> グリッサナ・サンダウオンチョン & ニレット・ルンラウオン
ID/IDs found in the database: g8vxgb
search> グリッサナ・サンダウオンチョン & ニレット・ルンラウオン
ID/IDs found in the LRU cache: g8vxgb
search> グリッサナ・サンダウオンチョン & ニレット・ルンラウオン
ID/IDs found in the LRU cache: g8vxgb
search> 彩京朋美 (cv.川澄綾子)、セルニア・伊織・フレイムハート (cv.中原麻衣)、四季鏡早苗 (cv.小清水亜美)、大地 薫 (cv.釘宮理恵)
ID/IDs found in the database: ft0q1_
search> John Lennon & The Plastic Ono Nuclear Band with Little Big Horns & The Philharmonic Orchestrange
ID/IDs found in the database: 3j0d5y
search> John Lennon & The Plastic Ono Nuclear

Did you mean?: 'john lennon & the plastic ono nuclear band with little big horns & the philharmonic orchestrange'
<Above are your RECOMMEDATIONS>

RESULT: ID FOR YOUR Input Query NOT FOUND !!!

search> depapepe meets ハチミツとクローバー
ID/IDs found in the database: 3f5bpc
search> depapepe
ID/IDs found in the database: 1rhnmt
search> hhh (ryu & dai)
ID/IDs found in the database: 1x0q6l
search> hhh (ryu & dai)
ID/IDs found in the LRU cache: 1x0q6l
search> a
ID/IDs found in the database: 2p6x0
```

2. Band Names not present in the Database (shows a few recommendations based on Prefix pattern)



```
@ Javadoc Declaration Console
Searcher [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jun 10, 2015, 10:39:09 PM)
search> phil collins; emilio cuervo; pablo perea; toni menguiano; miguel galindo; tomas alvarez
ID/IDs found in the LRU cache: dtx59h
search> phil collins

Did you mean?: 'phil collins & philip bailey'
'phil collins & marilyn martin'
'phil collins and glenn close'
'phil collins; emilio cuervo; pablo perea; toni menguiano; miguel galindo; tomas alvarez'
<Above are your RECOMMEDATIONS>

RESULT: ID FOR YOUR Input Query NOT FOUND !!!

search> камерный оркестр ленинградской государственной филармонии
ID/IDs found in the LRU cache: Эхnm8x
search> камерный оркестр ленинградской

Did you mean?: 'камерный оркестр ленинградской государственной филармонии'
<Above are your RECOMMEDATIONS>

RESULT: ID FOR YOUR Input Query NOT FOUND !!!

search> Rela

Did you mean?: 'reladi'
'relato consciente'
'relative silence'
'relatives menschsein'
'relatively clean rivers'
'relativity'
'relapsed'
'relanium & ivan weber'
'relayer'
'relais'
'relax 2000'
'relax & cosy'
'relaxed muscle'
'relaxo abstracto'
<Above are your RECOMMEDATIONS>

RESULT: ID FOR YOUR Input Query NOT FOUND !!!
```

3. Demonstrating caching the searches

```
@ Javadooc | Declaration | Console [X]
Searcher [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jun 10, 2015, 10:51:07 PM)

search> Overload
ID/IDs found in the database: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> Overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> Overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> Overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> Overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> ডাটাবেস মেন অউ অডার জালিফ
ID/IDs found in the database: 1wxn7_
search> ডাটাবেস মেন অউ অডার জালিফ
ID/IDs found in the LRU cache: 1wxn7_
search> Overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
search> ডাটাবেস মেন অউ অডার জালিফ
ID/IDs found in the LRU cache: 1wxn7_
search> クリッサナ・サンガウオンチョン & ニルット・ルンラウオン
ID/IDs found in the database: g8vxgb
search> ডাটাবেস মেন অউ অডার জালিফ
ID/IDs found in the LRU cache: 1wxn7_
search> overload
ID/IDs found in the LRU cache: 4jm18_ gf13v3 3j3d5k 1twzd4 3j35x4 8ls5b 3f05sj
```

4. Band names which are not found in the database & also no such patterns are present

[illegible]

<JUnit test run snapshot – NEXT>

C. Junit Sample Test Run – snapshot

The screenshot shows an IDE window with the following components:

- Package Explorer:** Shows the project structure with 'JUnit' and 'Searcher' packages.
- JUnit Test Run Summary:** Displays 'Finished after 41.348 seconds', 'Runs: 25/25', 'Errors: 0', and 'Failures: 0'.
- Test Results List:** A list of test methods and their execution times:
 - testSearches1 (3.686 s)
 - testSearches2 (3.016 s)
 - testSearches3 (3.894 s)
 - testSearches4 (2.042 s)
 - testSearches5 (2.024 s)
 - testSearches6 (1.781 s)
 - testSearches7 (1.938 s)
 - testSearches8 (2.255 s)
 - testSearches9 (0.983 s)
 - testSearches10 (1.362 s)
 - testSearches11 (1.848 s)
 - testSearches12 (0.948 s)
 - testSearches13 (1.224 s)
 - testSearches14 (2.050 s)
 - testSearches15 (0.982 s)
 - testSearches16 (1.372 s)
- Source Code:** The 'SearcherTester.java' file is open, showing several test methods. The method `testSearches25()` is highlighted in blue. It contains the following code:

```
183 @Test
184 public void testSearches25() {
185     List<String> actual = testSearchInstance.search("\\n");
186     List<String> expected = Arrays.asList();
187     assertTrue(actual.equals(expected));
188 }
```