CSR-Traversal vs. HashJoin Contest

Colombo Aurora Anna Francesca and Berzoini Raffaele

I. Introduction

The data-set provided by the Stanford Network Analysis Project [1] analyzes data coming from the POKEC social network. Our interest is only in the relationships between people in the social network, which are represented with an edge connecting a user and one of his friends.

II. Task 1

Our data storage is divided in two java classes, one of each represents a different way to store data. Each of them use IntArrayList [2] to collect users and friends, as the data type int covers a tinier portion of the memory than ArrayList, which employs Integer. Furthermore both of them permit to store all the data-set or just a portion of it, choosing its size.

A. Data Storage CSR

In the CSR class are present two methods to store data in CSR format (composed by two IntArrayList, *ptr* and *idx*).
The *CSR.DataStorageAll(String path)* method allows to store all the vertices and edges contained in [1] whereas the *CSR.DatastorageSubset(String path, int n_rel)* will store only a given number of edges (where *n_rel* stands for "number of relationships").
Both of the methods receive a String containing the path of the data-set.
They works in the same way:

1) *ptr*, *idx* IntArrayList and *scanner* Scanner instantiation. The *ptr* contains the positions of *idx* in which the first friend of each user is stored.
2) *current_user*, *userID* int initialisation.
3) A *while* loop running until EOF or until *n_rel* is reached.
4) A inner *while* loop and an *if* statement to update the *ptr* each time *scanner* switches from the current *userID* to the following one. This part of code manage also the absence of an user.
5) The *while* loop ends with the adding of the friend in the actual scanned row.
6) In the end a new value is added to *ptr* in order to know the number of friends of the last user.

B. Data Storage Table

Java class *IntTable* builds a struct, containing two IntArrayList, *user* and *friend*, which represent respectively the user's and his friend's ID.
IntTable.DataStorageAll(String path) aims to store all the data-set into the two IntArrayList, using a *while* loop and a scanner, which moves from an int to another in the .txt file.
IntTable.DataStorageSubset(String path, int n_rel) aims instead to store a subset of a given size through the variable *n_rel*, which represents the number of friendship relations chosen. This function includes a *for* loop, which has the same purpose of the previous *while* loop.

Taking into consideration the two structures described in II-A and II-B we can tell that CSR is a little bit more memory-friendly. In fact it needs less space in the *ptr* IntArrayList, that "corresponds" to the *user* IntArrayList of the Table format, because it contains only the position of the friends of a given user.

III. Task 2

Data is now functionally stored and can be easily used for some simple operations, such as CSR-Traversal and Hash Join, which can be described with the SQL query:

SELECT r1.user_id2
FROM relationship r1
WHERE r1.user_id1 = "input_ID".

So, given a data-set and a vertex ID, these functions return a list of all the immediate neighbours for that vertex. Moreover CSR-Traversal operates with the CSR data structure and Hash Join with the Table one.

A. Traverse

After having built the CSR structure, now we have two IntArrayList: *ptr* and *idx*.
Given the ID of a user, *ptr[ID]* returns the index position of the first friend in the *idx* IntArrayList.
To know how many edges (in other words, the number of friends) of the given user you need to perform the following difference:

$$d = ptr[ID+1] - ptr[ID] \tag{1}$$

Now the traverse functionality is completed by iterating from *idx[ptr[ID]]* for *d* times. The result is then stored inside an IntArrayList and returned to the calling function.

B. Hash Join

Hash Join operation for depth=1 needs two different functions:

- *HashMap()*;
- *depth_1(int userID)*.

The first one generates a second table, *map*, to quickly retrieve neighbours. Users ID contained in *user* II-B will become keys, i.e. attributes that can be univocally identified, while *friends* will be stored in blocks, called *friendsList*. Each *map* row, which is an ArrayList called *keys*, contains five *friendsList*, belonging to five different users ID.

Table I
REPRESENTATION OF THE HASHMAP STRUCTURE

map[0]	keys[0]	friendsList
	keys[1]	friendsList
	keys[2]	friendsList
	keys[3]	friendsList
	keys[4]	friendsList

The Hash Function, which links a user ID to its location in *map*, wasn't computationally implemented, but it's intrinsic in the *HashMap()*. Given a user ID, its *friendsList* position can be found as: *map.get(user_ID/5).get(user_ID%5)*, so that the division result gives the index for the *map* row, while its remainder gives the position in *keys*, through modulo operator. This is possible thanks to a *for* loop, which is used to read users ID from *user* II-B and storing friends in *friendsList*, and three *if* instances in it.

- The first one allows to store the *friendsList* into *keys* when the scanner switches to a different user, but both of them belong to the same *map* row (i.e. *user.get(i)%5 != user.get(i-1)%5 && user.get(i)/5 == user.get(i-1)/5*). This instance includes a *for* loop that creates empty IntArrayList for missing users ID.
- The second one has the same function of the previous one, but operates when *map* row index increments. It allows the storage of the *friendsList* into *keys* and *keys* into *map* and generates empty *keys* if the missing users are more than five.
- When the scanner reaches the last user, the third *if* instance permits the correct recollection.

The HashMap generation is now completed.

The real Hash Join operation is performed by *depth_1(int userID)* which is a quick access to the *map*. The method returns an IntArrayList that corresponds to *friendsList* if the user ID requested can be reached or an empty one if it is not reachable (e.g. nonexistent users or when only a subset is loaded).

Given the two different functionalities III-A and III-B the second one results faster because it has only to perform a double access to the HashMap.
A way to speed the traverse functionality up could have been dividing the *for()* loop into two or more Threads; however iterating through *idx* is fast enough to get the first Thread to terminate before the next one begins. This has been tested also with the "most popular" user[1].
The traverse method is located in the CSR class as:
private IntArrayList singleTraverse (int userID){}

IV. TASK 3

Both of the two methods described in III-A and III-B both return an IntArrayList containing the friends of a given user. For this reason the implementation of the join and traverse functionality for $depth > 1$ is the same.
The two methods are:

$$\textit{public ArrayList<IntArrayList> hashJoin(),} \tag{2}$$

$$\textit{public ArrayList<IntArrayList> multipleTraverse().} \tag{3}$$

Both of them need a user ID to start and a chosen depth to know the path[3] size before returning the results to the calling function.

The main method works as follow:

1) Instatiation of eigth[2] ArrayList<IntArrayList> (called *links*).
2) The singleTraverse() / depth_1() is executed in order to obtain the friends of the given user.
3) Seven Threads are declared. Each thread will perform the algorithm for $\frac{1}{8}$ of the given user's friends. The eighth thread is the *for* loop inside the described method.
4) Once the eight threads are completed all the ArrayList<> are concatenated and returned to *main()*.

The description of the *for* loop performed by each thread can be found in IV-A.

A. The followPath() method

The followPath method is recursive. It needs three arguments:

1) IntArrayList *path*: it contains the actual path starting from the given user and ending when depth reaches 0.
2) int *depth*: it is decreased every time *followPath()* is called.
3) ArrayList<IntArrayList> *links*: it will store the *path* once it gets to the end, that is when *depth* reaches 0.

The method has a simple structure: if *depth* reaches 0 it checks via the *duplicate(IntArrayList path)* function if the path doesn't contain any duplicate elements, in that case, the path is added to *links*. If $depth \neq 0$ a *singleTraverse() / depth_1()* is performed in order to obtain the n friends of the last user of the current path. Then each friend is added to a copy of the current path. At the end each copy is passed to *followPath(..., depth-1, ...)*.

B. Time and memory

Since the global operation of the two algorithms is very similar, the differences in time are due to III-A and III-B methods. Indeed it's been tested that the double access performed by III-B to the HashMap is faster than the double access to *ptr* followed by an iteration through *idx* made by III-A. However the needs of III-B to build an HashMap (*HashMap()* method) in order to operate leads to a delay of approximately 0.37 s. This makes the (3) faster for smaller depths while (2) is faster for longer ones.

[1]The most popular user is the nr. 5867 with 8763 edges

[3]By "path" we mean the combination of nodes connected together where $a_1 \neq a_2 \neq ... \neq a_n$. With n equal to the depth set by the user.

[2]We used eigth different ArrayList<IntArrayList> because the algorithm uses eigth threads and ArrayList<>.add() is not Thread safe

This differences are managed by an heuristic engine, described in V that discerns which method is faster in order to get the best execution time for the achievement of the desired path.

Memory wise we decided not to use ArrayList<Integer> to store the single paths. This decision was taken because an Integer needs 128 bits to store an int of 32 bits. In our algorithms representing int as object does not lead to better implementations. Also representing a user ID as String wouldn't have been more memory efficient because a String composed by only four characters (so from user 1000 onward) would have occupied an amount of memory equal or even bigger than the amount needed by an int. Furthermore as an object, a String needs also a pointer to its instance of 32 or 64 bits based on the OS of the computer. On the other hand in our situation a primitive variable like int was more than enough to operate with. Moreover to save even more memory we decided not to use Set<> to save paths, that would have simplified the individuation and consequent elimination of duplicates, because Sets works with Integer. So we implemented our own *duplicate()* function.

Our algorithm doesn't need intermediate results to get to the end of each path with the exception of the HashMap support of the *hashJoin()* method. Despite of these precautions some queries produce a large amount of results (e.g. the query with userID 1 and depth = 5 leads to more than 32 millions different paths) that sometimes cause memory bugs[4].
This can be solved by letting the JVM use more memory. However we also implemented a writer on .txt files to store the results in order not to run out of memory during the execution[5].

V. Task 4

The implementation of an heuristic allows our engine to switch efficiently between CSR and Hash Join in order to achieve the best execution time for the operation.

This implementation is based on several tests focused on depth, the number of user's friends and time. We called Switching Point the moment in which (2) becomes faster than (3). Analysing data, we found a pattern linking the Switching Point, the number of friends at depth=1 and the depth itself.

In Fig.1, the execution time of the two methods increments at different speeds: at the beginning CSR is the fastest, while at depth=4 you can observe the Switching point. We have analyzed a lot of regression techniques (e.g. linear, polynomial, logarithmic, exponential) and the most representative was the logarithmic one with an $R^2 = 0.839$.
The obtained equation is:

$$S.P. := 6.4 - 0.274 \ln(n\_friends) \quad (4)$$

where S.P. stands for the Switching Point.

[4]In particular: java.lang.OutOfMemoryError: Java heap space

[5]Note that writing on a .txt file is a lot slower than storing the results in the *ArrayList<IntArrayList> links*

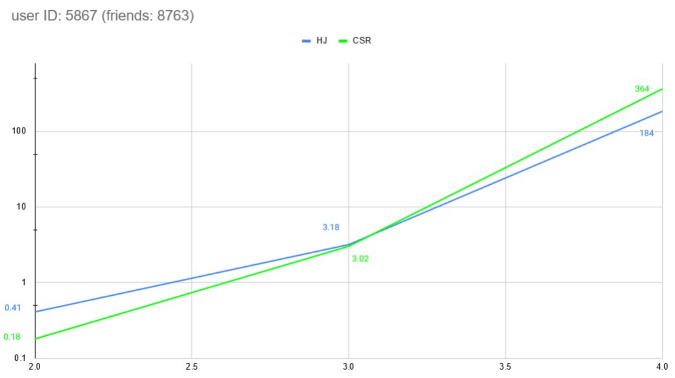

Figure 1. Switching Point at depth=4.

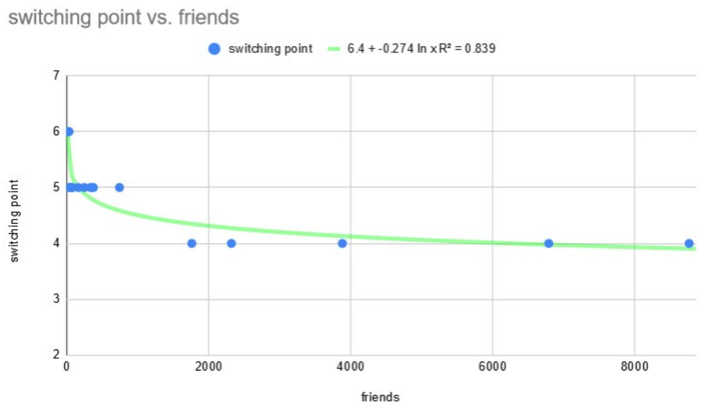

Figure 2. Switching Point for each number of friends.

The equation (4) has been traced in Fig.2: the blue points represent the actual Switching Point while the green curve symbolizes the predicted values.

Our implementation of the heuristic can be found in Main class as *Engine(CSR csr, IntTable table, int userID, int depth)*. It starts by calculating the number of friends (*n_friends*) of the given *userID* through *csr.numberOfFriends(userID)*, then by (4) the predicted S.P. is calculated and compared to depth. If S.P. is equal or smaller than depth, *hashJoin()* is performed else *multipleTraverse()* is run.

VI. Conclusion

In a social network like Pokec these operations are the basis for friends research, even though this implementation doesn't consider the common interests, locations, past activities of the users. In a more specific research, more complex database are used like the one that can be found in [1] as soc-pokec-profiles.txt, which contains users attributes.

It is impressive how many people a person can get in contact with meeting only friends' friends (e.g. user 1340 has 58 friends and could meet 4635 more).

References

[1] J. Leskovec, *Pokec social network*, 2012, https://snap.stanford.edu/data/soc-Pokec.html.

[2] IBM, *Class IntArrayList*, 2018, https://www.eclipse.org/collections/javadoc/9.1.0/org/eclipse/collections/impl/list/mutable/primitive/IntArrayList.html.