Advanced Data Bases

Report 1

Lukas Strack

July 22, 2022

1 Problem statement

In this Exercise we are supposed to join 2 given datasets with our own implementation of Hash Join and Sort-merge Join. The query is as follows:

$$follows \bowtie_{follows.object=friendsOf.subject} friendsOf \bowtie_{friendsOf.object=likes.subject} likes \bowtie_{likes.object=hasRevies.subject} hasRevies$$

2 Algorithm description

For this exercise we implemented 4 different Join algorithms.

1. Hash-Join
2. Sort-Merge-Join
3. Sort-Hash-Join (self creation)
4. Grace-Hash-Join

2.1 Hash-Join

Hash-Join describes a algorithm, which uses the good insert and lookup properties of a hash table. Therefore we scan the relation R and store every *value, key* pair in the hash table. In a second phase we iterate over S and use the constant lookup time to check for matching pairs in the hash table. In theory with out memory limitations, we would only need to scan each relation one time. For the first version of the hash-join we scan S every time the hash map exceeds our memory constraint.

1. For each tuple r in the build input R
   a) Add r to the in-memory hash table
   b) If the size of the hash table equals the maximum in memory-size:
      i. Scan the probe input S, and add matching join tuples to the output relation
      ii. Reset the hash table, and continue scanning the build input R
2. Do a final scan of the probe input S and add the resulting join tuples to the output relation

`https://en.wikipedia.org/wiki/Hash_join`

2.2 Sort-Merge-Join

Sort-Merge-Join describes a join algorithm, which uses the efficient merging property of sorted lists. Therefore we first have to sort the two relations R and S with respect to there keys. After this step we have to merge the two relations.

1. Initialize two "pointer" to the first elements of Relations R and S; $i := 0$, $j := 0$
2. Repeat until one "pointer" reached the limit
   a) if $r_i < s_j$:
      i. increment i
      ii. reset j to last mark
      iii. remove mark from S
   b) if $r_i > s_j$:
      i. increment j
   c) if $r_i = s_j$:
      i. add element to new relation
      ii. if S has no mark, set a mark
      iii. increment j

The challenge for this implementation is to create a Data Type which can store a array or list like element which do not fit in memory. For this reason i created the *BigList* class in my github project.

2.3 Sort-Hash-Join

In this variation of the Hash-Join we sort the first list to not fragment the list so much.

2.4 Grace-Hash-Join

`https://en.wikipedia.org/wiki/Hash_join#Grace_hash_join`
We create a hash map for both Relations and iterate over the keys of one relation and check the hash table of the second relation.

3 Dataset description

Since the dataset is described very detailed here `https://dsg.uwaterloo.ca/watdiv/` i decided to not copy all of this to this report. As brief overview: we have two datasets *100k.txt* and *watdiv.10M.nt* which are both instances of the *Waterloo SPARQL Diversity Test Suite WatDiv dataset* with *100.000* and *10.000.000* entries respectively.

4 Experiment

Source Code and results are available on: `https://github.com/Louquinze/DebisProject2` For this experiment we run all algorithms for 10 run on each dataset and measured the time.

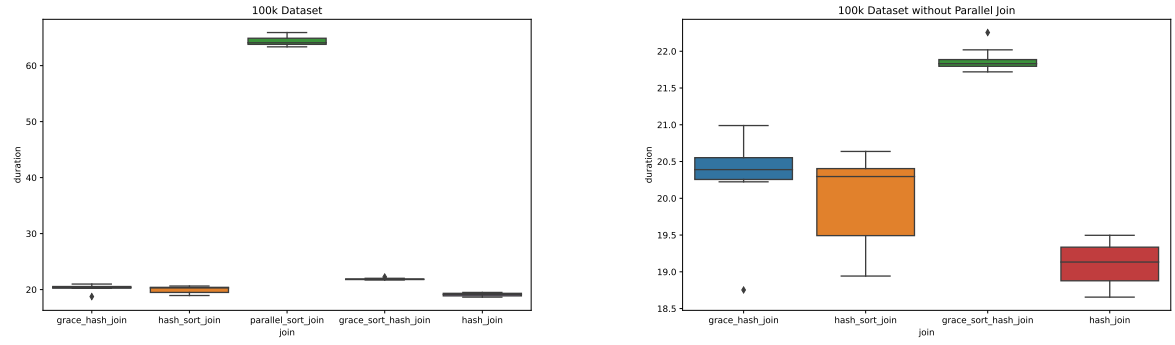

Figure 1: time in seconds for 100k dataset

In Figure 1 we can see that the merge-sort-join needs significant more time as the hash counter parts. For the hash based algorithms it looks like they are not relatively similar. With the normal hash join coming out as the best and the grace hash join as the worst.

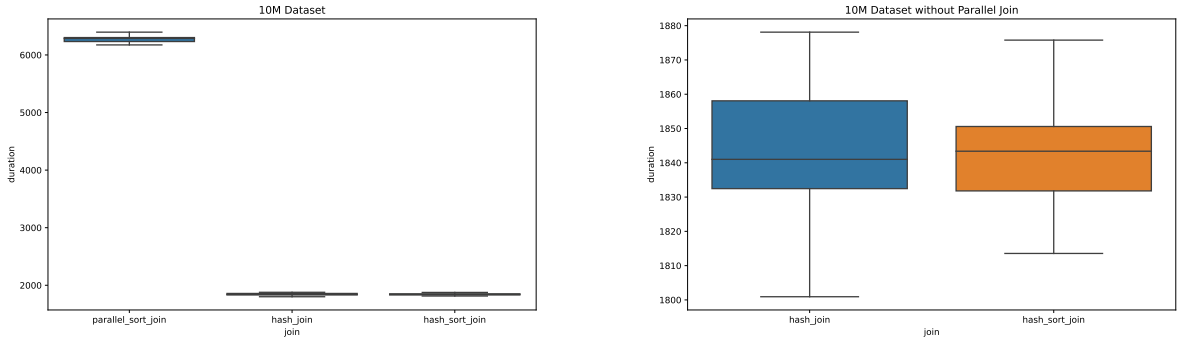

Figure 2: time in seconds for 10M dataset

First of all we see that the grace hash join did not finished. This is due to a programming error, where i used anohter hash table to manage the disk allocation. For the big dataset even this hash table got to huge for the memory.
Second, we see that the merge sort join again is significantly worse as the hash counterparts. For the hash base algorithms the two remaining algorithms do not differ much form each other and therefore i would consider them as similar.

4.1 Parallelizing code

I also tried to parallelize the merge algorthim in a way that each core can make join if more than one are available. But it turned out that the join *follows*/*firendsOf* needed a huge part of the time. This resulted in nearly no speed up for the algorithm.

4.2 human error

All results has to be seen with a grain of salt. The join implementation done by a single master student which also had to program and learn other stuff in the mean time. Therefore i was at a tight time budget and could not use as much time for testing and debugging as i liked to.

5 Conclusion

In my experiments we can clearly see that the hash based algorithms are clearly faster and easier to implement as the sort based algorithms. As soon as we need to manage a strategy to load and store data to the disk the process becomes very complicated.
In my opinion this the reason why the sort merge join and the grace hash join performed so bad. They relay on my own created database management which is pretty provisionally.
That said, the results for the $100k$ dataset should not differ as much since all the data can be stored in memory. Therefore we still can say that the hash algorithm is better as the sort/merge based algorithms.