Question 1. Data processing

For being able to make the algorithms to answer to the required query I have to make aggregation in the *friends(uid1,uid2)* and count the number of friends that each *uid1* has and in the next stage merge the results for counting with data in table *users(uid,network).* The aggregation will be done using both: sort-based and has-based algorithms.

# Sort-based

Before deciding of how to implement the algorithm I have made the assumption that the relationships from table *friends(uid1,uid2)* are unidirectional, meaning that *uid2* is friend of *uid1*  but do not know if *uid2*  also considers that *uid1*  is his friend unless we also have the entry *friends(uid2,uid1)* in the table. By making this assumption I make easier to count the number of friends for each person by only counting the number of apparitions for his uid in the first column in the table.

For this strategy, in the first phase, I have decided to use a mixture between Sort-Merge Join and the aggregation by using the Multi-Way External Merge Sort.

In the first stage I sort the list of *users* by the attribute *uid* by applying the usual merge-sort algorithm. After this the list of friends is sorted by using the Multi-Way External Merge Sort, meaning that blocks of the memory size are read from *friends* and inside of each block the entries are sorted by *uid1* and also the entries with same *uid1* are counted. Since uid2 does not matter anymore, the pairs (*uid1*,*uid2*) are turned into pairs (*uid1*,*count*).

In the next stage the sorted results are merged by attribute *uid* and the result for query are obtained.

**IO cost of the algorithm**:

The costs for the algorithm are:

This happens because after sorting F the size of the list will be same as U and perhaps even smaller if there are users that do not have friends at all. So in the first phase we have the costs for merge sorting list U and list F and at these are added the costs for merging the array U with the sorted array F which has same size as U after the merging with counting is done.

# Hash-based

For the hash-based strategy I have thought about a variant of the Grace Hash Join with some changes.

In the first phase both the *users* and *friends* table are partitioned and for these chunks with size square root of largest table divided by two are loaded into memory and they are split into partitions and the hash key that is used for partitions is the *uid*. After a block is partitioned into buckets and they are loaded into the second disk, another block with the same size is loaded into memory for speeding up the process.

The elements in buckets are grouped so in each bucket with elements from *friends* table with same uid or some specific ids (in case it is needed to don’t exceed a specific number of partitions) are loaded. And the buckets with *users* are created so each bucket contains *ids* from a specific interval (e.g. interval 1-10, 10-20, 20-30). I also have to take care that each bucket will not exceed the size limit of square root of the largest table divided by 2.

In the second phase each one bucket of *users* and one bucket of *friends* is loaded into memory and the entries from *friends* bucket with the same *key* are counted and their number is added for each *user.*

The main difference from the classic Grace Hash Join algorithm is that I count the number of entries with same key (key is same with uid1) and the output is list of users with their network and number of friends.

**IO cost of the algorithm**:

The total cost for this algorithm is 3\*U + 2\*F. This is because both U and F are read from the initial disk, then split into buckets and written on the second disk. After this, U and F are both read from the second disk and after that only users, each with the number of friends are written to the third storage disk because the friends are no longer needed for the result.

Question 2:

# 1. Local wait-for graphs

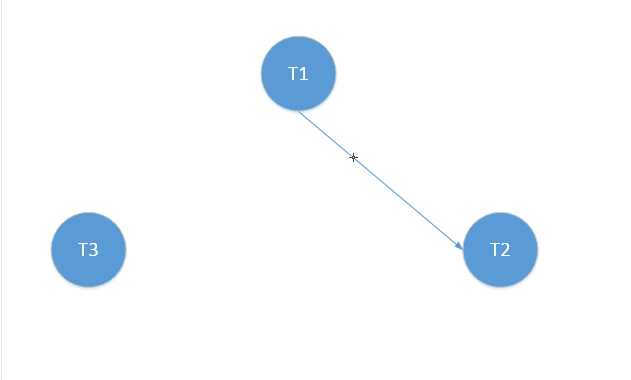


Figure Local wait-for graph for node 1

In node 1 there is precedence between transaction T1 and transaction T2 because T1 reads value from X and T2 writes information in X and to avoid any error then T2 has to wait until T1 finished the reading. There is no deadlock in this node because the precedence graph is acyclic

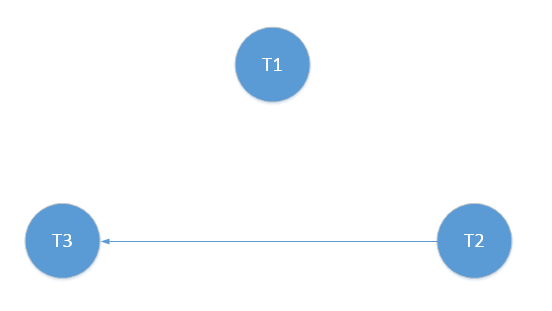


Figure Local wait-for graph for node 2

In this node there is precedence between T2 and T3 because transaction T2 reads value from B and T3 has to write value into B. In this case there is also no deadlock because the graph is acyclic.

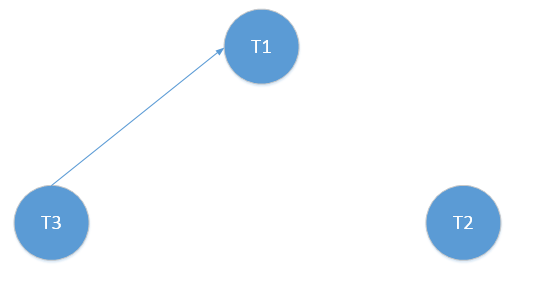


Figure Local wait-for graph for node 3

In node 3 there is precedence between T3 and T1 because T3 reads C and T1 writes into C. Once again, there is no deadlock because graph is acyclic.

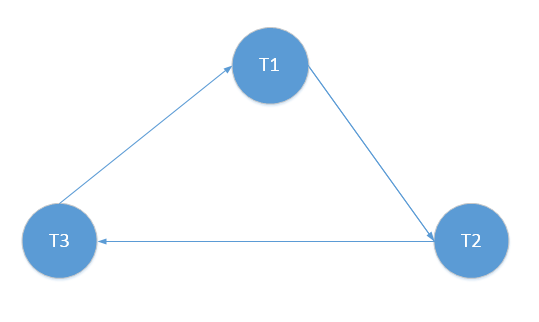


Figure Merge of graphs from Node1, Node2 and Node 3

In case I merge the graphs then it appears to be deadlock since the graph is acyclic.

# 2. Global wait-for graph

I have done the global wait-for graph with the 3 nodes and variables X, Y, A, B, C, D placed in each node. Outside the nodes I have the transaction T1, T2, T3 and the precedences.

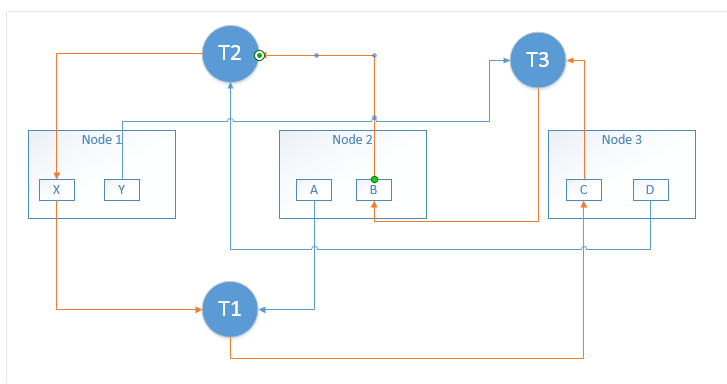


Figure Global wait-for graph

I explain how I have drawn the arrow:

* transaction T1 has to wait for variables X and A which it reads but variable C has to wait for T1 to finish writing
* transaction T2 has to read from D in node 3 and B in node 2 and it variable X has to wait until the transcation finish writing
* transaction T3 has to read to read record C from Node 2 and Y from node 1 and variable B has to wait for T3 to finish.

With yellow orange arrows I have drawn cycle that I have found in graph and this cycle can lead to deadlock. In order to avoid the deadlock I have to remove X, B and C.

# 