# CSE 232B Milestone Three Report

Yuning Hui - A53091905

# 1 Query Rewriter

## 1.1 Partitions

The first step of the rewriter is to identify navigation variables and their dependent variables. Navigation variables are variables whose associated XQueries begin with “doc”, and independent of other variables. The dependent variables begin with other variables. One navigation and all its dependent variables can form a closed partition. Only when two variables from different partitions occur in the same condition do we need to join two partitions. We can consider each navigation variable to be the parent of its dependent variables. Therefore a HashMap *invPartition* is used to store such relation. For example, <”$ta” -> “$a”> means “$a” is the parent of “$ta”. This reverse mapping can help easily find the corresponding navigation variable when examining the variables within condition clauses.

## 1.2 Resolving Non-Joinable Conditions

For all condition clauses with “*eq*” or “=”, they can be classified into two categories, joinable and non-joinable. Joinable condition clauses can initiate a join operator. The two operands of joinable condition clauses must be in two different partitions. We can check them by finding their parent navigation variable in HashMap *invPartition*. Non-joinable conditions, such as “*var eq var”* within same partition or “*var eq constant”* can be resolved by moving them into *where* clause. The *where* clause will be carried into the *for* loop within join clause to filter partition tuples.

## 1.3 Merging Partitions

### 1.3.1 Finding Joinable Conditions

The condition clauses in *where* clause provide explicit clues to join partitions. If two operand variables are in different partitions, the two partitions can be joined. If no joinable conditions can be found, the merging process will be over.

### 1.3.2 Constructing Join Clauses

If we decide to merge two partitions, we need to first find all variables within them. Then, we construct two *for* loops to execute the XQueries binding with each variable. The results of two

loops are tuples with variable values in them. Non joinable variables will be put into *where* clauses within the *for* loops, while joinable variables will be put into the attribute lists.

### 1.3.3 Merging Two Partitions

After the construction of join clause, the two partitions need to be merged into one partition. The idea of **union-find** algorithm can help to implement it. Recall that we store the mapping from dependent variables to their parent navigation variables by HashMap *invPartition.* We also check whether two variables are in the same partition by their parents. Therefore, we can simply change the parent of all variables in the second partition to the first partition.

## 1.4 Constructing Final Query

When there are no partitions to be joined, we can construct the final query. First, we assign each remaining partition a tuple variable, such as “$tuple”, “$tuple1”..., to iterate over each partition. Then, we create a *for* loop to wrap it and add prefixes to the original variables in the *return* clause. Here, we can find the prefix by first finding the partition of the variable and the assigned $tuple variable for that partition.

# 2 Hash Join Operation

The idea for hash join is to hash one operand and use the other operand to match the hash results. We use a HashMap from String Array to Node Array to store the hash results. The string array stores all variable values from attribute list as keys. The Node Array stores all query results with the same variable value combinations. After computing the HashMap, we just perform a get operation for each tuple in the second operand to search for matchings, then combine them.

# 3 Issues

Multiple condition clauses caused an issue for us. When dealing with *for* clause, we just iterated over all variables in a loop. However, when we iterate over all variables in *condition* clauses, the program does not work. It turned out that the *condition* clause are nested. They are not on the same level within the ParseTree. So we had to write a recursive function to find all condition clauses and extract all variables. The function name is *performGetCondList*. When we constructing the final *return* clause, the same problem raised again. We wrote a recursive function to access all nodes in *return* ParseTree to change all variable names.

Another issue is when to perform rewriting. Given there is no partially join, we can go through all condition variables, if all conditions are “var eq constant” or two variables within the same partitions, we will then stop and use the original query, because there is no need to rewrite.