Departmental Coversheet
Hillary term 2022 Mini-project
Paper title: Database System Implementation
Candidate Number: 1058016
Your degree: MSc Advanced Computer Science

1. (a) The data structure in this paper is a six-column triple table. The subject, predicate, and object of a triple are encoded as integers in the first three columns, R_s, R_p, and R_o, respectively. Conceptually, there are three linked lists, an sp-list that connects all triples with the same R_s grouped by R_p, an op-list that associates all triple with the same R_o grouped by R_p, and a p-list that relates all triples with the same R_p without any grouping. In the table, the last three columns, N_{sp}, N_{op}, and N_p, store the next-pointes, which are going to be the row number in the triple table in the actual implementation.

   Six index maps are also maintained. I_s, I_p, and I_o store the head of the sp-list, p-list, and op-list, respectively. I_{sp} maps s and p to the first occurrence of the triple with the same s and p in the sp-list. So does I_{op}. I_{spo} stores the row number of each triple in the table.

   (b) ADD(Triple t) consists of two parts; first, append a new row to the end of the RDF-index table, and second, associates the new row with all six index maps and alters the pointer columns, N_{sp}, N_{op}, and N_p, to point to the correct row.

   Since there is no need to worry about the concurrency in our setting, the RDF-index table is maintained as a fixed size vector of int list of size 6 with each position stands for different columns. Therefore, for each time we add a triple into the table, we simply append it to the last. However, if the triple is already in the I_{spo} map, we skip it. When the size reaches its limit, the entire table will resize.

Algorithm 1 ADD (t)

if t in I_{spo} **then** ▷ $t = (s, p, o)$ is a triple.
  return
end if
i = # of elements in the triple-table
if $i + 1 >$ the size of the triple-table **then**
  resize the triple table
end if
T_{new} = $[t.s, t.p, t.o, -1, -1, -1]$ ▷ The last three columns are left for update later.
triple-table[i] = T_{new}
$I_{spo}[t] = i$
Update remaining indexes

The columns, N_{sp}, N_{op}, and N_p, are maintained in a linked-list-like manner and are updated simultaneously with the index maps. Take N_{op} for example, if T_{new} does not appear in the I_o and I_{op}, it means that T_{new} is a new triple that the triple table never see before. Therefore, we insert T_{new} into the index map I_o and I_{op} (case1). If we found a T with $T.o = T_{new}.o$ and $T.p = T_{new}.p$, then we insert T_{new} after T, and point the next of T_{new} to the original next of the T (case3). A special case is that when there is no next for T and it is handled in case 2 of the Algorithm (2). The case of N_{sp} and N_p is similar.

Algorithm 2 Update $I_{op}(T_{new})$

T = the first triple with $T.o = T_{new}.o$ and $T.p = T_{new}.p$
if T does not exist **then** ▷ Case 1
  make T_{new} the head of I_o and I_{op}
end if
if T does not have T_{next} **then** ▷ Case 2
  make T_{new} the head of I_o and I_{op}
  $T_{new}.N_{op} = T$
end if
if T has T_{next} **then** ▷ Case 3
  $T_{next} = T.N_{op}$
  $T_{new}.N_{op} = T_{next}$
end if

(c) Let $t = \langle s, p, o \rangle$ be a triple pattern and X, Y, and Z be free variables. There are two inputs for EVALUATE; t, the matching pattern, and *index*, the previous state. There are also two outputs, t', the answer to the match pattern, *index*, the current state. The index are usually the pointer to the next triple, but there are two sentinels; *EndOfNode* will tell the caller there is no more next triple to search and *EndSearch* stands for an illegal search. The general frame work is shown in Algorithm (3).

Algorithm 3 Framework

while *index* $\neq$ *EndOfNode* and *index* $\neq$ *EndSearch* **do**
  index, $t' \leftarrow$ Evaluate(t, *index*)
end while

Inside EVALUATE, there are four categories we need to discuss, from no free variable to all three are variables.

When there is not free variable, EVALUATE will check if t is in I_{spo} map, if it is return the triple and set the index to be *EndOfNode*. Otherwise, it will return nothing and set the index be *EndSearch*. The pseudo-code is shown in Algorithm (4).

Algorithm 4 Evaluate $\langle s, p, o \rangle$

Require: $t = \langle s, p, o \rangle$
if t in I_{spo} **then**
  index $\leftarrow$ *EndOfNode*, $t' \leftarrow t$
else
  index $\leftarrow$ *EndSearch*, $t' \leftarrow$ Null
end if

When there is one free variable, there are three cases: $\langle X, p, o \rangle$, $\langle s, X, o \rangle$, and $\langle s, p, X \rangle$. The first and the last cases are similar. For $\langle X, p, o \rangle$, in the first search, we use the index map, I_{op}, to locate the head of the triple with predicate and object the same as o and p, then we return the first match and the next pointer at N_{op}. For all subsequent evaluate, we traverse N_{op} list until there is no match. The pseudo-code is suggested in Algorithm (5).

Algorithm 5 Evaluate $\langle X, p, o \rangle$

Require: $t = \langle X, p, o \rangle$
if hash($t.p, t.o$) does not appear in I_{op} **then**
  index $\leftarrow$ *EndSearch*
end if
if it is the first search **then** ▷ locate the head
  $t' \leftarrow$ the triple I_{op} points to, *index* $\leftarrow t'.N_{op}$.
else ▷ traverse N_{op} list
  $t' \leftarrow t.N_{op}$, *index* $\leftarrow t'.N_{op}$
end if
if $t'.o$ and $t'.p$ are not we are looking for **then** ▷ the end of grouped by.
  index $\leftarrow$ *EndSearch*
end if

$\langle s, p, X \rangle$ is similar. Since we don't have index on s and o, $\langle s, X, o \rangle$ needs some extra works. To match this pattern, we can traverse either sp-list or op-list and skip the triples does not match on s and o. The choose of the list depends on the size of $I_s[s]$ and $I_o[o]$ and we use the smaller one. The pseudo-code is in Algorithm (6).

Algorithm 6 Evaluate $\langle s, X, o\rangle$

Require: $t = \langle s, X, o\rangle$, where X is a special code stands for a free variable.
  if $|I_s[s]| < |I_o[o]|$ **then**
    do
      if it is the first search **then**
        $t' \leftarrow$ the triple I_{sp} points to, *index* $\leftarrow t'.N_{sp}$.
      else
        $t' \leftarrow t.N_{sp}$, *index* $\leftarrow t'.N_{sp}$
      end if
    while $t'.s \neq s$ and $t'.o \neq o$ and *index* $\neq$ *EndOfNode*
  else
    do
      if it is the first search **then**
        $t' \leftarrow$ the triple I_{op} points to, *index* $\leftarrow t'.N_{op}$.
      else
        $t' \leftarrow t.N_{op}$, *index* $\leftarrow t'.N_{op}$
      end if
    while $t'.s \neq s$ and $t'.o \neq o$ and *index* $\neq$ *EndOfNode*
  end if
  if $t'.s \neq s$ and $t'.o \neq o$ **then** ▷ corner case
    index = *EndSearch*
  end if

When there are two free variables, there are also three cases: $\langle X, Y, o\rangle$, $\langle X, p, Z\rangle$, and $\langle s, Y, Z\rangle$, where X, Y, and Z can be equal. Their idea is similar; $\langle X, Y, o\rangle$, for example, we first find $I_o[t.o]$ and traverse the op-list till the end. If $X = Y$, then we skip those triples with $p \neq o$. The pseudo-code is given in Algorithm (7)[1].

Algorithm 7 Evaluate $\langle X, Y, o\rangle$

Require: $t = \langle X, Y, o\rangle$, where X and Y are special codes stand for variables.
  do
    if it is the first search **then**
      $t' \leftarrow$ the triple I_o points to, *index* $\leftarrow t'.N_{op}$.
    else
      $t' \leftarrow t.N_{op}$, *index* $\leftarrow t'.N_{op}$
    end if
  while $X = Y$ and $t'.p \neq t'.s$ and *index* $\neq$ *EndOfNode*
  if $X = Y$ and $t'.p \neq t'.s$ **then** ▷ stop traversing when $t'.p \neq t'.s$ when the pattern asks $X = Y$
    index = *EndSearch*
  end if

The idea for the other two patterns is similar; starts from I_p or I_s map and traverse the p-list or sp-list.

When there are three free variables, the paper suggests to match, for example, the patterns like $\langle X, Y, Z\rangle$, we need to iterate over the triple table; if we want $X = Y$, we skip those $X \neq Y$. However, this is not efficient. Therefore, I modified a little bit and the idea and pseudo-code are discussed in 2.a, where I put everything that is different from paper there.

[1]Even I found myself a little confused by what I wrote in the do-while loop, so this footnote might be helpful. After a few minutes of thinking, I found only when we want $X = Y$, the condition $t'.p \neq t'.s$ will make the loop start and try to locate the one with $t.p = t.s$. If $X \neq Y$, the loop will never start and return immediately.

2. (a) RDF indexing data structure that implements Add and Evaluate functions.

This component is included in RDF_index.cpp, in which ADD and EVALUATE are implemented as suggested in the problem 1. There are a few things that is slightly different from the paper.

1) I used XXHASH[2] by Facebook instead of Jenkings hashing, because it achieves state-of-the-art excellent performance on both long and small inputs and it is true. Jenkings hashing is implemented in RDF_index.h and is commented. I did some tests and observed that Jenkings is about 10% slower than XXHASH.

2) Instead of open addressing, I eventually choose std::unordered_map for index maps I_{sp}, I_{op} and I_{spo}. I had an open addressing hash implemented in HashTable.cpp and HashTable.h (attached in the submission), but it was much slower than the unordered_map (by about 20% or more).

3) To match the patterns like $\langle X, Y, Z\rangle$, the paper suggests to iterate over the triple table; if we want $X = Y$, we skip those $X \neq Y$. I modify this a little bit to improve the efficiency. Again, $\langle X, X, Z\rangle$, for example, we first iterate I_s, and for each s in I_s, we find if I_{sp} includes hash(s, s), if yes, we traverse over the triple table. As shown in the pseudo-code in Algorithm (8)[3].

Algorithm 8 Evaluate $\langle X, X, Z\rangle$

```
for s in I_s do
    i = hash(s, s)
    if i in I_sp then
        Evaluate_SPZ(s, s)
    end if
end for
```

$\langle X, Y, Y\rangle$ and $\langle X, Y, X\rangle$ are similar. For $\langle X, X, X\rangle$, we only need to traverse s in I_s and find if $\langle s, s, s\rangle$ is in I_{spo} as shown in Algorithm (9).

Algorithm 9 Evaluate $\langle X, X, X\rangle$

```
for s in I_s do
    Evaluate_SPO(s, s, s)
end for
```

(b) The engine for evaluating BGP SPARQL queries.

This part is in the file SPARQL_engine.cpp and it is implemented strictly based on the model answer. A few additional lines are added for printing or improving performance.

(c) The greedy join order optimization query planner.

This part can be sought in the file query_planner.cpp. I see what the model answer is trying to say, but I believe there are some minor mistakes, so I made some slight changes, but it is still $O(n)$, where n is the number of triple patterns we try to fit.

There were three things I modified. Firstly, the last three lines should go to the outside while loop. Secondly and most importantly, I changed the criteria for updating the new triple patterns. In the model answer, the criteria is

$$t_{best} = \perp \text{ or } score < score_{best} \text{ and either } var(t) = \emptyset \text{ or } var(t) \cap B \neq \emptyset. \tag{1}$$

It is troublesome because it will always take the first unprocessed triple pattern as the best pattern and compare this with the remaining. However, it might cause some issues in many cases. For example, the

[2]https://cyan4973.github.io/xxHash/

[3]The actual implementation is slightly different. I put the check of if i in I_{sp} at the begging of the Evaluate_SPZ to reduce some code redundancy, but they are equivalent.

following triple patterns

$$\begin{aligned}\langle X, \quad 1, \quad 2\rangle\\ \langle Y, \quad 2, \quad 3\rangle\\ \langle X, \quad 4, \quad Y\rangle\end{aligned}$$

will produce the plan $\langle X, 1, 2\rangle \mapsto \langle Y, 2, 3\rangle \mapsto \langle X, 4, Y\rangle$ because even after we replace the X in $\langle X, 4, Y\rangle$, this triple pattern still has higher selectivity than $\langle Y, 2, 3\rangle$. Therefore, this plan will create an unnecessary cross product, and I believe the criteria (1) meant choosing the triple pattern that has the lowest selectivity and includes some variables that are in the processed patterns. Thus, I made some modifications to the algorithm, and it will 1) as long as an unprocessed pattern has a variable that is processed, we will always consider this pattern and 2) if this pattern has lower selectivity than the previous one, we choose this pattern as the best triple. With this modification, the plan becomes $\langle X, 1, 2\rangle \mapsto \langle X, 4, Y\rangle \mapsto \langle Y, 2, 3\rangle$.

Thirdly, there are some cases that $|\langle X, 1, 2\rangle| = 10,000$ but $|\langle Y, 2, 3\rangle| = 10$. If we ignore the size of the pattern, we might produce a lot of unnecessary joins, but we can store this information while we are updating I_{op}. Therefore, I also record the size of $I_{op}[op]$ and $I_{sp}[sp]$, and we choose the best triple pattern based on both selectivity and size. Our final query plan is $\langle Y, 2, 3\rangle \mapsto \langle X, 4, Y\rangle \mapsto \langle X, 1, 2\rangle$, which is the most optimal now. Notice that only I_{op} and I_{sp} are considered because they are the most common two cases, and recording the size of other index maps would produce a lot of overhead in creating and indexing the database. The majority of the overhead is due to updating unordered_map and vector.

The pseudo-code goes as the following.

Algorithm 10 New-Plan-Query(U)

$P \leftarrow []$, $B \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
  $t_{best} \leftarrow \perp$, $score_{beset} \leftarrow 100$, intersected $\leftarrow$ false
  for each unprocessed tripple pattern $t \in U$ **do**
    $score \leftarrow$ the position of t in $\prec$ where the variables in B are considered bounded
    if $t_{best} = \perp$ **then**
      intersected = $(var(t) \cap B \neq \emptyset)$
      $t_{best} \leftarrow t$, $score_{best} \leftarrow score$
    end if
    if t and t_{best} are both of the form $\langle X, p, o\rangle$ **then** ▷ $\langle s, p, X\rangle$ is similar.
      if $I_{op}[t] < I_{op}[t_{best}]$ **then**
        $t_{best} \leftarrow t$, $score_{best} \leftarrow score$
      end if
    end if
    if intersected **then**
      if $score < score_{best}$ and either $var(t) = \emptyset$ or $var(t) \cap B \neq \emptyset$ **then**
        $t_{best} \leftarrow t$, $score_{best} \leftarrow score$
      end if
    else
      if $var(t) = \emptyset$ or $var(t) \cap B \neq \emptyset$ **then** ▷ Make sure that the next pattern share some variables.
        $t_{best} \leftarrow t$, $score_{best} \leftarrow score$, intersected $\leftarrow$ true
      end if
    end if
  end for
end while

I also did some experiments showing that the strategy works. The testing protocol is the same as the one in question 3. From table 1, we see a significant improvement in most of the queries. However, updating

	Model Answer	New Query Plan
q1	4.69	0.095
q2	4579.05	36.433
q3	12.623	0.107
q4	1.649	0.773
q5	18.714	2.708
q6	8.351	8.417
q7	30227.6	4.295
q8	264.673	63.044
q9	17057.8	65.302
q10	18.469	0.092
q11	0.916	0.873
q12	0.703	0.334
q13	16.833	17.902
q14	6.336	6.789

Table 1: Time took to process and evaluate the query by the model answer and the new query plan in ms on the data set LUBM-001-mat.ttl. Both two are run without O3 flag to avoid some internal optimizations affect the results.

the additional index makes the load 20% slower from 723 ms to 930 ms without O3 flag.

(d) The component for parsing and importing Turtle files.
This part is included in Turtle_handler.cpp.

(e) The parser for SPARQL queries
This part is included in query_parser.cpp.

(f) The component implementing the command line
This part is included in interface.cpp.

3.a (a) Hardware and Software Configuration:

i. Model Identifier: MacBookPro14,3
ii. Processor Name: Quad-Core Intel Core i7
iii. Processor Speed: 2.9 GHz
iv. Number of Processors: 1
v. Total Number of Cores: 4
vi. L2 Cache (per Core): 256 KB
vii. L3 Cache: 8 MB
viii. Memory: 16 GB
ix. Operating System: macOS Big Surf, Version 11.6 (20G165)
x. Compiler Version: Apple clang version 11.0.0 (clang-1100.0.33.8)

(b) We set a timeout if one instruction takes more than 3 minutes. The following protocol can be found in protocol.cpp. If you want to run the load test, fill in the correct path to the data and uncomment the lines. It will take about 15 minutes to run.

Loading Data Test Protocol:

i. Turn off all irrelevant applications.
ii. Start a clean terminal.
iii. Load dataset 001 and markdown the time. If timeout, mark TO.
iv. Delete all objects.
v. Load dataset 010 and markdown the time. If timeout, mark TO.
vi. Delete all objects.
vii. Load dataset 100 and markdown the time. If timeout, mark TO.
viii. Delete all objects.
ix. Go back to step ii repeat this process for 10 times.
x. Take the average.

To eliminate the cache effect, I load 3 datasets one by one for 10 times instead of running a single dataset for 10 times. This will flush most of caches. Also, I should play my phone and don't touch my laptop while running.

(c) We choose COUNT instead of SELECT and for each query. We set a timeout if one instruction takes more than 3 minutes.

Loading Data Test Protocol:

i. Turn off all irrelevant applications.
ii. Start a clean terminal.
iii. MAKE all and run the output file.
iv. Load one dataset.
v. Run 14 queries one by one and markdown the time.
vi. Go back to the step v and repeat 10 times.
vii. Take the average.

To avoid the cache effect, I run 14 queries one by one for 10 times instead of running a single queries for 10 times.

3.b The time needed to load and index the data and the time needed to produce all query answers for each RDF graph and query are showed in Table 2 and Table 3, respectively. O3 flag is used[4].

	LUBM-001-mat	LUBM-010-mat	LUBM-100-mat
Load & Index Time	242.153	4648.9	52543.5

Table 2: Time needed to load and index the data in ms.

[4]Looking at Table 1 and Table3, I am very surprised by how large O3 can improve.

	LUBM-001-mat	LUBM-010-mat	LUBM-100-mat
q1	0.0135	0.0173	0.0274
q2	5.3407	81.835	973.923
q3	0.0175	0.0261	0.037
q4	0.1237	0.1609	0.1897
q5	0.4445	0.6328	0.777
q6	1.1039	21.381	185.05
q7	0.6371	9.4067	118.804
q8	10.1652	12.1391	14.2116
q9	9.1657	132.945	1640.17
q10	0.0136	0.021	0.0286
q11	0.1625	0.219	0.2554
q12	0.0495	0.4495	0.6043
q13	2.8269	46.3258	635.152
q14	0.9463	16.8604	150.809

Table 3: Time needed to produce all query answers in ms.

3.c q1: This query consists of two triple patterns, and they are both in the form of $\langle X, p, o\rangle$. My query plan picks the second one as a start because it is considerably smaller (1874 v.s. 4). Otherwise, it will be very slow, as Table 1 suggested.

q2: This query contains two groups of the forms $\langle X, p, o\rangle$, and $\langle X, p, Z\rangle$, and each group has three triples. The plan is to find a $\langle X, p, o\rangle$, then a $\langle X, p, Z\rangle$ that matches the first pick, and pick whatever matches the variable, Z, in the remaining unprocessed patterns. It picks not only the least selectivity pattern but also the smallest pattern in each step, resulting in a very short running time (compared to the model solution in Table 1).

q3: This one is similar to q1, but they have different sizes in each pattern. In LUBM-001-mat.ttl, q3 has 5999 possible matches in the first pattern and 6 in the second; q1 has 1874 in the first and 4 in the second. Therefore, q3 is slightly slower than q1.

q4: Though this query looks terrifying, the speed is fast because all patterns share the same X in p, and besides this, there is at most one free variable in each. Therefore, once X is picked, the remaining is just to check if Y can go with X.

q5: This one is similar to q1 and q3 but is much slower because the search space is large. This one has 8330 possibilities for the first pattern and 719 for the second, so even starting from the second one would slow down by a lot.

q6: This query simply matches just one pattern. No plan is needed. From the smallest to the largest dataset, the result sizes are 7790, 99566, and 1048532. So the time increases as we expected.

q7: There are two triples like $\langle X, p, o\rangle$. Since the first one has the size 7790 and the second one is 1627, starting at the second one, the performance is much better (see Table 1).

q8: All three datasets produce 7790 results, so the time is about the same. We also have a reasonable query plan because pattern 1 has 7790 matches in the smallest dataset but 1048532 matches in the largest dataset. However, pattern 3 has 239 possible matches among all datasets, which is what the plan chooses to start.

q9: It is like q8, but the first three patterns vary in size among three datasets and increase by 12x. So the time increases as the dataset goes large.

q10: Same as q1 and q3.

q11: Like q10, the performance of q1, q3, q10, and q11 are good because they choose the pattern with the smallest size to start.

q12: It is almost like q11. It first finds the smallest of the form $\langle X, p, o\rangle$ and matches $\langle X, p, Y\rangle$. Then all others are just $\langle s, p, o\rangle$ and can be verified very quickly.

q13: The plan first takes $\langle X, p, o\rangle$ and then $\langle s, p, X\rangle$. There are 8330 choices for the first pattern but just one for the second. If we do the second one first, it will be quick, but the selectivity is more important than the actual size by the greedy assumption in the paper.

q14: Same as q6. The running time increases as the size increases.