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 alter 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 $T.o = T_{new}.p$, 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 M_{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.o = 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
```

There

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. There are a few things that is slightly different from the paper.

   1) I used XXHASH[1] by Facebook instead of Jenkings hashing, because it achieves state-of-the-art excellent performance on both long and small inputs.

   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 so I changed it after one update.

   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 (3).

Algorithm 3 Evaluate $\langle X, Y, Z\rangle$

```
for s in I_s do
    i = hash(s, s)
    if i in I_sp then
        Evaluate_SPZ(s, s)                ▷ Evaluate_SPZ
    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 (4).

Algorithm 4 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 files SPARQL_engine.cpp. And 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.

   I see what the model answer is trying to say but I believe there are some minor mistakes so I change it slightly but it is still $O(n)$, where n is the number of triple patterns we try to fit.

   There were two 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 following triple patterns

$$\begin{array}{lll} \langle X, & 1, & 2\rangle \\ \langle Y, & 2, & 3\rangle \\ \langle X, & 4, & Y\rangle \end{array}$$

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

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 $\langle Y, 2, 3\rangle$. However, 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. Therefore, the new pseudo-code goes as the follow.

Algorithm 5 New-Plan-Query(U)

$P \leftarrow [], B \leftarrow \emptyset$
while $U \neq \emptyset$ **do**
    $t_{best} \leftarrow \perp$, $score_{beset} \leftarrow 100$, intersected = 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 = ($var(t) \cap B \neq \emptyset$)
            $t_{best} \leftarrow t$, $score_{best} \leftarrow score$
        else 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 = true
            end if
        end if
    end for
end while

In the new query plan, still, we take the first unprocessed triple pattern as our best plan at the beginning. However, we will mark down if the first pattern shares some common variables with some processed patterns. If it is, we will select the least selectivity pattern that also share some common variables in the remaining as in the model answer. However, if the first triple pattern does not share any processed variable, we pick the first pattern that shares some processed variables and then we search for the least selectivity pattern as before. The new plan in the example will now become $\langle X, 1, 2\rangle \mapsto \langle X, 4, Y\rangle \mapsto \langle Y, 2, 3\rangle$.

I also did some experiments showing that the strategy works. The testing protocol is the same as the one in the question 3.

(d) The component for parsing and importing Turtle files.

This part is included in query_parser.cpp. My parser not only read the query but also check if the triple patterns are correct such as if the query is asking for something that is not presented in the database, it will throw a warning right away instead of feeding it into the query engine to create overhead.

(e) The parser for SPARQL queries

This part is included in interface.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

	Model Answer	New Query Plan
q1	7	8
q2	10107	22
q3	33	25
q4	2	2
q5	66	37
q6	36	25
q7	64267	133
q8	801	128
q9	46215	250
q10	31	29
q11	1	1
q12	0	0
q13	33	35
q14	19	18

Table 1: Time took to process and evaluate the query by the model answer and the new query plan in ms.

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) Test Protocol:

i. Turn off all irrelevant applications.
ii. Start a clean terminal.
iii. make all and run the output file.
iv. If it is measuring the time needed to load and index the data, markdown the time and repeat this process for 10 times. Take off the highest and the lowest and find the average.
v. To measure the time needed to evaluate the query, we choose COUNT instead of SELECT and for each query, we markdown the time and repeat this process for 10 times. Take off the highest and the lowest and find the average.

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.

	LUBM-010-mat	LUBM-010-mat	LUBM-010-mat
Load & Index Time	1064	11879	122583

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

	LUBM-001-mat	LUBM-010-mat	LUBM-100-mat
q1	8	95	817
q2	22	393	25387
q3	25	261	2533
q4	2	26	211
q5	37	349	3489
q6	25	254	2376
q7	133	1375	14250
q8	128	1060	11367
q9	250	2618	26922
q10	29	328	3305
q11	1	13	119
q12	0	2	27
q13	35	322	3485
q14	18	194	1809

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

3.c (a) There