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

To run the code, try make all. There will be an output file RDF_DB.out. Run the file and you can enter the database interface.

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.

   Apart from the table, 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 triple with a brand new $T_{new}.s$ and $T_{new}.p$. 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 framework 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, from no free variable to three variables.

When there is not a 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].

[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.

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.

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 4% slower than XXHASH in average[3].

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)[4].

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).

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

[3] For loading, XXHASH and Jenkings took 314.359, 4026.82, and 323.914, 4206.85, and 58582.4 ms on three datasets, respectively, achieving 2%, 4%, and 8% speed up on each.

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

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 $O(n^2)$, where n is the number of triple patterns we try to fit.

There were four things I modified.

Firstly, the last three lines should go to the outside while loop.

Secondly, 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

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

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 plan. Notice that only I_{op} and I_{sp} are considered because they are the most common cases, and recording the size of other index maps would produce overhead in indexing. Since the size of $I_{op}[t]$ and $I_{sp}[t]$ can be recored while we are inserting triple, the cost of maintaining the size is negligible[5].

Lastly, the original query plan was based on the following precedence relation $\prec$ on selectivity of triple patterns:

$$(s, p, o) \prec (s, ?, o) \prec (?, p, o) \prec (s, p, ?) \prec (?, ?, o) \prec (s, ?, ?) \prec (?, p, ?) \prec (?, ?, ?).$$

Now, since we have maintained the size of $(?, p, o)$ and $(s, p, ?)$, a new order of the selectivity should be used:

$$(s, p, o) \prec (s, ?, o) \prec (?, p, o) = (s, p, ?) \prec (?, ?, o) \prec (s, ?, ?) \prec (?, p, ?) \prec (?, ?, ?).$$

[5]There are many tricks to save memory. For example, let's say I_{sp} is an unordered_map<int, int>, where the first int is the hash key and the second int is the index of row number. For a table size smaller than 2^{24} rows, we can use the higher 7 bits to store the size and lower 24 bits to store the index. Other trick might apply also.

Therefore, when we are precessing these two patterns, instead of comparing the selectivity, we can directly compare the actual size of the triple patterns and select the smaller one. For example, for the following triple patterns,

$$\langle X, \quad 1, \quad 2\rangle$$
$$\langle 2, \quad 3, \quad Y\rangle$$

with $|\langle 2,3,Y\rangle| > |\langle X,1,2\rangle|$ will produce the following query plan,

$$\langle 2, \quad 3, \quad Y\rangle$$
$$\langle X, \quad 1, \quad 2\rangle$$

The pseudo-code goes as the following.

Algorithm 10 New-Plan-Query(U)

```
P ← [], B ← ∅
while U ≠ ∅ do
    t_best ←⊥, score_beset ← 100, intersected ← false
    for each unprocessed tripple pattern t ∈ U do
        score ← the position of t in ≺ where the variables in B are considered bounded
        if t_best =⊥ then
            intersected = (var(t) ∩ B ≠ ∅)
            t_best ← t, score_best ← score
        end if
        if t and t_best are of the form ⟨X, p, o⟩ or ⟨s, p, X⟩ then
            t_best ← the one with a smaller size
        end if
        if intersected then
            if score < score_best and either var(t) = ∅ or var(t) ∩ B ≠ ∅ then
                t_best ← t, score_best ← score
            end if
        else
            if var(t) = ∅ or var(t) ∩ B ≠ ∅ then        ▷ Make sure that the next pattern share some variables.
                t_best ← t, score_best ← score, intersected ← true
            end if
        end if
    end for
end while
```

From Table 1, we see a significant improvement in most of the queries. The testing protocol is the same as the one in question 3. To do the test, uncomment lines 20-25 in query_planner.cpp and comment lines 26-45 and MAKE TEST. The analysis is included in 3.c.

(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:

	Model Answer (ms)	New Query Plan (ms)	Improvement (%)
q1	1.0781	0.0137	78.6934307
q2	1132.55	4.495	251.957731
q3	3.3425	0.0166	201.355422
q4	0.3796	0.1035	3.66763285
q5	5.1757	0.4164	12.429635
q6	1.3064	0.965	1.35378238
q7	7093.21	0.5548	12785.1658
q8	70.2673	8.4899	8.2765757
q9	3783.54	7.3296	516.200065
q10	5.0822	0.011	462.018182
q11	0.2163	0.1319	1.6398787
q12	0.1445	0.0394	3.66751269
q13	3.828	0.0075	510.4
q14	0.9798	0.8059	1.2157836

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. The new plan achieves more than 1000% speed up in average on the 14 queries.

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 some lines. It will take about 15 minutes to run.

Loading Data Test Protocol:

i. *Restart computer* and 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. *Restart computer* and 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.

	LUBM-001-mat (ms)	LUBM-010-mat (ms)	LUBM-100-mat (ms)
Load & Index Time	314.359	4026.82	53525.3

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

	LUBM-001-mat (ms)	LUBM-010-mat (ms)	LUBM-100-mat (ms)
q1	0.0152	0.0143	0.0248
q2	4.4338	72.8246	708.436
q3	0.0162	0.0239	0.0274
q4	0.1119	0.1494	0.1827
q5	0.3815	0.5832	0.7189
q6	0.974	19.8416	186.464
q7	0.0674	0.0898	0.1306
q8	8.3644	11.2896	14.8031
q9	6.1915	100.174	1376.28
q10	0.0106	0.0202	0.0278
q11	0.1376	0.2023	0.2621
q12	0.0409	0.4233	0.5893
q13	0.0075	0.0388	0.5113
q14	0.7846	15.3656	141.389

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 with the smallest size is picked, the remaining is just to check if a Y can go with the 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. It also illustrate that we have an excellent 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. Therefore, by choosing pattern 3 as a start, we ensure that the time does not change as the size of pattern 1 changes.

q9: It is like q8, but the sizes of the first three patterns increase by 12x from the smallest dataset to the largest. Therefore, even if we pick the smallest parttern to start with, the time will still increase.

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: There are 8330 choices for the first pattern but just one for the second. If we do it in order, it will be slow. However, the plan first takes $\langle s, p, X \rangle$ and then $\langle X, p, o \rangle$ because they have the same selectivity but different in sizes.

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