CS764 Interim Course Report
Ranked Enumeration of Conjunctive Query Results

Group members: Tien-Lung Fu, Pan Wu

Nov 18, 2019

1 Description of the Problem

For many data processing applications, enumerating query results according to an order by a ranking function is fundamental task. For example, users want to extract the top patterns from a an edge-weighted graph, and the rank of each pattern is the sum of the weights of the edges. Ranked enumeration also exists in SQL queries with an ORDER BY clause. Usually, users want to see the first k results as quickly as possible without predetermined value of k. Our project is according to given ranking function, We investigate the enumeration of top-k answers for conjunctive queries against relational databases. Our main task is to design and implement data structure and algorithm that allow for efficient enumeration after a pre-processing phase. We designed and implemented a novel priority queue based algorithm with near-optimal delay and non-trivial space guarantees that are output sensitive and depend on structure of the query. Our algorithms are divided into two phases: the pre-processing phase, where the system constructs a data structure that can be used later and the enumeration phase, when the results are generated. All of our algorithms aim to minimize the time of the pre-processing phase, and guarantee a logarithmic delay during enumeration.

2 Related Work

Top-k ranked enumeration of join queries has been studied extensively by the database community for both certain and uncertain databases. Most recent work has focused on enumerating twig-pattern queries over graphs. Our work departs from this line of work in 2 aspects: (1) use of novel techniques that use query decompositions and clever tricks to achieve better space requirement and formal delay guarantees (2) our algorithm are applicable to arbitrary hypergraphs as compared to simple graph patterns over binary relations. The most related works are [1] and [2]. Algorithm in [1] is fundamentally different from

ours. It uses an adaptation of Lawler-Murty's procedure to generate candidate output tuples which is also a source of inefficiency given that it ignores query structure. [2] presented a novel anytime algorithm for enumerating homomorphic tree patterns with worst case delay and space guarantees where the ranking function is sum of weights of input tuples that contribute to an output tuple. Their algorithm also generates candidate output tuples with different scores and sorts them via a priority queue. However, the candidate generation phase is expensive and can be improved substantially.

3 Implementation

We next discuss in detail the preprocessing and enumeration phase of the algorithm.

Preprocessing.

Algorithm 1: Preprocessing Phase

```
1  foreach t ∈ V(T) do
2  |  materialize the bag B_t
3  perform full reducer pass on materialized bags in T
4  forall t ∈ V(T) in post-order traversal do
5  |  foreach valuation v in bag B_t do
6  |  |  u ← v[key(B_t)]
7  |  |  if Q_t[u] is NULL then
8  |  |  |  Q_t[u] ← new priority queue
9  |  |  ℓ ← []
10 |  |  foreach child s of t do
11 |  |  |  ℓ.append(&Q_s[v[key(B_s)]].top())
12 |  |  Q_t[u].insert(⟨v, ℓ, ⊥⟩)
```

We define $key(B_t)$ to be the common variables that occur in the bag B_t and its parent.

The algorithm consists of two steps. The first step is to materialize each bag B_t, and then we apply a full reducer pass to remove all tuples from the materialized bags that will not join in the final result.

The second step initializes the hash map with the priority queues for every bag in the tree. We traverse the decomposition in a post-order fashion, and do the following. For a leaf node t, notice that the algorithm does not enter the loop in line 10, so each valuation v over B_t is added to the corresponding priority queue as a $Cell$ of the triple $\langle v, [], null \rangle$. For a non-leaf node t, we take each valuation v on $key(B_t)$ and form a $Cell$ with the top rank elements from the corresponding priority queues within its children, as line 11 shown. The $Cell$ is then added to the corresponding priority queue of the bag. Observe that the root node r has only one priority queue, since $key(r) = \{\}$.

Consider the running example as below:

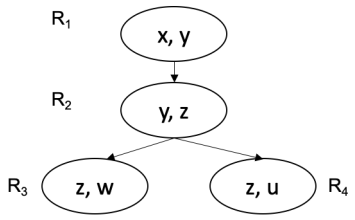
$$Q(x, y, z, w) = R_1(x, y)R_2(y, z)R_3(z, w)R_4(z, u)$$

where the ranking function is the sum of the weights of each input tuple. Consider the following decomposition for our running example.

id	x	y	weight_1
1	1	1	1
2	2	1	2

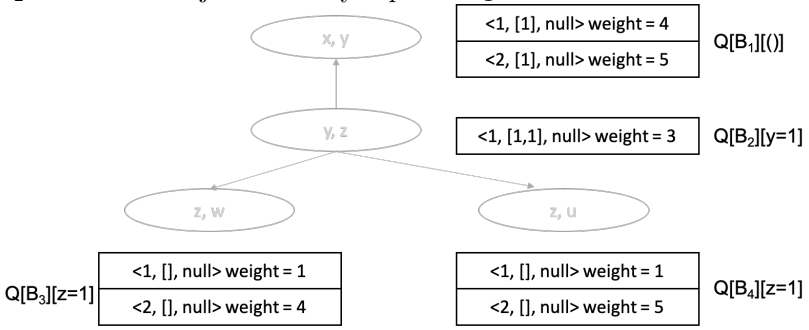
id	y	z	weight_2
1	1	1	1
2	3	1	1

id	z	w	weight_3
1	1	1	1
2	1	2	4



id	z	u	weight_4
1	1	1	1
2	1	2	5

For the instance shown above and the query decomposition that we have fixed, relation R_i covers bag B_i, $i \in [4]$. Each relation has size $N = 2$. Since the relations are already materialized, we only need to perform a full reducer pass, which can be done in linear time. This step removes tuple $(3, 1)$ from relation R_2 as it does not join with any tuple in R_1.



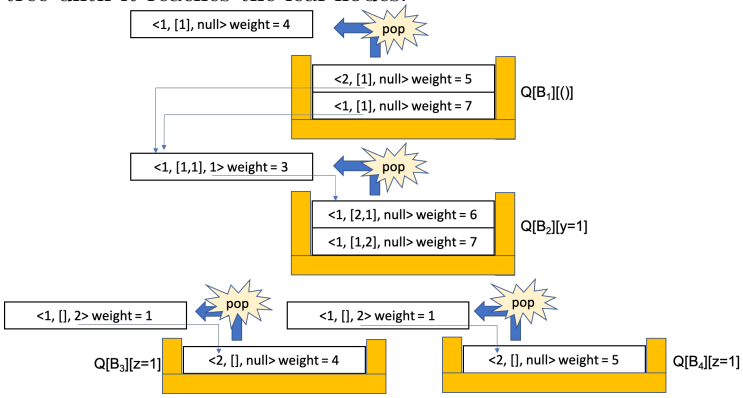
The above graph shows the state of priority queues after the preprocessing step. For convenience, v in each $Cell$ $\langle v, [p_i], next\rangle$ is shown using the tuple id and p_i represents the top tuple id of the i-th child. The $Cell$ in a memory location is followed by the partial aggregated score of tuple formed by creating the tuple from the pointers in the cell recursively. For instance, the score of the tuple formed by joining $(y = 1, z = 1) \in R_2$ with $(z = 1, w = 1) \in R_3$ and $(z = 1, u = 1) \in R_4$ is 1+1+1=3 (shown as $\langle 1, [1, 1], null\rangle weight = 3$ in the figure). Each $Cell$ in every priority queue points to the top element of the priority queue of child nodes that are joinable. Note that since both tuples in R_1 join with the sole tuple from R_2, they point to the same $Cell$ $\langle 1, [1, 1], null\rangle$.

Enumeration.

Algorithm 2: Enumeration Phase

```
1  PROCEDURE enum()
2      while 𝔔_r[()] is not empty do
3          output 𝔔_r[()].top()
4          topdown(𝔔_r[()].top(), r)
5  PROCEDURE topdown(c, t)
6      /* c = ⟨v, [p_1, ..., p_k], next⟩ */
7      u ← v[key(ℬ_t)]
8      if next = ⊥ then
9          𝔔_t[u].pop()
10         foreach child t_i of t do
11             p'_i ← topdown(*p_i, t_i)
12             if p'_i ≠ ⊥ then
13                 𝔔_t[u].insert(⟨v, [p_1, ..., p'_i, ... p_k], ⊥⟩)
14         if t is not the root then
15             next ← &𝔔_t[u].top()
16     return next
```

The main idea of the algorithm is that, whenever we want to output a new tuple, we can simply obtain it from the top of the priority queue in the root node (node r is the root node of the tree decomposition). Once we do that, we need to update the priority queue by popping the top, and inserting if necessary new valuations in the priority queue. This will be recursively propagated in the tree until it reaches the leaf nodes.



The above graph shows the state of the data structure after one iteration in $enum()$. The first answer returned to the user is the topmost tuple from $Q[B_1][()]$. *Cell* $\langle 1, [], null\rangle$ $weight = 4$ is popped from $Q[B_1][()]$. We recursively call *topdown* for child node B_2 and $\langle 1, [1, 1], null\rangle$ $weight = 3$. The *next* for this cell is also *null* and we pop it from $Q[B_2][y = 1]$. At this point, $Q[B_2][y = 1]$ is empty. The next *topdown* call is for B_3 and $\langle 1, [], null\rangle$ $weight = 1$. The *next* for $\langle 1, [], null\rangle$ $weight = 1$ is updated to $\langle 2, [], null\rangle$ $weight = 4$ which leads to creation and insertion of $\langle 1, [2, 1], null\rangle$ $weight = 6$ into $Q[B_2][y = 1]$. After the last *topdown* call for B_4 and $\langle 1, [], null\rangle$ $weight = 1$, we can get the result as the graph shows.

4 Plan about Next Step

Given a user-defined ranking function, we will verify the correctness of our implementation. Furthermore, we will compare the performance between our implementation and standard SQL engine. Finally, we will also compare the performance between our implementation and [2].

5 Reference

[1] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In International Workshop on Next Generation Information Technologies and Systems, pages 141–152. Springer, 2006.
[2] X. Yang, D. Ajwani, W. Gatterbauer, P. K. Nicholson, M. Riedewald, and A. Sala. Any-k: Anytime top-k tree pattern retrieval in labeled graphs. In Proceedings of the 2018 World Wide Web Conference on World Wide Web, pages 489–498. International World Wide Web Conferences Steering Committee, 2018.