Optimization of Kleene Closure Regular Path Query Processing Based on Node Clustered Index

Lulu Yang[1], Tenglong Ren[1], Xiaowang Zhang[1,2,*], Fan Feng[1] and Guopeng Zheng[1]

[1] College of Intelligence and Computing, Tianjin University, Tianjin 300350, China
[2] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China

Abstract

The *regular path query* (RPQ) consisting of Kleene closure (∗) is the most complex and difficult to handle due to the infinite recursion of Kleene closure. The infinite recursive process of Kleene closure can be preprocessed. However, the current optimization methods are less concerned with reducing query time through preprocessing. In this paper, we propose Node Clustered Index (NCI), and generating NCI in the preprocessing can solve the reachability problem on graphs. Moreover, we design a Kleene closure pattern tree decomposition algorithm, which combines with selective design to generate locally optimal query plans. Experimental results show that NCI can largely optimize Kleene closure regular path queries, and the RDF data sizes have little impact on query efficiency.

Keywords

SPARQL, RPQ, Kleene Closure, Node Clustered Index

1. Introduction

Kleene closure regular path queries have higher expressiveness than ordinary regular path queries. In real life, RPQ has been used in many fields, such as friendships in social networks, signalling pathways in protein interaction networks, and connections between two different organisms in bioinformatics. Besides the methods [1, 3] based on automata, there are also methods [2, 4, 5, 6] based on path index to optimize Kleene closure regular path queries. The representative work [5] is to use the landmark index to label constrained accessibility query, which calculates the accessibility information between all individual nodes and landmarks. Such work pays little attention to the infinitely recursive nature of Kleene closure. The tree structure proposed in work [2] solves the infinite recursion problem of Kleene closure, but its tree structure stores irrelevant nodes on the path, and it takes time to traverse the tree structure during query execution. Our work is optimized based on [2], and a new data structure is proposed, which directly clusters the path results and reduces the storage of unrelated nodes to optimize time and space.

In order to solve the problem of infinite recursion of Kleene closure in RPQ, we designed the preprocessing storage structure, and the system architecture is shown in Fig. 1. The contributions of this paper are as follows:

ISWC'22: The 21th International Semantic Web Conference, October 23–27, 2022, Hangzhou

*Corresponding author.
luluyang@tju.edu.cn (L. Yang); tenglongren@tju.edu.cn (T. Ren); xiaowangzhang@tju.edu.cn (X. Zhang); 2119216025@tju.edu.cn (F. Feng); guopengzheng@tju.edu.cn (G. Zheng)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)
CEUR-WS.org/Vol-3254/paper355.pdf

- We propose a new type of node clustered index (NCI). The NCI is built in the data preprocessing, mainly to store the results of queries with Kleene closure in the form of clusters.
- We design a regular expression decomposition algorithm, which can decompose complex regular expressions into ternary trees in units of Kleene closure.
- We propose a selective design to optimize the query execution.

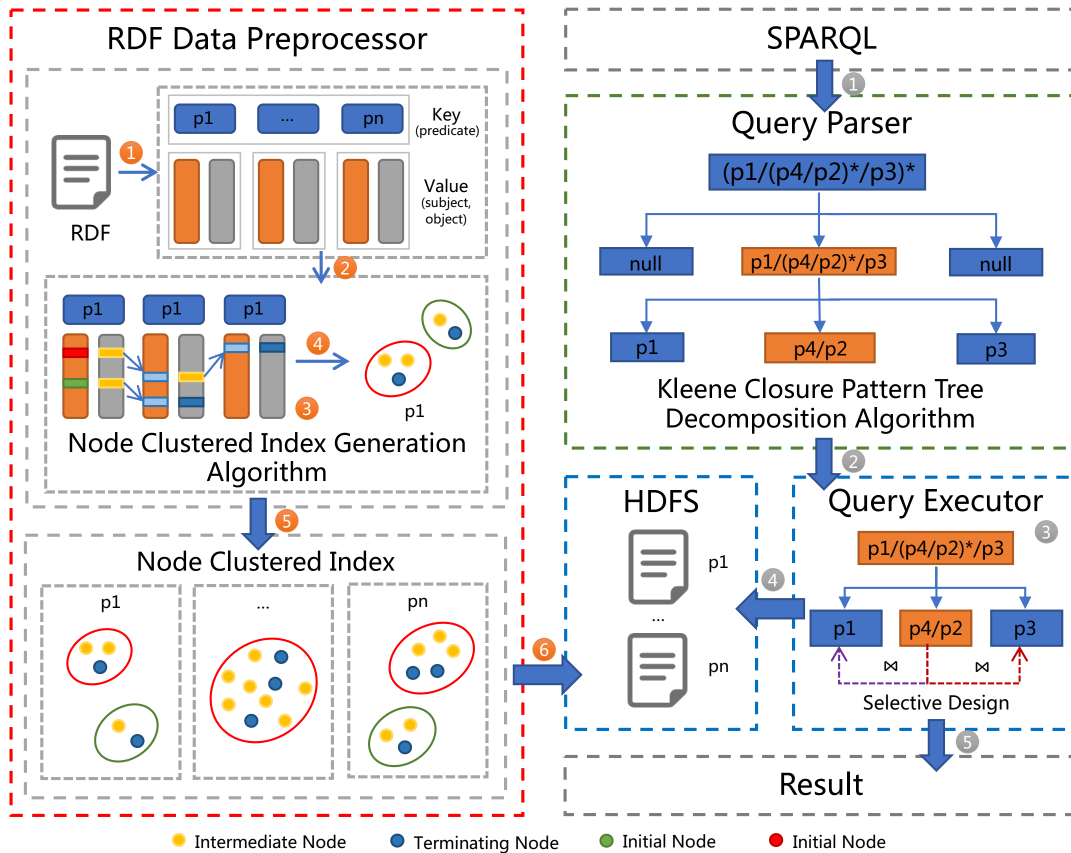

Figure 1: System Architecture. The **RDF Data Preprocessor** outputs the *NCI* through the *Node Clustered Index Generation Algorithm*. The **Query Parser** outputs the regular expression pattern tree through the *Kleene Closure Pattern Tree Decomposition Algorithm*. The **Query Executor** combines *selective design* to parse the pattern tree from the bottom up. The specific problem is identified when the system cannot handle the query correctly.

2. Query Optimization

2.1. Node Clustered Index

The NCI is a data storage method, and the specific details depend on its implementation. In short, the NCI can cluster the query results of the predicate (or expression) consisting of the Kleene closure. Algorithm 1 demonstrates the NCI generation process.

Taking the predicate *knows* as an example, the specific process of the NCI generation algorithm is as follows:

1. Get the subject-object table (value) of the current predicate *knows* (key) based on key-value storage.
2. Get the starting point table and the ending point table of the node cluster, where the starting points refer to the subject that does not appear in the object column, and the ending points refer to the object that does not appear in the subject column.
3. Traverse the starting point table to get the object corresponding to the current starting point (subject), take this object as the connection point, and carry out the self-join of the subject-object table until the object exists in the ending point table. Save all objects, and we can get the NCI with this starting point as the core.
4. Repeat step 3 until the starting points are traversed.

The result of the predicate *knows* consisting of Kleene closure can be aggregated data (the set of nodes on the path) through Algorithm 1. When we query (*knows*)$*$, the system returns the set of results directly.

2.2. Query Plan Generation

The query executor optimizes queries by parsing the pattern tree of the *Kleene Closure Pattern Tree Generation Algorithm* combined with the *Selective Design* to generate the query plan from the bottom up.

2.2.1. Kleene Closure Pattern Tree Decomposition Algorithm

If the complex(nested) Kleene closure query is directly queried as a whole, it will undoubtedly cost much time due to the infinite recursion of Kleene closure. Therefore, we propose a unique Kleene closure pattern tree decomposition algorithm.

The basic idea is to decompose the expression consisting of Kleene closure as a unit and decompose three leaf nodes: $prefix$, $infix$, and $postfix$, as shown in the Query Parser section of Fig. 1. The $infix$ refers to the string modified by the current Kleene closure, the $prefix$ refers to the string before the current Kleene closure unit, and the $postfix$ refers to the string located after the current Kleene closure unit. The algorithm is a recursive process until there is no expression consisting of Kleene closure in the leaf nodes. Finally, we can get a complete pattern tree.

Algorithm 1 Node Clustered Index Generation Algorithm

Input: $exper$;

Output: $nciSet$;

1: $keyValueTable$, $startPointSet$, $endPointSet$, $nciSet$, $interSet \leftarrow \emptyset$;
2: $keyValueTable = getKeyValue(exper)$;
3: Separate $keyValueTable$ to get $keySet$, $valueSet$;
4: $startPointSet = keySet - valueSet$;
5: $endPointSet = valueSet - keySet$;
6: **for all** $startPoint \in startPointSet$ **do**
7: $\quad interSet \leftarrow \emptyset$;
8: $\quad valuePoint = keyValueTable(startPoint)$;
9: $\quad interSet.add(valuePoint)$;
10: $\quad interNode = valuePoint$;
11: $\quad$ **while** true **do**
12: $\quad\quad interNode = SELFJOIN(keyValueTable, interNode)$;
13: $\quad\quad interSet.add(interNode)$;
14: $\quad\quad$ **if** $interNode \in endPointSet$ **then**
15: $\quad\quad\quad$ *break*;
16: $\quad\quad$ **end if**
17: $\quad$ **end while**
18: $\quad nciSet.add(interSet)$;
19: **end for**
20: **return** $nciSet$.

2.2.2. Selective Design

We use the bottom-up scheme for the parsed pattern tree to join the results. The result of the root node is related to the three-leaf nodes so that we can consider the optimal join order locally. We design equations 1 and 2 to determine the join order of $prefix$, $postfix$, and $infix$.

$$sel(prefix) = \frac{T(prefix)}{T(infix)} \tag{1}$$

$$sel(postfix) = \frac{T(postfix)}{T(infix)} \tag{2}$$

Where $sel(\cdot)$ denotes the selection degree of the current leaf node, and $T(\cdot)$ denotes the current leaf node table size. The specific implementation is to compare $sel(postfix)$, $sel(prefix)$, and who is the smallest. The benefits of this selective design are as follows:

1. In the case of 0 matches, the number of joins between the two tables is minimal.
2. In the case of full matches, the maximum amount of useful data in $infix$ will be filtered out while minimizing the number of joins.

3. Experiment and Evaluation

We ran experiments on Intel (R) Xeon (R) CPU E5-4607 v2@2.60GHz, 4 cores, and 62 GB RAM. We used the dataset (LUBM[1]) to compare the SPARQL query system (JENA, Blazegraph, Virtuoso) on the query set[2] (covering various types[3] of Kleene closures). Also, to evaluate the system's scalability, we compared it with different data sizes (10, 100, 200) of LUBM. The query results are shown in Fig. 2, and our proposed system is NCI_RPQ.

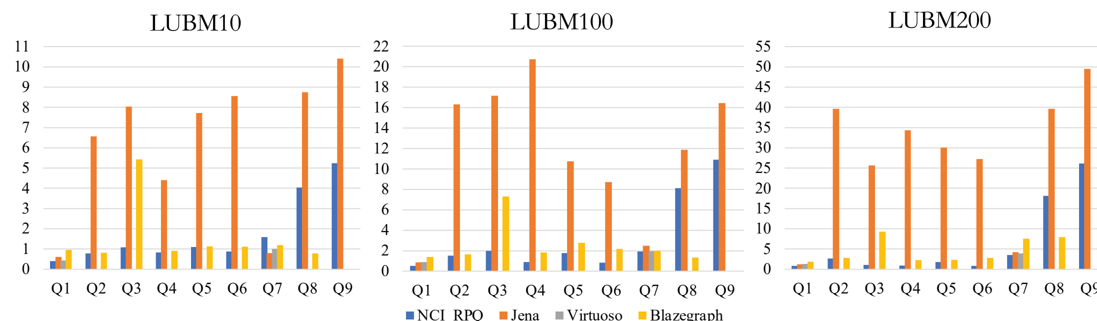

Figure 2: Query time (s) for queries on different data sizes.

The experimental results show that the efficiency of NCI_RPQ is less affected by the data sizes, especially compared with Jena, and the average query efficiency of NCI_RPQ is increased by at least 8.2 times when querying Q2-Q6, which is all due to NCI. In addition, compared to all other systems, the NCI_RPQ can handle various types of Kleene closure regular path queries due to the decomposition algorithm's strong applicability.

4. Conclusion

Our methods solve the problem of infinite recursion of Kleene closure in RPQ, and has great scientific research value. However, our work is still inadequate, and further optimization of NCI storage space and selective design based on deep learning are subsequently considered.

References

[1] Arroyuelo D , Hogan A , Navarro G , et al. Time- and Space-Efficient Regular Path Queries on Graphs[J]. 2021.

[2] Fan Feng, Optimization of Kleene Closure Regular Path Query Based on Tree Structure[Master Thesis], Tianjin University, 2022.

[3] Yakovets N , Godfrey P , Gryz J . Query Planning for Evaluating SPARQL Property Paths[C]// International Conference on Management of Data. ACM, 2016.

[1]http://swat.cse.lehigh.edu/projects/lubm

[2]https://github.com/luer9/RPQ_QUERY/blob/main/RPQ_QUERY.md

[3]Four types: general (Q1), single predicate (Q2, Q3), expression (Q4, Q5, Q6), complex (Q7, Q8, Q9).

[4] Liu B , Wang X , Liu P , et al. PAIRPQ: An Efficient Path Index for Regular Path Queries on Knowledge Graphs[J]. Springer, Cham, 2021.

[5] Valstar L D J , Fletcher G H L , Yoshida Y . Landmark Indexing for Evaluation of Label-Constrained Reachability Queries[C]// the 2017 ACM International Conference. ACM, 2017.

[6] Fletcher G H L , Peters J , Poulovassilis A . Efficient regular path query evaluation using path indexes[C]// Extending Database Technology. OpenProceedings.org, 2016.