Multi Reachability Query Processing

Jiefeng Cheng, Jeffrey Xu Yu, and Bolin Ding

The Chinese University of Hong Kong, China
{jfcheng,yu,blding}@se.cuhk.edu.hk

Abstract. There is a need to efficiently navigate on a large graph to find a certain piece of information by some structure constraints. Reachability query could be the main means for such a purpose and existing approaches studied an efficient primitive called reachability join. In this paper, we focus on processing the query containing multi reachability joins (R-joins). We studied the processing ability of the up-to-date alternative that extended the well-known tree-specific method, namely, twig-stack join algorithm, to be applicable on DAGs. Due to the rapid increasing of the transitive closure size for graphs in stead of trees, however, we found the proposed dynamic programming approach that combines R-join primitives into an optimized plan based on cost tends to be superior over the alternative and more stable in terms of the density of the underneath DAGs than the alternative. According to the extensive experiments we've done, our method is faster than the existing work by at least two orders of magnitudes.

1 Introduction

With the rapid growth of World-Wide-Web, new data archiving and analyzing techniques bring forth a huge volume of data available in public, which are graph structured in nature including hypertext data, semi-structured data and XML [1]. A graph provides great expressive power for people to describe and understand the complex relationships among data objects. As a major standard for representing data on the World-Wide-Web, XML provides facilities for users to view data as graphs with two different links, the parent-child links (document-internal links) and reference links (cross-document links). In addition, XLink (XML Linking Language) [7] and XPointer (XML Pointer Language) [8] provide more facilities for users to manage their complex data as graphs and integrate data effectively. Besides, RDF [3] explicitly describes semantical resource in graphs.

Upon such a graph, a primitive operation, reachability join (or simply R-join) was studied [11, 6]. In brief, a reachability join, $A \rightarrow D$, denoted R-join, is to find all the node-pairs, $(a, d)$, in the underneath large data graph such as $d$ is reachable from $a$, denoted $a \sim d$, and the labels of $a$ and $d$ are $A$ and $D$ respectively. Such R-join helps users to find information effectively without requesting them to fully understand the schema of the underneath graph. We explain the need of such R-join using an XML example. In Figure 1, it shows a graph representation (Figure 1 (b)) for an XML data (Figure 1 (a)). In Figure 1 (b), solid links represent document-internal links whereas dashed links represent cross-document links. We consider Figure 1 (b) as a graph with all links being treated in the same way. With R-join, we can easily find all the topics that a researcher...
is interested in using researcher→topic. However, it would be difficult for a user to find the same information using XPath queries, because XPath supports document-internal links using a descendants-or-self-axis operator // and cross-document links using value-matching based on a notion called ID/IDREF in XML. It cannot find such information using an XPath query, researcher//topic, because topic is a child of proj, and there is an ID/IDREF from researcher to proj. XPath requests users to fully understand the schema and understand that the two different kinds of links are processed in two different ways in XML data. In this paper, we focus ourselves on optimizing and processing multi R-join queries.

A query with more than one R-joins could be naturally be represented by a query graph. To process such a query graph, there are existing approaches [5, 12] that extended the well-known tree-specific method, namely, twig-stack join algorithm [4], to be applicable on DAGs. We studied the processing ability of the most up-to-date one method from such a family and found its performance is quite sensitive to the density of the underneath DAG. We give explanation for this observation. This makes our first contribution. For the second contribution, we found the proposed dynamic programming approach that combines R-join primitives into an optimized plan based on cost tends to be superior over the alternative and more stable in terms of the density of the underneath DAGs than the alternative. Our last contribution is that we conducted extensive experimental studies on multi R-joins queries processing using XMark data [9], which confirms that the efficiency of our approach.

The rest of paper is organized as follows. Section 2 gives our problem statement on multi reachability join query processing. Section 3 introduces the existing processing technique extended from tree-specific join approach. Together with the motivation for our approach, we discuss drawbacks of such an approach for multi R-joins queries processing. In Section 4, we briefly introduces the multiple interval encoding for DAGs, followed by a detailed description about fitting the existing R-join primitive and dy-
namic programming paradigm into a cost-based optimizer for multi R-join queries. Section 5 reports the performance evaluation on our proposed method. Section 6 concludes this paper.

2 Multi Reachability Joins

We consider a database as a directed node-labeled data graph $G = (V, E, L, \phi)$. Here, $V$ is a set of elements, $E$ is a set of edges, $L$ is a set of labels, and $\phi$ is a mapping function which assigns a node a label. Given a label $l \in L$, the extent of $l$ is defined as a set of nodes in $G$ whose label is $l$, denoted $ext(l)$. Below, we use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of a graph $G$, respectively. Such a data graph example is shown in Figure 1 (b).

A reachability join, $A \rightarrow D$, called R-join, is to find all the node-pairs, $(a, d)$, in the data graph $G$ such that $d$ is reachable from $a$, denoted $a \sim d$, and $\phi(a) = A$ and $\phi(d) = D$. We also use $D\leftarrow A$, instead of $A\rightarrow D$, if needed. $A\rightarrow D \equiv D\leftarrow A$. In this paper, we concentrate on processing conjunctive multi R-join queries in the form of

$$A\rightarrow B \land B\rightarrow C \land \cdots \land X\leftarrow Y$$

The following holds for R-joins.

- **Asymmetric:** $A\leftarrow B \neq B\leftarrow A$.
- **Transitive:** If $A\leftarrow B \land B\rightarrow C$ hold, then $A\leftarrow C$.
- **Associative:** $(A\leftarrow B)\leftarrow C \equiv A\leftarrow (B\leftarrow C)$.

A multi R-joins query can be represented as a directed query graph, $G_q(V_q, E_q, L_q, \lambda)$. Here, $V_q$ is a set of nodes. The node-label of a node $v \in V_q$ is represented as $\lambda(v)$. An edge $v \rightarrow u$ represents a R-join $A \rightarrow D$, where the labels of $v$ and $u$ are $A$ and $D$, respectively. A graph representation of a multi R-joins query, $A\leftarrow C \land B\rightarrow C \land C\rightarrow D$, is shown in Figure 2.

We evaluate a query graph $G_q(V_q, E_q, L_q, \lambda_q)$ over a data graph $G(V, E, L, \phi)$. The result of the query graph, $G_q$, denoted $\mathcal{R}(G_q)$, consists of a set of $n$-ary tuples. A tuple consists of $n$ nodes in the data graph $G$, if the query graph $G_q$ has $n$ nodes ($|V(G_q)| = n$), in the form of $t = [v_1, v_2, \ldots, v_n]$, where there is a one-to-one mapping between $v_i$ in $t$ and $u_i$ in $V(G_q)$ such that $\phi(v_i) = \lambda(u_i)$. In addition, all nodes in the $n$-ary tuple $r$ satisfy all the reachability join conditions specified in the query graph $G_q$.

**Example 1** Fig. 2 represents a simple multi R-joins query as a directed graph. This query graph has a node labeled **Institute**, a node labeled **researcher** and a node label

\[ A\rightarrow B \land B\rightarrow C \land \cdots \land X\leftarrow Y \]

The chain query $A\rightarrow B \land B\rightarrow C \land \cdots \land X\leftarrow Y$ is abbreviated to $A\rightarrow B\rightarrow C\rightarrow \cdots \rightarrow X\leftarrow Y$.
topic. And two edges are in the query graph. The edge from Institute node to researcher node requires that the data node pair \((i, r)\), \(i \in \text{ext(Institute)}\) and \(r \in \text{ext(researcher)}\), such that \(i \sim r\), should be returned; in the same time, the edge from researcher node to topic node requires that the data node pair \((r, t)\), \(r \in \text{ext(researcher)}\) and \(t \in \text{ext(topic)}\), such that \(i \sim r\), should be returned.

### 3 Twig-Join-Extended Approach and Our Motivation

In this section, we briefly review an existing approaches to process multi \(R\)-joins based on an interval-based coding scheme and give the motivation for our method. The Interval-based encoding scheme is widely used for processing queries over an XML tree, where a node \(v\) is encoded with a pair \([s, e]\) where \(s\) and \(e\) together specifies an interval. Given two nodes, \(u\) and \(v\) in an XML tree, \(u\) is an ancestor of \(v\), \(u \prec v\), if \(u.s < v.s\) and \(u.e > v.e\) or simply \(u\)’s interval contains \(v\)’s. Twig-Join is proposed in [4] to efficiently find all matches for a twig query against XML trees.

#### 3.1 TwigStack\(\mathcal{D}\) for Directed Acyclic Graphs

Recently, as an effort to extend Twig-Join in [4] to be workable on graphs, Chen et al. studied multi \(R\)-join query processing(called pattern matching) over a directed acyclic graph (DAG) in [5].

The test of a reachability relationship in [5] is broken into two parts. First, like the existing interval-based techniques for processing pattern matching over an XML tree, they first check if the reachability relationships can be identified over a spanning tree generated by depth-first traversal of a DAG. Table 2(a) lists the intervals from a spanning tree over the DAG of our running example. Second, for the reachability relationship that may exist over DAG but not in the spanning tree, they index all non-tree edges( named remaining edges in [5]), and all nodes being incident with any such non-tree edges in a data structure called SSPI in [5]. Thus, all predecessor/successor relationships that could not be identified by the intervals alone could be found with the help of SSPI. For our running example, Table 2(b) shows SSPI.

For example, the whole procedure to find the predecessor/successor relationship of \(17 \sim 21\) in the DAG of Fig. 1 is like this. First, we check the containment of tree intervals for 17 and 21, but we could not identify \(17 \sim 21\). Then, because 21 has entries
of predecessor in SSPI, we again try to find a reachability relationship between 17 and all 21’s predecessors in SSPI by checking the containment of tree interval for 17 and that of each of 21’s predecessors in SSPI. However, we still could not identify such a relationship relationship that far. Again, we recursively repeat the last step and we finally find that the tree interval of 17 contains that of 19. So we successfully identified 17 → 21. Note that in this method, the reachability relationship between two nodes does not exist only if none reachability relationship could be found recursively for all predecessors throughout the whole SSPI.

To process R-joins over a DAG, Chen et al. proposed a stack-based approach, denoted TwigStackD, overall in two parts.

1. Using Twig-Join algorithm in [4] to find all matches exist in the spanning tree of a given DAG for a twig/DAG query. In the mean while,
2. for each nodes popped out from stacks used in Twig-Join algorithm, TwigStackD buffers all nodes which participate in any partial solution in a bottom-up fashion, and maintains all reachability links among those nodes. When a top-most node that participates a solution found, TwigStackD enumerates in the buffer pool and outputs all full solutions headed by that node.

Example 2 For our running example, we use the query in Fig. 2 to demonstrate TwigStackD algorithm, which uses SSPI index and operate on three interval lists for Institute, researcher, and researcher. The Twig-Join part of TwigStackD will found no solution. Consider the point when the cursor from Twig-Join part arrives to node 17, which is the last node to be scanned in the researcher list with minimum start value. At that time, the buffer pool for topic contains node 20 and 21 (Recall that TwigStackD grows up solutions in a bottom-up fashion and so 20 and 21 are partial solutions). When the Twig-Join part of TwigStackD still could not find matches on 17 and pop out it, the second part of TwigStackD check if 17 could reach any nodes in the buffer pool of topic, that is, 20 and 21. TwigStackD finds 17 → 20 does not exist. Because entries in SSPI relevant to node 20 are recursively traversed, that is, 20, 13 and 4, and for all of them, no associated intervals found to be contained in that of 17. Then TwigStackD continues to check if 17 → 21 and it is true known from the above description. So 17 is added to the researcher buffer pool. Similar procedure goes for node 3 upon which TwigStackD find the top-most node that leads to the solution [3, 17, 21].

3.2 Processing Ability of TwigStackD— Our Motivation

In order to identify a reachability relationship between two nodes, say, a and d. TwigStackD need to recursively search on SSPI to check if an ancestor of d could be reached by a. This overhead over a DAG can be costly. Consider the DAG of $2n - 1$ nodes in Fig. 2, where the solid line is edges in the spanning tree generated by a depth-first search, and dashed lines are the remaining edges. Then the processing for the reachability relationship from node $n + 1$ to node $n$ will require the check if $n$ is reachable, one by one, from node $n + 1$ to node $2n - 1$, and there are total $n - 2$ times checking, since all of them are listed as the predecessors of $n$ in SSPI. So, the time for such an operation is in $O(|V|)$. Recall the second step of TwigStackD, where relying on SSPI, the reachability relationship between every node popped from stacks and every nodes in the buffer pool
are checked. Therefore, the I/O cost of using TwigStackD to process R-joins queries is considerable.

Fig. 4 shows the performance of TwigStackD on 5 DAGs, which are obtained by including in the resulting spanning tree 10%, 20%, 30%, 40% and 50% of the remaining edges that are not in the spanning tree from an original DAG. The original DAG is obtained by collapse all strong connected components in a graph that is from a small XMark dataset with a factor 0.01 (16K nodes) and take both XML tree edge and ID/IDREF links as the edges in the graph. Queries Q1 and Q4 listed in Fig. 7 (a) and (d) were used here. For the TwigStackD processing, all curves are titled Q1(TSD) and Q4(TSD).

Fig. 4 (a) shows the I/O number to process each query increased dramatically as more edges being added to the underlying DAG. As an example, for query Q4, the I/O number increased by 4606 from 10% to 20% on the y-axis, while it increased by 38881 from 40% to 50% on the y-axis. When 5 times number of remaining edges being included, the I/O number increased about 35 times. As for the number of index seek of SSPI, namely the number of times to seek an leaf page from the B+-Tree that implements the SSPI, which is showed in Fig. 4 (c), this value increased by 616052 from 10% to 20% on the y-axis, while it increased by 5201991 from 40% to 50% on the y-axis. The correlation coefficient for such two types of measurements is as hight as above 0.999, which speaks such an behavior for the number of I/Os during processing is mainly coursed by the number of index seek of SSPI. Similar situation for processing time could also be observed in Figure 4 (c), since the I/O number is the dominating factor for total processing cost. This test empirically showed that TwigStackD performs better for DAGs with fewer remaining edges, but its performance degrades rapidly when more edges being included in the underneath DAG.

One reason of the drawbacks of SSPI could be that to avoid storage of transitive closure of a DAG, TwigStackD trades time efficiency for space optimality. Thus, we choose to use a multiple interval encoding scheme for a DAG proposed in [2] which tries to efficiently manage the transitive closure for a DAG on both time and space. Besides, there exists efficient join algorithm[11] that processes a size 2 R-join query. Therefore, the main motivations to our approach is as follows,
– Using more efficient encoding scheme in [2] that requires not so large space as much as the transitive closure but still efficiently handles reachability queries for DAGs;
– Extending the existing size 2 R-join query processing scheme to be applicable to multi R-join queries with arbitrary size.
– Using long-established dynamic programming techniques to combine efficient R-join primitives into an optimized plan that evaluates multi R-join queries.

Fig. 4 (a) and (b) gives us a preliminary view on the efficiency of our approach, where curves are titled \( Q1(DP) \) and \( Q4(DP) \). It is obvious that our approach is by far not so sensitive as TwigStackD is to the density of the DAG and the overall performance of our approach is far better than TwigStackD. For example, for \( Q4 \) processing over the five DAGs, all numbers of I/Os are below 200 and the elapse times are below 1 second.

4 A New Dynamic Programming Approach
Dynamic programming has been widely used and studied as an effective paradigm for query optimization [10] and the query optimizers of nowadays commercial systems such as DB2 still depend on this paradigm. In this section, we show how to use dynamic programming to optimize and process multi R-joins queries.

4.1 A Multiple Interval Encoding
Agrawal et al. proposed an interval-based coding for encoding DAG [2]. Unlike the approaches that assign a single code, \([s : e]\), for every node in tree, Agrawal et al. assigned a set of intervals and a postorder number for each node in DAG. Let \( I_u = \{[s_1 : e_1], [s_2 : e_2], \ldots, [s_n : e_n]\} \) be a set of intervals assigned to a node \( u \), there is a path from \( u \) to \( v \), \( u \leadsto v \), if the postorder number of \( v \) is contained in an interval, \([s_j : e_j]\) in \( I_u \). The interval-based coding for the graph in Figure 1 (b) is given in Table 1. For the same example of 17 \( \leadsto 21 \) in the DAG of Fig. 1, it could be identified by the 21’s \( poId \), 12, and one interval associated with 17, \([12 : 13]\), since 12 is contained in \([12 : 13]\).

Based on [2], Wang et al. studied processing R-join over a directed graph [11]. In brief, given a directed graph, \( G \). First, it constructs a DAG \( G' \) by condensing a maximal strongly connected component in \( G \) as a node in \( G' \). Second, it generates codes for \( G' \) based on [2]. All nodes in a strongly connected component in \( G \) share the same code assigned to the corresponding representative node condensed in \( G' \). Given a R-join, \( A \leadsto D \), two lists \( Alist \) and \( Dlist \) are formed respectively. \( Alist \) encodes every node \( v \) as \((v, s:e)\) where \([s : e] \in I_v \). A node of \( A \) has \( n \) entries in the \( Alist \), if it has \( n \) intervals. \( Dlist \) encodes each node \( v \) as \((v, po_v)\) where \( po_v \) is the postorder number. Note: \( Alist \) is sorted on the intervals \([s : e]\) by the ascending order of \( x \) and then the descending order of \( y \), and \( Dlist \) is sorted by the postnumbers in ascending order. Wang et al. proposed to merge-join the nodes in \( Alist \) and \( Dlist \) and scans the two lists once.

4.2 System Overview
The join algorithm in [11] is not ready to be extended to process multi R-joins. Consider \( A \leadsto D \land D \leadsto E \). For processing \( A \leadsto D \), \( Dlist \) needs to be sorted based on the postnumbers, because \( D \) is descendant. For processing \( D \leadsto E \), \( Dlist \) needs to be sorted based on
\(s\) followed by \(e\) for all \((v, s, e)\), because \(D\) is an ancestor. Also, recall, for \(A \rightarrow D\), \(A\) list needs to encode every node \(v\) as \((v, s, e)\) where \([s : e] \in I_v\). The intervals of \(I_v\) list for all \(v\) in \(A\) list must be maintained in multi \(R\)-joins processing. We will give out system overview first and then explain that how we remedy this problem. Overall, our \(R\)-Joins query processing could be divided into three functioning parts, estimation, planning and execution. The database stores Interval Lists, that is, all \(A\) lists and \(D\) lists, Encoding \(Dict\), which supports retrieval of \(I_v\) and \(po_i\) for any node, and some statistics about the data, which is mainly used to estimate the cost of a given \(R\)-join.

The execution part of our query engine contains another 3 types of operations besides the join primitive for \(A \rightarrow D\),

\(\alpha(A)\): Given a list of node vectors in the form of \((v_1, v_2, \ldots, v_l)\) and each \(v_i\) is in the extension associated with \(A\), it attaches each interval \([s, e] \in I_v\) and obtain a number of \((v_1, v_2, \ldots, v_l, s : e)\) from every vector \((v_1, v_2, \ldots, v_l)\) and sorts the resulting list to obtain an \(A\) list from this vector.

\(\delta(D)\): Similarly, it results in a temporary \(D\) list.

\(\sigma(A, D)\): Given a list of node vectors in the form of \((v_1, v_2, \ldots, v_l)\) and each \(v_i/v_j\) is in the extension associated with \(A/D\), it select out those vectors satisfying \(v_i \rightarrow v_j\).

We explain the other parts for our \(R\)-Joins query processing in the following sections.

### 4.3 \(R\)-join Size Estimation

We introduce a simple but effective way to estimate the answer size for a sequence of \(R\)-joins. We need two presumption for our estimation: (1) For any pair-wise \(R\)-join, say \(A \rightarrow D\), every pair of instance \((a, d)\), where \(a \in ext(A)\) and \(d \in ext(D)\), is joinable with the same probability. (2) Consider two \(R\)-joins, say \(A \rightarrow B\) and \(B \rightarrow C\), for any three instance \((a, b, c)\), where \(a \in ext(A)\), \(b \in ext(B)\), and \(c \in ext(C)\), the two events \(E_1 = \{a \text{ is joinable } b\}\) and \(E_2 = \{b \text{ is joinable with } c\}\) are independent.

Suppose the answer size for \(R\)-joins \((R_1 \rightarrow \ldots \rightarrow R_l)\) is \(M\) and the answer size for the pairwise \(R\)-join \(R_h \rightarrow R_{h+1}\) is \(N\), we will show the answer size for \((R_1 \rightarrow \ldots \rightarrow R_l) \land (R_h \rightarrow R_{h+1})\) \((1 \leq h \leq l)\), could be estimated as \(\frac{M \times N}{|R_h|}\), where \(|R_h|\) is the cardinality for the extension of \(R_h\).

Suppose \(r_j\) is an instance from \(ext(R_j)\), and let \(Join(\cdot)\) denote the event that instances are joinable. Then because presumption (2), we have

\[
\Pr(Join(r_1, r_2, \ldots, r_{h+1})) \\
= \Pr(Join(r_1, r_2) \land Join(r_{h}, r_{h+1})) \\
= \Pr(Join(r_1, r_2)) \cdot \Pr(Join(r_{h}, r_{h+1})).
\]

And because of presumption (1), we have

\[
\Pr(Join(r_1, r_2)) \approx \frac{M}{|R_1| \cdot |R_2| \cdot |R_l|} \\
\Pr(Join(r_{h}, r_{h+1})) \approx \frac{N}{|R_h| \cdot |R_{h+1}|}.
\]
So the estimated answer size of \((R_1 \rightarrow \ldots \rightarrow R_i) \leftarrow R_{i+1}\) can be
\[
\text{EST} = |R_1||R_2|\ldots|R_i||R_{i+1}| \Pr(\text{Join}(r_1, r_2, \ldots, r_{i+1})) \\
= |R_1| \ldots |R_{i+1}| \frac{M}{|R_h|} N \\
= \frac{M \times N}{|R_h|}
\]

So we will be able to estimate the answer size for all such \(R\)-joins by conveniently memorizing all pairwise \(R\)-join size and all label’s extension cardinalities in the database catalog.

**Example 3** For our running example, the first join is \(\text{Institute} \rightarrow \text{research}\), thus \(M = 3\). For \(\text{Institute} \rightarrow \text{research} \rightarrow \text{topic}\), since \(N = 3\) and \(|\text{ext(research)}| = 5\), so the estimated result set size is \(\frac{3 \times 3}{5} = 1.8\). The same result could be calculated if \(\text{research} \rightarrow \text{topic}\) is taken as the first join.

### 4.4 \(R\)-join Plan Enumeration

We use dynamic programming style optimization to enumerate a set of equivalent plans that combine the operations mentioned in Sec. 4.2 to evaluate a query graph \(G_q\) against a database graph \(G\). We briefly outline the procedure of searching such plans and its execution to evaluate \(G_q\).

Given a query graph \(G_q\), only left-deep tree plans are searched as a common practice for a reasonable search space. Recall: in \(G_q\), a node represent a label and an edge represents \(\rightarrow\). An \(R\)-join, \(A \rightarrow D\), is represented as an edge from \(A\) to \(D\). Initially, subgraphs \(G_2\) with two nodes connected by an edge are considered. Here, \(V(G_2) = \{v, u\}\) and \(E(G_2) = \{(v, u)\}\) or \(E(G_2) = \{(u, v)\}\) depending on whether it is for \(v \rightarrow u\) or \(u \rightarrow v\).

In the next step, it considers to add one more edge. That is, it considers a subgraph \(G_3\) with three nodes and two edges, such as \(V(G_3) = V(G_2) \cup \{u\}\) and \(E_3\) includes all the edges in \(E(G_2)\) plus one edge with at least one incident node in \(V(G_2)\). The last step repeats until it includes all the nodes and edges in the original query graph \(G_q\) and we could get a sequence of subgraphs \((G_2, G_3, \ldots, G_m)\) and a sequent of edges being added

![Fig. 5. Searching for an Optimal Plan using DP](image1)

<table>
<thead>
<tr>
<th>Start</th>
<th>End</th>
<th>(R)-join</th>
<th>Result Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>(s_1)</td>
<td>(I \rightarrow R)</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>(s_0)</td>
<td>(s_2)</td>
<td>(R \rightarrow T)</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>(s_1)</td>
<td>(s_f_1)</td>
<td>(R \rightarrow T)</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(s_f_1)</td>
<td>(I \rightarrow R)</td>
<td>1</td>
<td>19</td>
</tr>
</tbody>
</table>

![Fig. 6. DP on \(G_q\)](image2)
\( (e_2, e_2, \ldots, e_m) \). Regarding a subgraph in the sequence, say, \( G_i \) and the edge to be added to the subgraph, which should be \( e_i \) or more specifically, \((u_i, v_i)\), there are 3 cases:

- Only \( u_i \) exists in \( V(G_i) \), in this case, an \( \alpha \) operation is needed and followed by a join for \( u_i \rightarrow v_i \) and the cost is calculated as
  \[
  C_I = C_\alpha \cdot |R(G_i)| + C_{\leftrightarrow}(\varepsilon \cdot |R(G_i)| + |Dlist_{v_i}|)
  \]

- Only \( v_i \) exists in \( V(G_i) \), in this case, an \( \delta \) operation maybe needed and followed by a join for \( u_i \rightarrow v_i \), since the \( Dlist \) for \( v_i \) maybe obtained by the output from the preceding join. When \( \delta \) operation is needed, the cost is calculated as
  \[
  C_{II} = C_\delta \cdot |R(G_i)| + C_{\leftrightarrow}(|R(G_i)| + |Alist_{u_i}|)
  \]

  The first term in \( C_{II} \) can be eliminated if no \( \delta \) operation needed.

- Both \( u_i \) and \( v_i \) exist in \( V(G_i) \), in this case, an \( \sigma \) operation is needed. The cost is calculated as
  \[
  C_{III} = C_\sigma \cdot |R(G_i)|
  \]

In these cost formulae, values for \(|Alist_{u_i}|\) and \(|Dlist_{v_i}|\) are obtained from the statistics in database catalog. The intermediate result by evaluating the query graph \( G_i \) is represented as \( R(G_i) \). We estimate the value of \(|R(G_i)|\) according to section 4.3. The explanation for other factors are as follows,

- \( C_\alpha \): factor to approximate the cost of \( \alpha \) operation by the cardinality of the node vectors;
- \( C_\delta \): factor to approximate the cost of \( \delta \) operation by the cardinality of the node vectors;
- \( C_\sigma \): factor to approximate the cost of \( \sigma \) operation by the cardinality of the node vectors;
- \( C_{\leftrightarrow} \): factor to approximate the length of an \( Alist \) by the cardinality of the node vectors.

### 4.5 The Algorithm

In our dynamic programming style optimization, two basic components in the solution space are \( \textit{statuses} \) and \( \textit{moves} \).

#### Algorithm 1 DP Algorithm to Generate Plan

\( l \) is a priority queue of status, sorting statues in the increasing order of \( \text{cost}(S) \).

1: Initialize queue \( l \) as \( \emptyset \);
2: Add \( S_0^i \) into \( l \);
3: while \( l \) is NOT empty do
4:   \( S = l.\text{first} \);
5:   \( \text{Delete} l.\text{first} \) from \( l \);
6:   if \( S \) is a \textit{Final Status} then
7:     Output plan \( \mathcal{P} \) backward from \( l \); \textbf{Terminate} this Algorithm;
8:   for each move from \( S \) to \( S' \) do
9:     if \( S' \notin l \) then
10:       Insert \( S' \) into \( l \);
11:     else
12:       Update \( \text{cost}(S') \) and \( l \);
A status, $S$, specifies a subquery, $G_s$, as an intermediate stage in generating a query plan. To be more specific, a subquery of $G_q$ is a subgraph $G_s$, where $V(G_s) \subseteq V(G_q)$ and $E(G_s) \subseteq E(G_q)$. Note: $G_s$ does not necessarily be a connected graph if without the left-deep tree restriction.

A move from one status (subquery $G_{s_i}$) to another status (subquery $G_{s_j}$) considers an additional R-join in $G_{s_j}$ that does not appear in $G_{s_i}$, toward finding the entire query plan for $G_q$. The next status is determined based on a cost function which results in the minimal cost, in comparison with all possible moves. The process of moving from one status to another results in a left-deep tree which is the R-join order selection result.

We could estimate the cost for each move by those cost formulae in Sec. 4.4. Each status $S$ is associated with a cost function, denoted $\text{cost}(S)$, which is the minimal accumulated estimated cost to move from the initial status $S_0$ to the current status $S$. Such accumulated cost of a sequence of moves from $S_0$ to $S$ is the estimated cost for evaluating the subquery $G_S$ being considered under the current status $S$. The goal of dynamic programming is to find the sequence of moves from the initial status $S_0$ toward the final status $S_f$ with the minimum cost, $\text{cost}(S_f)$, among all the possible sequences of moves.

The rest task is to find a sequence of moves, namely a plan $\mathcal{P}$, from the start status $S_0^+$ to a final status $S_f^+$, and minimize the cost $\text{cost}(\mathcal{P})$. Our algorithm is outlined in Algorithm 1. We simply apply Dijkstra’s algorithm for the shortest path problem into our search space, aiming to find a “shortest” path from $S_0$ to any $S_f^+$, where nodes represent statues, edges represent moves, and the length of an edge is the cost of one move. We omit further explanation about Algorithm 1.

Example 4 For our running example, Figure 5 shows two alternative plans for evaluating the query $I \bowtie R \rightarrow T$, both containing two moves. The status $S_0$ is associated with a NULL graph, while $S_1$ and $S_2$ are respectively associated with two two graphs with two connected nodes, and $S_3$ is associated with the $G_q$ and thus to be a final status. Details steps in the searching for an optimal plan is showed in Figure 6, where each row of the table lists a move in the solution space. The first column is the status where to start the move and the second column is the status where the move reaches. The third column is the R-join that will be processed in that move, while the number of results generated after the R-join is the fourth column.

5 Performance Evaluation

In this section, we use two sets of tests to show the efficiency of our approach. The first set of tests is designed to compare our dynamic programming approach (denoted DP) with algorithm [5] (denoted TSD). The second set of tests further confirms the the ability to scale of our approach. We implemented all the algorithms using C++ on top of an experimental database system as done in [6]. We configure the buffer of the database system to be 2MB. A PC with a 3.4GHz processor, 2GB memory, and 120G harddisk running Windows XP is used to carry out all tests.

We generated 20M, 40M, 60M, 80M and 100M size XMark datasets [9] using 5 different factors, 0.2, 0.4, 0.6, 0.8, and 1.0 respectively, and named each dataset by its
size. In these XML documents, we treat parent-child edges and ID/IDRE edges without difference to obtain graphs and collapse the strong connected components in graphs to get DAGs. The details of the datasets are given in Fig. 8. In Fig. 8, the first column is the dataset name. The second and third columns are the node number and edge number of the resulting DAG respectively. The forth column is the multiple interval labeling size, while the last column shows the average number of intervals per node in the DAG. Throughout all experiments, we use the 4 multi R-join queries listed in Fig. 7, where the label I stands for interest, C for category, L for listitem, D for description and P for parlist.

5.1 TwigStackD vs. Our Approach

We test all queries over the same dataset described in Section 3.2 for the purpose of compare TwigStackD algorithm to our approach. We show two set of figures that show the elapsed time, number of I/Os and memory used to process each query. The first set of figures show the performance on the DAG with 10 percent remaining edges added, which are listed in Fig. 9 (a)-(c), and the second set of figures show the performance on the DAG with 50 percent remaining edges added 50 percent, which are listed in Fig. 9 (d)-(f).

As shown in Fig. 9, our approach significantly outperformed TwigStackD, in terms of elapsed time, number of I/O accesses, and memory consumption. The sharp difference becomes even greater for the denser DAG, due to the fast performance degradation of TwigStackD when the edge number in the DAG increased. For example, consider Q3, TwigStackD used 16.7 times of elapsed time and 8.7 times of I/O accesses than those for our approach when 10 percent remaining edges being added, but when 50 percent remaining edges being added, they two rate becomes 2922.3 and 266.4 respectively. The memory usage of TwigStackD is unstable, and can range from 60MB to 900MB for the 4 queries, because needs to buffer every node that can potentially be in one final solution and thus largely depends on the solution size. And it could also be observed that generally, the larger query needs more memory for the increased needs of buffer pools by TwigStackD.
5.2 Scalability Test of Our Approach

With the size of the dataset increasing from 20M to 100M, we tested the scalability performance for our approach and Fig. 10 shows the results. Both the number of I/Os and memory usage increase evenly as the size of underlying DAGs increases. However, for the processing time of each query when the data size increased, its variation is not so uniformly. A main reason for this observation is the CPU overhead caused by sorting in $\alpha$ and $\delta$ operations, for different distribution of the data may result different number of those operations for the same query. However, there is no abrupt change for the processing and the overall performance is still acceptable and query could be done within tens of seconds.

Fig. 9. Compare on the DAG with 10% and 50% Remaining Edge Included

Fig. 10. Scalability Test
6 Conclusion

In this paper, we study processing multi reachability joins (R-joins) queries on DAGs. For the most up-to-date existing approach, the TwigStackD algorithm processes the reachability relationships in a spanning tree and those induced by remaining edges in a DAG using different procedures. The performance of this method degrades rapidly when the edges in a DAG increase. We extend the existing size 2 R-join query processing scheme to be applicable to multi R-join queries with arbitrary size. The dynamic programming algorithm is employed to combine efficient R-join primitives into an optimized plan that evaluates multi R-join queries. We conducted extensive performance studies and confirmed the efficiency of our approach.

References