Fast Graph Pattern Matching

Jiefeng Cheng¹ Jeffrey Xu Yu¹ Bolin Ding¹ Philip S. Yu² Haixun Wang²

¹ The Chinese University of Hong Kong, Hong Kong, China, {jfcheng,yu,blding}@se.cuhk.edu.hk
² T. J. Watson Research Center, IBM, USA, {psyu,haixun}@us.ibm.com

Abstract

Due to rapid growth of the Internet technology and new scientific/technological advances, the number of applications that model data as graphs increases, because graphs have high expressive power to model complicated structures. The dominance of graphs in real-world applications asks for new graph data management so that users can access graph data effectively and efficiently. In this paper, we study a graph pattern matching problem over a large data graph. The problem is to find all patterns in a large data graph that match a user-given graph pattern. We propose a new two-step R-join (reachability join) algorithm with filter step and fetch step based on a cluster-based join-index with graph codes. We consider the filter step as an R-semijoin, and propose a new optimization approach by interleaving R-joins with R-semijoins. We conducted extensive performance studies, and confirm the efficiency of our proposed new approaches.

1 Introduction

A graph provides great expressive power to describe and understand the complex relationships among data objects. With the rapid growth of World-Wide-Web, new data archiving and analyzing techniques, there exists a huge volume of data available in public, which is graph structured in nature including hypertext data, semi-structured data [1]. RDF also allows users to explicitly describe semantic resource in graphs [7]. In [27], Shasha et al. highlighted algorithms and applications for tree and graph searching including graph/subgraph matching in data graphs. The demand increases to query graphs over a large data graph. In this paper, we study a graph pattern matching problem that is to retrieve all patterns in a large graph, \( G_D \), that match a user-given graph pattern, \( G_p \), based on reachability. As an example, based on business relationships, a graph pattern can be specified as to find Supplier, Retailer, Whole-seller, and Bank such that Supplier directly or indirectly supplies products to Retailer and Whole-seller, and all of them receive services from the same Bank directly or indirectly over a large data graph which can be obtained from the Web. Similar needs also stem from finding web-services connection patterns in WWW, finding relationships in social networks [3], finding research collaboration patterns, and finding research paper citation connection in archived bibliography datasets.

The graph pattern matching problem can be considered as an extension of finding twig-patterns (tree patterns) over \( XML \) tree. However, the existing techniques for processing twig-patterns over \( XML \) tree [8, 14] cannot be effectively applied to handle graph pattern matching over a large directed graph. It is because a graph does not have the nice property such that every two nodes are connected along a unique path. In a large data graph, a node, \( v_i \), can reach another node \( v_j \), while the same \( v_j \) is possibly reachable from \( v_i \).

Contributions of this paper: We propose processing graph pattern matching as a sequence of R-join (reachability join) upon a graph database which stores a data graph in tables. We propose a new two-step R-join algorithm with a filter step and fetch step, based on a new cluster-based join-index with graph codes for reachability checking. Furthermore, we consider the first filter step as an R-semijoin, and propose a new optimization approach to optimize a sequence of R-joins/R-semijoins. We conducted extensive performance studies, and confirm the efficiency of our proposed new approaches.

Organization: We give the problem statement in Section 2. In Section 3, we discuss our R-join/R-semijoin approach. We propose a new two-step R-join algorithm (filter/fetch) based on which an R-semijoin is introduced. We propose a new R-join/R-semijoin order selection approach in Section 4. In Section 5, two existing approaches are discussed. We conducted extensive performance studies using large datasets and report our findings in Section 6. Related work is given in Section 7. Section 8 concludes the paper.

2 Problem Statement

In this section, we give our problem statement following the discussions on data graph and graph pattern.

A data graph is a directed node-labeled graph \( G_D = (V,E,\Sigma,\phi) \). Here, \( V \) is a set of nodes; \( E \) is a set of edges (ordered pairs); \( \Sigma \) is a set of node labels, and \( \phi \) is a mapping function which assigns each node, \( v_i \in V \), a label \( l_j \in \Sigma \). We use \( \text{label}(v_i) \) to denote the label of node \( v_i \). Given a label \( X \in \Sigma \), the extent of \( X \), denoted as \( \text{ext}(X) \), is the set
of all nodes in \( G_D \) whose labels are the same X. A simple data graph, \( G_D \), is shown in Figure 1 (a). There are 5 labels, \( \Sigma = \{A, B, C, D, E\} \). In Figure 1 (a), a node in an extent \( \text{ext}(X) \) is represented as \( x_i \), where \( x \) is a small letter of \( X \) with a unique number \( i \) to distinguish it from others in \( \text{ext}(X) \). For example, \( \text{ext}(C) = \{e_0, c_1, c_2, c_3\} \).

In the following, we use \( V(G) \) and \( E(G) \) to denote the set of nodes and the set of edges in graph \( G \).

A graph pattern is a connected directed node-labeled graph \( G_q = (V_q, E_q) \), where \( V_q \) is a subset of labels (\( \Sigma \)), and \( E_q \) is a set of edges (ordered pairs) between two nodes in \( V_q \). An edge \( (X, Y) \in E(G_q) \) represents a reachability condition, denoted \( X \rightarrow Y \), for \( X, Y \in V_q \). A reachability condition, \( X \rightarrow Y \), requests two nodes \( v_i \) and \( v_j \) in \( G_D \), for label \( (v_i) = X \) and label \( (v_j) = Y \), if \( v_j \) is reachable from \( v_i \), denoted \( v_i \rightarrow v_j \). A match in \( G_D \) matches graph pattern \( G_q \) if it satisfies all the reachability conditions conjunctively specified in \( G_q \). Note: \( X \rightarrow Y \) and \( Y \rightarrow Z \) implies \( X \rightarrow Z \).

A graph pattern is a connected directed node-labeled graph \( G_q = (V_q, E_q) \), where \( V_q \) is a subset of labels (\( \Sigma \)), and \( E_q \) is a set of edges (ordered pairs) between two nodes in \( V_q \). An edge \( (X, Y) \in E(G_q) \) represents a reachability condition, denoted \( X \rightarrow Y \), for \( X, Y \in V_q \). A reachability condition, \( X \rightarrow Y \), requests two nodes \( v_i \) and \( v_j \) in \( G_D \), for label \( (v_i) = X \) and label \( (v_j) = Y \), if \( v_j \) is reachable from \( v_i \), denoted \( v_i \rightarrow v_j \). A match in \( G_D \) matches graph pattern \( G_q \) if it satisfies all the reachability conditions conjunctively specified in \( G_q \). Note: \( X \rightarrow Y \) and \( Y \rightarrow Z \) implies \( X \rightarrow Z \).

A New Join-Based Approach

In this paper, given a graph pattern \( G_q \), we propose graph matching as a sequence of joins, where each reachability condition, \( X \rightarrow Y \in E(G_q) \), is a join, called \( R \)-join (for reachability join).

Such an \( R \)-join is possible based on a graph labeling called 2-hop reachability labeling [17]. A 2-hop reachability labeling over graph \( G_D \) assigns every node \( v \in V \) a label \( L(v) = (L_{in}(v), L_{out}(v)) \), where \( L_{in}(v), L_{out}(v) \subseteq V \), and \( u \sim v \) is true if and only if \( L_{out}(u) \cap L_{in}(v) \neq \emptyset \). A 2-hop reachability labeling for \( G_D \) is derived from a 2-hop cover of \( G_D \). In brief, given \( G_D \), the 2-hop cover minimizes a set of \( S(U_w, w, V_w) \), as a set cover problem. Here, \( w \in V(G_D) \) is called a center, and \( U_w, V_w \subseteq V(G_D) \). \( S(U_w, w, V_w) \) implies that, for every node, \( w \in U_w \) and \( v \in V_w \), \( w \sim v \) and \( v \sim w \), and therefore \( w \sim v \). Consider Figure 1, an example is \( S(L_{in}(w), L_{out}(v)) = S(\{b_0, b_1\}, c_2, \{e_0\}) \).

There are several implementations to find such 2-hop cover for \( G_D \) [23, 24, 15]. The 2-hop cover update problem is addressed in [24]. We proposed a fast algorithm to compute 2-hop cover [15].

Let \( \mathcal{H} = \{S_{w_1}, S_{w_2}, \ldots\} \) be the set of 2-hop cover computed, where \( S_{w_i} = S(U_{w_i}, w_i, V_{w_i}) \) and all \( w_i \) are centers. The 2-hop reachability labeling for a node \( v \) is \( L(v) = (L_{in}(v), L_{out}(v)) \). Here, \( L_{in}(v) \) is a set of centers \( w_i \), where \( v \) appears in \( V_{w_i} \), and \( L_{out}(v) \) is a set of centers \( w_i \) where \( v \) appears in \( U_{w_i} \).

Based on the 2-hop reachability labeling, we store graph \( G_D \) into a database, \( \text{GDB} \), by taking a node-oriented representation. There are \( |\Sigma| \) tables for \( G_D \). A table \( T_X \), for a label \( X \in \Sigma \), has three columns named \( x, in, \) and \( out \). The reachability condition \( X \rightarrow Y \) is represented with a row \( x \rightarrow in = Y \) and \( x \rightarrow out = Y \). Correspondingly, \( X \rightarrow Y \) is represented with a row \( Y \rightarrow in = x \) and \( Y \rightarrow out = x \).

For each node \( x_i \in \text{ext}(X) \), \( \subseteq V(G_D) \), there is a tuple in table \( T_X \). The \( X \) column keeps the node identifier \( x_i \). The \( in \) and \( out \) columns keep its \( \text{in}(x_i) \) and \( \text{out}(x_i) \), respectively. We assume that the \( X \) column is the primary key of the table, because a node in \( G_D \) is uniquely identified with a node identifier. We call \( T_X \) a base table if it is the table for a label \( X \in \Sigma \).

Example 3.1: A graph database \( \text{GDB} \) for \( G_D \) (Figure 1) is shown in Figure 2 (a). There are five tables: \( T_A(A, in, out) \), \( T_B(B, in, out) \), \( T_C(C, in, out) \), \( T_D(D, in, out) \), and \( T_E(E, in, out) \). For a tuple \( x_i \) in table \( T_X \), we make 2-hop reachability labeling compact by removing \( x_i \) from its \( \text{in}_{x_i} \) and \( \text{out}_{x_i} \) columns. Hence, \( \text{in}_{x_i} = \text{in}_{x_i} \cup \{x_i\} \) and \( \text{out}_{x_i} = \text{out}_{x_i} \cup \{x_i\} \). Below, we call \( \text{Join}(x_i) \) and \( \text{Out}(x_i) \) graph codes for \( x_i \), denoted \( \text{in}(x_i) \) and \( \text{out}(x_i) \). The reachability, \( x_i \sim y_i \), returns true, if \( \text{out}(x_i) \cap \text{in}(y_i) \neq \emptyset \).

3.1 R-Join

Given two base tables in \( \text{GDB} \), a reachability condition, \( X \rightarrow Y \), in a graph pattern \( G_q \), can be processed as an \( R \)-join between two tables, \( T_X \) and \( T_Y \).

\[
T_R \leftarrow T_X \Join T_Y
\]
Here, an $R$-join implies that, for every $x_i \in \text{ext}(X)$ and $y_j \in \text{ext}(Y)$, $x_i \sim y_j$ holds, if the reachability condition, $X \rightarrow Y$, is evaluated to be true using the graph codes. A pair, $(x_i, y_j)$, appears in the temporal table $T_R$, if $x_i \sim y_j$ is true ($\text{out}(x_i) \cap \text{in}(y_j) \neq \emptyset$).

Consider $T_B \bowtie_{E} T_E$, $(b_0, e_7)$ appears in the result, because $\text{out}(b_0) = \{b_0, c_1\}$, $\text{in}(e_7) = \{c_1, e_7\}$, and $\text{out}(b_0) \cap \text{in}(e_7) \neq \emptyset$.

In general, an $R$-join over any two tables, $T_R$ and $T_S$, with a reachability condition, $X \rightarrow Y$, can be specified. Note: $X$ ($Y$) is the column in the base table $T_X$ ($T_Y$), that may appear in a temporal table because of a previous $R$-join. Here, $T_R$ and $T_S$ can be either a base or a temporal table.

$$T_{RS} \leftarrow T_{R} \bowtie_{X \rightarrow Y} T_{S}$$

Therefore, a graph pattern, $G_q$, can be specified as a sequence of $R$-joins followed by a projection to project the columns for every label $X \in V(G_q)$.

In this paper, we concentrate ourselves on query processing and optimization over multi $R$-joins, and focus on discussions of finding an optimal query plan that is represented as a left-deep tree [29] in which an $R$-join is either between two base tables or between a temporal table and a base table. As shown in Eq. (3) and Eq. (4) below, $T_X$ and $T_Y$ represent base tables, and $T_R$ represents either a base or a temporal table.

$$T_{R} \bowtie_{X \rightarrow Y} T_{Y}$$

$$T_{Y} \bowtie_{X \rightarrow Y} T_{R}$$

As a special case, a self-$R$-join is a join that can be processed as a selection,

$$T_{R} \bowtie_{X \rightarrow Y} T_{R}$$

where $T_R$ can be a base/temporal table. The following holds for $R$-joins. $T_{R} \bowtie_{X \rightarrow Y} T_{S} \equiv T_{S} \bowtie_{X \rightarrow Y} T_{R}$ (Commutative), $(T_{R} \bowtie_{X \rightarrow Y} T_{S}) \bowtie_{W \rightarrow Z} T_{T} \equiv T_{R} \bowtie_{X \rightarrow Y} (T_{S} \bowtie_{W \rightarrow Z} T_{T})$ (Associative). Given a table $T_R$ and suppose $T_R$ keeps tuples that satisfy two reachability conditions, $A \rightarrow B$ and $B \rightarrow D$. Then the tuples in $T_R$ satisfy $A \rightarrow D$ (Transitive).

### 3.2 A Cluster-Based $R$-Join Index

Like a $\theta$-join, an $R$-join needs to check the reachability condition $X \rightarrow Y$ at run time, which incurs high cost. We propose a join-index approach, which is to index all tuples $x_i$ and $y_j$ that can join between two tables, $T_X$ and $T_Y$. With such a join-index, an $R$-join can be efficiently implemented as to fetch the results.

We build a cluster-based $R$-join index for a data graph $G_D$ based on the 2-hop cover computed, $H = \{S_{w_1}, S_{w_2}, \ldots\}$, using our fast algorithm in [15], where $S_{w_i} = S(U_{w_i}, w_i, V_{w_i})$ and all $w_i$ are centers. It is a B+-tree in which its non-leaf blocks are used for finding a given center $w_i$. In the leaf nodes, for each center $w_i$, its $U_{w_i}$ and $V_{w_i}$, denoted $F$-cluster and $T$-cluster, are maintained. We further divide $w_i$’s $F$-cluster and $T$-cluster into labeled $F$-subclusters/$T$-subclusters where every node, $x_i$, in an $X$-labeled $F$-subcluster can reach every node $y_j$ in a $Y$-labeled $T$-subcluster, via $w_i$. It is important to note that, in our cluster-based $R$-join index, we keep node identifiers (tuple identifiers) instead of pointers to tuples in base tables. With this arrangement, we can answer some $R$-join without accessing base tables. If there is a need to access a base table, we use the primary index built on the base table.

Together with the cluster-based $R$-join index, we design a $W$-table in which, an entry $W(X, Y)$ is a set of centers. A center $w_i$ will be included in $W(X, Y)$, if $w_i$ has a non-empty $X$-labeled $F$-subcluster and a non-empty $Y$-labeled $T$-subcluster. It helps to find the centers, $w_i$, in the cluster-based $R$-join index, that have an $X$-labeled $F$-subcluster and a $Y$-labeled $T$-subcluster.

Example 3.2: The $G_{DB}$ for $G_D$ (Figure 1) is shown in Figure 2. Figure 2 (a) shows the five base tables, Figure 2 (c) shows the clustered-based $R$-join index, and Figure 2 (b) shows the clustered-based $R$-join index, and Figure 2 (b)
**Algorithm 1 HPSJ** \( (T_X, T_Y, X \leftrightarrow Y) \)

1. \( C \leftarrow W(X, Y) \) using the \( W \)-table;
2. \( R \leftarrow \emptyset \);
3. for each \( w_k \in C \) do
   4. \( X_k \leftarrow \text{getF}(w_k, X) \) using the cluster-based \( R \)-join index;
   5. \( Y_k \leftarrow \text{getT}(w_k, Y) \) using the cluster-based \( R \)-join index;
   6. \( R \leftarrow R \cup (X_k \times Y_k) \);
4. end for
5. return \( R \);

**Algorithm 2 HPSJ+** \( (T_R, T_Y, X \leftrightarrow Y) \)

1. \( T_{W} \leftarrow \text{Filter}(T_{W}, X \leftrightarrow Y) \);
2. \( T_{R,R} \leftarrow \text{Fetch}(T_{W}, X \leftrightarrow Y) \);
3. return \( T_{R,R} \);
4. Procedure Filter \( (T_R, X \leftrightarrow Y) \)
   5. \( T_{W} \leftarrow \emptyset \);
   6. for each tuple \( r_i \) in \( T_R \) do
      7. \( X_i \leftarrow \text{getCenters}(x_i, X, Y) \) where \( x_i \) is in \( X \) column in \( r_i \);
   8. insert \( (r_i, X_i) \) into \( T_{W} \) if \( X_i \neq \emptyset \);
   9. end for
10. return \( T_{W} \);
11. Procedure Fetch \( (T_{W}, X \leftrightarrow Y) \)
12. \( T_{R,R} \leftarrow \emptyset \);
13. for each \( (r_i, X_i) \) in \( T_{W} \) do
   14. for each \( w_k \in X_i \) do
      15. \( Y_i \leftarrow \text{getT}(w_k, Y) \) using the cluster-based \( R \)-join index;
      16. \( T_{R,R} \leftarrow T_{R,R} \cup (r_i \times Y_i) \);
   17. end for
18. end for

shows its \( W \)-table. The cluster-based \( R \)-join index (Figure 2 (c)) has six centers, \( a_0, b_0, c_1, c_2, \) and \( c_3 \). The \( W \)-table (Figure 2 (b)) tells where \( R \)-join can find its centers in the cluster-based \( R \)-join index.

Consider \( T_A \cong T_B \). The entry \( W(A, B) \) keeps \{ \( a_0 \} \), which suggests that the answers can only be found in the clusters at the center \( a_0 \). As shown in Figure 2 (c), the center \( a_0 \) has an \( A \)-labeled \( F \)-subcluster \{ \( a_0 \} \), and a \( B \)-labeled \( T \)-subcluster \{ \( b_2, b_3, b_4, b_5, b_6 \} \). The answer is the Cartesian product between these two labeled subclusters.

### 3.3 R-Join Algorithms

We first outline an \( R \)-join algorithm (Algorithm 1) between two tables discussed in [16], and then discuss a new two-step \( R \)-join algorithm (Algorithm 2) between a temporal table and a base table proposed in this paper.

The **HPSJ** algorithm (Algorithm 1) processes an \( R \)-join between two base tables, \( T_X \cong T_Y \). First, it gets all centers, \( w_k \), that have a non-empty \( X \)-labeled \( F \)-subcluster and a non-empty \( Y \)-labeled \( T \)-subcluster, using the \( W \)-table, and maintains it in \( C \) (line 1). Second, for each center \( w_k \in C \), it conducts three things. (1) It obtains its \( X \)-labeled \( F \)-subcluster, using \( \text{getF}(w_k, X) \), and stores them in \( X_k \) (line 4). (2) It obtains its \( Y \)-labeled \( T \)-subcluster, using \( \text{getT}(w_k, Y) \), and stores them in \( Y_k \) (line 5). Both (1) and (2) are done using the cluster-based \( R \)-join index. (3) It conducts Cartesian product between \( X_k \) and \( Y_k \), and saves them into the answer set \( R \) (line 6). The output of an \( R \)-join between two base tables is a set of pairs \((x_i, y_j)\) for \( x_i \sim y_j \). It is important to note that there is no need to access base tables because all the nodes are maintained in the cluster-based \( R \)-join index to answer the \( R \)-join.

In order to process multi-\( R \)-joins, we need a way to process an \( R \)-join between a temporal table and a base table. In general, a temporal table \( T_R \) has columns which are all the labels that are involved in the previous \( R \)-joins. Its tuples satisfy all the previous \( R \)-joins. We propose a new two-step \( R \)-join algorithm in Algorithm 2, called **HPSJ+**. It processes \( T_R \cong T_Y \), where \( T_R \) is a temporal table that has an \( X \) column, and \( T_Y \) is a base table that has a \( Y \) column. Below, we discuss the **HPSJ+** algorithm in detail. A join algorithm can be implemented in a similar manner like Algorithm 2 to process \( T_R \cong T_X \), where \( T_R \) is a temporal table that has a \( Y \) column, and \( T_X \) is a base table that has an \( X \) column.

The **HPSJ+** algorithm takes three inputs, a temporal table \( T_R \), a base table \( T_Y \), and an \( R \)-join condition \( X \leftrightarrow Y \). In **HPSJ+**, first, it calls a procedure \( \text{Filter}(T_R, X \leftrightarrow Y) \) to filter \( T_R \) tuples that cannot be possibly joined with \( T_Y \) using \( W \)-table, and stores them into \( T_W \) (line 1). Second, it calls a procedure \( \text{Fetch}(T_{W}, X \leftrightarrow Y) \) to fetch the \( R \)-join results using the cluster-based \( R \)-join index. We do not need to access the base table \( T_Y \), because the needed nodes are stored in the cluster-based \( R \)-join index. The details of the two procedures are given below.

In \( \text{Filter}(T_R, X \leftrightarrow Y) \), first, it initializes \( T_W \) to be empty (line 5). Second, in a for-loop, it processes every tuple \( r_i \) in \( T_R \) iteratively (line 6-9). In every iteration, it obtains a set of centers, \( X_i \), for \( x_i \) in the \( X \) column in \( r_i \), where every center \( w_k \) in \( X_i \) must have some \( y_j \in T_Y \) in its \( T \)-cluster (line 7). It is done using getCenters \((x_i, Y)\) below:

\[
\text{getCenters}(x_i, X, Y) = \text{out}(x_i) \cap W(X, Y)
\]  

As shown in Eq. (6), \( \text{out}(x_i) \) is a set of centers \( w_k \) that \( x_i \) can reach. It needs to access the base table \( T_X \) using the primary index. We use a working cache to cache those pairs of \((x_i, \text{out}(x_i))\), in our implementation to reduce the access cost for later reuse. \( W(X, Y) \) is the set of all centers, \( w_k \), such that some \( X \)-labeled nodes can reach \( w_k \) and some \( Y \)-labeled nodes can be reached by \( w_k \). The intersection of the two sets is the set of all centers such that \( x_i \) must be able to reach some \( y_j \in \text{ext}(Y) \). If \( X_i \neq \emptyset \), it implies that \( x_i \) must be able to reach some \( y_j \) (line 6), and therefore the pair of \((r_i, X_i)\) is inserted into \( T_W \) (line 8). Otherwise, it can be pruned.

In \( \text{Fetch}(T_{W}, X \leftrightarrow Y) \), it initializes \( T_{R,R} \) as empty (line 12). For each pair of \((r_i, X_i) \) in \( T_W \), it obtains its \( Y \)-labeled \( T \)-subcluster, using \( \text{getT}(w_k, Y) \), stores them in \( Y_i \) (line 15), conducts Cartesian product between \( r_i \) and \( Y_i \), and puts them into \( T_{R,R} \) (line 16).

As an example, consider \((T_B \cong C \cong T_C) \cong D \cong D \text{TD}\)
to access $G_{DB}$ (Figure 2). First, Algorithm 1, processes $T_B \Join_{B \rightarrow C} T_C$ and results in a temporal table, $T_{BC} = \{(b_0, c_1), (b_2, c_1), (b_3, c_2), (b_4, c_2), (b_5, c_3), (b_6, c_3)\}$. Note: only the clusters maintained in the three centers $W(B, C) = \{c_1, c_2, c_3\}$ need to be used (Refer to Figure 2 (b)). Next, Algorithm 2 processes $T_{BC} \Join_{C \rightarrow D} T_D$. In the filter, the two tuples $(b_3, c_2)$ and $(b_4, c_2)$, in $T_{BC}$ are pruned because $\text{out}(c_2) = \{c_2\}$ and $W(C, D) = \{c_0, c_1, c_3\}$, and the intersection is empty (Eq. (6)). Fetch returns the final results, which are $\{(b_0, c_1, d_2), (b_0, c_1, d_3), (b_2, c_1, d_2), (b_2, c_1, d_3), (b_5, c_3, d_1), (b_5, c_3, d_2), (b_6, c_3, d_3)\}$.

3.4 R-Semijoins

Reconsider HPSJ$^+$ ($T_R, T_Y, X \rightarrow Y$) for an R-join between a temporal table $T_R$ and a base table $T_Y$. It can be simply rewritten as $\text{Fetch}(\text{Filter}(T_R, X \rightarrow Y), X \rightarrow Y)$ as given in Algorithm 2. Recall: the Filter prunes those $T_R$ tuples that cannot join any $T_Y$ using the $W$-table. The cost of pruning $T_R$ tuples is small for the following reasons. First, $W$-table can be stored on disk with a B*-tree, and accessed by a pair of labels, $(X, Y)$, as a key. Second, the frequently used labels are small in size and the centers maintained in $W(X, Y)$ can be maintained in memory. Third, the number of centers in a $W(X, Y)$ on average is small. Fourth, the cost of getCenters (Eq. (6)) is small with caching and sharing (Remark 3.1). We consider Filter () as an R-semijoin Eq. (7).

$$T_R \Join_{X \rightarrow Y} T_Y = \pi_{T_R}(T_R \Join_{X \rightarrow Y} T_Y) \quad (7)$$

Here, label $X$ appears in the temporal table $T_R$ and label $Y$ appears in the base table $T_Y$.

$$T_R \Join_{X \rightarrow Y} T_X = \pi_{T_R}(T_R \Join_{X \rightarrow Y} T_X) \quad (8)$$

Eq. (8) shows a similar case where label $Y$ appears in the temporal table $T_R$ and label $X$ appears in the base table $T_X$.

The R-semijoin discussed in this work is different from the semijoin discussed in distributed database systems which is used to reduce the dominant data transmission cost over the network at the expense of the disk I/O access cost. In our problem, there is no such network cost involved. A unique feature of our R-semijoin is that it is the first of the two steps in an R-join algorithm. In other words, it must process R-semijoin to complete R-join. Below, we use $\Join$ denote Filter() as an R-semijoin and $\Join_{X \rightarrow Y}$ denote Fetch(). Then, we have

$$T_R \Join_{X \rightarrow Y} T_S \equiv (T_R \Join_{X \rightarrow Y} T_S) \Join_{X \rightarrow Y} T_S \quad (9)$$

It is worth noting that the cost for both sides of Eq. (9) are almost the same.

Consider $((T_B \Join_{B \rightarrow C} T_C \Join_{C \rightarrow D} T_D) \Join_{C \rightarrow E} T_E)$. Suppose we process $T_B \Join_{B \rightarrow C} T_C$ first, and maintain the result in $T_{BC}$. It becomes $T_{BC} \Join_{C \rightarrow D} T_D \Join_{C \rightarrow E} T_E$. Then,

$$((T_B \Join_{B \rightarrow C} T_D) \Join_{C \rightarrow D} T_D) \Join_{C \rightarrow E} T_E = (((T_B \Join_{B \rightarrow C} T_D) \Join_{C \rightarrow D} T_D) \Join_{C \rightarrow E} T_E $$

The conditions used in the two $R$-semijoins are $C \rightarrow D$ and $C \rightarrow E$. Both access $C$ in table $T_{BC}$. If we process the two $R$-semijoins one followed by another, we need to scan the table $T_{BC}$, get another temporal table $T_{BC}'$, and then process the second $R$-semijoin against $T_{BC}'$. Instead, we can process the two $R$-semijoins together, which only requests to scan $T_{BC}$ once. The Filter cost can also be shared. It can be done by simply modifying Filter. Due to space limit, we omit the details.

Remark 3.1: (R-Semijoins Processing) In general, a sequence of R-semijoins, $((R \Join_{X_1 \rightarrow Y_1} \cdots \Join_{X_k \rightarrow Y_k})$ can be processed together by one-scan of the temporal table $T_R$ under the following conditions. First, it is a sequence of R-semijoins, and there is no any R-join in the sequence. Second, let $C_i$ be a reachability condition, $X_i \rightarrow Y_i$. Either all $X_i$ or all $Y_i$ are the same for a label appearing in $T_R$. □

4 Order Selection

In this section, we focus ourselves on R-join/R-semijoin order selection. We maintain the join sizes and the processing costs for all R-joins between two base tables in a graph database. In order to find an optimized left-deep tree query plan, we estimate the cost for a self-R-join (Eq. (5)), which can be done as a selection, and a join between a temporal table and a base table. We adopt the similar techniques to estimate joins/semijoins used in relational database systems. Note: our approaches is not independent on a cost model. The cost parameters are listed in Table 1.

$$|T_{RS}| = |T_X| \cdot |T_{XY}| \cdot |T_{Y}| \quad (10)$$

$$|T_{RS}| = |T_X| \cdot |T_{XY}| \cdot \frac{|T_Y|}{|T_{XY}|} \quad (11)$$

$$|T_{RS}| = |T_X| \cdot \frac{|T_{XY}| \cdot |T_Y|}{|T_{XY}|} \quad (12)$$

Eq. (10) estimates the size of a self-R-join (Eq. (5)), with condition $X \rightarrow Y$, using the join selectivity for the R-join $T_X \Join_{X \rightarrow Y} T_Y$ between two base tables $T_X$ and $T_Y$ (the second term on the right side). Eq. (11) and Eq. (12) estimate the join size for R-joins (Eq. (3) and Eq. (4)), respectively.
is the estimated cost for evaluating the subquery $G_S$ being considered under the current status $S$. Its search space is bounded by $O(2^m)$, where $m$ is the number of edges in $G_Q$.

### 4.2 Interleave R-Joins with R-Semijoins

Recall: $\bowtie$ is equivalent to $\bowtie (\text{Filter}())$ followed by $\bowtie (\text{Fetch}())$. In this section, we propose a new dynamic programming solution by interleaving R-joins with R-semijoins, or in precise, by interleaving $\bowtie \times \bowtie \bowtie$.

Here, we define a status, $S$, as a four element tuple, $(\mathcal{E}, \mathcal{L}, B^{in}, B^{out})$. A minimum-cost plan $P$ is associated with a status which is a sequence of $\bowtie \times \bowtie \bowtie$ being determined. We explain the four elements below. First, $\mathcal{E}$ is the set of edges ($R$-joins) in $E(G_q)$, that are already included in $P$ associated with $S$. Note: an edge $X \bowtie \bowtie Y$ is said to be included in $\mathcal{E}$, if its corresponding $\bowtie \bowtie \bowtie$ and $\bowtie$ are both included in $P$. Second, $\mathcal{L}$ is the set of labels that appear in the left-hand side of an $R$-semijoin or any side of an $R$-join. Third, $B^{in}$ ($B^{out}$) is a set of labels, where each label $X \in B^{in}$ ($\in B^{out}$) indicates that the graph code in (out) in the base $T_X$ is cached and can be used to process any remaining $\bowtie \bowtie \bowtie$, that has not been considered in the plan $P$ yet. It is important to note that $\mathcal{E}$ is only related to $\bowtie \bowtie \bowtie$ (both $\bowtie \bowtie \bowtie$ and $\bowtie$), and the other two elements, $B^{in}$ and $B^{out}$, are only related to $\bowtie$. There are three possible moves: a move by an additional $\bowtie$ (Filter), a move by an additional $\bowtie \bowtie \bowtie$ (Fetch), and a move by an additional $R$-join (Join). We call them, Filter-move, Fetch-move, and R-join-move, respectively. Note: the R-join-move is designed to use HPSJ (Algorithm 1) to $R$-join the initial two base tables, and the other two moves are design to HPSJ+ (Algorithm 2).

**Filter-move:** It corresponds to the addition of a new label, $X$, into $B^{in}$ ($B^{out}$) due to the inclusion of $X \bowtie \bowtie Y$ ($Y \bowtie \bowtie X$), where $X \in \mathcal{L}$, if $\mathcal{L} \neq \emptyset$, and $X \bowtie \bowtie Y$ ($Y \bowtie \bowtie X$) has not been included yet. When moving to $(\mathcal{E}, B^{in} \cup \{X\}, B^{out})$ (or to $(\mathcal{E}, B^{in}, B^{out} \cup \{X\})$), it does not only append $T_R \bowtie \bowtie T_S$ (or $T_R \bowtie \bowtie T_S$), but also all other $\bowtie \bowtie Y$ on $X$ to maximize the cost sharing (Remark 3.1). All possible $R$-semijoins can be considered.

**Fetch-move:** Consider the status $S = (\mathcal{E}, \mathcal{L}, B^{in}, B^{out})$, all unfinished Fetch are in $E(G_q) - \mathcal{E}$. Let $S'$ be a unfinished Fetch, a move from $S$ to $S' = (\mathcal{E} \cup \{(X \bowtie \bowtie Y)\}), B^{in}, B^{out})$ appends $P \bowtie \bowtie \bowtie$ if its $\bowtie \bowtie \bowtie$ has been included. Note: $\bowtie \bowtie \bowtie$ is included if either $X$ is in $B^{out}$, or $Y$ is in $B^{in}$. As a special case, if both $X \in B^{out}$ and $Y \in B^{in}$, $\bowtie \bowtie \bowtie$ is a self R-join, which can be processed in this status together.

**R-join-move:** Consider the status $S = (\mathcal{E}, \mathcal{L}, B^{in}, B^{out})$, $i$
all unfinished Fetch are in $E(G_q) - \mathcal{E}$. Let $\mathcal{S}_0$, be a unfinished $R$-join, a move from $\mathcal{S}$ to $\mathcal{S}' = (\mathcal{E} \cup \{X \rightarrow Y\})$, $B^{in}$, $B^{out}$ appends $\mathcal{S}_0$ into $\mathcal{P}$. Note: this $R$-join-move is only allowed to move from the initial status $\mathcal{S}_0$ to another status.

Consider the query graph, $G_q$, in Figure 1. Figure 3 illustrates several moves for finding the minimum-cost $R$-join/R-semijoin plan from $\mathcal{S}_0$. A status is shown in a block in Figure 3 with the following attributes: (i) subgraph of $G_q$ being considered, (ii) a plan in the form of left-deep tree for (i), and (iii) $B^{in}$ and $B^{out}$. Those subgraphs in a dot-circled in (i) shows $\mathcal{L}$. The edges appear in a dot-circled is $\mathcal{E}$. Initially, the start status $\mathcal{S}_0 = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$. From $\mathcal{S}_0$, there are 4 possible $R$-join-moves, because there are $n = 4$ edges in Figure 1, plus possible Filter-moves. In Figure 3, it shows two moves from $\mathcal{S}_1$: $\mathcal{S}_1$ (Filter-move) and $\mathcal{S}_2$ (R-join-move). In $\mathcal{S}_1$, $\mathcal{E} = \emptyset$, and $\mathcal{L} = \{A, C\}$, its plan $\mathcal{P}$ is shown in the part (ii), $T_C \prec T_E$, and its $B^{in}$ and $B^{out}$ are shown in the part (iii). In $\mathcal{S}_2$, $\mathcal{E} = \{A \rightarrow C\}$, and $\mathcal{L} = \{A, C\}$, its plan $\mathcal{P}$, $T_A \prec T_C$, is shown in the part (ii), and its $B^{in}$ and $B^{out}$ are shown in the part (iii). From $\mathcal{S}_1$, there are two possible Filter-moves to either $\mathcal{S}_2$ or $\mathcal{S}_3$. Consider the Filter-move from $\mathcal{S}_1$ to $\mathcal{S}_2$. Because $C \in L$ in $\mathcal{S}_2$, it adds $C$ into $B^{in}$ (getting $C$’s graph code in) in $\mathcal{S}_2$. Let the resulting temporal table of $\mathcal{S}_1$ be $T_R$. In $\mathcal{S}_3$, it adds two new $\prec$ into the plan, $(T_R \prec T_A) \prec T_B \prec T_C$ to be processed together to share the processing cost (make use of $C$’s graph code in).

Time/Space Complexity: Consider the number of statuses, $(\mathcal{L}, \mathcal{E}, B^{in}, B^{out})$. Because $\mathcal{L}$ contains all labels appeared in the previous statuses, provided the initial $n$ $R$-join-moves, where $n = |V(G_q)|$, $\mathcal{L}$ fully determines $\mathcal{E}$. Furthermore, consider the number of combinations for $(\mathcal{L}, B^{in}, B^{out})$, which determine the number of statuses. Note that $B^{in} \cup B^{out} \subseteq \mathcal{L}$. Thus regarding a node $v_q \in V(G_q)$, there are 5 possible cases: 1) $v_q \not\in \mathcal{L}$; 2) $v_q \in \mathcal{L}$, $v_q \not\in B^{in}$, $v_q \not\in B^{out}$; 3) $v_q \in \mathcal{L}$, $v_q \in B^{in}$, $v_q \not\in B^{out}$; 4) $v_q \in \mathcal{L}$, $v_q \not\in B^{in}$, $v_q \in B^{out}$; 5) $v_q \in \mathcal{L}$, $v_q \in B^{in}$, $v_q \in B^{out}$. There are in total 5$^n$ combinations. Therefore, the total number of statuses is $n \cdot 5^n$. The space complexity is $O(n \cdot 5^n)$. There are $m$ possible moves from each status, hence the total time complexity is $O(mn \cdot 5^n)$. The time complexity becomes $O(mn \cdot 3^n)$, if $B^{in}$ and $B^{out}$ is replaced by a single set as $B^{in} \cup B^{out}$, where our previous discussions of moves fit as well with the implication that the $X_{in}$ and $X_{out}$ columns of a base table $T_X$ are accessed with the other each time.

As a closely related issue of this problem, Wu et al. in [29] studied a tree-structured query graph for accessing XML data which is tree structured. The time complexity of their algorithm is $O(n^2 \cdot 2^n)$. In this paper, we study graph pattern matching over a large data graph. The time complexity of our solution is reasonable comparing the time complexity of $O(n^2 \cdot 2^n)$ for accessing a large XML tree.

## 5 Two Existing Approaches

In this section, we discuss two existing approaches for graph pattern matching. One is a holistic based approach for a graph pattern against a subclass of directed graphs, directed acyclic graphs (DAG) [11]. The other is sort-merge based multi-join approach to process a graph pattern against a directed graph [28].

### 5.1 A Holistic Based Approach

Chen et al. in [11] studied graph pattern matching over a directed acyclic graph (DAG) instead of a directed graph that we are studying in this paper. Both graph patterns and data graphs are DAGs in [11]. As an approach along the line of Twig-Join [8], Chen et al. used the interval-based encoding scheme, which 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, $v_i$ and $v_j$ in an XML tree, $v_i$ is an ancestor of $v_j$, $v_i \prec v_j$, if $v_i.s < v_j.s$ and $v_i.e > v_j.e$ or simply $v_j$’s interval contains $v_i$’s.

The test of a reachability condition between two data nodes used in [11] is broken into two phases. In the first phase, like the existing interval-based techniques for processing graph pattern matching over an XML tree, they first check if the reachability condition can be identified over a spanning tree generated by depth-first traversal of DAG. In the second phase, in order to find the reachability conditions that can not be referred in the spanning tree, they keep all non-tree edges (named remaining edges) in [11] and all
nodes being incident with any such non-tree edges in a data structure called SSPI (Surrogate and Surplus Predecessor Index). Thus, all predecessor/successor relationships that can not be identified by the intervals alone can be found with the help of SSPI.

The algorithm proposed in [11] is a stack-based algorithm, called TwigStackD. For the first phase, it uses Twig-Join algorithm in [8] to find all DAG graph patterns found in the spanning tree. For the second phase, for each node popped out from stacks used in Twig-Join algorithm, TwigStackD buffers all nodes which at least match a reachability condition in a bottom-up fashion, and maintains all the corresponding links among those nodes. When a top-most node that matches a reachability condition, TwigStackD enumerates the buffer pool and outputs all fully matched patterns. TwigStackD performs well for very sparse DAGs. But, its performance degrades noticeably when the DAG becomes dense, due to the high overhead of accessing edge transitive closures.

### 5.2 Sort-Merge Based Multi Join

Wang et al. studied processing $T_X \Join^Y_T Y$ over a directed graph [28] and proposed a join algorithm, called IGMJ. First, it constructs a DAG $G'$ by condensing a maximal strongly connected component in $G_D$ as a node in $G'$. Second, it generates a multi-interval code for a node in $G'$ based on the approach given in [2]. As its name implies, the multi-interval-based code for encoding DAG [2] is to assign a set of intervals and a postorder number to each node in DAG $G'$. Let $I_v = \{[s_1, e_1], [s_2, e_2], \cdots, [s_n, e_n]\}$ be a set of intervals assigned to a node $v$, there is a path from $v_i$ to $v_j$, $v_i \prec v_j$, if the postorder number of $v_j$ is contained in an interval $[s_k, e_k]$ in $I_{v_i}$. Note: nodes in a strongly connected component in $G$ share the same code assigned to the corresponding representative node condensed in DAG $G'$.

In the IGMJ algorithm, given $T_X \Join^Y_T Y$, two lists $Xlist$ and $Ylist$ are formed respectively. Here, in $Xlist$, every node $x_i$ has $n$ entries, if it has $n$ intervals in $I_{x_i}$. In $Ylist$, every node $y_j$ is encoded by the postorder number $p_{0y_j}$. Note: $Xlist$ is sorted on the intervals $[s, e]$ by the ascending order of $s$ and then the descending order of $e$, and $Ylist$ is sorted by the postorder number in ascending order. Then, IGMJ evaluates $T_X \Join^Y_T Y$ against DAG $G'$ by a single scan on the $Xlist$ and $Ylist$. If $x_i \prec y_j$ is satisfied, then every node that is contracted to $v_i$ can reach every node that is contracted to $v_j$ in the data graph $G_D$.

It needs extra cost to use the IGMJ algorithm to process multi $R$-joins, because it requests that both $T_X (\text{ext}(X))$ and $T_Y (\text{ext}(Y))$ must be sorted. Otherwise, it needs to scan two input tables multiple times to process an $R$-join. Consider an example. For processing $T_A \Join^D_T B$, $Dlist$ needs to be sorted based on the postnumbers, because $D$-labeled nodes are the nodes to be reached. Let the temporal table $T_R$ keep the result of $(T_A \Join^D_T B)$. Then, for processing $(T_R \Join^E_T E)$, it needs to sort all $D$-labeled nodes in $T_R$, based on their intervals, $[s, e]$, because $D$-labeled nodes now become the nodes to reach others. The main extra cost is the sorting cost.

### 6 Performance Evaluation

We conducted extensive experimental studies to study the performance of our two $R$-join/$R$-semijoin approaches, namely DP and DPS. Both use the HPSJ and HPSJ+ algorithms to process $R$-joins. Here, DP performs $R$-join order selection only (Section 4.1). DPS performs the optimal order selection by interleaving $R$-joins with $R$-semijoins (Section 4.2).

We compare DP and DPS with the holistic-based approach discussed in Section 5.1, denoted as TSD, and the multi $R$-joins approach discussed in Section 5.2 using a multi-interval code, denoted as INT-DP. The TSD is based on the TwigStackD algorithm [11], and can be only used to find graph matching over a special class of directed graphs, namely, directed acyclic graph (DAG). The INT-DP is based on the IGMJ algorithm [28] to process $R$-joins. We use dynamic programming for $R$-join order selection with INT-DP, as discussed in Section 4.1. We have implemented all the algorithms using C++ on top of the minibase database system developed at Univ. of Wisconsin-Madison.

We generated five large graphs based on XMark benchmark [25]. First, we generate five XML datasets using five factors, 0.2, 0.4, 0.6, 0.8, and 1.0, and name them as 20M, 40M, 60M, 80M, and 100M, respectively. Here, $m$ means the dataset is $m$ megabyte in size. Second, for each dataset, we generate a large graph by treating both document-internal links (parent-child) and cross-document links (ID/IDREF) as edges in the same manner. The details of the five databases are given in Table 2. In Table 2, the first column is the dataset name, the second and third columns are the numbers of nodes and edges, in the corresponding graphs, respectively. The forth column is the 2-hop cover size, while the last column shows the average size of graph codes using 2-hop cover.

We tested a large number of graph patterns as illustrated in Figure 4. We conducted our testing on a PC with a 3.4GHz Pentium processor, and 120GB hard disk running Windows XP. Note: the buffer size we used in our testing is 1MB for I/O access where the PC has 2GB memory. In the following, the reported elapse time includes both query optimization time and query processing time.
We first compare the two basic R-join order selection approaches, TSD and DP, with the holistic-based approach INT-DP. We used nine path-patterns and nine tree-patterns. A path-pattern has a linear structure (Figure 4(a), 4(c), and 4(h)). For the nine path-patterns, the 3-node path-patterns are P1, P2, and P3; the 4-node path-patterns are P4, P5, P6; and the 5-node path-patterns are P7, P8, P9. For treepatterns, Figure 4(d) shows the shape of T1 to T3. Figure 4(j) shows the shape of T4 to T6. Figure 4(k) shows the shape of T7 to T8. Figure 4(l) shows the shape of T9.

We tested these graph patterns using a small XMark dataset with a factor 0.01 (16K nodes), because TSD has difficult to answer graph patterns over a large graph [11]. We compared TSD and INT-DP, respectively. It is because that INT-DP outperforms TSD, with the holistic-based approach for all patterns because DP needs less I/O cost. INT-DP needs to sort for R-joins, and therefore needs extra I/O cost.

As shown in Figure 5, both R-join based approaches, INT-DP and DP, significantly outperform TSD, in terms of elapsed time. For example, TSD spends 1,668 and 9,709 times of elapsed time as the amount that INT-DP and DP used to process P2, respectively. It is because that TSD needs to buffer every node that can possibly be in one final solution. DP outperforms INT-DP for all patterns because DP needs less I/O cost. INT-DP needs to sort for R-joins, and therefore needs extra I/O cost.

In the following, we focus on our R-join approaches, DP and DPS, over directed graphs.

### 6.1 R-Join vs Holistic over DAG

6.2 R-Join/Semijoin over Directed Graphs

We tested DP and DPS using query structures listed through Figure 4(a) to Figure 4(h) by enumerating all possible patterns with different labels. For most queries, DP spends over five times of I/O cost than what DPS spends.
The optimization of a single select-project-join query in a centralized relational DBMS is outlined in [19]. Optimization for join processing are surveyed in [22], [29] studied optimizing multiple structural join for XML tree-structured data.

Semijoins has also been studied in distributed database systems [6], which reduces the dominate data transmission cost over the network at the expense of the disk I/O access cost. Semijoin full reduction is discussed in [5]. A two-step approach to optimize queries using join and semijoin is discussed [12], by adding semijoins to a join sequence. An approach that considers both semijoins and joins in query optimization is reported in [13], however, the overall complexity can be as high as $O(3^{|E|}|V|^{-1})$ in [13]. In this paper, we propose dynamic programming strategies to deal with both R-joins/R-semijoins together with the overhead manageable.

Surveys on recursive query processing strategies can be found in [4]. In this paper, we show how to use graph coding and a join-index to process graph matching that avoids recursive query processing.

8 Conclusion

We proposed new R-join/R-semijoin processing and optimization techniques for the graph pattern matching problem. Given a graph pattern, $G_q$, where an edge represents a reachability condition that can be processed by a R-join, we proposed a new filter/fetch R-join algorithm, based on a new cluster-based join-index. By taking the first step as an R-semijoin, we optimize such a query by optimizing the R-joins/R-semijoins sequence. A unique feature of our R-semijoin/R-join approach is that R-semijoin is the first step of R-join so that there is a minimal overhead to process R-semijoins. We proposed a new optimization approach by interleaving R-joins with R-semijoins. We conducted extensive performance studies using large data graphs, and confirmed the effectiveness and efficiency of our approach.

References