Hash-based Subgraph Query Processing Method for Graph-structured XML Documents[*]

Hongzhi Wang
Harbin Institute of Tech.
wangzh@hit.edu.cn

Jianzhong Li
Harbin Institute of Tech.
lijzh@hit.edu.cn

Jizhou Luo
Harbin Institute of Tech.
luojizhou@hit.edu.cn

Hong Gao
Harbin Institute of Tech.
honggao@hit.edu.cn

ABSTRACT

When XML documents are modeled as graphs, many research issues arise. In particular, there are many new challenges in query processing on graph-structured XML documents because traditional query processing techniques for tree-structured XML documents cannot be directly applied. This paper studies the problem of structural queries on graph-structured XML documents. A hash-based structural join algorithm, HGJoin, is first proposed to handle reachability queries on graph-structured XML documents. Then, it is extended to the algorithms to process structural queries in form of bipartite graphs. Finally, based on these algorithms, a strategy to process subgraph queries in form of general DAGs is proposed. Analysis and experiments show that all the algorithms have high performance. It is notable that all the algorithms above can be slightly modified to process structural queries in form of general graphs.

1. INTRODUCTION

XML has become the *de facto* standard for information representation and exchange over the Internet. In many applications, an XML document needs to be modeled as a graph more naturally than a tree. For example, the XML document of the relationship of publications and authors adapts to graph structure since one paper may have more than one author and one author may have more than one paper. A fragment of such information is shown in Fig 1. Obviously, the graph-structured XML document can be represented in tree structure by duplicating the element with more than one incoming paths. But it will result in redundancy. If the information in Fig 1 is represented with a tree-structured XML document, the element "author" will be duplicated.

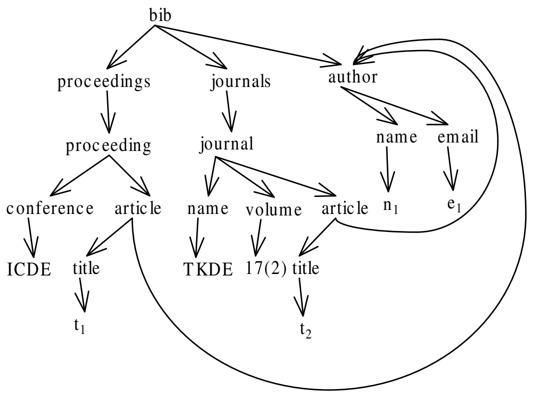


Figure 1: An Example of Graph-structured XML

Among the queries on graph-structured XML documents, the subgraph queries are widely used. A subgraph query on graph-structured XML documents (subgraph query for short) is to retrieve the subgraphs matching the graph in the query. For instance, a subgraph query on the graph-structured XML document in Fig 1 is to retrieve the names of authors with publications both in proceedings and journals. Another subgraph query on the XML document in Fig 1 is to retrieve the name of the journal with an author who published papers in the conference ICDE. Such queries are difficult to represent with traditional tree-structured queries.

It is a big challenge to process subgraph queries efficiently. All the four kinds of traditional methods of processing structural queries on tree-structured XML documents, structural join based methods[2], holistic Twigjoin based methods[3], structural index based methods[14, 12] and subsequence matching based methods[23, 19], cannot be used to process subgraph queries.

The structural join based methods and the holistic Twigjoin based methods both depend on the labelling scheme specially for tree-structured XML documents. The encoding scheme of the graph-structured XML documents is totally different from that of the tree-structured XML documents. As a result, they cannot be used to process subgraph queries.

The structural index based methods of processing structural queries on tree-structured XML documents require that the size of the index must be very small. However, the indices of the graph-structured XML documents are very large in general. Thus, the structural index based methods

[*]Support by the Key Program of the National Natural Science Foundation of China under Grant No.60533110; the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303000; the National Natural Science Foundation of China under Grant No.60773068 and No.60773063

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

cannot be used to process the subgraph queries efficiently.

The subsequence matching based methods require that the tree-structured XML documents and the queries on the documents must be converted into sequences before query processing. It is difficult to covert a graph-structured XML document into a sequence so that it is not easy to process subgraph queries using the subsequence matching based methods.

A few methods has been proposed to process some kinds of subgraph queries on XML data in form of some special kinds of graphs. A method, called StackD, is presented in to [4] to process twig queries on DAG-structured data. It is a modification of a holistic TwigJoin based method. However, StackD focuses on tree-structured twig queries and is not suitable for queries in form of complex graphs. Additionally, when there are many edges in the graph, StackD should maintain a very large data structure. In this case, it needs very huge main memory space, which is not practical and it becomes inefficient. Another modification of the holistic TwigJoin-based method is presented in [27] to process twig queries on graph-structured data. However, it only works on a kind of special graphs, i.e. st-planar graphs [11], but not suitable for other graphs. In summary, current methods cannot process general subgraph queries effectively or efficiently.

To process general subgraph queries effectively and efficiently, a new method based on labeling scheme [17] is proposed in this paper. The reasons of choosing the labeling scheme [17] are as following.

- It contains only intervals and identifications (ids). All intervals and ids are numerical values so that there is an order on them, which makes query processing easier.
- It is compatible with the adjacent labeling scheme so that it can be used to process queries with both reachability and adjacent edges. We will discuss this in details in Section 6.
- By slightly modification, the labeling scheme can be used to process graphs with circles.

Our proposed method is designed step by step. First, a hash-based join algorithm, HGJoin, is proposed for processing reachability queries on graph-structured XML. Second, the HGJoin algorithm is extended to the IT-HGJoin and T-HGJoin algorithms to process the reachability queries with multiple ancestors or multiple descendants. Then, Bi-HGJoin, the combination of IT-HGJoin and T-HGJoin, is designed to process queries in form of complete bipartite graphs. Finally, based on all the above algorithms, the method for processing subgraph queries in form of DAGs is proposed.

Without losing generality, this paper will focus on subgraph queries in form of DAG with only reachability relationships on edges for the convenience of discussion. With a slight modification, the method can be used to process any general subgraph queries.

The contributions of this paper are as follows:

- Based on the reachability scheme in [17], a family of hash-based join algorithms is presented as basic operators of subgraph query processing.
- For structural queries in form of general graphs, an efficient method is presented. The basic idea is to split a query into bipartite subqueries each of which can be processed by some HGJoin algorithm. In order to find effective splitting strategy, a cost-based query optimization strategy is presented with some acceleration strategies .
- The extensive experimental results show that the proposed algorithms outperform the existing algorithms and our query splitting strategy is effective and efficient.

The rest of this paper is organized as follows: Section 2 introduces some background knowledge. Section 3 presents the basic version of HGJoin algorithm for reachability query with one ancestor and one descendant. Section 4 illustrate the algorithms for queries in form of bipartite graphs. The strategy of processing queries in form of general DAGs is proposed in Section 5. The extensions of our method for the general subgraph queries are discussed in Section 6. Experimental results and analysis are shown in Section 7. Related work is discussed in Section 8 and Section 9 concludes this paper.

2. PRELIMINARIES

In this section, the background and notations used in this paper are presented.

2.1 Graph-structured XML Data and Queries

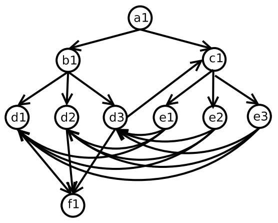
With IDREF-ID in an XML document representing reference relationship, an XML document can be considered as a tagged directed graph. Elements and attributes in an XML document is mapped to the nodes in a graph. Directed nesting relationships and reference relationships in an XML document is mapped to the edges in a graph. For example, an XML document is shown in Fig 2(a) and its graph structure is shown in Fig 2(b).

In a graph-structured XML document, structural queries are defined based on the structural relationship between nodes. In the graph structure G of an XML document, two nodes a and b satisfy *adjacent relationship* if and only if an edge from a to b exists in G; two nodes a and b satisfy *reachability relationship* if and only if a path from a to b exists in G. A *reachability query* $a \to d$ is to retrieve all pairs of nodes (n_a,n_d) in G where n_a has tag a, n_d has tag d and n_a and n_d satisfy reachabilty relationship in G. For example, the result of reachability $a \to e$ in the graph in Fig 2(b) includes (a,e_1), (a, e_2), (a,e_3). *Adjacent queries* can be defined similarly. The combination of multiple reachability or adjacent relationships forms subgraph queries.

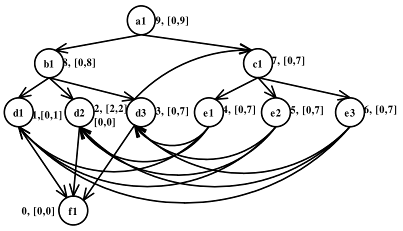
A subgraph query is a tagged directed graph Q=$\{V, E, tag, rel\}$, where V is the node set of Q; E is the edge set of Q; the function $tag : V \to TAG$ is the tag function (TAG is the set of tags); the function $rel : E \to AXIS$ shows the structural relationships between the nodes in the query ($AXIS$ is the set of relationships between nodes; $PC \in AXIS$ represents adjacent relationship; $AD \in AXIS$ represents reachability relationship). The directed graph (V,E) is called the *query graph.* If $ab \in E$ and $rel(ab) = PC$, then ab is called an *adjacent edge.* If $ab \in E$ and $rel(ab) = AD$, then ab is called a *reachability edge.* In V, the nodes without incoming edges are called *sources* and the nodes without outgoing

```
< a id="a1 >
< b id="b1" >
< d id="d1" f="f1"/ >
< d id="d2" >
< f id="f1"/ >
</d>
<d id="d3" f="f1" c="c1"/ >
< /b >
< c id="c1" >
< e id="e1" d="d1" d="d2" d="d3"/ >
< e id="e2" d="d1" d="d2" d="d3"/ >
< e id="e3" d="d1" d="d2" d="d3"/ >
< /c >
< /a >
```

(a) An XML Fragment

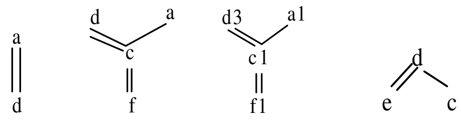


(b) The Graph of Fig 2(a)



(c) The Reachability code of Fig 2(a)

Figure 2: An Example of Graph-structured XML



(a) (b) (c) (d)

Figure 3: Example Queries

edges are called sinks. For simplicity, a subgraph query is represented as $Q = (V, E)$.

The result of a subgraph query $Q = \{V_Q, E_Q\}$ on a graph $G = \{V_G, E_G\}$ is $R_{G,Q} = \{g = (V_g, E_g)|V_g \subset V_G$ and $\exists$*bijective map* $f_g : V_g \to V_Q$ and $\exists$*injective map* $p_g : V_g \to V_G$ satisfying $\forall n \in V_g, tag(n) = tag(f_g(n)) = tag(p_g(n))$; $\forall e = (n_1, n_2) \in E_g, p_g(n_1), p_g(n_2)$ satisfy the relationship $rel(f(n_1), f(n_2))\}$.

For example, Fig 3(b) shows a subgraph query, in which a single line represents an adjacent edge and a double line represents a reachability edge. The result of such a query on the graph shown in Fig 2(b) is the graph in Fig 3(c).

2.2 Reachability Labelling Scheme

The labelling scheme for a graph-structured XML document is to judge the reachability relationship between any two nodes in an XML document without retrieving other information, such that subgraph queries can be processed efficiently. The labelling scheme used in this paper is an extension [24] of that in [17].

The reachability labelling scheme can be generated in the following steps:

- Each strongly connected component in G is contracted to one node to convert G to a DAG D.
- An optimum tree covering T of the DAG D is found. A depth-first traversal from the root of T accesses all nodes to generate the post-order of each node. Note that during the traversal, when a node n_C generated from a strongly connected component $C \subset G$ is accessed, if the post order of last accessed node is pc, then $pc + 1$, $pc + 2$, $\cdots$, $pc + |V_c|$ are assigned to n_C (where V_C is the set of nodes in C). Then, each node $n \in T$ is assigned a number id and an interval $[x,id]$, where id is the post order of n; x is the smallest post order of descendants of n in T.
- All the nodes in D are traversed in the reversed topological order. When a node n is reached, the interval sets of n's children in D are copied to that of n. Intersected intervals in the interval set of n are merged.
- $\forall n \in C$, its interval set is that of n_C; its id is one of the ids of n_C. Each node in C has a different id.

When such steps are finished, each node n in G is assigned a number $n.id$ and a set of intervals I_n. In [24], it is proved that a node a reaches a node b if and only if $b.id$ belongs to some interval of I_a. For example, the labelling scheme of the graph in Fig 2(b) is shown in Fig 2(c). Since the id of $f1$ is in the interval [0,0] of $d2$, it can be judged that $d2$ and $f1$ satisfy reachability relationship.

In order to retrieve pairs of nodes satisfying a reachability query based on reachability labelling scheme, the following storage strategy is applied. For each tag $t \in TAG$ of an XML document, two lists, $t.Alist$ and $t.Dlist$, are stored. N_t is the set of nodes with tag t. $t.Dlist$ is the list of the elements in set $\{n.id|n \in N_t\}$ sorted incrementally. $t.Alist$ is the elements in set $\{(val, n.id)|n \in N_t, [x, y] \in I_n, val = x \text{ } or \text{ } val = y\}$ sorted by the first item.

From the third step of encoding, for each node n, I_n has no overlapping intervals. Therefore, $tag(n).Alist$ has the following property which is the base of the algorithms in the following sections.

Proposition 1. *For a node n, all the elements in tag(n). Alist with the second item equals to n.id form an ordered list* $(val_{i_1}, n.id)$, $(val_{i_2}, n.id)$, $\cdots$, $(val_{i_k}, n.id)$. *In such an ordered list,* $l \le s \le u \le r \le v \le k$ *do not exist such that both* $[val_{i_s}, val_{i_r}]$ *and* $[val_{i_u}, val_{i_v}]$ *belong to* I_n.

Additionally, in order to judge whether all the intervals of one node has been processed, for each node n, a tuple (*null*, *id*) is inserted next to the last tuple of $tag(n).Alist$ with the second item equals to $n.id$, where *null* represents empty.

To process the queries with predicates and build-in functions, as the preprocessing step, the labelling schemes are filtered with the predicates and build-in functions before the processing of subgraph query.

In order to analyze the complexity of algorithms in this paper, $N = max_{t \in TAG}|\{n \in V_g|tag(n) = t\}|$. The set of the second items of all elements in $Alist$ is ID_{Alist}. Obviously, for each Alist and Dlist, $|ID_{Alist}| \le N$, $|Dlist| \le N$.

3. HASH-BASED JOIN ALGORITHM FOR GRAPH-STRUCTURED XML (HGJOIN)

In this section, a hash-based join algorithm is presented to process reachability queries in form of $a \to d$ on graph-structured XML data based on the reachability labelling scheme.

Algorithm 1 HGJoin

```
$a$=$Alist$.begin
$d$=$Dlist$.begin
while $a \neq Alist$.end AND $d \neq Dlist$.end do
  if $val_a \leq id_d$ then
    if $id_a \notin H$ then
      $H$.insert($id_a$)
      $a = a$.nextNode
    else if $val_a < id_d$ then
      $H$.delete($id_a$)
      $a = a$.nextNode
    else
      for $\forall id \in H$ do
        append ($id$, $id_d$) to $OutputList$
      d=d.nextNode;
  else
    for $\forall id \in H$ do
      append ($id$, $id_d$) to $OutputList$
    $d$=$d$.nextNode;
```

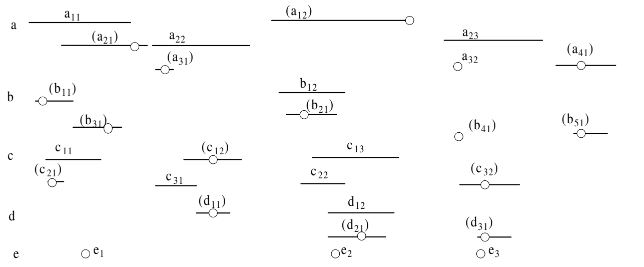


Figure 4: Example data

Based on the storage strategy, suppose $Alist = tag(a).Alist$ and $Dlist = tag(d).Dlist$. Intuitively, for any $id_j \in Dlist$, if two tuples in Alist, (x, id_i) and (y,id_i), satisfy $x \leq id_i \leq y$ and $[x, y]$ is an interval of the labelling scheme of some node, then (id_i, id_j) belongs to the result set. Proposition 1 assures that the query can be processed with scanning Alist and Dlist alternately only once. During the scan, a hash table H is used to store ids of nodes satisfying the condition that for each node n, the start point x of some interval in I_n has been scanned while corresponding end point y is not met. Proportion 1 shows that before such y is met, none of the start points of other intervals in I_n will be met. When a corresponding y is met, $n.id$ will be removed from H. The step is shown as following. At first, cursors a and d are assigned to Alist and Dlist, respectively. During the scan of Alist, if the current tuple $(val_a$,$id_a)$ satisfies $val_a \leq id_d$ and $id_a \in H$, it means that the end position of some interval is met and such an interval is impossible to contain id_d. Since the elements in Dlist is in incremental order, and such an interval is impossible to contain other unscanned elements in Dlist. So id_a is removed from H and a is updated. If $val_a = id_a$ and $id_a \in H$ or $val_a > id_d$, it means that id_d is contained in some interval with val_a as the end point. In such an instance, partial results are outputted and the scan is switched to Dlist. During the scan of Dlist, if $id_d < val_a$, or $val_a = id_a$ and $id_a \in H$, then based on the properties of elements in H, for $\forall id \in H$, (id, id_d) is outputted and d is updated; if $id_d > val_a$, or $val_a = id_d$ but $id_a \notin H$, it means that val_a is possibly the start position of an interval containing id_d and the scan is switched to Alist. The pseudo-code of HGJoin is shown in Algorithm 1.

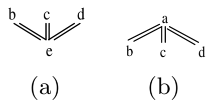


Figure 5: Query Examples

EXAMPLE 1. *In Fig 4, we give the intuition of Alist and Dlist, where a, b, c, d and e are tags. The elements with tag a are a_1, a_2, a_3 and a_4. Each interval of a_i is represented by a line segment and denoted by a_{ij}. The start point of the line segment is the start position of the interval and the end point of the line segment is the end position of the interval. The position of element id in the interval is represented by a circle. For example, $a_1.id$ is in the end position of interval a_{12} and $a_2.id$ is in the middle of the interval a_{21}; other symbols have the same meanings. For a reachablity query $a \rightarrow b$, corresponding Alist has an ordered list with first items sorted in form of (the end point) of a_{ij}, $a_i.id$ and Dlist=$\{b_1.id, b_3.id, b_2.id\}$. After Alist and Dlist scanned with HGJoin algorithm, the outputted result is ($a_1.id$, $b_1.id$), ($a_1.id$, $b_3.id$), ($a_2.id$, $b_3.id$), ($a_1.id$, $b_2.id$) in order.*

Complexity Analysis The time cost of HGJoin algorithm has three parts, the cost of operations of H, the cost of disk I/O and the cost of result outputting. The time cost is $Cost = \frac{|Alist|}{2} \cdot (cost_I + cost_D) + |result| \cdot cost_{out} + (\frac{|Alist| \cdot |entry_A|}{|B|} + \frac{|Dlist| \cdot |entry_D|}{|B|}) \cdot cost_{I/O}$ with each item corresponding to each part, where the cost of insertion and deletion of H once are $cost_I$ and $cost_D$, respectively, $|entry_A|$ and $|entry_D|$ are the sizes of each tuple in Alist and Dlist, respectively, B is the size of a disk block, $cost_{I/O}$ is the cost of accessing each disk block and $cost_{out}$ is the cost of outputting one tuple.

The space cost of HGJoin algorithm is the space cost of the hash table H. Therefore, the space complexity is the largest size of H during the algorithm. In the worst case, all the elements in ID_{Alist} are in H. Therefore, the space cost of HGJoin is no more than N.

4. EXTENSIONS OF HGJOIN

HGJoin can be extended to process some special cases of subgraph queries. For a subgraph query Q=(V, E, tag, rel), if $V = V_s \bigcup \{d\}$, $d \notin V_s$, and $E = \{(a, d)|a \in V_s\}$, then Q is an IT-query. If $V = a \bigcup V_s$, $a \notin V_s$, and $E = \{(a, d)|d \in V_s\}$, then Q is a T-query. In this section, we will present two extensions of HGJoin algorithm, IT-HGJoin and T-HGJoin to process IT-query and T-query, respectively.

4.1 Algorithm for IT-queries

In this section, HGJoin is extended to process IT-queries.

For an IT-query Q, suppose its sources are $a_1, \cdots, a_k$ and its sink is d. Let $Alist_i = tag(a_i).Alist$, $Dlist = tag(d).Dlist$, $i \in 1, 2, \cdots, k$. Similar as HGJoin algorithm, IT-HGJoin algorithm scans $Alist_1$, $\cdots$, $Alist_k$ and $Dlist$ alternatively once and obtains the results of an IT-query. During the scan, a hash table H_i is assigned to each $Alist_i$, the function of which is the same as H in $HGJoin$ algorithm. In the algorithm, cursors l_1, l_2, $\cdots$, l_k point to the current scanned position of $Alist_1$, $Alist_2$, $\cdots$, $Alist_k$, respectively. A cursor l points to the current scanned position of $Dlist$. The algo-

Algorithm 2 IT-HGJoin

```
for $i = 1$ to $k$ do do
  $a_i = Alist_i.begin$
d = $Dlist$.begin
while $a_i \neq Alist_i.end$ do
  for $i$=1 to $k$ do
    while $val_{a_i} < id_d$ do
      if $id_{a_i} \in H_i$ then
        $H_i$.insert($id_{a_i}$)
      else
        $H_i$.delete($id_{a_i}$)
      $a_i = a_i.next$
    while $val_{a_i} = val_d$ AND $id_{a_i} \notin H_i$ do
      $H_i$.insert($id_{a_i}$)
      $a_i = a_i.next$
  if none of the hash tables are not empty then
    $OutputList \bigcup = OutputTuples(H_1, \cdots, H_k, id_d)$
    for each $a_i$ do
      while $val_{a_i} = val_d$ AND $id_{a_i} \in H_i$ do
        $H_i.delete(id_{a_i})$
        $a_i = a_i.next$
      d = d.next
```

rithm scans $Alist_1$, $Alist_2$, $\cdots$, $Alist_k$ in turn. id_{l_i}s in pairs (val_{l_i}, id_{l_i}) satisfying $val_{l_i} \leq id_l$ and $id_{l_i} \notin H_i$ are inserted to H_i and id_{l_i}s in pairs (val_{l_i}, id_{l_i}) satisfying $val_{l_i} \leq id_l$ and $id_{l_i} \in H_i$ are removed from H_i. After $Alist_k$ is processed, the scan is switched to *Dlist*. Such processing is similar as the Dlist scan in HGJoin algorithm. At that time, the node corresponding to each *id* in any H_i is an ancestor of the node corresponding to id_l. If any of H_i is not empty, it means that with its ancestors, the node with id_l matches the descendent in the query. Such subgraphs are outputted as partial results. Since each tuple in $H_1 \times H_2 \times \cdots \times H_k$ corresponds to each group of different ancestors of id_l, all the tuples in $H_1 \times H_2 \times \cdots \times H_k \times \{id_l\}$ should be outputted. In the step of result output, function OutputTuples(H_1,$\cdots$, H_k, id_d) is invoked. After such partial results are outputted, the scan is switched to $Alist_1$, $\cdots$, $Alist_k$. IT-HGJoin algorithm is shown in Algorithm 2.

In the implementation of IT-HGJoin algorithm, the size of $|H_1 \times H_2 \times \cdots \times H_k|$ may be very large. In order to store partial results efficiently, latency processing strategy is applied. That is, H_1, $\cdots$, H_k and corresponding id_l are stored respectively. The Cartesian production is not performed until such partial result will be used.

Complexities Analysis Obviously, the worst space complexity is kN. Similar as the analysis of HGJoin, the time complexity $Cost = \sum \frac{|Alist_i|}{2} \cdot (cost_I + cost_O) + |result| \cdot cost_{out} + (\frac{\sum |Alist_i| \cdot |entry_A|}{|B|} + \frac{|Dlist| \cdot |entry_D|}{|B|}) \cdot cost_{I/O}$

4.2 Algorithm for T-queries

In this section, an algorithm for processing T-queries is presented. In a T-query Q, the source is a and sinks are d_1,$\cdots$, d_k. Let *Alist*=tag(a).*Alist*, $Dlist_i$=tag(d_i).*Dlist*, $i \in \{1, 2, \cdots, k\}$. Since in Q, the source a has multiples descendants d_1, $\cdots$ d_k and in the reachability labelling scheme, all the nodes with tags $tag(d_1)$, $\cdots$ $tag(d_k)$ do not belong to the same interval of a node with $tag(a)$, in order to obtain the result of Q, all results of reachability query $a \rightarrow d_i$ where $i \in \{1, \cdots, k\}$ should be obtained and the join operation is performed on such intermediate results.

HGJoin can be applied to process reachability queries $a \rightarrow d_i$. For the interest of efficiency, the k way scans of *HGJoin* algorithm are combined. That is, during the scans on Alist, $Dlist_1$, $\cdots$, $Dlist_k$ are processed together. Such that all intermediate results can be obtained by scanning all lists only once. In order to make a join operation efficient, a hash table IHT_i is assigned to each $Dlist_i$. When a bucket in some IHT_i is full, the intermediate results in such a bucket are sorted based on the first items (the id value of the node matching a) and written out to the disk.

When the intermediate results are obtained, all tuples in form of $(id, id_1) \in IHT_1$, $\cdots$, $(id, id_k)IHT_k$ are joined to generate a tuple, $(id, id_1, \cdot, id_k)$, the partial result of IT-query. Obviously, the cost of join operation increases fast with $|IHT_i|$. In order to reduce the cost of join, the size of IHT_i should be decreased during join.

The strategy in T-Join algorithm is that during the scan of Alist, when the end sign (*null*,*id*) (see Section 2.2) is met, it means that the following steps of the scan will not generate intermediate result in form of (id, $*$). Therefore, the join operation can be performed on current IHT_1,$\cdots$,IHT_k to generate all results in form of (id, id_1, $\cdots$, id_k). Then the tuples with form (id, $*$) are deleted from IHT_1,$\cdots$,IHT_k and corresponding disk blocks are merged.

Even though the above strategy can minimize the size of $|IHT_i|$ during join operation, frequent join operations will make an algorithm inefficient. Additionally, after each join operation, the number of disk blocks to be merged is very limited. In order to make it more efficient, the "join latency" strategy is applied in T-HGJoin algorithm. That is, during T-HGJoin algorithm, an *ancestor table A* with fixed size is maintained. During the scan of Alist, when end sign (*null*,*id*) is met, *id* is inserted to *A*. When *A* is full or the scan of Alist is finished, the join operation is performed on current IHT_1, $\cdots$, IHT_k to generate partial query results with form (id,$*$,$\cdots$,$*$) where *id* is the id of any element in *A*. Then intermediate results with form (id,$*$) are deleted and for each bucket in any IHT_i, mergable disk blocks are merged.

In order to reduce the space cost of partial results storage, the latency processing strategy similar as IT-HGJoin can also be applied.

Example 2. *The processing of the T-query in Fig 5(b) on the data in Fig 4 is considered. For easy discussion, suppose that each bucket can only contain two tuples and each IHT uses hash function hash(x)=x mod 2. It means that each IHT has only two buckets. The size of ancestor table A is 2. When T-HGJoin algorithm accesses tuple (null, a_1.id), generated intermediate results are shown in Fig 6(a). a_1.id is inserted to A. At that time, A is not full, so the join operation is not performed. When the tuple (null,a_3.id) is scanned, the intermediate results are shown in Fig 6(b). a_3.id is inserted to A and A is full, so the join operation is performed on the intermediate results with only a_1 and a_3 and results (a_1, b_1, c_2, d_2), (a_1, b_2, c_2, d_2) and (a_1, b_3, c_2, d_2) are generated. After join, all intermediate results in IHTs related to a_1 and a_3 are deleted. Current intermediate result is shown as Fig 6(c).*

Complexities Analysis The time cost of T-HGJoin includes 4 parts, the cost of operations on H, the disk I/O cost of accessing Alist and all *Dlist*s, the cost of processing intermediate results and the cost of outputting final results. In the worst case, since each element in Alist is the ancestor of any element in each $Dlist_i$, $|IHT_i| \leq N^2$. Therefore, the worst time cost of T-HGJoin is $Cost = \frac{|Alist|}{2} \cdot (cost_I + cost_O) + (\frac{|Alist| \cdot |entry_A|}{|B|} + \frac{\sum |Dlist_i| \cdot |entry_D|}{|B|}) \cdot cost_{I/O} + (k \cdot N^2 \cdot$

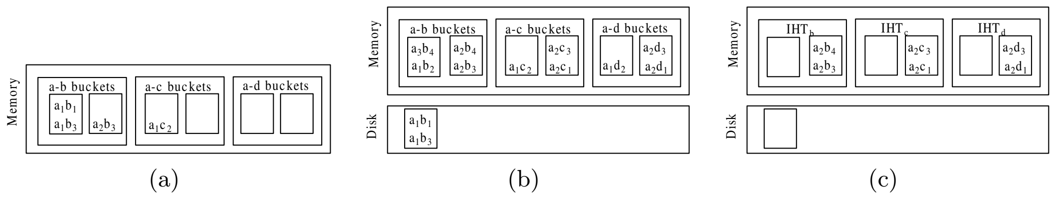


Figure 6: Intermediate Results in different Steps of Example 2

$cost_I + 2\cdot k\cdot(\frac{|entry_{IHT}|\cdot N^2}{|B|} - n_b)\cdot cost_{I/O} + [(\frac{|entry_{IHT}|\cdot N^2}{|B|\cdot n_b} - 1) + (\frac{N^2}{n_b})^k \cdot cost_{join}]\cdot N) + |result_F|\cdot cost_{out}$, each item of which corresponds the cost of each part. The cost analysis of the former two and last parts is similar as that of HGJoin. In the third item, $cost_{join}$ is the cost of generating one output tuple.

The main memory cost of T-HGJoin algorithm includes the space for the hash table H during the scan of Alist and main memory used to store intermediate results in IHT_i. With the symbols discussed above, the main memory space cost of T-HGJoin is $N + k\cdot n_b\cdot|B|$ in the worst case.

4.3 Algorithm for Bipartite Queries

In this section, the processing algorithm for bipartite subgraph queries is presented. At first, the algorithm for the bipartite subgraph queries in a special case that all descendants share the same ancestor (*CBi* for brief) is presented and then that of bipartite subgraph queries is presented. Suppose that the sources of a CBi query are a_1, $\cdots$, a_m and the sources are d_1,$\cdots$, d_n. Let $Alist_i$=tag(a_i).Alist, $Dlist_j$=tag(d_j).Dlist, $i \in \{1, \cdots, m\}$, $j \in \{1, \cdots, n\}$. In the algorithm, cursors l_1, $\cdots$, l_m points to the current scanned position of $Alist_1$, $\cdots$, $Alist_m$, respectively. t_1, $\cdots$, t_n points to the current scanned positions of $Dlist_1$, $\cdots$, $Dlist_n$, respectively.

Similar as IT-HGJoin, the algorithm assigns a hash table H_i for each source a_i. Similar as T-HGJoin, the algorithm assigns a hash table IHT_j for the intermediate results corresponding to each sink d_j.

Bi-HGJoin algorithm includes two alternative steps. In the first step, similar as IT-HGJoin, the algorithm scans $Alist_1$,$\cdots$, $Alist_m$ in turn and inserts id_{l_i} in the pair (val_{l_i}, id_{l_i}) satisfying $val_{l_i} \leq min(id_{t_j})$ $(1 \leq j \leq n)$ and $id_{l_i} \notin H_i$ to H_i. When $Alist_m$ is processed, the algorithm switches to process the $Dlist_j$ with the smallest id_{t_j}. If the hash tables H_1,$\cdots$,H_m is not empty, each tuple (h_1, $\cdots$, h_m, id_{t_j})$\in$ $H_1 \times \cdots \times H_m \times \{id_{t_j}\}$ is inserted to the bucket with hash value hash(h_1, , $\cdots$, h_m) of IHT_j. The second step is similar as the join of intermediate results in T-HGJoin. When in each $Alist_i$, the end sign ($null$, h_i) of h_i is met, where $1 \leq i \leq m$, the combination of ancestor (h_1, $\cdots$, h_m) is inserted into the ancestor table A. When A is full or all scans on Alists have been finished, for each combination (h_1, $\cdots$, h_m) in A, the buckets with hash value hash(h_1, $\cdots$, h_m) in IHT_1,$\cdots$, IHT_n are joined to generate partial query result with form (h_1, $\cdots$, h_m, id_1, $\cdots$, id_n). Then corresponding intermediate results are deleted and disk blocks in such buckets are merged. When the join operation is finished, the first step resumes. The above steps are repeated until all elements in any $Alist_i$ have been scanned.

Such algorithm cannot process general bipartite subgraph queries. Since sinks may have different sources, it is difficult to find a hash function to assure effective execution of a join operation. Intuitively, such a problem can be processed with following steps. Based on the cost model presented in Section 5.1, a query is split to some CBi subqueries. Then Bi-HGJoin is invoked to process CBi subqueries to obtain intermediate results. At last, intermediate results are joined together to obtain final query results. For example, the bipartite subgraph query shown in Fig 7(b) can be split to CBi subqueries in Fig 7(c) and Fig 7(d). When intermediate results are obtained, the equal join is performed on f elements to obtain final results.

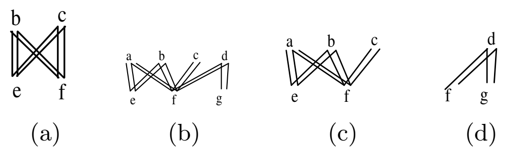


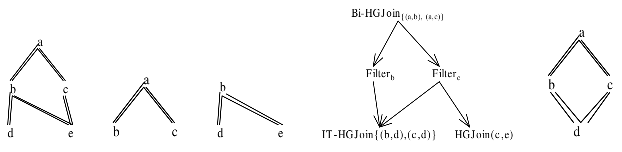
Figure 7: An Example for Complex Bipartite Query

5. DAG SUBGRAPH QUERY EVALUATION

In this section, we present a hash-based evaluation strategy for structural queries in form of DAGs.

The direct processing of a general DAG subgraph query requires not only large main memory space but also large disk space. The efficiency is affected. Hence the strategy presented in this section does not process general subgraph queries directly but also splits a DAG subgraph query to some CBi-queries. Each CBi subquery is processed to obtain intermediate results. Then, join operations are executed to obtain final results. Such join is performed with sort-merge algorithms. For example, to process the query in Fig 8(a), it is split to the subquerires: q_{11} in Fig 8(b), q_{12} in Fig 8(c) and reachability query $c \rightarrow e$. Then labelling schemes with tags tag(b) and tag(c) are filtered with intermediate results of q_{11}. Then subquery q_{12} is processed on filtered labelling schemes to obtain intermediate results. The reachability query is processed on filtered data to obtain intermediate results. At last, the join operation is performed on such three groups of intermediate results to obtain final results.

Obviously, the key of the above strategy is how to split the query and construct the query plan. A query plan can be modelled as a DAG D=(V, E), where each node in V represents an operation (possible operations includes HGJoin, IT-HGJoin, T-HGJoin and Bi-HGJoin, Filter and sort-merge operations). The results of the operation in arrow tail is the input of operation in arrow head. For example, in Fig 8(d), $T - HGJoin_{\{(a,b),(a,c)\}}$ represents that



(a) (b) q_{11} (c) q_{12} (d) (e)

Figure 8: Example of Query Plan

T-HGJoin algorithm is performed on the labelling schemes with tag $tag(a)$, $tag(b)$ and $tab(c)$. $Filter_c$ represents that the labelling schemes with tag $tag(c)$ are filtered to eliminate the labelling schemes which are not in the intermediate results of $T-HGJoin_{\{(a,b),(a,c)\}}$. Other operations have the same meanings.

At first, we design the cost model. Then query plan generation algorithm and query optimization accelerate strategy are presented.

5.1 The Cost Model

The query plan of a subgraph query has multiple choices. In order to generate an efficient query plan, a query optimizer is required. As the base of the query optimization, the cost model is presented in this section. For each operation in the query plan, its cost has two parts, execution time and required main memory. The former represents the execution efficiency of query plan and the latter is the main memory space which is required during query plan execution. The estimation of the cost of sort-merge join operation has been studied extensively and the time cost of HGJoin and IT-HGJoin can be estimated as the time complexity in Section 3 and Section 4.1, respectively. This section focuses on the cost model of T-HGJoin and Bi-HGJoin.

The cost model of T-HGJoin is similar as time analysis in Section 4.2. With intermediate size estimation techniques[15, 16], for $\forall a \in ID_{Alist}$ and $\forall i \in \{1, 2, , k\}$, $P_{a,i}$, the number of tuples related to a in the intermediate results of join operation of $Alist$ and $Dlist_i$, can be estimated. Additionally, such technique can be used to estimate $S_{a,i}$, the number of disk blocks of the intermediate results related to a in IHT_i at the time when $(null,a)$ is met in $Alist$. Therefore, intermediate results related to a can be estimated as $S_a = \sum_{i\in 1,\dots,k} S_{a,i}$. When intermediate results related to a are joined, $S_{a,1}$,$\cdots$,$S_{a,k}$ times of disk blocks distributed in $IHT_1, \cdots, IHT_k$ are required to be accessed with Nest_loop method, respectively. In such step, the number of disk blocks to be accessed is estimated as $NL_a = \sum_{i\in 1,\dots,k} S_{a,i}$. Based on such estimations, the time cost of T-HGJoin is estimated as $Cost = \frac{|Alist|}{2} \cdot (cost_I + cost_O) + (\frac{|Alist|\cdot|entry_A|}{|B|} + \frac{\sum|Dlist_i|\cdot|entry_D|}{|B|}) \cdot cost_{I/O} + cost_I \cdot \sum_{i=1}^{k} selectivity(A, D_i) + 2 \cdot \sum_{a\in ID_{Alist}} S_a \cdot cost_{I/O} + \sum_{a\in A} NL_a + |result_F| \cdot cost_{out}$, where $sel(A, D_i)$ is the result number of join on A and D_i.

The estimation of Bi-HGJoin is similar as T-HGJoin. The number of intermediate result generation with the join on $Alist_1$, $\cdots$, $Alist_m$ and $Dlist$ is denoted by sel(A_1, $\cdots$, A_m, D_i). The number of intermediate results related to the combination of nodes a_1, $\cdots$, $a_m(a_i \in ID_{Alist_i}$ is denoted by $S_{a_1\cdots,a_m}$. The number of disk blocks for joining intermediate results with a_1,$\cdots$, a_m is denoted by $NL_{a_1,\cdots,a_m}$. The estimation of these parameters is similar as those of T-HGJoin. The time of Bi-HGJoin operation can be estimated as $Cost = \frac{\sum|Alist_i|}{2} \cdot (cost_I + cost_O) + (\frac{\sum|Alist|\cdot|entry_A|}{|B|} + \frac{\sum|Dlist_i|\cdot|entry_D|}{|B|}) \cdot cost_{I/O} + cost_I \cdot \sum_{i=1}^{n} sel(A_1, \cdots, A_m, D_i) + 2 \cdot \sum_{a_i\in A_i} S_{a_1,\cdots a_m} \cdot cost_{I/O} + \sum_{a_i\in A_{j_i}} NL_{a_1,\cdots,a_m} + |result_F| \cdot cost_{out}$.

The main memory required by an operation includes fixed main memory and alterable main memory. The fixed main memory is the buffer for the intermediate result. The size of such part equals to a fixed value $fixed\_mm$. The alterable part is the main memory of the hash tables corresponding to Alists during query processing. The size of such part is linear with the maximum number of ancestors of descendants during query processing. $ancestor_d$ represents the number of ancestors in Alist(s) of $d \in Dlist$. $|entry_H|$ is the size of data element in hash table H. The space cost of the operation is $cost_{mm} = fixed_{mm} + \max_{d\in Dlist}(|ancestor_d|) \cdot |entry_H|$.

5.2 Algorithms for Query Plan Generation

In this section, the process of query execution is represented by state graph and then optimal query plan is generated as the shortest path generation algorithm on the state graph.

Suppose (V_Q,E_Q) is the query graph of a subgraph query Q (m=$|E_Q|$). The directed graph $G^* = (V_{G*}, E_{G*})$ is the *state graph* of the subgraph query Q, where $V_{G^*} = \{g|g = (V_g, E_g)\ and\ E_g \subset E_Q\}$. There is a directed edge from $g \in V_{G^*}$ to $g' \in V_{G^*}$ if and only if a subquery $Q_{gg'} = (V_{gg'}, E_{gg'})$ belonging to one of reachability query, T-query, IT-query and CBi-query exists with $E_g - E_{gg'} = E_{g'}$. The node in G^* representing the query graph (V_Q, E_Q) is called the *start state* of G^*, denoted by g_0. The node (V_Q, ϕ) in G^* is called *end state* of G^*, denoted by g_m. Other elements in V_{G^*} are called *intermediate states* of G^*.

Obviously, in G^*, any path g_0=g_{i_1},g_{i_2},$\cdots$, $g_{i_k} = g_m$ ($k \leq m$) from g_0 to g_m corresponds to a processing course of the query Q. Such course processes subqueries $Q_{g_{i_j} g_{i_{j+1}}} 1 \leq j < k$) step by step. The query plan describing such course is generated with following steps. For the first edge in the path $g_{i_1} g_{i_2}$, an operation O_E (where $E = E_{g_{i_1} g_{i_2}}$) is constructed based on the operation type $O \in \{HGJoin, T-HGJoin, IT-HGJoin, Bi-HGJoin\}$ of $Q_{g_{i_1} g_{i_2}}$.

It is supposed that the query plan for g_0=g_{i_1},g_{i_2},$\cdots$,g_{i_j} has been generated and the collection of sets of intermediate results is denoted by $\mathbb{B}_j$. The edge $g_{i_j} g_{i_{j+1}}$ in the path is considered. At first, based on the operation type $O \in \{HGJoin, T-HGJoin, IT-HGJoin, Bi-HGJoin\}$ of $Q_{g_{i_j} g_{i_{j+1}}}$, an operation O_E is added to the query plan (where $E = E_{g_{i_1} g_{i_2}}$). Then each node $a \in V_{g_{i_1} g_{i_2}}$ is considered. If some set of intermediate results exists in $\mathbb{B}_j$, then an operation $Filter_a$ is added to the query plan. An edge from corresponding intermediate result set is added to $Filter_a$ and another edge from $Filter_a$ is added to O_E. Then intermediate results set of O_E is added to $\mathbb{B}_j$. At that time, if there is mergable intermediate results set in $\mathbb{B}_j$, then an operation sort-merge is added to the query plan and an edge from corresponding intermediate is added result set to new added sort-merge operation. At the same time, the unmerged intermediate results set is deleted from $\mathbb{B}_j$ and the merged intermediate result set is inserted. The above steps are repeated until no intermediate result sets can be merged. Let $\mathbb{B}_{j+1} = \mathbb{B}_j$. The query plan of other edges is generated until all the edges in the path are considered.

During the generation of a query plan for the path $g_0 =$

g_{i_1},g_{i_2},$\cdots$,g_{i_k}=g_m, a group of operations are added to the query plan for $g_{i_j}g_{i_{j+1}}$. $wg_{i_j}g_{i_{j+1}}$, the sum of time cost of such operation group, and the maximum space cost w are considered. If w is not larger than available main memory space, then let the weight of edge $g_{i_j}g_{i_{j+1}}$ equal to $w_{g_{i_j}g_{i_{j+1}}}$. Otherwise, such group of operations is infeasible and the weight of edge $g_{i_j}g_{i_{j+1}}$ equals to $+\infty$. In such a way, any edge gg' in G^* is unique, since whatever the path to g is, the collection of sets of intermediate results sets are the same. Therefore, the operation added to gg' is the same.

From the above discussion, it can be seen that a path from g_0 to g_m in the weighted query graph G^*=(V_{G^*},E_{G^*}) corresponds to a query plan with the weighted length as the cost of the query plan. Therefore, the generation of the optimal query plan is to find the shortest path from g_0 to g_m in G^*. Such problem can be solved with Dijkstra algorithm [1]. For the interests of space, details of the algorithm is omitted. From the construction of G^*, it can be known that $|V_{G^*}| = 2^m$. Since the time complexity of Dijkstra algorithm with n nodes in the graph is O(n^2), the time complexity for the generation of the shortest path with Dijkstra algorithm is $O(2^{2m})$. Note that m is the number of edges in the query graph, for the graph with smaller graphs such algorithm is efficient.

5.3 Query Optimization Acceleration

In section 5.2, when the size of G^* is large, both the time and space complexity of optimal query plan generation with Dijkstra algorithm will be very large. In this section, some acceleration rules are presented.

In the following discussion, the weighted length of the current shortest path from g_0 to g is denoted by w_g with initial value $+\infty$. Once g is reached in the algorithm, w_g is updated. Since Dijkstra algorithm uses Best-first expanding strategy, when g is chosen to expend, the shortest path from g_0 to g is obtained. Based on such property, the following rules can be obtained. The former can halt the algorithm to reduce the run time. The latter will delete the states impossibly belonging to the shortest path to reduce the space cost.

Rule 1 In the Dijkstra algorithm, if the selected state g equals to g_m or $w_g \geq w_{g_m}$, then the current shortest path from g_0 to g_m is outputted and the algorithm halts.

Rule 2 In the Dijkstra algorithm, if the selected state is g and each outgoing edge gg' of g satisfies $w_g + w_{gg'} > w_{g'}$ ($w_{gg'}$ is the weight of the edge gg'), then g is deleted from the data structure of the Dijkstra algorithm.

Proposition 2. *Rule 1 and Rule 2 will not affect the result of query optimization.*

Intuitively, the time cost of executing a complex operation directly is often smaller than the sum of the time cost of the series of simple operation split from such operation. For example, a CBi-subquery can be split to some T-subqueries or IT-subqueries, but the sum of the run time of these subqueries is often larger than the run time of execution of CBi directly. Therefore, in the Dijkstra algorithm, the states with all descendants expanded can be neglected to accelerate query optimization. This is Rule 3.

Rule 3 For current state g and $\exists gg' \in E_{G^*}$ in Dijkstra algorithm, if some descendant state of g' has been inserted into the data structure, then the expanding of g' will not be performed.

For a child state g' of g, the selectivity of g is defined as the selectivity of the operations corresponding to the edge gg'. When the current state g chosen to expand with Dijkstra algorithm has multiple children states, only the children with higher selectivities are chosen. So that query optimization will be accelerated. Then we have the following rule.

Rule 4 For each expansion state g in Dijkstra algorithm, all the children of g are sorted by the selectivities. When the expansion is performed for C times or all the children have been processed, the expansion of g is finished.

Even though Rule 3 and Rule 4 will result in non-optimal query plan. These two rules can reduce the time complexity of query optimization effectively.

Proposition 3. *With Rule 3 and Rule 4, the complexity of Dijkstra algorithm is $O(C \cdot 2^m)$ in the worst case.*

6. DISCUSSIONS

In this section, we present the discussions about two variations to make our query processing method to support subgraph queries in form of general graphs. One is how to make the method to support subgraph query in form of graphs with circles. The other is how to make the method to support subgraph queries with adjacent relationships.

6.1 Evaluate Queries with Circles

In this section, we present a strategy to adapt the family of HGJoin to support subgraph queries with circles. Such strategy is based on the feature that all reachability labelling scheme of the nodes in the same strongly connected component(SCC for brief) share the same interval sets.

The basic idea is to identify all the SCCs in the nodes in the graph related to the query with labelling schemes. An id is assigned to each SCC and such id is also assigned to the nodes belong to such SCC. All edges in the SCC in the query are deleted. Each separate part of the query is processed individually. Then the results of these parts are joined together with the id of SCC based on the nodes corresponding to the query nodes in the same SCC.

6.2 Evaluate Queries with Adjacent Edges

Subgraph queries with adjacent edges can be processed in the algorithms similar to the family of HGJoin.

An adjacent labelling schema is assigned to each node. The generation of adjacent labelling scheme is that for each node with postid i, interval [i,i] is assigned to each of its parents. The benefit of such scheme is that the judgement of adjacent relationship is same as that of reachability labelling scheme so that the processing techniques for subgraphs with only reachability relationships can be applied to evaluate structural queries with adjacent edges.

If a query node n as an ancestor has both reachability and adjacent outgoing edges, n should be split to two query nodes with only reachability and adjacent outgoing edges, respectively. It is because different intervals are used to judge reachability and adjacent relationships. Considering only outgoing edges is because the judgements of reachability and adjacent relationship use the same *postid*s. When the query processing method is applied, for the query node with reachability outgoing edges, intervals in reachability scheme are used, while for query node with adjacent outgoing edges, intervals in adjacent scheme are used. The algorithm is same as the corresponding one in the family of HGJoin.

Table 1: Statistics of the XMark Datasets

factor	0.1	0.5	1.0	1.5	2	2.5
Size(M)	11	56	113	170	226	284
#Nodes(K)	175	871	1681	2524	3367	4213
#Edges(K)	206	1024	1988	2985	3982	4981
Num(M)	0.98	4.9	9.71	14.7	19.6	24.7

Table 2: The Quality of Query Plan

Query	OPT	MAX	MIN	AVG
XMQC1	26923	169797	55641	101715
XMQC2	62720	234640	66953	154243.8

7. EXPERIMENTAL EVALUATION

In this section, we present the results and analysis of part of our extensive experiments on the algorithms in this paper.

7.1 Experimental Setup

Experiments were run on Pentium 3GHz with 512M memory. We implemented all our algorithms in this paper.

The dataset we tested is the XMark benchmark [20]. It can be modeled as a graph with complicated schema and circles. Documents are generated with factors 0.1 to 2.5. Their parameters are shown in Table 1, where Num is the number of numbers in the labelling scheme in the storage.

In order to evaluate the algorithms on graphs in various forms, we also use synthetic data generated with 2 parameters: the number of nodes with each tag (*node number* for brief) n and the probability between nodes with two different tags(*edge probability* for brief) p. The data set generated in this way is called the random dataset. All the graphs in the random dataset have 8 tags in order $\{A, B, C, D, E, F, G, H\}$. The graphs in the random dataset may be DAGs or general graph (GG for brief). For a GG, the probability between each pair of nodes with any tag is p. For a DAG, only the probability of an existing edge from a node with smaller tag to a node with larger tag is p but the probability of the edges in inverted direction is 0. We use run time as the measure of algorithms (RT for brief).

For queries on XMark, we choose two queries for each algorithm, one contains all the nodes not in the circle with smaller selectivity, the other one contains some nodes in the circle with larger selectivity. The queries for HGJoin are XMQS1:$text \rightsquigarrow emph$ and XMQS2:$person \rightsquigarrow bold$. Queries for IT-HGJoin and T-HGJoin are in Fig 9(a), Fig 9(b) and Fig 9(c), Fig 9(d), respectively. For the comparison with the algorithm in [4], we also design complex twig queries XMQW1 and XMQW2 shown in Fig 9(c) and Fig 9(d). In order to study the performance of query optimization deeply, we design two complex structured queries XMQC1 and XMQC2 in Fig 9(i) and Fig 9(j), respectively. Since Bi-HGJoin algorithm is the combination of IT-HGJoin and T-HGJoin and its features are represented by the experiments of these two algorithms, due to space limitation, the experimental results specially for Bi-HGJoin are not shown. In order to make query graphs clear, without confusion, in the query graphs we use arrows to represent AD relationships. Some of these queries are from real instances while some of them are synthetic. For example, XMQS1 represents the query for retrieving the text and the emph part belonging to it and XMQT1 is to retrieve the text with all emph, bold and keywors in it.

For queries on the random dataset, since the selectivity is mainly determined by edge probability, we choose one query for each algorithm. The query for HGJoin is RQS: $A \rightsquigarrow E$. Queries for IT-HGJoin and T-HGJoin are in Fig 9(e) and Fig 9(f). Twig query and complex query are RQW and RQC, shown in Fig 9(k) and Fig 9(l).

Since the query processing methods for the queries in form of circle or with adjacent relationships are the extensions of that for DAG queries, the features of these algorithms are similar. Due to space limitation, the experimental results of such queries are not shown in this paper.

7.2 Comparisons

For comparison, we implemented stackD algorithm in [4]. The comparison is performed on 10M XMark data and random XML document in form of DAG and general graph with 4096 nodes and edge probability of 0.1, 0.8, 0.4, representing sparse edges, dense edges and the density of edge between those two instances. Note that in such case, the edge probabilities 0.1, 0.4 and 0.8 correspond to the ratio of the numbers of edges and the numbers of nodes of 250, 922 and 1794. The results are shown in Fig 10 From the results, it can be seen that the efficiency of our algorithm outperforms StackD significantly, especially when the density of edges is large. For random graphs with high edge density, the difference is the most significant. It has two reasons. The first is that when the edge of a graph is dense, one interval may be shared by many nodes; our method can process same intervals of nodes with same tag together while StackD processes them separately. The second is when nodes have many intervals, stackD has to maintain a very large data structure, the operations on which is costly.

7.3 The Effectiveness of Query Optimization

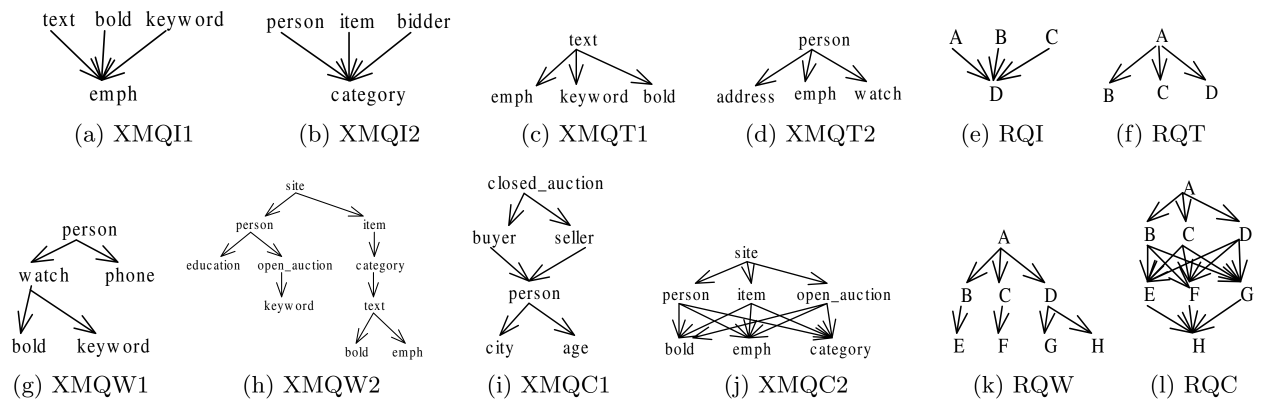
To validate the effectiveness of the query optimization, we check the quality of query plans and the efficiency of query optimization. In the experiments in this section, we fixed the available memory to 1M and performed query optimization on the 50M XMark document.

7.3.1 The Quality of Query Plan

To validate the quality of query plans, we compare the execution time of the query plan generated by the optimizer with those of 10 random query plans. The results are shown in Table 2, where the unit of time is ms and OPT is the execution time of query plan optimized with rule4 with $C = 4$. The maximal, minimal and average run time of 10 randomly generated query plans are shown in the columns of MAX, MIN and AVG, respectively. From the results, the query optimal strategy always avoids the worst query plan and obtains a better query plan than random plans do.

7.3.2 The Efficiency of Query Optimization

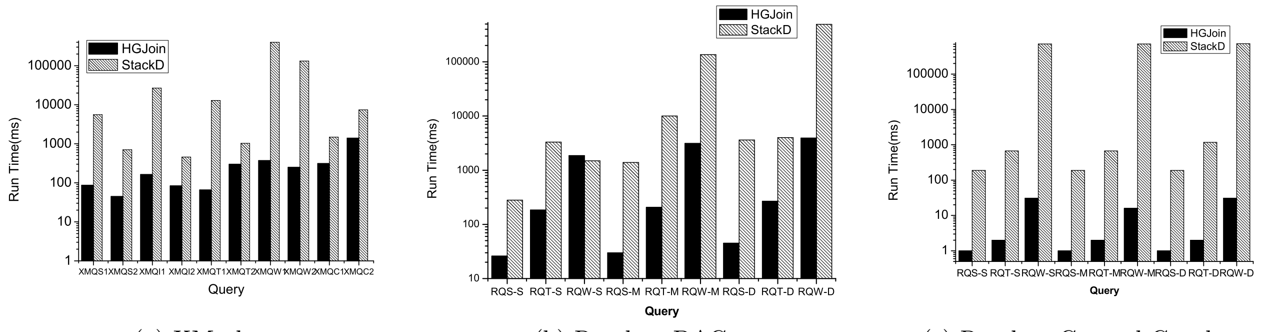
To check the efficiency of query optimization, we compare the optimization time of XMQC1 and XMQC2 with various acceleration rules. The result is shown in Table 3, where *timei* represents the optimization time of the optimization with $Rule_i$, respectively. EXE-Time is the run time of query plan obtained by query optimizer with rule4. Here in rule4, C is set to be 4. The unit of time is ms. From the results, it can be seen that our rules can reduce query optimization time effectively and comparing with query plan execution



(a) XMQI1 (b) XMQI2 (c) XMQT1 (d) XMQT2 (e) RQI (f) RQT

(g) XMQW1 (h) XMQW2 (i) XMQC1 (j) XMQC2 (k) RQW (l) RQC

Figure 9: Query Set



(a) XMark (b) Random DAG (c) Random General Graph

Figure 10: Results of Comparison

Table 3: Efficiency of Query Optimization

Query	time1	time2	time3	time4	EXE-Time
XMQC1	47	16	2	1	17328
XMQC2	104968	16109	35	34	62720

time, optimized query optimization time is very small.

7.4 Changing Parameters

7.4.1 Scalability

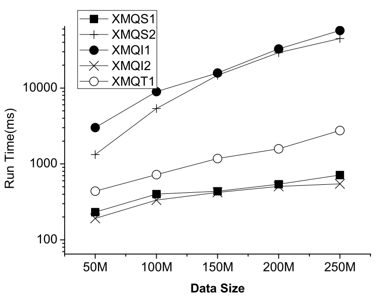
The scalability experiment is to test the run time with the document in the same schema but with various size. In our experiments, for XMark, we change the factor from 0.5 to 2.5 and the results are shown in Fig 11(a) and Fig 11(b). Run time of Fig 11(a) is in log scale. From the results, the run time of XMQS1, XMQS2, XMQI1, XMQI2 and XMQW2 changes almost linearly with the data size. When data size gets larger, the process times of XMQT1, XMQT2 , XMQW1, XMWC1 and XMWC2 increase fast. It is because major parts of these queries are related to some circle or bipartite subgraph in XML data. The results of such part are as the Cartesian production of related nodes and the number of results and intermediate results increase in the power of the number of query nodes. Therefore, the processing time increases faster than linearly. Since the time complexity is related to the size of results, it is inherent.

For the random dataset, experiments on DAG were performed. Node number factors are changed with fixed edge probabilities 0.1. The results are shown in Fig 11(c). Note that run time and node number axes of Fig 11(c) are in log scale. For the same reason discussed in the last paragraph, the query processing time of RQS, RQI, RQT and RQW increases faster than linearly with node number. The run time of RQC does not increase significantly with node number because with query optimizer, RQC is performed bottom-up and the selectivity of subquery ($\{E, F, G\}$, H) is very small.

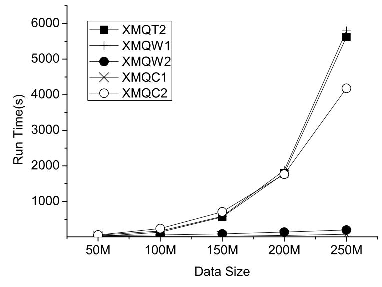
As a result, the query processing time increases faster than linearly only when the size of final results increases faster.

For the random dataset, we performed experiments on DAGs and changed edge probabilities from 0.1 to 1.0 with fixed node number 4096. The results are shown in Fig 11(d), it shows that the run time of RQS, RQI, RQT and RQW does not change significantly with the number of edges. It is because when the edges become dense, more intervals are copied to ancestors and the intervals of all nodes trend to be the same. Based on our data preprocessing strategy, same intervals are merged. Therefore, the query processing time of these queries does not change a lot. RQC is an exception. When the edge probability changes from 0.2 to 0.3, the run time changes significantly. It is because RQC is complex and only when the density of edges reaches a threshold, the number of results becomes large.

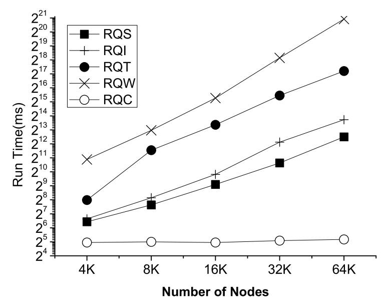
We also performed experiments on general graphs with



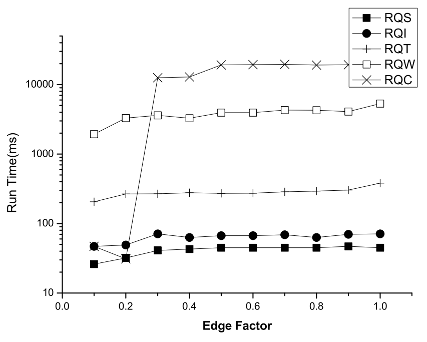
(a) Scalability-XMark1



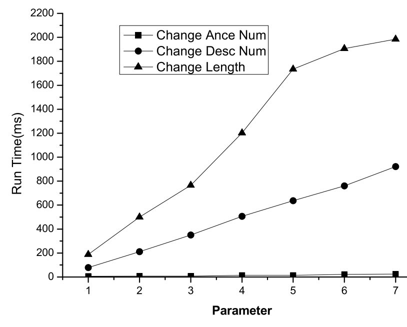
(b) Scalability-XMark2



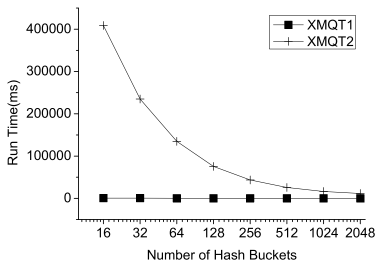
(c) Change DAG Nodes



(d) Change DAG Edges



(e) Change Query Form



(f) Change Bucket Num

Figure 11: Results of Changing Parameters

edge probabilities fixed to 0.1 and node number varying from 4K to 64K, experiments on general graphs with node number fixed to 4096 and edge probabilities varying from 0.1 to 1.0. The run time is small (0ms to 5ms) and does not change significantly. It is because in such cases, almost all the nodes in a graph are in the same strongly connected component. With our data preprocess strategy, their interval sets are the same and contain just one interval. Due to space limitation, we do not show the results.

7.4.2 Changing the Form of Queries

In this section, we test the efficiency change of our method with the forms of queries. All experiments were performed on a random dataset with node number 4096 and edge probability 0.1.

We test the query efficiency with the change of the number of ancestors and descendants in the queries of T-HGJoin and IT-HGJoin from 1 to 7, respectively. The queries for IT-HGJoin algorithm use H as the descendant and $\{A\}$, $\{A, B\}$, $\cdots$, $\{A, \cdots, G\}$ are the ancestor sets, respectively. The queries for T-HGJoin use A as the ancestor and $\{B\}$, $\{B, C\}$, $\cdots$, $\{B, \cdots, H\}$ are the ancestor sets, respectively. The run time axes are both in log scale. We also test the query efficiency with the change of the length of path query from 2 to 8. The queries are $A \to B$, $A \to B \to C$, $\cdots$, $A \to \cdots \to H$. The results of these three experiments are shown in Fig 11(e).

From these results, the run time of our algorithm is nearly linearly to the number of ancestors or descendants. It is because with the hash sets, all ancestors of one descendant can be outputted directly from the hash set without useless comparisons with other nodes.

7.4.3 Changing the Number of Buckets in Hash Table

The number of buckets of the hash table is an important factor of T-HGJoin. We vary bucket numbers from 16 to 2048. The results of XMQT1 and XMQT2 on 50M XMark are shown in Fig 11(f). The number of hash buckets has little effect on the efficiency of XMQT1. It is because nodes corresponding to the four query nodes are all in the tree-structure of an XML graph. The coding of each node has only one interval. So there are only three intervals to process at the same time. But for XMQT2, the run time is nearly logarithmic related to the number of bucket. It is because during the processing of XMQT2, there are many intermediate results in the hash table. More buckets will reduce not only disk I/O but also the cost of sort and join.

8. RELATED WORK

The reachability labelling schemes of a DAG or a graph include [17, 28, 8] and [5]. A survey of labeling schemes on DAG is presented in [21].

A 2-hop reachability label is presented in [28]. In [18], a 2-hop label is used to process the reachability query in complex XML document collections. [5] presents an approximate algorithm for the computation of 2-Hop labelling by finding densest subgraphs. HLSS labelling is presented in [8]. This labelling strategy obtains ($preorder$, $postorder$) for each node and then computes 2-Hop labelling on remaining edges. The labelling scheme presented in [22] obtains ($preorder$, $postorder$) for each node at first and then computes a transmit closure matrix for remaining edges. With preorder and postorder, the size of such matrix can be reduced. The algorithms in this paper are based on an extended version of the labelling scheme in [17]. It is because such scheme avoids costly set comparison and matrix looking up and is suitable for the computation of (ancestors, descendent) pairs from two node sets. Additionally, such labelling scheme is compatible with adjacent labelling scheme so that it is also suitable to process subgraph queries with both adjacent and reachability relationships.

With efficient coding, XML queries can also be evaluated on-the-fly using the join-based approaches. Structural join and twig join are such operators and their efficient evaluation algorithms have been extensively studied [26, 13, 7, 9, 6, 25] [3, 10]. Their basic tool is the labelling schemes that enable efficient checking of structural relationship of any two nodes. TwigStack [3] is the best twig join algorithm to answer all twig queries without using additional index. The idea of these papers can be referenced to process query on graph. But these algorithms cannot be applied on the labelling schemes of a graph directly.

9. CONCLUSIONS AND FURTHER WORK

When XML documents are modeled as graphs, many challenging research issues arise. In this paper, we consider the problem of efficient structural query evaluation which is to match a subgraph in the graph structure of an XML document. Based on a reachability labelling scheme, we present a hash-based structural join algorithm, HGJoin, to handle reachability queries for graph-structured XML data. As the extensions of HGJoin, two algorithms are presented to process reachability queries with multiple ancestors and single descendants or single ancestors and multiple descendants, respectively. As the combination of these two algorithms, the query processing algorithm for subgraph queries in form of bipartite graphs is presented. With these algorithms as basic operators, we present a query processing method for subgraph queries in form of DAGs. In this paper, we also discuss how to extend the method to support subgraph queries in the form of general graphs. Analysis and experiments show that our algorithms outperform the existing algorithm.

Several issues for further exploration and experimentation are raised by this work. First, it would be worthwhile to design efficient index to accelerate the query processing. Second, how to generate more efficient query plans is an interesting problem. The last but not the least, the maintenance of the labelling scheme is another import topic for future research. We plan to investigate these directions in our future work in this area.

10. REFERENCES

[1] *Introduction to Algorithms.* MIT Press, Cambridge MA, 1990.

[2] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. ICDE 2002.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. SIGMOD 2002.

[4] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. VLDB 2005.

[5] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. EDBT 2006.

[6] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. VLDB 2002.

[7] T. Grust. Accelerating XPath location steps. SIGMOD 2002.

[8] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. CIKM2005.

[9] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural join. ICDE 2003.

[10] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed xml documents. VLDB 2003.

[11] T. Kameda. On the vector representation of the reachability in planar directed graphs. *Information Process Letters*, 3(3):78–80, 1975.

[12] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. SIGMOD 2002.

[13] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. VLDB 2001.

[14] T. Milo and D. Suciu. Index structures for path expressions. ICDE 1999.

[15] N. Polyzotis and M. N. Garofalakis. Structure and value synopses for XML data graphs. VLDB 2002.

[16] N. Polyzotis, M. N. Garofalakis, and Y. E. Ioannidis. Selectivity estimation for XML twigs. ICDE 2004.

[17] H. V. J. Rakesh Agrawal, Alexander Borgida. Efficient management of transitive relationships in large data and knowledge bases. SIGMOD 1989.

[18] G. W. Ralf Schenkel, Anja Theobald. Hopi: An efficient connection index for complex xml document collections. EDBT 2004.

[19] P. Rao and B. Moon. Prix: Indexing and querying xml using prüfer sequences. ICDE 2004.

[20] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. VLDB 2002.

[21] M. S. S. T. Vassilis Christophides, Dimitris Plexousakis. On labeling schemes for the semantic web. WWW 2003.

[22] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. ICDE 2006.

[23] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying xml data by tree structures. SIGMOD 2003.

[24] H. Wang, W. Wang, X. Lin, and J. Li. Labeling scheme and structural joins for graph-structured xml data. APWeb 2005.

[25] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. ICDE 2003.

[26] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. SIGMOD 2001.

[27] V. J. T. Zografoula Vagena, Mirella Moura Moro. Twig query processing over graph-structured xml data. WebDB 2004.

[28] H. K. U. Z. Edith Cohen, Eran Halperin. Reachability and distance queries via 2-hop labels. SODA 2002.