Shortest Path Computing in Relational DBMSs

Article *in* IEEE Transactions on Knowledge and Data Engineering · February 2013

DOI: 10.1109/TKDE.2013.43

Jun Gao
Peking University

Shortest Path Computing in Relational DBMSs

Jun Gao, Jiashuai Zhou, Jeffrey Xu Yu, Tengjiao Wang

Abstract—This paper takes the shortest path discovery to study efficient relational approaches to graph search queries. We first abstract three enhanced relational operators, based on which we introduce an *FEM* framework to bridge the gap between relational operations and graph operations. We show new features introduced by recent SQL standards, such as *window function* and *merge statement*, can improve the performance of the *FEM* framework. Second, we propose an edge weight aware graph partitioning schema and design a bi-directional restrictive BFS (breadth-first-search) over partitioned tables, which improves the scalability and performance without extra indexing overheads. The final extensive experimental results illustrate our relational approach with optimization strategies can achieve high scalability and performance.

Index Terms—Relational Database, Graph, Shortest Path

1 Introduction

With the rapid growth of graphs, graph search faces more challenges. Nowadays, graph data emerge in different domains, such as web graphs, social networks, knowledge graphs, etc. Graph search is highly needed in applications over graphs. Specifically, graph search seeks a sub-graph(s) meeting the specific purposes, such as the shortest path between two nodes [13], the minimal spanning tree [23], the salesman traveling path [10], and the like. We also observe that these graphs are always exceedingly large and keep growing at a fast rate. When a graph is too big to be fit into main memory, the existing in-memory approaches [13], [22], [23] to graph search must be re-examined, since the I/O becomes the key factor in the graph operations on the external disk memory.

Existing disk-based methods have limitations to support general graph search queries when graphs exceed the main memory limitation. One straightforward way is to build native graph databases from scratch. Neo4j is a representative one [2], which can store large graphs into the database, and provide primitive operations, such as graph traversal or shortest path discovery to end users. However, the stability, maturity and performance of graph database systems should be continuously improved, as graph database systems have to implement complex components, including storage, index, query evaluation, and query optimization, etc. Some kinds of indices are devised to support specific search queries over large graphs. For example, a shortest path index over planar graphs is designed on the external memory [9]. These methods face difficulty in supporting general search queries. We notice that the MapReduce framework [17] and its open source implementation Hadoop [1] can process large graphs stored in a distribute file system over a cluster of commodity servers. However, the low latency of users' queries is not the main concern in the design of Hadoop [1].

- *Jun Gao, Jiashuai Zhou, Tengjiao Wang are with the Key Laboratory of High Confidence Software Technologies, Ministry of Education & School of EECS, Peking University, China.*
  E-mail: $\{gaojun, zjiash, tjwang\}$@pku.edu.cn
- *Jeffrey Xu Yu is with the Department of System Engineering, Chinese University of Hong Kong, Hong Kong. Email: yu@se.cuhk.edu.hk.*

RDB (Relational Database) provides a promising infrastructure to support graph search. After more than 40 years of development, RDB is mature and stable enough, and plays a key role in information systems. We can see that RDB and the graph data management have many overlapped functionalities, including storage, data buffer, index, and optimizations, etc. In addition, RDB has already shown its flexibility in managing other complex data types, such as XML data [19], [14]. As for graph data, RDB can be used to support specific graph search queries, like reachability query [28] and BFS [27]. The extension of RDB to graph search queries is especially useful in applications when both graph and relational operations are needed.

However, it still requires substantial efforts to support generic graph search queries efficiently in the RDB context. First, graph search queries have various forms. It is not flexible to implement each query separately. We had better find a mechanism to support the evaluation of generic graph search in the relational context. Second, the semantic mismatch between relational operations and graph operations has a significant impact on the performance of graph search on the RDB. Graph operations always follow a node-at-a-time fashion in order to lower the search space. During the search we need to do logic and arithmetic computation, make choices based on aggregated results, and record necessary information [13], [23], [10]. In contrast, relational operations take a set-at-a-time fashion, and only restricted operations, such as projection, selection, join, aggregation, etc, are allowed on the relational tables [15].

This paper focuses on the shortest path discovery for two reasons. First, it plays a key role in many applications. For example, it can reveal how the relationships between two individuals are built in a social network [25]. It also can find the path with the minimal length in a transportation network. Second, it is a representative graph search query, which has a similar evaluation pattern to other search queries, such as the minimal spanning tree construction [23].

We have proposed a relational approach to shortest path discovery in the previous version [18]. In this paper, we make substantial extensions to further improve the scalability and performance of the relational approach. We introduce a weight aware edge table partitioning schema, and design a restrictive BFS strategy over partitioned tables. The strategy can improve both the scalability and performance significantly without needing extra index construction and space overhead.

To sum up, we investigate the techniques for the shortest path discovery used in the RDB context. We expect our method can not only deliver benefits to the shortest path computation, but also shed lights on the efficient SQL implementation for other graph search queries. The contributions of this paper can be summarized as follows:

- We abstract three key operators, namely F, E and M-operator, and then provide a generic graph searching framework *FEM*. We find new features, such as *window function* and *merge statement* introduced by recent SQL standards, can simplify the expression and improve the performance. We also show the basic method to discover the shortest path using the *FEM* framework. (Section 3)
- We propose a novel weight aware edge table partitioning schema and a bi-directional restrictive BFS over the partitioned tables. Specifically, we define an extended E-operator using basic F and E-operator, which expands paths by scanning *partial* partitioned edge tables. We also show the issue of incomplete search caused by the restricted BFS and then give its solution. We will see that the restricted BFS over the partitioned tables can yield correct results with a high performance, scalability and no extra space cost. (Section 4)
- We conduct extensive experiments over synthetic and real-life data to illustrate the effectiveness and efficiency of our *FEM* framework along with its optimizations. The results also show that our relational approach has overwhelming advantages to the existing native graph database in handling shortest path discovery. (Section 5)

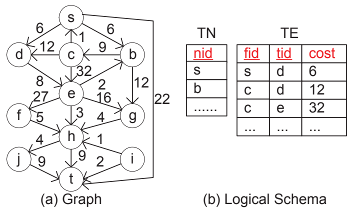
The remainder of this paper is organized as follows: Section 2 presents the preliminary knowledge. In Section 3, we abstract three operators and propose an *FEM* framework. Section 4 focuses on the restrictive BFS over the partitioned tables. Section 5 reports experimental results on real and synthetic data. Related work is reviewed in Section 6. Section 7 concludes the paper.

2 PRELIMINARY

In this section, we first give notations and relational schema of graphs, and then show new SQL features used in our path finding method.

2.1 Graph Notations and Logic Relational Schema

This paper studies weighted (directed or undirected) graphs. Let $G = (V, E)$ be a graph, where V is a node set and E is an edge set. Each node $v \in V$ has a unique node identifer. Each edge $e \in E$ is represented by $e = (u, v)$, $u, v \in V$. $w(e)$ is the weight for the edge e. A path $p = u_0 \rightsquigarrow u_x$ from u_0 to u_x is a sequence of edges $(u_0, u_1), (u_1, u_2), \dots, (u_{x-1}, u_x)$, where $e_i = (u_i, u_{i+1}) \in E$ $(0 \leq i < x)$. The length of a path p, $len(p)$, is the sum of the weights of its constituent edges. The shortest distance from u to v is denoted by $\delta(u, v)$.



TN

nid
s
b
......

TE

fid	tid	cost
s	d	6
c	d	12
c	e	32
...	...	...

Fig. 1. Relational Representation of Graph

The logical relational schema for a graph is illustrated in Figure 1. Let $G = (V, E)$ be a graph. *TN* table is to represent nodes V. We can record *nid* for the node's identifer and other attributes in *TN* table. *TE* table is to store edges E. For an edge $(u, v) \in E$, the identifiers of node u and v, as well as the weight of the edge, are recorded by *fid*, *tid* and *cost* field respectively. As this paper studies the shortest path discovery, only *TE* table is used in the following operations.

2.2 SQL Features

In this paper, in addition to utilizing standard features in SQL statements, we leverage two new SQL features, window function and merge statement, which are now supported by commercial database systems, like Oracle, DB2, and SQL server. These features are the "short-cut" for combinations of some basic relational operators. More importantly, compared with the general-purpose statements implemented by the traditional features, the statements with these new features are more specific, which then provide chances for them to gain a higher performance.

Listing 1. Syntax for New Features

```
1:Window function
<WindowFunction> :: = <Aggregation>
  Over ( [ Partition By <expr>, ... ]
         [ Order By <expression> ] )
2: Merge statement
Merge Into tablename Using table On (condition)
When Matched Then
 Update Set col1 = val1 [, col2 = val2 ...]
When Not Matched Then
 Insert (col1[,col2 ...]) Values (val1[, val2 ...])
```

The window function [1], introduced by SQL 2003, returns aggregate results for *each* tuple in a set, in contrast to traditional aggregation functions which obtain aggregate results for a set. In addition, non-aggregate attributes are allowed by the window function to be along with the aggregate ones in the *select* clause, even if these attributes are not in the *group-by* clause. Window function also supports aggregate functions such as *rank*, *row_num*, which are related to the tuple position in a sorted tuple sequences specified by an *order-by* clause. The syntax for window function is described in Listing 1(1).

The merge statement [2] adds new tuples and updates the existing ones in a target table from a source table (or SQL results, or a view). It is officially introduced by SQL 2008, but is supported earlier by different database vendors due to the need in loading data into data warehouse. The main

1. http://en.wikipedia.org/wiki/SQL:2003
2. http://en.wikipedia.org/wiki/Merge_(SQL)

part of merge statement specifies actions (*e.g.,insert,delete* and *update*) under different relationships between the source and target. The *update* action is invoked when the tuples in the source match the tuples in the target, and the *insert* action is performed when there are no matched tuples in the target for the tuples in the source. A merge statement can be equivalently rewritten by one *update* statement followed by an *insert* statement. The redundant table scan cost in two separate statements can be avoided by one merge statement. The syntax for merge statement is referred in Listing 1(2).

3 Relational *FEM* Framework for Shortest Path Discovery

In this section, we introduce a generic graph search framework *FEM*, and leverage it to realize the classic Dijkstra's shortest path discovery algorithm on RDB.

3.1 A Generic Framework for Graph Search

Graph search queries seek the sub-graph(s) meeting the specific requirements. For example, reachability query answers whether there exists a path between two given nodes [12]. The shortest path query locates the shortest path between two given nodes [22], [3]. The minimal spanning tree query returns a tree covering all nodes with the minimal sum of edge weights [23]. The salesman path query gets a shortest tour that visits each city exactly once [10].

Many graph search algorithms show a common pattern. Due to a large search space, they always utilize greedy ideas. We observe that most of these greedy algorithms can fit into a generic iterative processing structure. A visited node set A_1 is initialized first according to different purposes. Then an iterative searching starts. We illustrate the i^{th} iteration in Figure 2. Let the **visited nodes** A_i record all nodes encountered in the graph search so far; the **frontier nodes** F_i are selected from the visited nodes with the certain criteria (application dependent, $F_i \subseteq A_i$); the **expanded nodes** E_i are the next visited nodes from the frontier nodes (generally neighboring subset of F_i); and the next visited ones A_{i+1} are obtained by merging the newly expanded nodes E_i and A_i. The iterations continue until the target sub-graph(s) can be discovered. We refer to such a generic processing structure as an *FEM* framework.

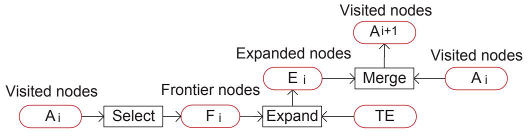


Fig. 2. Conversion Between Nodes in the i^{th} Iteration

Before we proceed to the details of implementing relational operators for the *FEM* framework, we first exploit it to briefly describe two representative graph search algorithms: *Dijkstra's shortest path algorithm* and *Prim's minimal spanning tree algorithm*. In Dijkstra's algorithm for shortest path discovery [13] using the *FEM* framework, each node is annotated with $d2s$ to record the distance from the source node s, and a flag f indicating whether the node is finalized or not. We initially add the source node s into a visited node set with $s.d2s = 0$ and $s.f = false$. We then start an iterative path expansion for the target node. In each iteration, we *select* a non-finalized node u ($u.f = false$) from all visited nodes with the minimal $d2s$ as a frontier node, finalize u, *expand* u (visiting its neighbors), and *merge* the newly expanded nodes into the visited nodes.

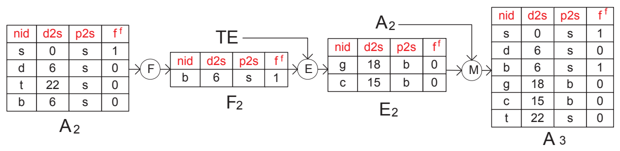
Another case is to construct a minimal spanning tree T by Prim's algorithm [23]. Each node u is represented by $u(p2s, w, f)$. Here $p2s$ is the parent of u, w is the edge weight from $p2s$ to u, if u is in T. f is a flag for whether u is in T. The visited nodes are initialized by any node u with $u.f = false$ and $u.w = 0$. In each iteration in building T, we *select* a node u with $u.f = false$ and the minimal edge weight, add u into T by changing $u.f = true$, make further *expansion* from u, and *merge* the expanded nodes into the visited ones. In the merge operation, the expanded nodes can be discarded directly if they have been included ($f = true$) in T. The iterations repeat until all nodes have been in T.

Note that there are other operations besides the three basic operations (select, expand, and merge) in the graph search. For example, the recovery of the shortest path or the minimal spanning tree, and the termination detection, are also needed in the *FEM* framework. However, these operations are generally auxiliary under the *FEM* framework and their computational costs are quite minimal compared with the three main operations. The exploration on the shortest path search in the remainder of the paper will illustrate the details of these additional operations, and also demonstrate the key functionality of the three operations.

3.2 *FEM* Operators for Shortest Path Discovery

To realize a graph search query using the *FEM* framework on RDB, we need three operators which can be expressed by relational algebra: i) F-operator to *select* frontier nodes from the visited nodes; ii) E-operator to *expand* from the frontier nodes; and iii) M-operator to *merge* the newly expanded nodes into the visited ones. As discussed above, attributes on nodes and operations may be various for different graph search queries. In the remainder of the paper, we will focus on efficiently realizing the shortest path discovery in the *FEM* framework. The new techniques we developed for shortest paths in the *FEM* framework can be in general applied or extended to deal with other graph search queries.

Now, we start with Dijkstra's algorithm for the shortest path discovery as an example to describe its F, E and M-operator. Each node u in the visit nodes A_i, the frontier nodes F_i and the expanded nodes E_i can be represented by $(nid, d2s, p2s, f^f)$, where i is the number of iterations, nid is for $u's$ identifier, $d2s$ is for the distance from the source node to u, $p2s$ is for the predecessor node of u to the source node, and f^f is for the sign indicating whether u is finalized or not. Take the shortest path discovery from s to t in the graph in Figure 1 as an example. We add s to A_1 first. d, t, b will be added into A_2 next. Figure 3 shows the visited nodes, frontier nodes, and expanded nodes in the 2-nd iteration of the shortest path discovery.



nid	d2s	p2s	f^f
s	0	s	1
d	6	s	0
t	22	s	0
b	6	s	0

nid	d2s	p2s	f^f
b	6	s	1

nid	d2s	p2s	f^f
g	18	b	0
c	15	b	0

nid	d2s	p2s	f^f
s	0	s	1
d	6	s	0
b	6	s	1
g	18	b	0
c	15	b	0
t	22	s	0

Fig. 3. F, E and M-operator in the 2-nd Iteration

Below we formally describe these operators based on the relational algebra.

Definition 1: F**-operator:** $F(A_i) \leftarrow \sigma_{nid=mid} A_i$ returns frontier nodes F_i from visited nodes A_i in the i^{th} expansion.

In Dijkstra's algorithm, we select the node with the minimal $d2s$ among all non-finalized nodes, and assign its identifier to mid. Take Figure 3 as an example, node b is selected with the minimal $d2s = 6$, and we assign the identifer of b to mid. The computation of mid can be done by an auxiliary operation before F-operator. In order to enhance the utility of our framework, we assume that there may exist multiple frontier nodes after F-operator. For example, an optimization strategy used in the next section may produce multiple frontier nodes with a revised predicate in F-operator. After F-operator, we use another auxiliary operation to adjust $f^f = 1$ for the node identified by mid.

Definition 2: E**-operator:** $(F_i)E(\mathit{TE})$ returns the expanded nodes E_i based on the frontier nodes $F_i(r, d2s, p2s, f^f)$ and $\mathit{TE}(r, x, w)$ table in the i^{th} expansion.

$$(1)\quad minCost(x, c) \leftarrow_x \mathcal{G}_{min(d2s+w)} \Pi_{(x,d2s,w)}(F_i(r, d2s, p2s, f^f) \bowtie TE(r, x, w));$$

$$(2)\quad E_i \leftarrow \Pi_{(x,d2s+w,r,0)} \sigma_{c=d2s+w} minCost(x, c) \bowtie F_i(r, d2s, p2s, f^f) \bowtie TE(r, x, w);$$

(r, x, w) in *TE* table is for an edge from r to x with its weight w .

In E-operator, $minCost(x, c)$ preserves the expanded nodes with the minimal distances. Since the expanded nodes may be reached by different paths, and only the ones with the minimal distance ($d2s + w$) are needed, we use an aggregate function to find them. Note that $minCost(x, c)$ contains the newly expanded nodes with their costs, but lacks their parents $p2s$, which are required in the recovery of the full path. However, we cannot simply put the non-aggregate attribute $p2s$ into the *select* clause in $minCost(x, c)$, due to the constraint on aggregation functions in relational algebra. Thus, another join operation is needed to find $p2s$ from *TE*. Take Figure 3 as an example, we make an expansion from the frontier nodes $\{b\}$, and get the newly expanded nodes g and c.

Definition 3: M**-operator:** $(E_i)M(A_i)$ returns visited nodes A_{i+1} based on expanded nodes $E_i(x, d2s_e, p2s_e, f^f_e)$ and visited nodes $A_i(x, d2s_a, p2s_a, f^f_a)$ as follows:

$$(1)\quad A_i \leftarrow A_i - \Pi_{x,d2s_a,p2s_a,f^f_a}(\sigma_{d2s_e<d2s_a} (E_i(x, d2s_e, p2s_e, f^f_e) \bowtie A_i(x, d2s_a, p2s_a, f^f_a));$$

$$(2)\quad E_i \leftarrow E_i - \Pi_{x,d2s_e,p2s_e,f^f_e}(\sigma_{d2s_e>d2s_a} (E_i(x, d2s_e, p2s_e, f^f_e) \bowtie A_i(x, d2s_a, p2s_a, f^f_a));$$

$$(3)\quad A_{i+1} \leftarrow A_i \cup E_i;$$

In M-operator, we remove the visited nodes from A_i, whose distances are larger than those of the corresponding newly expanded nodes in E_i. Then we add partial newly expanded nodes into the visited node set. The newly added nodes in A_{i+1} actually have two parts, the nodes new to A_i and the nodes with smaller distances than their counterparts in A_i.

3.3 SQL Implementation for Operators

Now, we discuss the SQL implementation for these operators. We use a table *TA* to store all visited nodes. The attributes nid, $d2s$, $p2s$, f^f in *TA* have the same meanings as those in Section 3.2. Since F, E and M-operator are based on relational algebra, we can use SQL to express them. However, the direct translation will result in a poor performance, especially for E and M-operator. For example, E-operator is implemented by an aggregate function over the join results with *group by* clause. In addition, the location of the parent node $p2s$ of each expanded node still needs another join operation. Moreover, when there are multiple paths with the same minimal $d2s$ to an expanded node, we have to keep only one path due to the primary key constraint on nid in *TA* table. As for M-operator, we have two different actions according to whether the expanded nodes have been in the visited nodes or not. In the SQL implementation, we might use an update statement followed by an insert statement with a *not exists* sub-query. Such a kind of expression is not only verbose but also inefficient.

Listing 2. SQL in Path Finding
```
1://Initialize TA with source node
Insert into TA(nid,d2s, p2s, f^f) values(s, 0, s, 0);
2://Locate the next frontier node
Select top 1 nid from TA where f^f=0
 and d2s=(select min(d2s) from TA where f^f=0);
3://E—operator in the i^th forward expansion
Create view er as
 select nid, p2s, cost
   from ( select TE.tid, TE.fid,
            TE.cost+TA.d2s,  row_number() over
               (partition by TE.tid order by
                  TE.cost+TA.d2s asc) as rownum
            from TA, TE
            where TA.nid=TE.fid and TA.nid= mid)
         tmp (nid, p2s, cost,rownum)
   where rownum=1
4://M—operator in the i^th forward expansion
Merge TA as target using er as source
 on source.nid=target.nid
when matched and target.d2s>source.cost then
 update set d2s=source.cost, p2s=source.p2s, f^f=0
when not matched by target then
 insert (nid, d2s, p2s, f^f)
  values(source.nid, source.cost, source.p2s, 0);
```

Fortunately, we find new features including window function and merge statement introduced by recent SQL standards can simplify the expression as well as improve the performance. As for our case, we can partition all occurrences of expanded nodes by the identifer nid, sort them with their $d2s$ and select the tuple with the minimal $d2s$ by using an aggregate function $row\_number() = 1$ in each partition. We see that the window function can avoid another extra join operation to locate the parent node $p2s$ of a currently expanded node. It can also handle the case when the node are reached by multiple paths with the same minimal distance. As for the M-operator, we can use one merge statement to combine two separated insert and update statements.

The SQL statement used in the path expansion is illustrated in Listing 2. The 1-st statement adds the source node s into *TA* nodes initially, where f^f is set to 0 (non-finalized). The

E-operator can be expressed by the 3-rd statement. It joins the frontier nodes specified by $nid = mid$ (F-operator) and *TE* table, where mid is discovered by the 2-nd statement. The tuples with the minimal $d2s + c$ among all occurrences for the same node can be located by the function row_number over the sorted tuples. In the 4-th SQL statement for M-operator, we use one statement to merge expanded nodes into *TA* table. When the newly expanded nodes are new to existing visited nodes in *TA*, we directly add them into *TA*. When the newly expanded nodes have smaller distances from the source node, the nodes in the *TA* are replaced by the newly expanded ones.

3.4 Shortest Path Discovery in *FEM* Framework

Here, we present Dijkstra's algorithm for shortest path discovery in Algorithm 1 as a case to show the functionality of the *FEM* framework. Besides the key SQLs shown above, we also need auxiliary SQLs in Listing 3. The method can run on the client side, which connects to the underlying RDB via JDBC or ODBC. In the running time, only few variables are kept on the client side, and the RDB carries out time-consuming tasks.

Algorithm 1: Shortest Path Discovery in *FEM* Framework

```
  Input: Source node $s$, target node $t$, graph $G = (V, E)$.
  Output: The shortest path between $s$ and $t$.
1 Initialize TA using the SQL in Listing 2(1);
2 while true do
3     Locate $mid$, the ID for the next frontier node $u$,
      using the SQL in Listing 2(2);
4     Expand paths using the SQLs in Listing 2(3,4) with
      $mid$;
5     if the number of affected tuples is 0 then
6         Break;
7     Finalize the node $u$ using the SQL in Listing 3(2)
      with $mid$;
8     if there exist results for the SQL in Listing 3(1) then
9         Break;
10 Find edges in the shortest path $p$ along $p2s$ link using
   the SQL in Listing 3(3) iteratively;
11 Return $p$;
```

We initialize *TA* table first with the source node. We then start iterations to find the shortest path from line 2 to line 9. We issue an SQL on *TA* table to locate mid, the ID for the next frontier node in line 3, and use mid to compose SQLs for F, E and M-operators in path expansion in line 4. After that, we extract the number of affected tuples from SQL communication area of database (SQLCA) in line 5, and terminate iterations when *TA* is not updated. Otherwise, we finalize the node with its identifier mid. We then detect whether the target node has been finalized or not. Once the iterations are terminated, we recover the full path from the source node to the target node using SQLs along the $p2s$ link iteratively.

There are at most n iterations for the path finding in Algorithm 1 in the worst case, where n is the number of nodes in the graph. In each iteration, we have 4 separate SQLs. Thus, we at most issue $4n$ SQLs in the shortest path discovery.

Listing 3. Auxiliary SQLs in Path Finding

```
1://Detect termination
 Select * from TA where f^f=1 and nid=t;
2://Finialize the frontier node
 Update TA set f^f=1 where nid=mid
3://Locate the predecessor node
 Select p2s into u from TA where nid=u;
```

4 Bi-directional Restrictive BFS over Partitioned Tables

In this section, we first show the basic idea behind the optimizations in the relational context. Then, we discuss bi-directional search, propose a weight aware edge table partitioning schema, revise F, E, and M-operator, and introduce new termination actions in the restrictive BFS search. Finally, we present the complete algorithm on bi-directional restrictive BFS over partitioned tables.

4.1 Basic Idea of Optimizations in Relational Context

An important aspect of relational approaches lies in its evaluation fashion: in the RDB context, the set-at-a-time evaluation is more suitable than the node-at-a-time fashion for the same search space, since the former can enable RDB to fully adopt the intelligent scheduling to access disk and exploit the data loaded in the buffer, and thus to make a better evaluation plan [15], [20]. For example, let us consider the E-operator (the 3-rd SQL in Listing 2). As we expand from one node in an E-operator in Algorithm 1 each time, we have to issue n SQL statements for loading the edges of n nodes separately in the worst case. Such a node-at-a-time operation is very inefficient due to the redundant I/O cost for accessing edges of different nodes when they are stored in the same data block in RDB.

From another viewpoint, the performance is not satisfactory either if we solely focus on the set-at-a-time evaluation fashion. An extreme case is BFS strategy. BFS typically follows the set-at-a-time evaluation fashion, since it views all newly visited nodes as frontier nodes, and makes expansions from these nodes along all their adjacent edges in one operation. However, BFS may re-expand the nodes which have been expanded before, and incurs larger search space (or more visited nodes).

We have to make a balance between selecting all frontier nodes (BFS) and selecting one frontier node in one expansion (the classic Dijkstra's algorithm) in the relational context. In this paper, we take a strategy of the restrictive BFS. That is, we allow multiple nodes be frontier nodes, and search from these frontier nodes along their partial incident edges first, specifically, edges with smaller weights. In such a way, the expansion can be done by a set-at-a-time fashion, and the visited nodes have fewer chances to be re-expanded compared with these in BFS. Most importantly, the enormous cost in scaning edge table can be reduced in the restrictive BFS, if we partition the edge table according to edge weights.

In addition, the strategies which lower the search space while still taking the set-at-a-time evaluation fashion are highly preferable in the relational context. One of such strategies is the bi-directional search [11], which makes both expansions from the source node and the target node, and locates paths when the forward expansion meets the backward expansion. The bi-directional search not only lowers the search space, but also can be incorporated into the restrictive BFS easily. In the relational context, the bi-directional search can reduce the total computational cost contributed by F, E, and M-operator, as fewer nodes need to be visited.

4.2 Bi-directional Search

The introduction of the bi-directional search into the relational context makes changes to the basic path discovery method. First, as the forward expansion and the backward expansion are allowed in the bi-directional search, we use two tables *TA*^{f} and *TA*^{b} to record visited nodes in both directions respectively.

Second, the expansion direction needs to be selected in each iteration in the bi-directional search. We choose the direction with fewer frontier nodes in order to reduce the search space. The number of frontier nodes after each iteration, n^f, can be computed by issuing an extra SQL on the forward visited node table *TA*^{f}. Another way is to extract the number of affected tuples from SQLCA after the SQL statement for M-operator is evaluated, and use it as n^f.

Third, the bi-directional search can prune the search space with discovered paths, especially when multiple frontier nodes are allowed. Specifically, we locate the maximal finalized distance l_i^f in the latest $i^{th}(i \geq 1)$ forward expansion from *TA*^{f}. In the i^{th} expansion, l_i^f is the distance to the source node of the frontier node for the classic Dijkstra's algorithm, or the minimal distance to the source node of all newly expanded nodes for the BFS search. Similarly, we compute the maximal finalized distance l_j^b in the latest j^{th} backward expansion. We also compute the minimal distance $minCost$ between the source node and the target node seen so far from a join result between *TA*^{f} and *TA*^{b}. We can incorporate these collected data into the E-operator to avoid unnecessary expansions. Take the forward expansion as an example. The nodes with their cost larger than $minCost - l_j^b$ do not need to be visited. Specifically, the rule can be described as follows:

Theorem 1: Let l_i^f, l_j^b and $minCost$ have the same meanings above, v be a frontier node in the forward expansion. We do not need to expand from v to its neighbor x, if $v.d2s + w(v,x) + l_j^b \geq minCost$.

Proof: We prove it by contradiction. Suppose that there exists a path $p' = s \rightsquigarrow t$ with $len(p') < minCost$ and p' has the prefix sub-path from s to x via v. Since $v.d2s + w(v,x) + l_j^b \geq minCost$, the distance from x to the target node t is less than l_j^b in p' to achieve $len(p') < minCost$. In such a case, x has already been finalized in the backward expansion since its distance to t is less than l_j^b, which indicates that the path p' with $len(p') < minCost$ has been already discovered. This contradicts that $minCost$ is the minimal distance discovered yet. Thus, x can be ignored safely in the bi-directional searching. ■

The existing studies give the termination condition in bi-directional shortest path discovery [11]. We can stop iterations when $minCost \leq l_i^f + l_j^b$. In such a case, $minCost$ is the shortest distance between the source node and the target node.

4.3 Weight Aware Edge Table Partitioning Schema

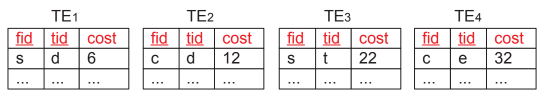
Table partitioning strategy is highly needed in handling large graphs. Let n be the number of nodes in a graph. The number of edges is n^2 in the worst case. We can imagine how large the edge table is for a real life graph. On such a large table, even primitive graph operations, such as the location of adjacent edges for a specified node, are very costly. The table partitioning strategy is then a key to the scalability for graph queries in RDB.

A straightforward schema is to partition the edge table with node IDs. Given pts partitioned tables, we randomly put an edge $e = (u, v)$ into a partitioned table via a hash function on $u.id$. However, such a schema brings no substantial improvement in addressing our problem. For example, in E-operator, we still have to union all partitioned tables before joining with the frontier nodes, as the edges are distributed randomly in partitioned tables.

In this paper, we propose a weight aware edge table partitioning schema. Instead of using meaningless node IDs, we partition the edge table according to edge weights. Intuitively, the cost-related graph operations, such as the minimal spanning tree, shortest path discovery, etc, prefer accessing the edges with smaller weights. Thus, we may have chances to implement these operations on a limited number of partitioned tables.

Definition 4: **Weight Aware Edge Table Partitioning Schema.** Let pts be the number of partitioned tables, $[w_{min}, w_{max}]$ be the range of edge weights. $[w_0, \ldots, w_{pts}]$ be a partitioning vector, where $w_0 = w_{min}$, $w_{pts} = w_{max} + 1$, and $w_{i+1} > w_i$ for $0 \leq i < pts$. For any edge e in a graph, e is put into the partitioned table *TE*$_i$, where $w_{i-1} \leq w(e) < w_i$. i is the index of the partitioned table for e.

The partitioning vector is a flexible mechanism to support both equal-width and equal-depth schema. With the equal-width partitioning, the differences, $w_{i+1} - w_i$, are almost equal for $0 \leq i < pts$ in the partitioning vector. When the equal-depth partitioning schema is used, the total number of edges in each partitioned table is nearly even. We show an equal-width table partitioning schema in Figure 4. Suppose the range of the edge weights is $[1, 40]$ and the number of partitioned tables, pts, is 4. The partitioning vector will be $[1, 11, 21, 31, 41]$. The table *TE*$_1$ preserves the edges with their weights from 1 to 10. And the edge from s to t with its weight 22 will be stored into the partitioned table *TE*$_3$.


TE1

fid	tid	cost
s	d	6
...	...	...

TE2

fid	tid	cost
c	d	12
...	...	...

TE3

fid	tid	cost
s	t	22
...	...	...

TE4

fid	tid	cost
c	e	32
...	...	...

Fig. 4. Equal-Width Weight Aware Partitioning Schema

4.4 Revised F, E, M-Operator in Restrictive BFS

Now, we start our restrictive BFS over the partitioned tables. Roughly speaking, all newly visited nodes are viewed as the frontier nodes. For each frontier node, we make expansions along its incident edges with smaller weights first, and delay expansions along the edges with larger weights. The location of specific edges can be accelerated by the weight aware edge table partitioning schema. In fact, the partitioning vector acts as an index, which can help to locate the edges with specified weights in the restrictive BFS.

We need to extend the visit node tables for recording sufficient information used in selecting specific partitioned tables, in order to support the bi-directional restricted BFS on partitioned tables. Take the forward expansion as an example. Each node u in the visited table TA^f takes the form of $(u.id, u.p2s, u.d2s, u.fwd)$. Here, $u.id$ is for the identifer of node u, $u.p2s$ is for the identifier of the parent node of u in the path, $u.d2s$ is for the distance between the source node and u. $u.fwd$ is set to 1 when u is the source node. After the i^{th} expansion, all newly visited nodes or nodes with reduced $d2s$ are with $fwd \leftarrow i+1$. The attribute fwd can be used as a criteria in the following operators on partitioned tables.

The basic F, E, and M-operator have to be revised to support the restrictive BFS. We first give an extended E-operator by combining the basic F and E-operator:

Definition 5: **Extended E-operator**. Let i be the number of the expansions, l be $\max(1, i-pts+1)$, $TE_1, \ldots, TE_{pts}$ be all partitioned tables and TE be the union of all partitioned tables. The extended E-operator between TA^f and TE can be defined as $\cup_{l \le k \le i} (\sigma_{fwd=k} TA^f)\ E(TE_{i-k+1})$, where E is the basic E-operator in Definition 2.

An extended E-operator unions the results of multiple basic E-operators. We give an intuitive example in Figure.5(a). Suppose we want to find the shortest path from node b to node t in Figure.1(a) on the partitioned tables in Figure.4. Initially, b is added into TA^f with $fwd \leftarrow 1$. In the 1-st path expansion, we select b $(fwd = 1)$ as the frontier nodes and search along the edge (b, c) in TE_1, but delay the expansion along the edge (b, g) in TE_2. In the 2-nd expansion, we search from c $(fwd = 2)$ and expand along the edges in TE_1. Meanwhile, the previous delayed expansion from b $(fwd = 1)$ along the edge (b, g) in TE_2 is conducted at this time.

The composition of E-operators in an extended E-operator is illustrated in Figure.5(b). Let pts be the number of partitioned tables, i be the current number of expansions. A node u is a frontier node when $u.fwd + pts - 1 \le i$, which indicates u may have to-be-searched edges. We divide all frontier nodes into different groups according to their fwd, and make joins with the corresponding partitioned tables. In the 1-st path expansion, we make one basic E-operator with TE_1, which contains the edges with the smallest weights. In the 2-nd path expansion, we make two basic E-operators. Similarly, we make i basic E-operators in the i^{th} expansion, until i reaches pts. After that, each extended E-operator includes pts basic E-operators between partial frontier nodes and partial edges.

We can see the advantages of the extended E-operator. First, the extended E-operator expands from multiple nodes in one operation. Thus, it follows the set-at-a-time evaluation fashion. Second, the extended E-operator incurs smaller search space than BFS. Instead of searching along all incident edges of frontier nodes, the extended E-operator searches along the edges with smaller weights in specific partitioned tables earlier, which can gain fewer chances in node re-expansions. Third, the extended E-operator can be implemented efficiently, as it requires join operations between *partial* visited nodes and *partial* edges. Finally, the extended E-operator is buffer-friendly. The edges with smaller weights have more chances in the buffer since they are accessed more frequently.

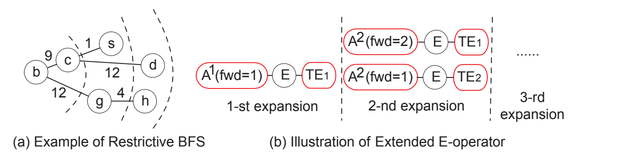


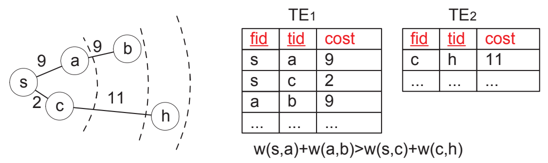
Fig. 5. Extended E-operator over the Partitioned Tables

The M-operator in the restrictive BFS is similar to the basic one, except that we may adjust the sign of fwd on nodes. That is, for the newly visited nodes and the nodes with reduced $d2s$ in the latest i^{th} expansion, their fwd will be set to $i+1$. In this way, fwd can be used to guide the selection of partitioned tables in the following expansions.

4.5 Termination Detection and Verification in Restrictive BFS

Although the extended E-operator can reduce overheads in the path search significantly, the delayed expansions may result in incomplete search in the shortest path discovery, if we still use the original termination condition $l_i^f + l_j^b \ge minCost$ (Section 4.2) in the restrictive BFS. In fact, the delayed expansions make an impact on the original computation rule for the maximal finalized distance(l_i^f and l_j^b). In addition, the candidate shortest distance $minCost$ might be not the shortest one, even when $l_i^f + l_j^b \ge minCost$ is satisfied. In the following, we point out the issues of completeness of search space, followed by their solutions.

First, we show the issues in the maximal finalized distance computing in the restrictive BFS. The maximal finalized distance is a key measure in the bi-directional search, which can be used in the bi-directional pruning rules and the termination condition. In BFS, the maximal finalized distance equals the minimal distance discovered in the latest expansion. However, the rule is no longer valid in the restrictive BFS, as illustrated in Figure.6. The minimal distance is 2 in the 1-st expansion. The minimal distance is 18 in the 2-nd expansion. Notice that the searching along the edge (c, h) is delayed. The minimal distance is 13 in the 3-rd expansion. We can observe that the minimal distance discovered cannot be used as the maximal distance finalized in the i^{th} expansion now, since the delayed expansion may produce a smaller one.



TE1

fid	tid	cost
s	a	9
s	c	2
a	b	9
...	...	...

TE2

fid	tid	cost
c	h	11
...	...	...

Fig. 6. Issue in Maximal Finalized Distance Computing

In order to address such an issue, we revise the computation rule for the maximal finalized distance in one expansion. Let

l_i^f be the maximal distance finalized after the i^{th} expansion, $min(d2s)_i$ be the minimal distance $d2s$ in *TA*^{f} with their $fwd = i$, $[w_0, \ldots, w_n]$ be the partitioning vector for the edge table. l_i^f can be computed recursively:

$$l_i^f = \begin{cases} min(min(d2s)_1, w_1) & \text{if } i = 1 \\ min(l_{i-1}^f + w_i - w_{i-1}, min(d2s)_i) & i \geq 2 \end{cases} \tag{1}$$

Theorem 2: The distance computed by Formula 1 is maximal finalized.

Proof: We show that Formula 1 is valid by induction. For the base case, let u be the visited node with the minimal $d2s$ in the 1-st expansion. If u exists, $u.d2s$ is the maximal finalized distance since the delayed expansions along edges with larger weights cannot lower $u.d2s$. If no node is visited, the delayed expansion cannot find a node with its $d2s$ less than w_1. Thus, the formula is correct for $i = 1$.

We assume the induction hypothesis that l_i^f is correct. We make the $(i+1)^{th}$ restrictive BFS expansion. The maximal finalized distance l_{i+1}^f comes from the $(i+1)^{th}$ expansion or the delayed expansion. In the first case, $min(d2s)_{i+1}$ is the minimal distance for all newly visited nodes in the $(i+1)^{th}$ expansion. In the second case, the delayed expansion cannot visit nodes with distances less than $l_i^f + w_{i+1} - w_i$, according to the rule of the extended E-operator and the properties of edge-weight aware partitioned tables. Thus, the formula is correct for $i \geq 2$. ■

Next, we illustrate an example in Figure.7 to show that the candidate shortest distance $minCost$ might be not shortest when the original termination condition $minCost \leq l_i^f + l_j^b$ is met. Still take the partitioned tables in Figure.4 as an example. After we make two forward expansions and two backward expansions, we get the maximal finalized distance $l_1^f = 6$ and $l_2^f = 14$ in the forward expansion, according to Formula 1. Similarly, we get the maximal finalized distance $l_1^b = 2$ and $l_2^b = 12$ in the backward expansion. We further compute the current minimal distance $minCost = 26$ from a path via d, e, and h. At that time, $l_2^f + l_2^b = minCost$, and then the iterations are terminated. However, 26 is not the shortest distance, since there exists a direct edge from s to t with its weight 22, whose expansion is delayed.

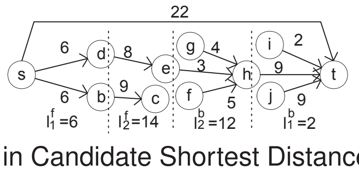


Fig. 7. Issue in Candidate Shortest Distance Computing

The above example shows that $minCost$ is larger than usual due to the delayed expansion. We cannot simply consider all edges in each expansion to address the issue, since it will retreat to the classic BFS method and totally lose the advantages of the restrictive BFS and partitioning schema. We address such a issue by launching a one-time verification after $l_i^f + l_j^b \geq minCost$ is satisfied. The verification is described as follows:

Definition 6: **Verification**. Let *TA*^{f} (*TA*^{b}) be all visited nodes in the forward (backward) expansion. *TE* be the union of all partitioned edge tables. The verification phase finds the minimal $u.d2s + w(u,v) + v.d2t$, where u is in *TA*^{f}, v is in *TA*^{b}, $w(u,v)$ is the weight of an edge $(u,v) \in$ *TE*.

Let $minCost_0$ be the minimal distance discovered in the path expansions when $l_i^f + l_j^b \geq minCost_0$ is satisfied, $minCost_1$ be the minimal distance discovered in the verification. The shortest distance $\delta(s,t)$ can be computed by Formula 2.

$$\delta(s,t) = min(minCost_0, minCost_1) \tag{2}$$

Theorem 3: The shortest distance can be computed by Formula 2 correctly.

Proof: We first show that all nodes in the shortest path have been finalized in forward or backward expansions, when $l_i^f + l_j^b \geq minCost_0$ is achieved in the restrictive BFS on the partitioned tables. We prove it by contradiction. Let x be a node in the shortest path p. Suppose that x is neither finalized in the forward expansion nor in the backward expansion. The distance from u to x is then more than l_i^f, and the distance from x to v is more than l_j^b. Thus, the length of p, $len(p)$, is larger than $l_i^b + l_j^f$, and then $len(p) > minCost_0$, which contradicts that p is shortest.

Next, let p be the shortest path from the source node s to the target node t. Since all nodes in p have been finalized in the forward or backward expansion, it is easy to know that p can be divided into at most three sub-parts, a sub-path p^f from s to u, an edge (u,v), and a sub-path p^b between v and t, where all nodes in p^f are finalized in the forward expansion, and all nodes in p^b are finalized in the backward expansion. $minCost_0$ is the shortest distance if there is no such an edge e or e has already been searched in the iterative path expansions. Otherwise, the shortest distance can be discovered as $minCost_1$ during the verification. ■

4.6 Complete Algorithm of Bi-Directional Restrictive BFS on Partitioned Tables

In this part, we show the complete SQL statement and algorithm for the bi-directional restrictive BFS on partitioned tables in Algorithm 2.

We insert the source node and target node into an empty *TA*^{f} table and *TA*^{b} table respectively in line 1. $minCost$ is for the minimal distance currently seen. The variable l_i^f for the maximal finalized distance in the latest forward expansion, n^f for the total number of frontier nodes, and i for the number of expansions in the forward expansion are initialized from line 3 to line 5. The counterpart variables in the backward expansion are also initialized in the same line.

We make path expansions in the iterations from line 6 to line 14. We select an expansion direction according to the number of frontier nodes, and show the details in the forward expansion from line 8 to line 11. We expand the paths using the 2-nd SQL in Listing 4, which merges the nodes in a view *ER* into the visited node set. *ER* contains the results of an extended E-operator in the 1-st SQL. Let i be the number of the forward expansions, pts be the number of the partitioned tables. The extended E-operator returns an union of $min(i, pts)$ basic E-operators. l in the 1-st SQL is $max(1, i - pts + 1)$, which is

Algorithm 2: Bi-directional Restrictive BFS on Partitioned Tables

```
Input: Source node $s$ and target node $t$, partitioned
       tables $TE_1,\ldots,TE_{pts}$ with a partitioning vector.
Output: The shortest path between $s$ and $t$.
1  Initialize $TA^f$ with node $s$ and $TA^b$ with node $t$;
2  $minCost \leftarrow +\infty$;
3  $i \leftarrow 1$, $j \leftarrow 1$;
4  $l_i^f \leftarrow 0$, $l_j^b \leftarrow 0$;
5  $n^f \leftarrow 1$, $n^b \leftarrow 1$;
6  while $l_i^f + l_j^b \leq minCost$ do
7      if $n^f \leq n^b$ then
8          Expand paths with the SQL in Listing 4(2);
9          $n^f \leftarrow$ the number of affected tuples from
           SQLCA;
10         Compute $l_i^f$ with the SQL in Listing 4(3) and
           Formula 1;
11         $i \leftarrow i+1$;
12     else
13         Similar actions from line 8 to line 11 for the
           backward expansion;
14     Locate $minCost$ using SQL in Listing 4(4);
15 Make verification with $TA^f$, $TA^b$, and $TE$ using SQL in
   Listing 4(5);
16 Compute the minimal distance with Formula 2;
17 Locate a node $xid$ in the shortest path with the SQL in
   Listing 4(6);
18 Find the sub-path $p_0$ from $s$ to $xid$ along $p2s$ links;
19 Find the sub-path $p_1$ from $xid$ to $t$ along $p2t$ links;
20 Return $p_0 + p_1$;
```

for the minimal value of fwd of all frontier nodes in the i^{th} expansion. We also incorporate the bi-directional pruning rule into the extended E-operator. Let $minCost$ be the minimal distance seen so far, l_j^b be the maximal distance finalized in the latest backward expansion. Only the nodes with their $d2s$ less than $minCost - l_j^b$ are considered in the forward expansion. We compute the maximal distance finalized l_i^f in the i^{th} forward expansion in line 10 using Formula 1. We further refine the minimal distance $minCost$ seen so far in line 14 using the 4-th statement.

When the condition $l_i^f + l_j^b \leq minCost$ in line 6 is satisfied, we terminate iterations and make a verification step using the 5-th statement. The shortest distance can be computed from the minimal distance discovered in the iterative path search and that in the verification phase, as shown in Formula 2. We then locate one node xid in the shortest path with the shortest distance $minCost$, and recover the full shortest path along the $p2s$ and $p2t$ links from xid respectively.

Analysis. Now we give the number of iterations used in the restrictive BFS over partitioned tables in Theorem 4.

Theorem 4: Let s be the source node and t be the target node. p be the shortest path between s and t. p can be discovered by $\sum_{e \in edges(p)}(part_{idx}(e))$ iterations, where $edges(p)$ is for the edge set of p, $part_{idx}(e)$ is for the index of the partitioned table for e.

Proof: Suppose we have made i forward expansions and j backward expansions in Algorithm 2, where $i + j = \sum_{e \in edges(p)}(part_{idx}(e))$. p can be found since each edge e in p will be expanded after $part_{idx}(e)$ expansions. We analyze whether the iterations can be terminated at that time. We know $minCost$ is $len(p)$, or $\delta(s,t)$, as p has been found. According to the Formula 1, l_i^f is the smaller one between the minimal distance $min(d2s)_i$ discovered in the latest i^{th} forward expansion and the minimal distance for delayed expansions. Since p has been discovered, $min(d2s)_i$ is no more than the minimal distance for delayed expansions. Then, l_i^f is $min(d2s)_i$. Similarly, l_j^b is $min(d2s)_j$ in the backward expansion. Thus, we have $l_i^f + l_j^b = minCost$, and the iterations can be terminated. ■

Listing 4. SQL in Expansion

```
1:extended $E$—operation in $i^{th}$ expansion
 Create view ER(id, p2s, d2s) as
   select TE.tid, TE.fid, TF.d2s+TE.cost
   from $TA^f$ as TF, $TE_{i-l+1}$ as TE
   where TF.fwd=$l$ and TF.nid=TE.fid
    and TF.d2s+TE.cost<($minCost - l_j^b$)
   union
   ....
   select TE.tid, TE.fid, TF.d2s+TE.cost
   from $TA^f$ as TF, $TE_1$ as TE
   where TF.fwd=$i$ and TF.nid=TE.fid
    and TF.d2s+TE.cost<($minCost - l_j^b$)
2:Make $i^{th}$ expansion with $F$, $E$, $M$—Operator
 Merge $TA^f$ as target
  using (select nid, p2s, cost
     from ( select er.id, er.p2s, er.d2s,
              row_number()over(partition by
              id order by d2s asc) as rownum
             from ER)
          tmp1 (nid, p2s, cost,rownum)
    where rownum=1
 ) as source(nid, p2s, cost)
 on source.nid=target.nid
 when matched and target.d2s>source.cost then
  update set d2s=source.cost, p2s=source.p2s, fwd=i+1
 when not matched by target then
  insert (nid, d2s, p2s, fwd)
    values( source.nid, cost, source.p2s, i+1);
3:Locate the minimal distance in forward searching
 Select min(d2s) from $TA^f$ where fwd=i+1;
4:Locate the minimal distance discovered currently
 Select min(d2s+d2t) from $TA^f$ TF, $TA^b$ TB
 where TF.nid=TB.nid;
5:Locate the minimal distance in verification phase
 Select min(d2s+cost+d2t) from $TA^f$ TF, $TA^b$ TB, TE
 where TF.nid=TE.fid and TE.tid=TB.nid
6:Locate a node in the shortest path
 Select nid into xid from $TA^f$ TF, $TA^b$ TB
 where TF.nid=TB.nid and d2s+d2t=minCost;
```

In general cases, the restrictive BFS over partitioned tables requires much fewer iterations than the classic Dijkstra' search. The restrictive BFS allows more nodes to be expanded in one operation, and then the maximal finalized distances l_i^f and l_j^b increase faster, which terminate the path expansions earlier. The reduction of iterations is more obvious on denser graphs. For example, the online social networks always have six degrees of separation. In such large dense graphs, a shortest path between any two nodes tends to contain fewer edges, which results in fewer iterations by Theorem 4.

5 Experimental Results

In this section, we experimentally evaluate the effectiveness and efficiency of our relational approach on both real and synthetic datasets extensively.

5.1 Experimental Setup

Implementation Details and Competitors. We compare 11 approaches related to this paper. These approaches are summarized in Table 1. We implement all methods in Java with JDK 1.6 and evaluate them on a server with 2× Intel Xeon E5620 64bit 2.4GHz processors, 24G of RAM, running Windows server 2008 R2 Standard. As for Neo4j [2], we take its major stable version neo4j-community-1.8 and use its built-in routing *GraphAlgoFactory.dijkstra* in the shortest path discovery.

DJ	Relational Dijkstra's algorithm in Algorithm 1
BDJ	Relational bi-directional Dijkstra's algorithm
BSDJ	Relational bi-directional set Dijkstra's algorithm [18]
BSEG	Relational bi-directional selective path search on SegTable in the previous version [18]
BBFS	Relational bi-directional BFS
BBFSn	Relational bi-directional BFS on ID based partitioned edge tables
BSDJn	Relational bi-directional set Dijkstra's algorithm on ID based partitioned edge tables
BRBFSw	Relational bi-directional restrictive BFS on weight aware partitioned edge tables in Algorithm 2
MDJ	In-memory Dijkstra's algorithm
MBDJ	In-memory bi-directional Dijkstra's algorithm
Neo4j	Dijkstra's algorithm implemented by Neo4j

TABLE 1
Methods to be compared

In order to show the applicability of our methods, we conduct experiments over two relational database systems: one commercial database system (denoted by *DBMS-x*) and another open source database system PostgreSQL 9.0 (denoted by *PostgreSQL*).

We build indices over the relational tables for the graph and intermediate results. Specifically, for the edge table *TE* or any partitioned edge table, we build clustered indices on fid in the tables. For the forward visited node table TA^f and the backward visited node table TA^b, we build unique clustered indices on nid on both tables. The meanings of fid and nid are given in Section 2.1 and Section 3.2 respectively.

Data Sets. We use 4 graph data sets in tests, including two real graphs *LiveJournal* and *Orkut*, and two synthetic graphs named *Random* and *Power*. *LiveJournal* is downloaded from Stanford's data collection[3], which is a friendship network of on line community LiveJournal. *Orkut* is a large social network graph [21]. *Random* graphs are generated as follows. Let n and m be the number of nodes and edges respectively, we randomly select the source and target node for m edges among n nodes. *Power* graph set is generated using *Barabasi Graph Generator v1.4*[4]. We assign the weights of edges in all graphs in two ways. The first is with *Uniform* weights that are drawn uniformly at random between 1 and 100. For the second one *Zipf*, we use Zipf distribution for the same range and assign the weights randomly. In Zipf's law with N elements, the frequency of the k^{th} element is $k^{-\alpha}/\sum_{n=1}^{N} n^{-\alpha}$, where α is a positive constant to control the skew of data distributions. We vary α from 0.2 to 1.0 in following experiments.

3. http://snap.stanford.edu/data/
4. http://www.cs.ucr.edu/ ddreier/barabasi.html

Some statistics of these graphs are summarized in Table 2. As for any synthetic graph, we suffix $xNyd$ to indicate that the graph is with x nodes and the average degree y. For example, *Random100mN3d* represents a *Random* graph with 100 million nodes and an average degree 3.

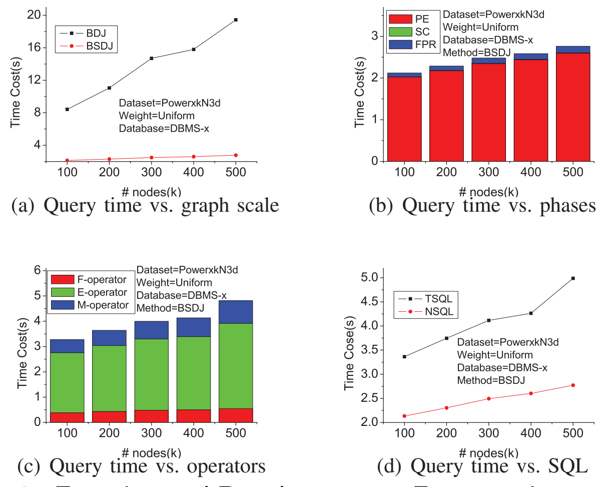
DataSet	# Nodes	# Edges
LiveJournal	4,847,571	43,110,428
Orkut	3,072,449	111,576,474
RandomxmNyd	5m-100m	5ym-100ym
Power$x$kNyd	100k-500k	100yk-500yk

TABLE 2
Statistics of Graph Data Sets

We study the issues related to the *FEM* framework and its optimizations. The experiments will be divided into 3 subparts. We first verify the effectiveness of the *FEM* framework and set-at-a-time evaluation fashion. We then study optimization strategies specific to shortest path discovery inside the *FEM* framework. Finally, we make extensible studies on our method with all optimization strategies. In the following experiments, we randomly generate 100 shortest path queries, and report the average time cost. The parameters used in the tests are listed inside figures.

5.2 *FEM* Framework and Set-at-a-time Fashion

In this part, we mainly answer the following questions: i) Can the *FEM* framework support Dijkstra's implementation in RDB? ii) What is the most expensive phase in path discovery? iii) Do the new SQL features boost the performance remarkably? The methods studied include DJ, BDJ and BSDJ in Table 1.



(a) Query time vs. graph scale (b) Query time vs. phases
(c) Query time vs. operators (d) Query time vs. SQL

Fig. 8. Experimental Results on *FEM* Framework

Figure 8(a) reports the results of DJ, BDJ, and BSDJ on *Power* graphs varying size from 100k to 500k. We omit the curves for DJ as DJ takes much more time cost than the other two. For example, DJ consumes about 15 minutes, while BDJ

	DJ		BDJ		BSDJ	
$\|V\|$	E	T	E	T	E	T
100k	>30k	>900	399	8.42	89	2.13
200k		>900	515	11.1	95	2.30
300k		>900	652	14.7	101	2.49
400k		>900	718	15.8	104	2.60
500k		>900	853	19.4	107	2.77

TABLE 3
E(# Expansions), T(Time:s) on Power Graphs

consumes 8.42 seconds, and BSDJ costs 2.13 seconds in a graph with 100k nodes.

In order to find the main factors to the evaluation cost, we collect the total number of expansions in the path finding and the time cost consumed by DJ, BDJ, and BSDJ in Table 3. It clearly shows that BSDJ takes the fewest expansions. The number of expansions in DJ is about 75 times bigger than that in BDJ, and 300 times bigger than that in BSDJ. Thus, the number of SQLs used by BSDJ are fewest among three. The results verify our claim before. The set-at-a-time evaluation fashion, which enables the RDB optimizer to produce a better evaluation plan, can beat the node-at-a-time evaluation fashion easily, when they have the same search space.

Figure 8(b) plots the time cost used in different phases in the path finding in BSDJ, including the path expansion (denoted by *PE*), the statistics collection (denoted by *SC*), and the full path recovery (denoted by *FPR*). We can see that the path expansion with three operators consumes most of the time. Next, we go deep into the path expansion. We directly translate F, E, and M-operator into separate SQLs, and collect their time cost. In Figure 8(c), we find E-operator takes about 70 percent of the time cost in the path expansion. It is mainly due to the high cost of join operation between the graph edge table and the visited node table in E-operator.

Figure 8(d) studies the query time cost affected by different SQL features. We implement path discovery with the new features including window function and merge statement (denoted by *NSQL*), and the traditional features including aggregate functions and insert/update for merge (denoted by *TSQL*). The results show that the *NSQL* method outperforms the *TSQL* method significantly. We believe that the advance of RDB makes it more powerful in handling the shortest path discovery and other complex graph operations.

Due to the fact that BSDJ outperforms DJ and BDJ thoroughly, we omit the curves for the latter two in the following. We will use new features of SQL in path finding unless explicitly mentioned.

5.3 Optimizations in Shortest Path Discovery

In this part, we first study the effectiveness of different optimizations, including BBFS, BSDJ, BSEG and BRBFSw listed in Table 1. We then attempt to find the impacts of database buffer, equal-width and equal-depth partitioning schemas, and the number of partitioned tables, on the performance of the shortest path discovery.

Figure 9(a) and Figure 9(b) compare the time cost consumed by BSDJ, BSEG(x), BBFS and BRBFSw(x) on two sets of large graphs with uniform weights. Here, x in BSEG(x) is for the index threshold, and we set an approximate optimal

	BSDJ			BSEG(10)			BBFS			BRBFSw(10)		
$\|V\|$	T	E	Vst	T	E	Vst	T	E	Vst	T	E	Vst
5M	5.0	173	3.7k	**3.3**	34	7.93k	8.2	32	126k	4.8	45	10.2k
10M	5.9	188	5.8k	**4.0**	36	12.0k	14	34	232k	5.5	47	15.4k
15M	6.2	192	6.5k	6.4	37	13.4k	16	34	270k	**6.0**	48	20.4k
20M	6.9	199	7.8k	9.6	38	17.5k	24	35	400k	**6.3**	49	22.3k
40M	17	208	10k	29	39	25.7k	31	37	606k	**8.7**	50	32.7k

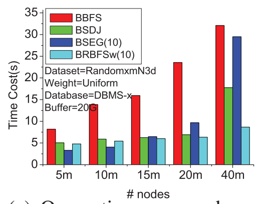
TABLE 4
T(Time:s), E(# expansions), and Vst (# visited nodes) on Random Graphs

threshold in our tests (more details are specified in the previous work [18]). x in BRBFSw(x) is for the total number of partitioned tables. We can observe that in bigger graphs, BRBFSw is fastest. The time cost in BRBFSw is nearly 1/4 of that in BBFS, 1/2 in BSDJ and 1/4 in BSEG across *Random40mN3d* graph, and the proportion to BBFS is about 1/10 on *Orkut* graph.

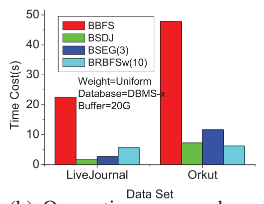
We make a deep analysis on the detailed time cost, the number of expansions and the number of visited nodes in different strategies in Table 4. We can see that the efficiency of BRBFSw on large graphs comes from that BRBFSw can reduce the number of SQLs at the cost of slightly enlarged search space. For example, the visited nodes in BRBFSw(10) are a little more than those in BSDJ, but the number of expansions in BRBFSw(10) is just 1/4 of that in BSDJ. We also see that although BBFS takes fewest expansions, BBFS is slowest in the tests due to its much more visited nodes. BSEG attempts to lower the number of SQLs with pre-computed segments. It works best when a graph is small, but its performance degrades sharply in handling large graphs as it incurs too many pre-computed segments. For example, BSEG generates 18,515,589 additional segments on *Random40mN3d* graph with $l_{thd} = 10$. By combining these experimental results, we know that we cannot consider the evaluation fashion (the number of expansion operations) or the search space (the size of intermediate nodes) separately, but strike to achieve a balance between them.

Next, we study the impact of different buffer sizes on the query performance. Figure 9(c) and Figure 9(d) compare the time cost used by BSDJ, BSEG, BBFS and BRBFSw varying different buffer sizes. We can see a major advantage of BRBFSw compared with BBFS, BSDJ and BSEG when the buffer is reduced. It is due to that the expansion in BRBFSw can be made on much smaller partitioned tables. In addition, BRBFSw is buffer-friendly. The partitioned tables with smaller indices have more chances to be in the buffer as they are accessed more frequently in extended E-operators.

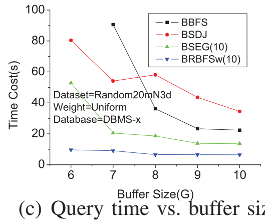
Figure 9(e) studies different effects of equal-width and equal-depth partitioning schema on the query performance across *Random* graphs with Zipf weight distributions varying skew α from 0.2 to 1.0. The results are a little surprising. The equal-depth schema does not have obvious advantages over the equal-width one on graphs with random skewed edge weights. We further analyze the size of partitioned tables, and find that the random skewed edge weights may result in non-skewed partitioned tables in the equal-width schema, as each partitioned table contains edges within a range of weights. For example, the maximal size of partitioned tables is only



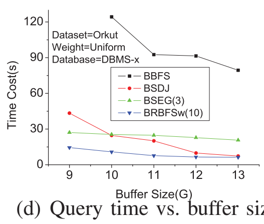
(a) Query time vs. graph scale



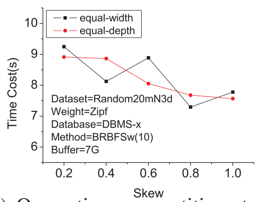
(b) Query time vs. graph scale



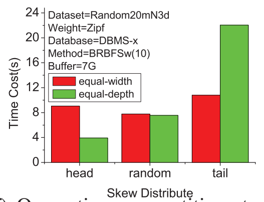
(c) Query time vs. buffer size



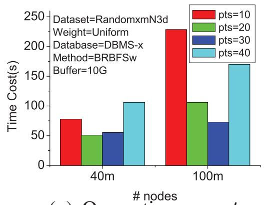
(d) Query time vs. buffer size



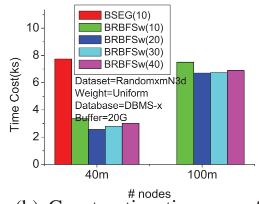
(e) Query time vs. partition strategy



(f) Query time vs. partition strategy



(g) Query time vs. pts



(h) Construction time vs. pts

Fig. 9. Experimental Results of Different Optimization Strategies

about twice the average size with $\alpha = 1.0$ in the equal-width schema. These non-skewed partitioned tables will alleviate the impacts of skewed edge weights on the equal-width schema in the restrictive BFS.

However, the above reasons cannot explain why the equal-width schema may obtain a better performance than the equal-depth one on skewed partitioned tables. We conjecture that the partitioned tables with smaller indices will have a bigger impact on the search space in the restrictive BFS. To verify the claim, we artificially assigned small weights(denoted by *Head*), random weights(denoted by *Random*), large weights(denoted by *Tail*) with high frequencies to edges and report the results in Figure 9(f). As expected, the equal-depth schema works better in the case of *Head*, while the equal-width schema wins in the case of *Tail*. Recall that BRBFSw expands along edges in partitioned tables with smaller indices first. In the case of *Head*, the partitioned tables with smaller indices contain more edges in the equal-width schema than those in the equal-depth one, and then BRBFSw visits more nodes in the following search, which hinders the performance on the equal-width schema. The results are opposite in the case of *Tail*.

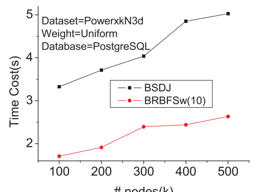
Figure 9(g) illustrates the query evaluation time cost varying pts, the number of partitioned tables. We can see the increase of pts first lowers the query cost and then incurs more query cost. In fact, the setting of pts has twofold impacts. On the one hand, a larger pts results in more smaller partitioned tables, which enables the optimizer in the RDB to produce a better buffer strategy. On the other hand, a larger pts means we need more expansions to locate the shortest path, which cuts down the query performance. In addition, as shown in Figure 9(g), a relatively larger pts (*e.g.,* pts=30) is appropriate for *Random100m* graph while a smaller pts (*e.g.,* pts=20) is suitable on *Random40m*. How to find an optimal pts over different graphs will be a future work of this paper.

Figure 9(h) studies the time cost in the weight aware edge table partitioning varying pts over two large *Random* graphs. We need to create partitioned tables $TE_i (1 \leq i \leq pts)$, then put edges into TE_i according to their weights, and finally build indices on TE_i. The overall time cost is insensitive to different settings of pts. Another observation is that table partitioning shows a much higher scalability than SegTable indexing [18]. For example, SegTable takes more than 5 hours on *Random100mN3d* graph in indexing.

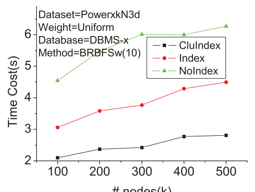
5.4 Extensive Studies

In this part, we study the impacts of different database engines and index strategies on the query performance. Finally, we compare our RDB approach with the in-memory ones and graph database Neo4j.

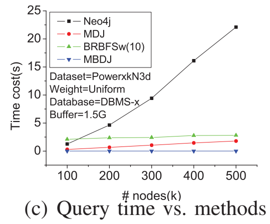
Figure 10(a) compares the query time consumed by BSDJ and BRBFSw on *PostgreSQL*. Since *PostgreSQL* supports the window function but cannot provide the *merge* statement, we use an *update* followed by an *insert* statement for the M-operator instead. The results on *PostgreSQL* are similar to those on the commercial database system, which also show that our method has good applicability.



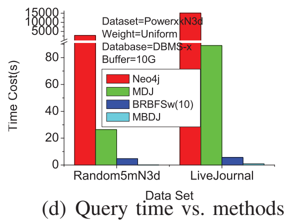
(a) Query time vs. database



(b) Query time vs. index



(c) Query time vs. methods



(d) Query time vs. methods

Fig. 10. Experimental Results on Extensive Study

In Figure 10(b), we conduct studies on different index strategies, including the clustered unique index (denoted by *CluIndex*), non-clustered unique index (denoted by *Index*), and no index (denoted by *NoIndex*) on fid in the edge table *TE*, as well as on nid in two visited node tables TA^f and TA^b. We can see that *CluIndex* achieves the best performance. *CluIndex* offers RDB a chance to implement the extended E-operator and the discovery of candidate shortest distances by the merge-sort join, in which only the merging cost is needed since the clustered indices have imposed orders on nodes.

Last, we compare our relational approach with the in-memory ones, including MDJ and MBDJ, as well as Neo4j in Table 1. The buffer size of RDB is set to be the same as the JVM memory used by in-memory approaches. Neo4j is embedded in a java application with the same JVM memory. In order to make comparisons fair, the time reported for in-memory approaches does not include the time in loading graph into memory, and the time for relational approaches and Neo4j graph database is collected after the database buffer becomes hot. In both Figure 10(c) and Figure 10(d), we find that the performance and scalability of Neo4j are not good on large graphs. In fact, we cannot run Neo4j on *Random5mN3d* and *Livejournal* with 10G memory and we set the JVM memory used in Neo4j to 20G. We believe the following strategies contribute to the improvement of our method. First, BRBFSw employs a bi-directional search strategy, which can reduce the search space significantly. Second, BRBFSw takes the set-at-a-time fashion, which can fully exploit the data loaded into the memory and then cut down the high cost in I/O operations. Third, BRBFSw adopts the table partitioning schema, which can greatly lower the join cost in path expansions.

In addition, we can see our BRBFSw algorithm is not as fast as MBDJ. It is not surprised as RDB is a general-purpose infrastructure in different information systems, and is not tailored for the graph data management. Furthermore, RDB cannot exploit all memory in the buffer for the shortest path discovery, since RDB keeps other information, such as system tables, in the buffer. We also notice that BRBFSw can outperform MDJ on large graphs and show a better scalability. These results also reveal that a proper evaluation strategy in the RDB context can still gain a satisfactory performance. We stress here that the main advantage of RDB lies in its scalability, stability and easy programming in the graph management.

5.5 Summary

To sum up, from the experimental results, we can draw the following conclusions: i) The *FEM* framework can be used to support graph search queries such as the shortest path discovery. The new features introduced by recent SQL standards can improve the performance of the *FEM* framework. ii) The bi-directional restrictive BFS on weight aware partitioned tables can improve both performance and scalability significantly. The choice of the equal-width and equal-depth partitioning schemas is related to the skew of partitioned tables and the size of partitioned tables with small indices. iii) Our relational approach is more efficient than the existing graph database Neo4j in the shortest path discovery on large graphs.

6 Related Work

Graph Search. Graph search queries are basic and important graph operations. Due to the large search space, many proposed approaches take a greedy idea, including Dijkstra's algorithm for the shortest path [13], Prim's algorithm for the minimal spanning tree [23], the salesman path discovery [10], and the like. These approaches follow a similar select-expand-merge pattern in the graph search.

Shortest path discovery is a representative graph search operation. Dijkstra's algorithm is a well-known online algorithm [13] to solve the single-source shortest path problem. The bi-directional search strategy [11] is an effective optimization on Dijkstra's algorithm. Different kinds of shortest path indices, such as the landmark index [22], 2-HOP related index [12], etc, can be computed offline and boost the performance in the running time. However, neither these online shortest path discovery nor index building methods consider the case when the graph cannot be fully loaded into main memory.

External Graph Operations. The external graph operations have been also studied to handle large graphs. An external shortest path index is proposed by [9] on planar graphs. MapReduce framework [17], [4] and its open source implementation Hadoop [1] achieve a high scalability in processing large graphs. The limitation of current MapReduce framework lies in its weak support to online query and expensive dynamic update of graphs. We also notice the studies on specific graph operations in the external memory. For example, the approximate minimum-cut [6] can be computed on the sampled graph. The cliques can be found on the partially loaded sub-graphs [16]. However, it is hard to extend these methods to support general graph search queries.

RDB based Graph Operations. RDB, as a scalable platform, can be extended to manage complex data, such as XML data [19], [14], [20], and to support sophisticated applications, such as data mining [5] and statistical data analysis [7]. An operation on these data or in applications is always translated into a sequence of SQLs, due to the limited expressive power of SQL. In the translation, an important optimization is to produce fewer SQLs [20]. We also incorporate such a feature in designing an efficient relational approach for graph search queries.

The studies on transitive closure and aggregation with recursion in the relational context can be viewed as pioneer works to support graph operations in RDB, as graph search queries, such as the shortest path discovery [26], [24], can be expressed using transitive closure, aggregation and selection. Different approaches are compared to find the full or partial transitive closure [26]. The aggregation with recursion can be computed efficiently in an incremental way [24] with the changes of underlying graph data. Compared with the previous works, this paper focuses on the shortest path discovery in the relational context with new features of SQL, and designs various optimizations specific to the problem, including the restrictive BFS and the table partitioning schema.

RDB based graph query processing is related to our work. The reachability query is evaluated by a stored procedure

[28]. The SQL based graph data mining approaches have been studied in [27], [8]. Different from the existing methods, we attempt to design a generic graph search framework in RDB, and improve the efficiency of the framework with new features of SQL. We also optimize the relational shortest path discovery by considering the search space reduction, RDB-friendly evaluation fashion and table partitioning schema.

7 Conclusions

This paper proposes a relational approach to discover the shortest path on large graphs. We first abstract a relational generic graph search framework *FEM* with three new operators, and employ the new features of SQL such as window function and merge statement to improve the performance of the *FEM* framework. Second, we optimize the basic method via the bi-directional restrictive BFS over weight aware partitioned edge tables, which can improve both performance and scalability significantly without extra overheads. The final extensive experimental results show that our approach can achieve high efficiency and scalability compared with the existing methods. Due to space limitations, the extension of the *FEM* framework to other complex graph operations will be our future work.

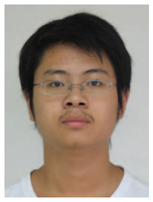
Acknowledgments

This work was supported by NSFC under Grant No. 61073018 and 61272156, the research grants Council of the Hong Kong SAR under No. 418512 and 419109, National High Technology Research and Development Program of China under No. 2012AA011002.

References

[1] *Apache Hadoop*. http://hadoop.apache.org.
[2] *Neo4j*. http://neo4j.org/.
[3] A.Goldberg and C.Harrelson. Computing the shortest path: search meets graph theory. In *SODA*, pages 156–165, 2005.
[4] B.Bahmani, K.Chakrabarti, and D.Xin. Fast personalized pagerank on mapreduce. In *SIGMOD*, pages 973–984, 2011.
[5] B.Zou, X.Ma, B.Kemme, G.Newton, and D.Precup. Data mining using relational database management systems. In *PAKDD*, pages 657–667, 2006.
[6] C.Aggarwal, Y.Xie, and P.Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2(1):862–873, 2009.
[7] C.Mayfield, J.Neville, and S.Prabhakar. Eracer: a database approach for statistical inference and data cleaning. In *SIMGMOD*, pages 75–86, 2010.
[8] C.Wang, W.Wang, J.Pei, Y.Zhu, and B.Shi. Scalable mining of large disk-based graph databases. In *SIGKDD*, pages 316–325, 2004.
[9] D.Hutchinson, A.Maheshwari, and N.Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126(1):55–82, 2003.
[10] D.Johnson and L.McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 215-310, 1997.
[11] D.Wagner and T.Willhalm. Speed-up techniques for shortest-path computations. In *STACS*, pages 23–36, 2007.
[12] E.Cohen, E.Halperin, H.Kaplan, and U.Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.
[13] E.Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.
[14] F.Tian, B.Reinwald, H.Pirahesh, T.Mayr, and J.Myllymaki. Implementing a scalable xml publish/subscribe system using a relational database system. In *SIGMOD*, pages 479–490, 2004.
[15] H.Garcia-Molina, J.Ullman, and J.Widom. *Database Systems: The Complete Book*. Prentice Hall Press, 2008.
[16] J.Cheng, Y.Ke, A.W.Fu, J.X.Yu, and L.Zhu. Finding maximal cliques in massive networks by h*-graph. In *SIGMOD*, pages 447–458, 2010.
[17] J.Dean and S.Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[18] J.Gao, R.Jin, J.Zhou, J.Xu, X.Jiang, and T.Wang. Relational approach for shortest path discovery over large graphs. *PVLDB*, 5(4):358–369, 2011.
[19] J.Shanmugasundaram, K.Tufte, C.Zhang, G.He, D.DeWitt, and J.Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.
[20] M.Benedikt, C.Chan, W.Fan, R.Rastogi, S.Zheng, and A.Zhou. Dtd-directed publishing with attribute translation grammars. In *VLDB*, 2002.
[21] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC*, San Diego, CA, October 2007.
[22] M.Potamias, F.Bonchi, C.Castillo, and A.Gionis. Fast shortest path distance estimation in large networks. In *CIKM*, pages 453–470, 2009.
[23] R.Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.
[24] R.Ramakrishnan and S. Sudarshan Kenneth A. Ross, Divesh Srivastava. Efficient incremental evaluation of queries with aggregation. In *SLP*, pages 204–218, 1994.
[25] R.Ronen and O.Shmueli. Soql: A language for querying and creating data in social networks. In *ICDE*, pages 1595–1602, 2009.
[26] S.Dar and R.Ramakrishnan. A performance study of transitive closure algorithms. In *SIGMOD*, pages 454–465, 1994.
[27] S.Srihari, S.Chandrashekar, and S.Parthasarathy. A framework for sql-based mining of large graphs on relational databases. In *PAKDD*, pages 160–167, 2010.
[28] S.Trißl and U.Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

Jun Gao received his B.E. and M.E. in Computer Science, from Shandong University, China, in 1997, 2000, and received his Ph.D in Computer Science, from Peking university in 2003. Currently he is a Professor in the School of Electronics Engineering and Computer Science, Peking University, China. His major research interests include web data management and graph data management.



Jiashuai Zhou received his B.S. degree from the Department of Computer Science, Peking University, China, in 2011. He is currently a graduate student in Department of Computer Science, School of Electronics Engineering and Computer Science, Peking University. His research interests include graph data management and web data management.

Jeffrey Xu Yu received his B.E., M.E. and Ph.D. in Computer Science, from the University of Tsukuba, Japan, in 1985, 1987 and 1990, respectively. Currently he is a Professor in the Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong. His major research interests include data mining, data stream, data warehouse, graph query processing and optimization.

Tengjiao Wang received his B.E. and M.E. in Computer Science, from Shandong University, China, in 1996, 1999, and received his Ph.D in computer science, from Peking university in 2002. Currently he is a Professor in the School of Electronics Engineering and Computer Science, Peking University, China. His major research interests include data mining, web data management, database system implementation.