GraphREL: A Decomposition-Based and Selectivity-Aware Relational Framework for Processing Sub-graph Queries

Sherif Sakr

NICTA and University of New South Wales
Sydney, Australia
Sherif.Sakr@nicta.com.au

Abstract. Graphs are widely used for modelling complicated data such as: chemical compounds, protein interactions, XML documents and multimedia. Retrieving related graphs containing a query graph from a large graph database is a key issue in many graph-based applications such as drug discovery and structural pattern recognition. Relational database management systems (RDBMSs) have repeatedly been shown to be able to efficiently host different types of data which were not formerly anticipated to reside within relational databases such as complex objects and XML data. The key advantages of relational database systems are its well-known maturity and its ability to scale to handle vast amounts of data very efficiently. RDMBSs derive much of their performance from sophisticated optimizer components which makes use of physical properties that are specific to the relational model such as: sortedness, proper join ordering and powerful indexing mechanisms. In this paper, we study the problem of indexing and querying graph databases using the relational infrastructure. We propose a novel, decomposition-based and selectivity-aware SQL translation mechanism of sub-graph search queries. Moreover, we carefully exploit existing database functionality such as partitioned B-trees indexes and influencing the relational query optimizers by selectivity annotations to reduce the access costs of the secondary storage to a minimum. Finally, our experiments utilise an IBM DB2 RDBMS as a concrete example to confirm that relational database systems can be used as an efficient and very scalable processor for sub-graph queries.

1 Introduction

Graphs are among the most complicated and general form of data structures. Recently, they have been widely used to model many complex structured and schemaless data such as XML documents [24], multimedia databases [18], social networks [3] and chemical compounds [16]. Hence, retrieving related graphs containing a query graph from a large graph database is a key performance issue in all of these graph-based applications. It is apparent that the success of any graph database application is directly dependent on the efficiency of the graph indexing and query processing mechanisms. The fundamental sub-graph

X. Zhou et al. (Eds.): DASFAA 2009, LNCS 5463, pp. 123–137, 2009.

search operation on graph databases can be described as follows: given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a graph query q expressed as *find all graphs* g_i *which belongs to the graph database* D *such that* q *is a subgraph of* g_i. Clearly, it is an inefficient and a very time consuming task to perform a sequential scan over the whole graph database D and then to check whether q is a subgraph of each graph database member g_i. Hence, there is a clear necessity to build graph indices in order to improve the performance of processing sub-graph queries.

Relational database management systems (RDBMSs) have repeatedly shown that they are very efficient, scalable and successful in hosting types of data which have formerly not been anticipated to be stored inside relational databases such complex objects [7,21], spatio-temporal data [8] and XML data [6,14]. In addition, RDBMSs have shown its ability to handle vast amounts of data very efficiently using its powerful indexing mechanisms. In this paper we focus on employing the powerful features of the relational infrastructure to implement an efficient mechanism for processing sub-graph search queries.

In principle, XPath-based XML queries [4] are considered to be a simple form of graph queries. Over the last few years, various relational-based indexing methods [5,6,11,23] have been developed to process this type of XML queries. However, these methods are optimized to deal only with tree-structured data and path expressions. Here, we present a purely relational framework to *speed up* the search efficiency in the context of graph queries. In our approach, the graph data set is firstly encoded using an intuitive *Vertex-Edge* relational mapping scheme after which the graph query is translated into a sequence of SQL evaluation steps over the defined storage scheme. An obvious problem in the relational-based evaluation approach of graph queries is the huge cost which may result from the large number of join operations which are required to be performed between the encoding relations. In order to overcome this problem, we exploit an observation from our previous works which is that the size of the intermediate results dramatically affect the overall evaluation performance of SQL scripts [12,19,20]. Hence, we use an effective and efficient pruning strategy to *filter* out as many as possible of the false positives graphs that are guaranteed to not exist in the final results first before passing the candidate result set to an *optional verification* process. Therefore, we keep statistical information about the *less frequently* existing nodes and edges in the graph database in the form of simple Markov Tables [2]. This statistical information is also used to influence the decision of relational query optimizers by selectivity annotations of the translated query predicates to make the right decision regarding selecting the most efficient join order and the cheapest execution plan to get rid of the *non-required* graphs very early out of the intermediate results. Moreover, we carefully exploit the fact that the number of distinct vertices and edges labels are usually far less than the number of vertices and edges respectively. Therefore, we try to achieve the maximum performance improvement for our relation execution plans by utilizing the existing powerful *partitioned B-trees* indexing mechanism of the relational databases [10] to reduce the access costs of the secondary storage to the minimum [13]. In summary, we made the following contributions in this paper:

1) We present a purely relational framework for evaluating directed and labelled sub-graph search queries. In this framework, we encode graph databases using an intuitive *Vertex-Edge* relational schema and translate the sub-graph queries into standard SQL scripts.
2) We describe an effective and very efficient pruning strategy for reducing the size of the intermediate results of our relational execution plans by using a simple form of statistical information in the form of Markov tables.
3) We describe our approach of using summary statistical information about the graph database vertices and edges to consult the relational query optimizers through the use of accurate selectivity annotations for the predicates of our queries. These selectivity annotations help the query optimizers to decide the right join order and choose the most efficient execution plan.
4) We exploit a *carefully tailored* set of the powerful partitioned B-trees relational indexes to reduce the secondary storage access costs of our SQL translation scripts to a minimum.
5) We show the efficiency and the scalability of the performance of our approach through an extensive set of experiments.

The remainder of the paper is organized as follows: We discuss some background knowledge in Section 2. Section 3 describe the different components of our relational framework for processing graph queries including the graph coding method, pruning strategy and the SQL translation mechanisms. We evaluate our method by conducting an extensive set of experiments which are described in Section 4. We discuss the related work in Section 5. Finally, we conclude the paper in Section 6.

2 Preliminaries

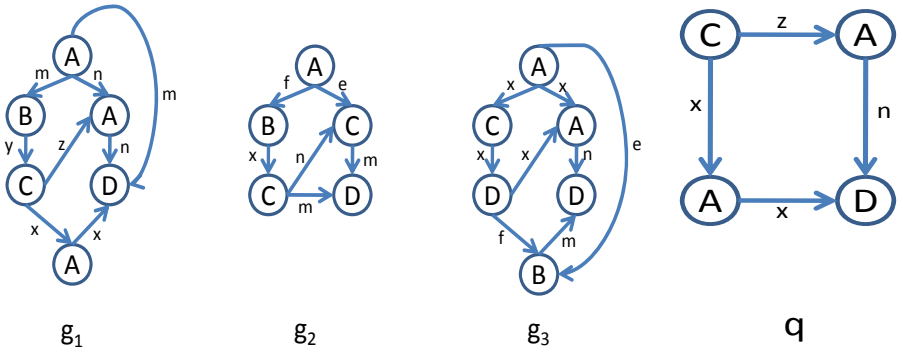
2.1 Labelled Graphs

In labelled graphs, vertices and edges represent the entities and the relationships between them respectively. The attributes associated with these entities and relationships are called labels. A graph database D is a collection of member graphs $D = \{g_1, g_2, ..., g_n\}$ where each member graph g_i is denoted as (V, E, L_v, L_e) where V is the set of vertices; $E \subseteq V \times V$ is the set of edges joining two distinct vertices; L_v is the set of vertex labels and L_e is the set of edge labels.

In principal, labelled graphs can be classified according to the direction of their edges into two main classes: 1) *Directed-labelled graphs* such as XML, RDF and traffic networks. 2) *Undirected-labelled graphs* such as social networks and chemical compounds. In this paper, we are mainly focusing on dealing with directed labelled graphs. However, it is straightforward to extend our framework to process other kinds of graphs. Figure 1(a) provides an example of a graph database composed of three directed-labelled graphs $\{g_1, g_2, g_3\}$.

2.2 Subgraph Search Queries

In principal, the subgraph search operation can be simply described as follows: given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a graph query q, it returns the



(a) Sample graph database (b) Graph query

Fig. 1. An example graph database and graph query

query answer set $A = \{g_i | q \subseteq g_i, g_i \in D\}$. A graph q is described as a sub-graph of another graph database member g_i if the set of vertices and edges of q form subset of the vertices and edges of g_i. To be more formal, let us assume that we have two graphs $g_1(V_1, E_1, L_{v1}, L_{e1})$ and $g_2(V_2, E_2, L_{v2}, L_{e2})$. g_1 is defined as sub-graph of g_2 , if and only if:

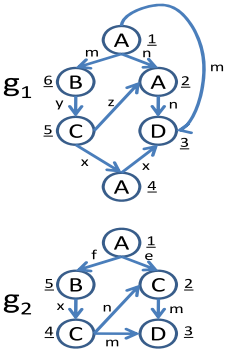
1) For every distinct vertex $x \in V_1$ with a label $vl \in L_{v1}$, there is a distinct vertex $y \in V_2$ with a label $vl \in L_{v2}$.
2) For every distinct edge edge $ab \in E_1$ with a label $el \in L_{e1}$, there is a distinct edge $ab \in E_2$ with a label $el \in L_{e2}$.

Figure 1(b) shows an example of graph query q. Running the example query q over the example graph database D (Figure 1(a)) returns an answer set consists of the graph database member g_1.

3 GraphREL Description

3.1 Graph Encoding

The starting point of our relational framework for processing sub-graph search queries is to find an efficient and suitable encoding for each graph member g_i in the graph database D. Therefore, we propose the *Vertex-Edge* mapping scheme as an efficient, simple and intuitive relational storage scheme for storing our targeting *directed labelled graphs*. In this mapping scheme, each graph database member g_i is assigned a unique identity *graphID*. Each vertex is assigned a sequence number (*vertexID*) inside its graph. Each vertex is represented by one tuple in a single table (*Vertices table*) which stores all vertices of the graph database. Each vertex is identified by the *graphID* for which the vertex belongs to and the *vertex ID*. Additionally, each vertex has an additional attribute to store the vertex label. Similarly, all edges of the graph database are stored in a single table (*Edges table*) where each edge is represented by a single tuple in this table. Each edge tuple describes the graph database member which the edge



graphID	vertexID	vLabel
1	1	A
1	2	A
1	3	D
1	4	A
1	5	C
1	6	B
2	1	A
2	2	C
2	3	D
2	4	C
2	5	B

Vertices Table

graphID	sVertex	dVertex	eLabel
1	1	2	n
1	1	3	m
1	2	3	n
1	4	3	x
1	5	4	x
1	6	5	y
1	5	2	z
1	1	6	m
2	1	2	e
2	2	3	m
2	4	3	m
2	4	2	n
2	5	4	x
2	1	5	f

Edges Table

Fig. 2. Vertex-Edge relational mapping scheme of graph database

belongs to, the id of the *source vertex* of the edge, the id of the *destination vertex* of the edge and the edge label. Therefore, the relational storage scheme of our *Vertex-Edge* mapping is described as follows:

- $Vertices(graphID, vertexID, vertexLabel)$
- $Edges(graphID, sVertex, dVertex, edgeLabel)$

Figure 2 illustrates an example of the *Vertex-Edge* relational mapping scheme of graph database. Using these mapping scheme, we employ the following SQL-based *filtering-and-verification* mechanism to speed up the search efficiency of sub-graph queries.

- **Filtering phase:** in this phase we use an effective and efficient pruning strategy to filter out as many as possible of the non-required graph members very early. Specifically, in this phase we specify the set of graph database members contain the set of vertices and edges which are describing the sub-graph query. Therefore, the filtering process of a sub-graph query q consists of a set of vertices QV with size equal m and a set of edges QE equal n (see Figure 1(b)) can be achieved using the following SQL translation template:

1 SELECT DISTINCT $V_1.graphID, V_i.vertexID$
2 FROM Vertices as V_1,..., Vertices as V_m, Edges as E_1,..., Edges as E_n
3 WHERE
4 $\forall_{i=2}^{m}(V_1.graphID = V_i.graphID)$
5 AND $\forall_{j=1}^{n}(V_1.graphID = E_j.graphID)$
6 AND $\forall_{i=1}^{m}(V_i.vertexLabel = QV_i.vertexLabel)$
7 AND $\forall_{j=1}^{n}(E_j.edgeLabel = QE_j.edgeLabel)$
8 AND $\forall_{j=1}^{n}(E_j.sVertex = V_f.vertexID$ AND $E_j.dVertex = V_f.vertexID)$;

(transTemplate)

Where each referenced table V_i (Line number 2) represents an instance from the table $Vertices$ and maps the information of one vertex of the set of vertices QV which is belonging to the sub-graph query q. Similarly, each referenced table E_j represents an instance from the table $Edges$ and maps the information of one edge of the set of edges QE which is belonging to the sub-graph query q. f is the mapping function between each vertex of

QV and its associated vertices table instance V_i. Line number 4 of the SQL translation template represents a set of $m-1$ conjunctive conditions to ensure that all queried vertices belong to the same graph. Similarly, Line number 5 of the SQL translation template represents a set of n conjunctive conditions to ensure that all queried edges belong to the same graph of the queried vertices. Lines number 6 and 7 represent the set of conjunctive predicates of the vertex and edge labels respectively. Line number 8 represents the edges connection information between the mapped vertices.

- **Verification phase:** this phase is an *optional* phase. We apply the verification process only if more than one vertex of the set of query vertices QV have the same label. Therefore, in this case we need to verify that each vertex in the set of filtered vertices for each candidate graph database member g_i is distinct. This can be easily achieved using their *vertex ID*. Although the fact that the conditions of the verification process could be injected into the SQL translation template of the filtering phase, we found that it is more efficient to avoid the cost of performing these conditions over each graph database members g_i by delaying their processing (*if required*) in a separate phase after pruning the candidate list.

Clearly, an obvious problem of the SQL translation template of the *filtering* is that it involves a large number of conjunctive SQL predicates $(2m + 4n)$ and join $(m + n)$ *Vertices* and *Edges* tables instances. Hence, although this template can be efficiently used with relatively small sub-graph search queries, most of relational query engines will certainly fail to execute the SQL translation queries of medium size or large sub-graph queries because they are too long and too complex (this does not mean they must consequently be too expensive). In the following subsections we will describe our approach to effectively deal with this problem by carefully and efficiently *decomposing* this complex one step evaluation step into a series of well designed relational evaluation steps.

3.2 Relational Indexes Support for Vertex-Edge Mapping Scheme

Relational database indexes have proven to be very efficient tools to speed up the performance of evaluating the SQL queries. Moreover, the performance of queries evaluation in relational database systems is very sensitive to the defined indexes structures over the data of the source tables. In principal, using relational indexes can accelerate the performance of queries evaluation in several ways. For example, applying highly selective predicates first can limit the data that must be accessed to only a very limited set of rows that satisfy those predicates. Additionally, query evaluations can be achieved using index-only access and save the necessity to access the original data pages by providing all the data needed for the query evaluation. In [15], He and Singh have presented an indexing scheme for processing graph queries which is very similar to R-Tree index structure. However, R-tree indexing technique is not commonly supported by many of the RDBMS systems where B-tree indexing is still the most commonly used technique. Therefore, in our purely relational framework, we use a *standard*, powerful and matured indexing mechanisms to accelerate the processing

performance of our SQL evaluation of the sub-graph queries, namely *partitioned B-tree indexes* and *automated relational index advisors*.

Partitioned B-tree Indexes Partitioned B-tree indexes are considered to be a slight variant of the B-tree indexing structure. The main idea of this indexing technique has been represented by Graefe in [10] where he recommended the use of low-selectivity leading columns to maintain the partitions within the associated B-tree. For example, in labelled graphs, it is generally the case that the number of *distinct* vertices and edges labels are far less than the number of vertices and edges respectively. Hence, for example having an index defined in terms of columns $(vertexLabel, graphID)$ can reduce the access cost of sub-graph query with only one label to one disk page which is storing a list of *graphID* of all graphs which are including a vertex with the target query label. On the contrary, an index defined in terms of the two columns $(graphID, vertexLabel)$ requires scanning a large number of disk pages to get the same list of targeted graphs. Conceptually, this approach could be considered as a horizontal partitioning of the *Vertices* and *Edges* table using the high selectivity partitioning attributes. Therefore, instead of requiring an execution time which is linear with the number of graph database members (graph database size), having partitioned B-trees indexes of the high-selectivity attributes can achieve fixed execution times which are no longer dependent on the size of the whole graph database [10,13].

Automated Relational Index Advisor Leveraging the advantage of relying on a pure relational relational infrastructure, we are able to use the ready made tools provided by the RDBMSs to propose the candidate indexes that are effective for accelerating our query work loads. In our work, we were able to use the *db2advis* tool provided by the DB2 engine (our hosting experimental engine) to recommend the suitable index structure for our query workload. Through the use of this tool we have been able significantly improve the quality of our designed indexes and to speed up the evaluation of our queries by reducing the number of calls to the database engine. Similar tools are available in most of the widely available commercial RDBMSs.

3.3 Statistical Summaries Support of Vertex-Edge Mapping Scheme

In general, one of the most effective techniques for optimizing the execution times of SQL queries is to select the relational execution based on the accurate selectivity information of the query predicates. For example, the query optimizer may need to estimate the selectivities of the occurrences of two vertices in one subgraph, one of these vertices with label A and the other with label B to choose the more selective vertex to be filtered first. Providing an accurate estimation for the selectivity of the predicates defined in our SQL translation template requires having statistical information that contain information about the structure of the stored graph data. Additionally, these statistics must be small enough to be processed efficiently in the short time available for query optimization and without any disk accesses. Therefore, we construct three Markov tables to store information about the frequency of occurrence of the distinct labels of vertices,

Vertex Label	Frequency
A	100
B	200
C	38
D	4
E	50
L	6
M	10
N	250
O	3
P	40
R	55

Markov Table summary of vertices labels

Edge Label	Frequency
a	40
c	5
e	28
l	54
m	140
n	3
o	20
p	15
x	8
y	60
z	15

Markov Table summary of edges labels

Edge Label Connection	Frequency
ab	3
ac	15
ae	45
ec	14
em	103
la	5
pc	18
px	45
xy	25
xz	2
za	1

Markov Table summary of pair-wise edge connections

Fig. 3. Sample Markov tables summaries of Vertex-Edge mapping

distinct labels of edges and connection between pair of vertices (edges). Figure 3 presents an example of our Markov table summary information. In our context, we are only interested in label and edge information with low frequency. Therefore, it is not necessary and not useful to keep all such frequency information. Hence, we summarize these Markov tables by deleting high-frequency tuples up to certain defined threshold $freq$. The following subsection will explain how we can use these information about the low frequency labels and edges to effectively prune the search space, reduce the size of intermediate results and influence the decision of the relational query optimizers to select the most efficient join order and the cheapest execution plan in our decomposed and selectivity-aware SQL translation of sub-graph queries.

3.4 Decomposition-Based and Selectivity-Aware SQL Translation of Sub-graph Queries

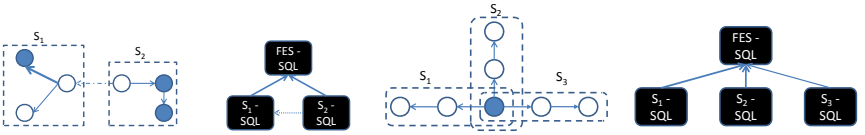
In Section 3, we described our preliminary mechanism for translating sub-graph queries into SQL queries using our *Vertex-Edge mapping*. As discussed previously, the main problem of this *one-step* translation mechanism is that it cannot be used with medium or large sub-graph queries as it generate SQL queries that are too long and too complex. Therefore, we need a *decomposition* mechanism to divide this large and complex SQL translation query into a sequence of intermediate queries (using temporary tables) before evaluating the final results. However, applying this decomposition mechanism blindly may lead to inefficient execution plans with very large, non-required and expensive intermediate results. Therefore, we use the statistical summary information described in Section 3.3 to perform an effective selectivity-aware decomposition process. Specifically, our decomposition-based and selectivity-aware translation mechanism goes through the sequence of following steps:

– **Identifying the pruning points.** The frequency of the labels of vertices and edges in addition to the frequency of edge connection play a crucial role in our decomposition mechanism. Each vertex label, edge label or edge connection

with low frequency is considered as a pruning point in our relational evaluation mechanism. Hence, given a query graph q, we first check the structure of q against our summary Markov tables to identify the possible *pruning* points. We refer to the number of the identified pruning points by NPP.

- **Calculating the number of partitions.** As we previously discussed, having a sub-graph query q consists of a m vertices and a set of n edges requires $(2m + 4n)$ conjunctive conditions. Assuming that the relational query engine can evaluate up to number of conjunctive condition equal to NC in one query then the number of partitions (NOP) can be simply computed as follows : $(2m + 4n)/NC$
- **Decomposed SQL translation.** Based on the identified number of pruning points (NPP) and the number of partitions (NOP), our decomposition process can be described as follows:
  - *Blindly Single-Level Decomposition.* if $NPP = 0$ then we blindly decompose the sub-graph query q into the calculated number of partition NOP where each partition is translated using our translation template into an intermediate evaluation step S_i. The final evaluation step FES represents a join operation between the results of all intermediate evaluation steps S_i in addition to the conjunctive condition of the sub-graphs connectors. The unavailability of any information about effective pruning points could lead to the result where the size of some intermediate results may contain a large set of non-required graph members.
  - *Pruned Single-Level Decomposition.* if $NPP >= NOP$ then we distribute the pruning points across the different intermediate NOP partitions. Therefore, we ensure a balanced effective pruning of *all* intermediate results, by getting rid of the non-required graph database member early which consequently results in a highly efficient performance. All intermediate results S_i of all pruned partitions are constructed before the final evaluation step FES joins all these intermediate results in addition to the connecting conditions to constitute the final result.
  - *Pruned Multi-Level Decomposition.*if $NPP < NOP$ then we distribute the pruning points across a first level intermediate results of NOP partitions. This step ensures an effective pruning of a percentage of NPP/NOP % partitions. An *intermediate* collective pruned step IPS is constructed by joining all these pruned first level intermediate results in addition to the connecting conditions between them. Progressively, IPS is used as an entry pruning point for the rest $(NOP - NPP)$ non-pruned partitions in a hierarchical multi-level fashion to constitute the final result set. In this situation, the number of *non-pruned* partitions can be reduced if any of them can be connected to one of the pruning points. In other words, each pruning point can be used to prune more than one partition (if possible) to avoid the cost of having any large intermediate results.

Figure 4 illustrates two example of our selectivity-aware decomposition process where the pruning vertices are marked by solid fillings, pruning edges are marked by bold line styles and the connectors between subgraphs are



(a) $NPP > NOP$ (b) $NPP < NOP$

Fig. 4. Selectivity-aware decomposition process

marked by dashed edges. Figure 4(a) represents an example where the number of pruning points is greater than the number of partitions. Figure 4(b) represents an example where one pruning vertex is shared by the different partitioned sub-graphs because the number of pruning points is less than the number of partitions.

– **Selectivity-aware Annotations.** In principal, the main goal of RDBMS query optimizers is to find the most efficient execution plan for every given SQL query. For any given SQL query, there are a large number of alternative execution plans. These alternative execution plans may differ significantly in their use of system resources or response time. Sometimes query optimizers are not able to select the most optimal execution plan for the input queries because of the unavailability or the inaccuracy of the required statistical information. To tackle this problem, we use our statistical summary information to give influencing hints for the query optimizers by *injecting* additional selectivity information for the individual query predicates into the SQL translations of the graph queries. These hints enable the query optimizers to decide the optimal join order, utilizing the most useful indexes and select the cheapest execution plan. In our context, we used the following syntax to pass the selectivity information to the DB2 RDBMS query optimizer:

SELECT *fieldlist* FROM *tablelist*
WHERE P_i SELECTIVITY S_i

Where S_i indicates the *selectivity* value for the query predicate P_i. These selectivity values are ranging between 0 and 1. Lower selectivity values (close to 0) will inform the query optimizer that the associated predicates will effectively prune the number of the intermediate result and thus they should be executed first.

4 Performance Evaluation

In this section, we present a performance evaluation of *GraphREL* as a purely relational framework for storing graph data and processing sub-graph queries. We conducted our experiments using the IBM DB2 DBMS running on a PC with 2.8 GHZ Intel Xeon processors, 4 GB of main memory storage and 200 GB of SCSI secondary storage. In principle, our experiments have the following goals:

1) To demonstrate the efficiency of using *partitioned B-tree* indexes and *selectivity injections* to improve the execution times of the relational evaluation of sub-graph queries.

2) To demonstrate the efficiency and scalability of our decomposition-based and selectivity-aware relational framework for processing sub-graph queries.

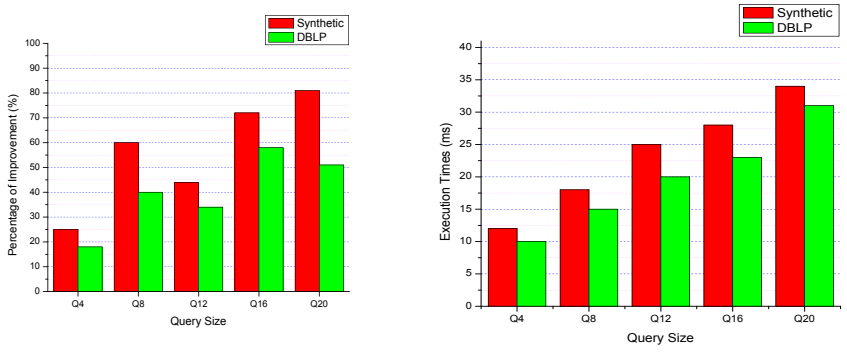
4.1 Datasets

In our experiments we use two kinds of datasets:

1) The real DBLP dataset which presents the famous database of bibliographic information of computer science journals and conference proceedings [1]. We converted the available XML tree datasets into labelled directed graphs by using edges to represent the relationship between different entities of the datasets such as: the ID/IDREF, cross reference and citation relationships. Five query sets are used, each of which has 1000 queries. These 1000 queries are constructed by randomly selecting 1000 graphs and then extracting a connected m edge subgraph from each graph randomly. Each query set is denoted by its edge size as Q_m.
2) A set of synthetic datasets which is generated by our implemented data generator which is following the same idea proposed by Kuramochi et al. in [17]. The generator allows the user to specify the number of graphs (D), the average number of vertices for each graph (V), the average number of edges for each graph (E), the number of distinct vertices labels (L) and the number of distinct edge labels (M). We generated different datasets with different parameters according to the nature of each experiment. We use the notation $DdEeVvLlMm$ to represent the generation parameters of each data set.

4.2 Experiments

The effect of using partitioned B-tree indexes and selectivity injections. Figures 5(a) and 5(b) indicate the percentage of speed-up improvement on the execution times of the SQL-based relational evaluation sub-graph queries using the partitioned B-tree indexing technique and the selectivity-aware annotations respectively. In these experiments we used an instance of a syntactic database that was generated with the parameters $D200kV15E20L200M400$ and a DBLP instance with a size that is equal to 100 MB. We used query groups with different edge sizes of 4,8,12,16 and 20. The groups with sizes of 4 and 8 are translated into one SQL evaluation step, the queries with sizes of 12 and 16 are *decomposed* into two SQL evaluation steps and the queries with of size 20 are *decomposed* into three SQL evaluation steps. The reported percentage of speed up improvements are computed using the formula: $(1 - \frac{G}{C})$ %. In Figure 5(a) G represents the execution time of the SQL execution plans using our defined set of the partitioned B-tree indexes while C represents the execution time of the SQL execution plans using the traditional B-tree indexes. Similarly, in Figure 5(b) G represents the execution time of the SQL execution plans with the injected selectivity annotations while C represents the execution time of the SQL execution plans without the injected selectivity annotations. The results of both experiments confirm the efficiency of both optimization techniques on both data sets. Clearly, using partitioned B-tree indexes has a higher effect on improving the



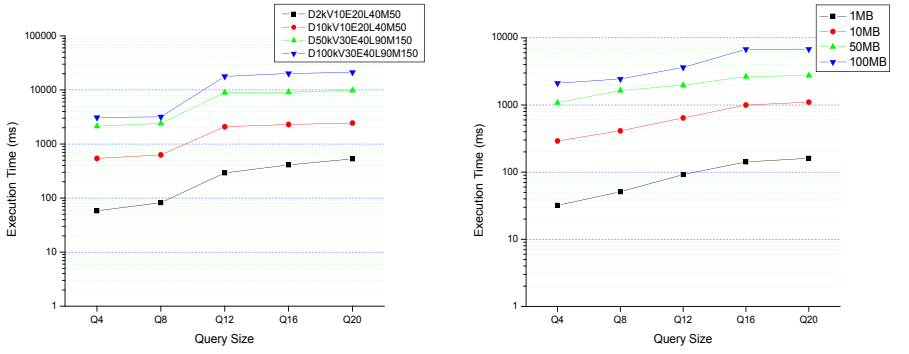
(a) Partitioned B-tree indexes (b) Injection of selectivity annotations

Fig. 5. The *speedup improvement* for the relational evaluation of sub-graph queries using partitioned B-tree indexes and selectivity-aware annotations

execution of the SQL plans because it dramatically reduce the access cost of the secondary storage while selectivity annotations only improve the ability to select the more optimal execution plans. Additionally, the effect on the synthetic database is greater than the effect on the DBLP database because of the higher frequency on the vertex and edge labels and thus reducing the cost of accessing the secondary storage is more effective. The bigger the query size, the more join operations are required to be executed and consequently the higher the effect of both optimization techniques on pruning the cost of accessing the secondary storage and improving the execution times.

Performance and Scalability. One of the main advantages of using a relational database to store and query graph databases is to exploit their well-know scalability feature. To demonstrate the scalability of our approach, we conducted a set of experiments using different database sizes of our datasets and different query sizes. For the DBLP data sets, we used different subsets with sizes of 1,10,50 and 100MB. For the synthetic datasets, we generate four databases with the following parameters: $D2kV10E20L40M50$, $D10kV10E20L40M50$, $D50kV30E40L90M150$ and $D100kV30E40L90M150$. For each dataset, we generated a set of 1000 queries. Figures 6(a) and 6(b) illustrate the average execution times for the SQL translations scripts of the 1000 sub-graph queries.

In these figures, the running time for sub-graph query processing is presented in the Y-axis while the X-axis represents the size of the query graphs. The running time of these experiments include both the *filtering* and *verification* phases. However, on average the running time of the verification phase represents 5% of the total running time and can be considered as have a negligible effect on all queries with small result set. Obviously, the figures show that the execution times of our system performs and scales in a near linear fashion with respect to the graph database and query sizes. This linear increase of the execution time starts to decrease with the very large database sizes (DBLP 100MB and Synthetic $D100kV30E40L90M150$) because of the efficiency of the partitioned B-tree indexing mechanism which



(a) Synthetic Dataset (b) DBLP Dataset

Fig. 6. The scalability of GraphREL

decouples the query evaluation cost from the total database size. . To the best of our knowledge, this work is the first to successful demonstrate a feasible approach of processing large subgraph queries up to 20 vertices and 20 edges over graph databases with such very large sizes up to 100 MB.

5 Related Work

Recently, graph database has attracted a lot of attentions from the database community. In [9], Shasha et al. have presented *GraphGrep* as a path-based approach for processing graph queries. It enumerates all paths through each graph in a database until a maximum length L and records the number of occurrences of each path. An index table is then constructed where each row stands for a path, each column stands for a graph and each entry is the number of occurrences of the path in the graph. The main problem of this approach is that many false positive graphs could be returned in the filtering phase. In addition, enumerating the graphs into a set of paths may cause losing some of their structural features. Some researchers have focused on indexing and querying graph data using data mining techniques such as: *GIndex* [22], *TreePi* [25] and *Tree+Δ* [26]. In these approaches data mining methods are firstly applied to extract the frequent subgraphs (*features*) and identify the graphs in the database which contain those subgraphs. Clearly, the effectiveness of these approaches depends on the quality of the selected features. In addition, the index construction time of these approach requires an additional high space cost and time overhead for enumerating all the graph fragments and performing the graph mining techniques. Moreover, all of these approaches deal with relatively small graph databases where they assume either implicitly or explicitly that the graph databases can completely or the major part of them fit into the main memory. None of them have presented a persistent storage mechanism of the large graph databases. In [27] Jiang et al. proposed another graph indexing scheme called *GString*. GString

approach focus on decomposing chemical compounds into basic structures that have semantic meaning in the context of organic chemistry. In this approach the graph search problem is converted into a string matching problem and specific string indices is built to support the efficient string matching process. We believe that converting sub-graph search queries into sting matching problem could be an inefficient approach specially if the size of the graph database or the sub-graph query is large. Additionally, it is not trivial to extend GString approach to support processing of graph queries in other domain of applications.

6 Conclusions

Efficient sub-graph query processing plays a critical role in many applications related to different domains which involve complex structures such as: bioinformatics, chemistry and social networks. In this paper, we introduced *GraphRel* as a purely relational framework to store and query graph data. Our approach converts a graph into an intuitive relational schema and then uses powerful indexing techniques and advanced selectivity annotations of RDBMSs to achieve an efficient SQL execution plans for evaluating subgraph queries. In principle GraphREL has the following advantages:

1) It employs purely relational techniques for encoding graph data and processing the sub-graph search query. Hence, it can reside on any relational database system and exploits its well known matured query optimization techniques as well as its efficient and scalable query processing techniques.
2) It has *no* required time cost for offline or pre-processing steps.
3) It can handle static and dynamic (with frequent updates) graph databases very well. It is easy to maintain the graph database members and *no* special processing is required to insert new graphs, delete or update the structure of existing graphs.
4) The *selectivity* annotations for the SQL evaluation scripts provide the relational query optimizers with the ability to select the most efficient execution plans and apply an efficient pruning for the non-required graph database members.
5) As we have demonstrated in our experiments, using the well-known *scalability* feature of the relational database engine, GraphREL can achieve a very high scalability and ensure its good performance over very large graph databases and large sub-graph queries.

In the future, we will experiment our approach with other types of graphs and will explore the feasibility of extending our approach to deal with similarity queries and the general subgraph isomorphism problem as well.

Acknowledgments

The author would like to thank Jeffery XY Yu for his valuable comments on earlier drafts of this paper.

References

1. DBLP XML Records, http://dblp.uni-trier.de/xml/
2. Aboulnaga, A., Alameldeen, A., Naughton, J.: Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In: VLDB (2001)
3. Cai, D., Shao, Z., He, X., Yan, X., Han, J.: Community Mining from Multi-relational Networks. In: PPKDD (2005)
4. Clark, J., DeRose, S.: XPath 1.0: XML Path Language (XPath) (November 1999), http://www.w3.org/TR/xpath
5. Cooper, B., Sample, N., Franklin, M., Hjaltason, G., Shadmon, M.: A Fast Index for Semistructured Data. In: VLDB (2001)
6. Yoshikawa, M., et al.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. TOIT 1(1) (2001)
7. Cohen, S., et al.: Scientific formats for object-relational database systems: a study of suitability and performance. SIGMOD Record 35(2) (2006)
8. Botea, V., et al.: PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data. GeoInformatica 12(2) (2008)
9. Giugno, R., Shasha, D.: GraphGrep: A Fast and Universal Method for Querying Graphs. In: International Conference in Pattern recognition (2002)
10. Graefe, G.: Sorting And Indexing With Partitioned B-Trees. In: CIDR (2003)
11. Grust, T.: Accelerating XPath location steps. In: SIGMOD, pp. 109–120 (2002)
12. Grust, T., Mayr, M., Rittinger, J., Sakr, S., Teubner, J.: A SQL: 1999 code generator for the pathfinder xquery compiler. In: SIGMOD (2007)
13. Grust, T., Rittinger, J., Teubner, J.: Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect. In: SIGMOD (2007)
14. Grust, T., Sakr, S., Teubner, J.: XQuery on SQL Hosts. In: VLDB (2004)
15. He, H., Singh, A.: Closure-Tree: An Index Structure for Graph Queries. In: ICDE (2006)
16. Klinger, S., Austin, J.: Chemical similarity searching using a neural graph matcher. In: European Symposium on Artificial Neural Networks (2005)
17. Kuramochi, M., Karypis, G.: Frequent Subgraph Discovery. In: ICDM (2001)
18. Lee, J., Oh, J., Hwang, S.e.: STRG-Index: Spatio-Temporal Region Graph Indexing for Large Video Databases. In: SIGMOD (2005)
19. Sakr, S.: Algebraic-Based XQuery Cardinality Estimation. IJWIS 4(1) (2008)
20. Teubner, J., Grust, T., Maneth, S., Sakr, S.: Dependable Cardinality Forecasts in an Algebraic XQuery Compiler. In: VLDB (2008)
21. Türker, C., Gertz, M.: Semantic integrity support in SQL: 1999 and commercial (object-)relational database management systems. VLDB J. 10(4) (2001)
22. Yan, X., Yu, P., Han, J.: Graph indexing: a frequent structure-based approach. In: SIGMOD (2004)
23. Yuen, L., Poon, C.: Relational Index Support for XPath Axes. In: International XML Database Symposium (2005)
24. Zhang, N., Özsu, T., Ilyas, I., Aboulnaga, A.: FIX: Feature-based Indexing Technique for XML Documents. In: VLDB (2006)
25. Zhang, S., Hu, M., Yang, J.: TreePi: A Novel Graph Indexing Method. In: ICDE (2007)
26. Zhao, P., Xu Yu, J., Yu, P.: Graph indexing: tree + delta = graph. In: VLDB (2007)
27. Zou, L., Chen, L., Xu Yu, J., Lu, Y.: GString: A novel spectral coding in a large graph database. In: EDBT (2008)