An Efficient Features-Based Processing Technique for Supergraph Queries

Sherif Sakr and Ghazi Al-Naymat

School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
{ssakr,ghazi}@cse.unsw.edu.au

Abstract. Graphs are widely used for modelling complicated data such as social networks, chemical compounds, protein interactions, XML documents and multimedia databases. To be able to effectively understand and utilize any collection of graphs, a graph database that efficiently supports elementary querying mechanisms is crucially required. Supergraph query is an important type of graph queries which has many applications in practice. Given a graph database $D$, the answer set of a supergraph query $q$ is computed by retrieving all graphs in $D$ which are fully contained in $q$. A primary challenge in computing the answers of graph queries is that pair-wise comparisons of graphs are usually hard problems. For example, subgraph isomorphism is known to be NP-complete. Clearly, the success of any graph database application is directly dependent on the efficiency of the graph indexing and query processing mechanisms. In this paper, we study the problem of using the relational infrastructure to achieve an efficient evaluation of supergraph queries. We rely on an effective and efficient layer of features-based summary structures, called graph features knowledge, to reduce the required number of pair-wise graph comparisons and boost the efficiency of query processing. Finally, we conduct an extensive set of experiments on real and synthetic data sets to demonstrate the efficiency and the scalability of our approach.

1 Introduction

Graphs are among the most complicated and general form of data structures. Recently, graphs have been widely used to model many complex structured and schemaless data such as semantic web [20], protein networks [16], social networks [4] and chemical compounds [18]. Retrieving related graphs containing a query graph from a large graph database is a key performance issue in all of these graph-based applications. Therefore, the success of any graph database application is directly dependent on the efficiency of the graph indexing and query processing mechanisms. In principle, there are two main types of graph queries:

1. **Subgraph queries:** This category searches for a specific pattern in the graph database. Formally, given a graph database $D = \{g_1, g_2, ..., g_n\}$ and a subgraph query $q$, the query answer set $A = \{g_i | q \subseteq g_i, g_i \in D\}$. 
2. **Supergraph queries**: This category searches for the graph database members of which their whole structures are contained in the input query. Formally, given a graph database \( D = \{g_1, g_2, \ldots, g_n\} \) and a supergraph query \( q \), the query answer set \( A = \{g_i|q \supseteq g_i, g_i \in D\} \).

Clearly, it is an inefficient and a very time consuming task to perform a sequential scan over the entire graph database \( D \) to check whether each graph database member \( g_i \) belongs to the answer set of a graph query \( q \). Hence, there is a clear necessity to build graph indices in order to improve the performance of processing graph queries. In recent years, the problem of subgraph query processing has attracted much research attention and many algorithms have been proposed [7, 11, 15, 17, 26, 29, 30, 32, 33]. Although the importance of the supergraph query processing problem in many practical application domains (e.g., chemical compounds), it has not been extensively considered in the research literature. Few approaches have been presented to deal with this problem [6, 31].

Relational database management systems (RDBMSs) have repeatedly been shown to be highly efficient, scalable and successful in hosting types of data which have formerly not been anticipated to be stored inside relational databases such as: complex objects [9, 24], spatio-temporal data [10] and XML data [8, 14]. In addition, RDBMSs have shown its ability to handle vast amounts of data very efficiently using its powerful indexing mechanisms.

In a previous work [21], we presented our approach for employing the powerful features of the relational infrastructure to achieve an efficient and scalable processing for subgraph queries. In this paper we extend our approach to efficiently execute the supergraph queries. The proposed approach involves utilizing a layer of compact feature-based summary structures of the underlying graph databases, called **graph features knowledge**, to reduce the required number of pair-wise graph comparisons and boost the efficiency of query processing. In summary, we made the following contributions:

1. We systematically investigate the use of the relational infrastructure for indexing graph databases and processing supergraph search queries. We propose an approach for extracting descriptive features for the underlying graph database members. The extracted features are then used to effectively filter the candidate graph database members and to get rid of many false positives.
2. We present a novel graph features-based mechanism for minimizing the number of subgraph isomorphism testings which are required during a relational processing for supergraph queries. The mechanism relies on a compact features summary structure of the underlying graph database.
3. 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 depicts an overview of our relational approach for storing graph databases. Section 4 describes the components of the summary layer of graph features knowledge. Section 5 presents our mechanism to utilize the graph features knowledge in order to reduce the number of subgraph isomorphism testings and improve the relational processing for supergraph
queries. We evaluate our method by conducting an extensive set of experiments which are described in Section 6. The related work is surveyed in Section 7 before we conclude the paper in Section 8.

2 Preliminaries

This paper focuses on the directed-labeled and connected simple graphs (we refer to them as graphs in the rest of the paper). However, the algorithms proposed in the paper can be easily extended to other kinds of graphs.

Definition 1. [Labeled Graph] A labeled graph \( g \) is denoted as \( (V, E, L_v, L_e, F_v, F_e) \) 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; \( L_e \) is the set of edge labels; \( F_v \) is a function \( V \to L_v \) that assigns labels to vertices and \( F_e \) is a function \( E \to L_e \) that assigns labels to edges.

Definition 2. [Graph Database] A graph database \( D \) is a collection of member graphs \( g_i \), where \( D = \{g_1, g_2, ..., g_n\} \).

In principle, there are two main types of graph databases. The first type consists of a few number of very large graphs such as the web graph and social networks. The second type consists of a large set of small graphs such as chemical compounds and biological pathways. This paper focuses on the second type of graph databases.

Definition 3. [Subgraph Isomorphism] Let \( g_1 = (V_1, E_1, L_{v1}, L_{e1}, F_{v1}, F_{e1}) \) and \( g_2 = (V_2, E_2, L_{v2}, L_{e2}, F_{v2}, F_{e2}) \) be two graphs, \( g_1 \) is defined as a graph isomorphism to \( g_2 \), if and only if there exists at least one bijective function \( f : V_1 \to V_2 \) such that: 1) for any edge \( uv \in E_1 \), there is an edge \( f(u)f(v) \in E_2 \). 2) \( F_{v1}(u) = F_{v2}(f(u)) \) and \( F_{v1}(v) = F_{v2}(f(v)) \). 3) \( F_{e1}(uv) = F_{e2}(f(u)f(v)) \).

The subgraph search and supergraph search problems in graph databases are formulated as follows.

![Fig. 1. An illustrating example of supergraph search queries in graph databases](image-url)
Definition 4. [Subgraph Search] Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a subgraph query $q$, the query answer set $A = \{g_i | q \subseteq g_i, g_i \in D\}$.

Definition 5. [Supergraph Search] Given a graph database $D = \{g_1, g_2, \ldots, g_n\}$ and a supergraph query $q$, the query answer set $A = \{g_i | q \supseteq g_i, g_i \in D\}$.

Figure 1 provides an illustrative example for the supergraph search problem in graph databases which is the main focus of this paper.

### 3 Relational Encoding of Graph Databases

The starting point of our approach for processing supergraph 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 labeled graphs. In this mapping scheme, each graph database member $g_i$ is assigned a
unique identity \textit{graphID}. Each vertex is assigned a sequence number (\textit{vertexID}) inside its graph. Each vertex is represented by one tuple in a single table (\textit{Vertices table}) which stores all vertices of the graph database. Each vertex is identified by the \textit{graphID} to which the vertex belongs and the \textit{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 (\textit{Edges table}) where each edge is represented by a single tuple in this table. Each edge tuple describes the graph database member to which the edge belongs, the id of the \textit{source vertex} of the edge, the id of the \textit{destination vertex} of the edge and the edge label. Therefore, the relational storage scheme of our \textit{Vertex-Edge} mapping is described as follows:

- \textit{Vertices}(\textit{graphID}, \textit{vertexID}, \textit{vertexLabel})
- \textit{Edges}(\textit{graphID}, s\textit{vertex}, d\textit{vertex}, \textit{edgeLabel})

Figure 2 illustrates an example of the \textit{Vertex-Edge} relational mapping scheme of a sample graph database. Figure 3 illustrates the encoding of a supergraph query \textit{q}. Evaluating the sample supergraph query against the sample graph database should return an answer set that contains the graph database members \{\textit{g}_1, \textit{g}_2\}. Let us assume a supergraph query \textit{q} with \textit{M} nodes and \textit{N} edges. The answer set of this query would thus contain all graph database members \textit{g}_i with \textit{x} nodes and \textit{y} edges where \(x \leq M, y \leq N, x > y\) and structurally contained in \textit{q}. Apparently, using a straightforward SQL-based approach for evaluating the answer of this type of queries is quite difficult and expensive. A traditional filtering-and-verification can go through the following steps:

- **Filtering phase:** This phase can be represented by a \textit{distinct union} statement between a set of SQL-based intermediate results which represent the set of graph database members for each pair of integer values \textit{x} and \textit{y} where \(x \leq M, y \leq N, x > y\) using the following SQL-based template:

  1. \textbf{SELECT} \textit{V}_1.\textit{graphID}  
  2. \textbf{FROM} \textit{Vertices} as \textit{V}_1, ..., \textit{Vertices} as \textit{V}_x, \textit{Edges} as \textit{E}_1, ..., \textit{Edges} as \textit{E}_y,  
  3. \textit{qVertices} as \textit{QV}_1, ..., \textit{qVertices} as \textit{QV}_x, \textit{qEdges} as \textit{QE}_1, ..., \textit{qEdges} as \textit{QE}_y  
  4. \textbf{WHERE}  
  5. \forall i=2 (\textit{V}_i.\textit{graphID} = \textit{V}_i.\textit{graphID})  
  6. AND \forall j=1 (\textit{V}_j.\textit{graphID} = \textit{E}_j.\textit{graphID})  
  7. AND \forall i=1 (\textit{V}_i.\textit{vertexLabel} = \textit{QV}_i.\textit{vertexLabel})  
  8. AND \forall j=1 (\textit{E}_j.\textit{edgeLabel} = \textit{QE}_j.\textit{edgeLabel});

where each referenced table \textit{V}_i (Line number 2) represents an instance from the table \textit{Vertices} and maps the information of one vertex of the set of vertices of the supergraph query \textit{q} represented by the tables \textit{QV}_i. Similarly, each referenced table \textit{E}_j represents an instance from the table \textit{Edges} and maps the information of one edge of the supergraph query \textit{q} represented by the tables \textit{QE}_j. Line number 5 of the SQL translation template represents a set of \(x - 1\) conjunctive conditions to ensure that all queried vertices belong to the same graph. Similarly, Line number 6 of the SQL translation template represents a set of \(y\) conjunctive conditions to ensure that all queried
edges belong to the same graph of the queried vertices. Lines number 7 and 8 represent the set of conjunctive predicates of the vertex and edge labels respectively. The distinct union statement is required to get rid of duplicate graphs which can appear in different intermediate results because they contain subgraphs with different sizes of the supergraph query $q$.

- **Verification phase**: This phase needs to ensure that each candidate graph database member is fully contained in the supergraph query $q$ and does not have any extra non-filtered nodes or edges which are not considered in the SQL-based template of the filtering phase. An obvious problem of the straightforward SQL translation template of the filtering phase in its current form is that it will return many false positive graphs because it considers any graph database member that have a subgraph of the supergraph query $q$ as a candidate result. The main reason behind this is that the filtering predicates of the SQL template is quite limited as it does not consider any information about the features or the topology of the supergraph query. Moreover, it has no mean check if these filtered graphs are fully contained in the supergraph query $q$ or not. In the next section, we will describe our approach to tackle this problem and dramatically improve the performance and the effectiveness of the SQL-based filtering phase of supergraph queries. Clearly, improving the effectiveness of the filtering phase will consequently reduce the cost of the verification phase.

## 4 Graph Features Knowledge

In order to improve the efficiency of the SQL-based approach for evaluating supergraph queries, we employ the idea of automatic creation and usage of higher-level data about data (metadata) [3, 5], here referred to as graph features knowledge. This information is used during the query optimization and execution phases to improve the performance of query evaluation. We propose two metadata layers: graph descriptors and aggregate graph descriptor.

### Graph Descriptors

In the first layer, we maintain a description record about the features of each graph database member in the graph database. Each graph descriptor includes the following components:

- **No. Nodes (NN)**: represents the total number of nodes in the graph.
- **No. Edges (NE)**: represents the total number of edges in the graph.
- **Unique Node Labels (NL)**: represents the number of unique node labels in the graph.
- **Unique Edge Labels (EL)**: represents the number of unique edge labels in the graph.
- **Maximum in-Node Degree (MID)**: represents the maximum number of incoming edges for a node in the graph.
- **Maximum out-Node Degree (MOD)**: represents the maximum number of outgoing edges for a node in the graph.
- **Maximum io-Node Degree (MD)**: represents the maximum number of total number of incoming and outgoing edges for a node in the graph.
Graph Descriptors

- **Maximum Path Length (MPL)**: where a path length in a graph represents a sequence of consecutive edges and the length of the path is the number of edges traversed.

Figure 4 illustrates an example of a graph descriptor table for a sample graph database. In fact, the components of the graph descriptor can vary from one database to another based on the characteristics of the underlying graph database members. For example, if there is a guarantee that each node or edge in has a unique label within a graph then we can avoid storing the components which represent the number of unique node and edge labels in each graph. It is also possible to add some more descriptive fields such as: the number of occurrences of the maximum path length or the number of occurrences of the maximum (in/out/io)-node degrees.

On the one hand, the more indexed features the higher the pruning power of the filtering phases. However, it also increases the space cost. On the other hand, the less indexed features lead to poor pruning power in the filtering process and consequently resulting in a performance hit in the query processing time.

In general, computing the components of our graph descriptor (except the maximum path length which is recursive in nature) can be achieved in a straightforward way using SQL-based aggregate queries. For example, computing the number of nodes graph descriptor component (NN) can be represented using the following SQL query:

```sql
SELECT graphID, count(*) as NN FROM Vertices GROUP BY graphID;
```

Similarly, computing the number of edges descriptor component (NE) can be represented using the following SQL query:

```sql
SELECT graphID, count(*) as NE FROM Edges GROUP BY graphID;
```
Aggregate Graph Descriptors In the second layer of our graph features knowledge, we maintain a compact aggregated version of the graph descriptors table in the following form:

\[(BID, NN, NE, NL, EL, MID, MOD, MD, MPL, graphCount)\]

In fact, the main idea behind that aggregate graph descriptors is that it groups the graph database members with the same features into one bucket identified by its bucket ID \((BID)\). Hence, it represents a guide summary of the features of the graph database members [12]. The size of this aggregated version is usually, by far, less than the size of the descriptor table and can easily fit into the main memory. Based on a defined threshold \(t\), we create an auxiliary inverted list [2] of all the related graph database members for each summary bucket with a number of graph database members \((graphCount)\) which is less than \(t\). The main goal of these inverted lists is to allow for cheap and quick access to the set of the few graphs which share the same features of that bucket.

5 Features-based SQL Evaluation of Supergraph Queries

In this section, we describe how to use the two layers of graph features knowledge (Section 4) to improve the SQL-based evaluation of supergraph queries. Figure 5 gives an overview of our approach for features-based SQL translation of supergraph queries which goes through the following main steps:

1. Database Preprocessing: In this step we analyze the underlying graph database to build the components of the layer of the graph features knowledge \((GFK)\).
2. **Determining The Pruning Strategy**: The query processor recognize the features of the input graph query and probes the components of the GFK with the query features to identify the pruning strategy of the filtering phase.

3. **Filtering**: It uses the outcome decisions from probing the GFK to apply an effective and efficient pruning strategy for filtering the graph database members, get rid of many false positive graphs (graphs that are not possible to belong that the query answer) and produce a list of graph database members which are candidate to constitute the query result.

4. **Verification**: It validates each candidate graph to check if it really belongs to the answer set of the query graph.

Clearly, the query response time consists of the filtering time and the verification time. The less candidates after the filtering, the faster the query response time. Hence, pruning power in the filter process is critical to the overall query performance. Therefore, Given a supergraph query $q$, the query evaluation process starts by computing the graph query descriptor ($qd$) then using the aggregated graph descriptors layer ($AGD$), we identify the set of buckets where the features of each bucket are fully contained in the graph query. In fact, the graph query descriptor is considered to be fully contained in a bucket if and only if the value of each component in the query descriptor is greater than or equal to its associated component in the bucket. This matching process is defined as follows.

**Definition 6.** [Bucket Match] Given an aggregate graph descriptor $AGD = \{b_1, b_2, ..., b_n\}$ and a supergraph query descriptor $qd$, the matched bucket set $MB = \{b_i | qd \subseteq b_i, b_i \in AGD\}$ where $qd \subseteq b_i$ iff $qd.NN \geq b_i.NN \land qd.NE \geq b_i.NE \land qd.NL \geq b_i.NL \land qd.EL \geq b_i.EL \land qd.MID \geq b_i.MID \land qd.MOD \geq b_i.MOD \land qd.MD \geq b_i.MD \land qd.MPL \geq b_i.MPL$.

Figure 6 illustrates an example of the matching process of a supergraph query descriptor against the buckets of an aggregate graph descriptors. Based on this matching process, the supergraph query evaluation strategy is determined as depicted in the steps of Algorithm 1 and Algorithm 2. The strategies of our supergraph query evaluation process are described as follows.
Require: $q$: Supergroup Query,  
$AD$: Aggregate Descriptors,  
$t$: Threshold of Bucket Size with an Inverted List,  

Ensure: SupStrategy: integer

$mb$: Array of Matched Aggregated Descriptors Entries

1: $qd := \text{ComputeGraphQueryDescriptor}(q)$;
2: $mb := \text{IdentifySupMatchedBuckets}(qd, AD)$;
3: if $mb.length = 0$ then
4:     SupStrategy := 1;
5:     return ;
6: end if
7: SupStrategy := 2;
8: for $i = 1$ to $mb.length$ do
9:     if $mb[i].graphcount > t$ then
10:        SupStrategy := 3;
11:        return ;
12: end if
13: end for

Algorithm 1: IdentifySupStrategy: Determines the supergraph query evaluation strategy.

- **Case SupStrategy1**: If the count of the matched buckets is equal to zero, we return an empty result set because none of the graph database members has features which are fully contained in the supergraph query descriptor.
- **Case SupStrategy2**: If the count of the matched buckets is greater than zero and the size of each matched bucket ($b_i.graphcount$) is less than the defined threshold $t$ then we unfold their associated inverted lists to retrieve and verify their related graph database members in order to compute the answer set of the supergraph query $q$.
- **Case SupStrategy3**: This case handles the situation when we have a number of matched buckets with a size less than the threshold $t$ and a number of matched buckets with a size greater than the threshold $t$. Similar to SupStrategy2, for each matched bucket with a size less than $t$, we convert the supergraph search problem into a subgraph search problem. We use the features of the matched bucket ($mb_s$) to apply a backtracking step which uses the features of the matched bucket to extract from the supergraph query ($q$) all subgraphs ($sqg$) with exactly the same features. We then treat each extracted subgraph ($eqg$) as a subgraph query where we issue a features-based SQL query using the translation template depicted in Figure 7 to retrieve all graph database members that fulfill the features of both the extracted subgraph ($eqg$) and the matched bucket ($mb_s$). Clearly, this backtracking step dramatically reduces the search space. In this template, the number of the instantiated instances of the $Vertices$ and $Edges$ tables are determined by the values
Require: \( D \): Graph Database, \( q \): Supergraph Query, 
\( GD \): Graph Descriptors, 
\( AD \): Aggregate Descriptors, 
\( t \): Threshold of Bucket Size with an Inverted List,

Ensure: \( ASGQ \): Answer set of the supergraph query \( (q) \)

1: \textbf{IdentifySupStrategy}(q, \( AD \), \( t \), \( SupStrategy \), \( mb \)) ;
2: if \( SupStrategy = 1 \) then
3: \hspace{1em} return empty ;
4: end if
5: if \( SupStrategy = 2 \) then
6: \hspace{1em} for \( i = 1 \) to \( mb.length \) do
7: \hspace{2em} for all \( g \in mb[i].\text{InvertedList}.\text{graphs} \) do
8: \hspace{3em} if Verify\((g, q)\) then
9: \hspace{4em} \( ASGQ.\text{ADD} (g) \) ;
10: \hspace{3em} end if
11: \hspace{2em} end for
12: \end for
13: return \( ASGQ \) ;
14: end if
15: if \( SupStrategy = 3 \) then
16: \hspace{1em} for \( i = 1 \) to \( mb.length \) do
17: \hspace{2em} if \( mb[i].\text{graphcount} \leq t \) then
18: \hspace{3em} for all \( g \in mb[i].\text{InvertedList}.\text{graphs} \) do
19: \hspace{4em} if Verify\((g, q)\) then
20: \hspace{5em} \( ASGQ.\text{ADD} (g) \) ;
21: \hspace{4em} end if
22: \hspace{3em} end for
23: \hspace{2em} else
24: \hspace{3em} \( sqg:= \text{ExtractFeaturedSubgraphs}(q, mb[i]) \)
25: \hspace{3em} for all \( eqg \in sqg \) do
26: \hspace{4em} \( cg:= \text{SQLFeaturedFilter}(D, eqg, GD, mb[i]) \) ;
27: \hspace{4em} for all \( g \in cg \) do
28: \hspace{5em} if Verify\((g, q)\) then
29: \hspace{6em} \( ASGQ.\text{ADD} (g) \) ;
30: \hspace{5em} end if
31: \hspace{4em} end for
32: \hspace{3em} end for
33: \hspace{2em} end if
34: \end for
35: return \( ASGQ \) ;
36: end if

\textbf{Algorithm 2}: Features-Based Evaluation of Supergraph Queries
SELECT DISTINCT $V_1$.graphID
FROM Vertices as $V_1$, ..., Vertices as $V_{mb_s}.NN$,
Edges as $E_1$, ..., Edges as $E_{mb_s}.NE$,
GraphDescriptors as GD
WHERE
$V_1$.graphID = GD.graphID
AND GD.NN = $mb_s$.NN
AND GD.NE = $mb_s$.NE
AND GD.NL = $mb_s$.NL
AND GD.EL = $mb_s$.EL
AND GD.MID = $mb_s$.MID
AND GD.MOD = $mb_s$.MOD
AND $∀_{i=2}^{m}$($V_1$.graphID = $V_i$.graphID)
AND $∀_{j=1}^{n}$($E_j$.graphID = $E_i$.graphID)
AND $∀_{j=1}^{n}$($V_j.vertexLabel$ = $QV_i.vertexLabel$
AND $E_j.sVertex$ = $V_f.vertexID$
AND $E_j.dVertex$ = $V_f.vertexID$);

Fig. 7. SQL Template for Features-Based Subgraph Search

of the number of nodes ($NN$) and the number of edges ($NE$) components in the identified bucket ($mb_s$) respectively. Each referenced table $V_i$ and $E_j$ maps the information of vertices and edges of the extracted subgraph of the supergraph query $q$ respectively. Lines number 7,8,9,10,11,12 and 13 use the graph descriptors information to retrieve only the graph database members which satisfy the features of the identified buckets. Lines number 14 and 15 of the SQL translation template represents a set of conjunctive conditions to ensure that all queried vertices and edges belong to the same graph respectively. Similarly, Lines number 16 and 17 represent the set of conjunctive predicates of the vertex and edge labels respectively. Lines number 18 and 19 represents the topology conditions of the edges between the mapped vertices where $f$ is the mapping function between each subgraph query vertex and its associated vertices table instance $V_i$.

In contrast to the verification phase described in Section 3, the features-based verification phase does not need to ensure that the candidate graph database members are fully contained in the supergraph query $q$ and do not have any extra non-filtered nodes or edges. This is already guaranteed by the conditions of the filtering phase. In principle, this phase is optionally required if and only if more than one vertex of the set of subgraph query vertices have the same label. In this case, it verifies that each vertex in the set of filtered vertices for each candidate graph database member is distinct. This can be easily achieved using their vertexID. Although the fact that the conditions of the verification process could be injected into the SQL translation template
of the features-based filtering phase, we found that it is more efficient to avoid the cost of performing these conditions over each graph database members by delaying their processing (if required) to a separate phase after pruning the candidate list.

6 Performance Evaluation

In this section, we present an extensive experimental study to demonstrate the efficiency and scalability of our proposed techniques. We conducted our experiments using the PostgreSQL RDBMS running on a PC with 3.2 GHZ Intel Xeon processors, 4 GB of main memory storage and 250 GB of SCSI secondary storage. In our experiments we used the following two kinds of datasets:

1) The real AIDS antiviral screen dataset (NCI/NIH) which is publicly available on the website of the Developmental Therapeutics Program [1]. For this dataset, different query sets are used, each of which has 1000 queries. These 1000 queries are constructed by randomly selecting 1000 graphs from the antiviral screen dataset and then extracting a connected $m$ edge subgraph from each graph randomly. Each query set is denoted by its edge size ($m$) as $Q_m$.

2) A set of synthetic datasets which is generated by our implemented data generator that follows the same idea proposed by Kuramochi and Karypis in [19]. 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. For example a data set $D5000V10E20L40M50$ means that the data set contains 5000 graphs, the average number of vertices in each graph is 10, the average number of edges in each graph is 20, there are 40 possible vertices labels and there are 50 possible edge labels.

Figure 8 shows the offline preprocessing cost of different datasets which are used in our experiments. Figure 8(a) shows the disk storage size of the graph features knowledge. The results of the experiments show that the storage size increases linearly w.r.t the size of the graph databases (more specifically the number of graph database members). Obviously, the main reason behind this is that each graph database member requires a new record in the graph descriptor table. In fact, the size of the aggregate graph descriptors is very small and could be neglected. The size increase of this aggregate component is not affected by the increase in the number of graph database members. However, it is affected by the number of unique features of the graph database members.

One of the main advantages of using a relational database to store and query graph databases is to exploit their well-know scalability feature. Figure 9 shows the query performances of our features-based SQL evaluation of supergraph queries. Figure 9(a) illustrates the average execution times for the features-based SQL translations of 1000 supergraph queries with different sizes on graph
databases with different characteristics. In this figure, the running time for the supergraph query processing is presented in the Y-axis, the X-axis represents the different query sizes. The running time of these experiments include both the \textit{filtering} and \textit{verification} phases. However, on average the running time of the verification phase represents 3\% of the total running time and can be considered with negligible effect on all queries with small result sets.

In practical, the SQL translation template of the \textit{filtering} phase involves join operations between several instances of the encoding relations. Hence, for the buckets with large values for the \textit{NN} and \textit{NE} components, some of the relational query engines may fail to execute their SQL translations because they are too long or too complex (this does not mean they must consequently be too expensive). Therefore, we rely on our selectivity-aware decomposition algorithm previously introduced in [21] to tackle this problem. Since the main focus of this paper on graph databases which contain large sets of small graphs, the occurrences of these patterns are not frequent in our experiments. In labeled graphs, it is generally the case that the number of \textit{distinct} vertices and edges labels are far less than the number of vertices and edges respectively. In order to improve the performance of the filtering phase, we created a partitioned B-tree indexes [13] over the vertices and edges labels to reduce the access costs of the secondary storage. The experiment results show that the execution times of our approach performs and scales in a near linear fashion with respect to the graph database and query sizes.

Although the size of the graph features knowledge can be considered as an additional overhead, it is really worth to pay as it significantly improves the performance of query evaluation especially when considering its relatively cheap and scalable construction time (Figure 8(b)). Figure 9(b) indicates the percentage of speed-up improvement on the execution times of the features-based SQL evaluation of supergraph queries in comparison to the straightforward approach.
The reported percentage of speed up improvements are computed using the formula: \((1 - \frac{F}{C})\%\) where \(F\) represents the execution time our features-based SQL translation while \(C\) represents the execution time of the straightforward SQL translation. Clearly, using our approach improves the query performance with several order of magnitudes. The percentage of improvement increases linearly with respect to the increase in the database and query sizes due to the effectiveness of the filtering phase of our features-based approach to get rid of many of the false positives. In particular, we specify the reasons behind the improvement of the features-based evaluation of supergraph queries as follows:

- It effectively reduces the size of the intermediate results. Matching the information of the graph descriptors with the features of the identified buckets dramatically reduces the size of each intermediate result and gets rid of many false positive candidates.

- It avoid having an intermediate result for each pair of values \(x\) and \(y\) where \(x > y\), \(x \leq\) the number of the query nodes and \(y \leq\) the number of query edges (Section 3). Instead, the intermediate results are determined by the extracted subgraph from the supergraph queries according to the features of the matched buckets which are fully contained in the features of the supergraph query. Thus, these intermediate results are ensured to very effectively pruned. Moreover, evaluating the intermediate results for some of the matched buckets can be avoided if their sizes (\(\text{graphCount}\)) are less than the defined threshold \(t\).

- The intermediate results are disjoint and duplicate-free. Each candidate graph database member can belong to only one of the intermediate result based on the matching of its graph descriptor with the unique features of each identified bucket. Disjointness and duplicate-free play an effective role in reducing the total number of records in the intermediate results.
7 Related Work

Recently, graph database has attracted a lot of attention from the database community. There are many graph indexing techniques have been recently proposed to deal with the problem of subgraph query processing [21, 23, 29, 32]. They can be mainly classified into the following two main approaches:

- Non Mining-Based Graph Indexing Techniques: These techniques focus on indexing the whole constructs of the graph database instead of indexing only some selected features (GraphGrep [11], Closure-Tree [15], gString [17], GraphREL [21], GDIndex [26]). The main criticisms of these approaches are: 1) They can be less effective in their pruning power. 2) They may need to conduct expensive structure comparisons in the filtering process and thus degrades the filtering efficiency. Therefore, these techniques need to employ efficient filtering and pruning mechanisms to overcome these limitations.

- Mining-Based Graph Indexing Techniques: These techniques apply graph mining methods [25, 27, 28] to extract some features from the graph database members (gIndex [29], FG-Index [7], TreePi [30], Tree-δ [32]). An inverted index is created for each feature. Answering a subgraph query $q$ is achieved through two steps: 1) Identifying the set of features of the subgraph query. 2) Using the inverted index to retrieve all graphs that contain the same features of $q$. The rationale behind this type of query processing techniques is that if some features of graph $q$ do not exist in a data graph $G$, then $G$ cannot contain $q$ as its subgraph. Clearly, the effectiveness of these filtering methods depends on the quality of mining techniques to effectively identify the set of features.

In [21] Sakr has presented a purely relational framework for processing graph queries named GraphREL. The main optimization technique of GraphREL is based on the observation that the size of the intermediate results dramatically affects the overall evaluation performance of SQL scripts [22]. Therefore, GraphREL keeps statistical information about the less frequently existing nodes and edges in the graph database. For a graph query $q$, the maintained statistical information is used to identify the highest pruning point on its structure (nodes or edges with very low frequency) to firstly filter out, as many as possible, the false positives graphs that are guaranteed to not exist in the final results before passing the candidate result set to an optional verification process.

Few approaches have been presented to deal with the supergraph search problem. Chen et al. [6] have presented an approach named cIndex. The indexing unit of this approach is the subgraphs which are extracted based on the rarity of their occurrence in historical query graphs. Sometimes, the extracted subgraphs are very similar to each other. cIndex uses a redundancy-aware selection mechanism to sort out the most significant and distinctive contrast subgraphs between database graphs and queries. Zhang et al. [31] proposed an approach for a compact organization of graph database members named GPTree. In this approach, all of the graph database members are stored into one graph where
the common subgraphs are stored only once. Based on the containment relationship between the sets of the extracted features, a mathematical approach is used to determine the ordering of the feature set which can reduce the number of subgraph isomorphism tests during query processing.

8 Conclusion

Efficient supergraph query processing plays a critical role in many applications domains which involve complex structures such as: bioinformatics and chemical compounds. In this paper, we studied the problem of using the relational infrastructure to achieve an efficient evaluation of supergraph queries. Our approach converts a graph into an intuitive relational schema and uses a compact summary structures of the underlying graph databases to achieve efficient SQL execution plans for evaluating supergraph queries. We presented an algorithm for features-based SQL-based evaluation of supergraph queries. The proposed algorithm reduces the required number of pair-wise graph comparisons and boosts the efficiency of relational-based query processing. We evaluated the performance of our approach with extensive experiments using real and synthetic datasets. The results show that our approach is up to several orders of magnitude faster than the straightforward solution.

References

27. X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In ICDM, 2002.