Noname manuscript No.
(will be inserted by the editor)

Fast Graph Query Processing with a Low-Cost Index

James Cheng · Yiping Ke · Ada Wai-Chee Fu · Jeffrey Xu Yu

Received: date / Accepted: date

Abstract This paper studies the problem of processing *supergraph queries*, that is, given a database containing a set of graphs, find all the graphs in the database of which the query graph is a *supergraph*. Existing works usually construct an index and perform a *filtering-and-verification* process, which still requires many subgraph isomorphism testings. There are also significant overheads in both index construction and maintenance. In this paper, we design a graph querying system that achieves both fast indexing and efficient query processing. The index is constructed by a simple but fast method of extracting the commonality among the graphs, which does not involve any costly operation such as graph mining. Our query processing has two key techniques, *direct inclusion* and *filtering*. Direct inclusion allows partial query answers to be included directly without candidate verification. Our filtering technique further reduces the candidate set by operating on a much smaller projected database. Experimental results show that our method is significantly more efficient than the existing works in both indexing and query processing, and our index has a low maintenance cost.

James Cheng
School of Computer Engineering, Nanyang Technological University, Singapore
E-mail: jamescheng@ntu.edu.sg

Yiping Ke (Corresponding Author)
Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong.
E-mail: ypke@se.cuhk.edu.hk

Ada Wai-Chee Fu
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong.
E-mail: adafu@cse.cuhk.edu.hk

Jeffrey Xu Yu
Department of Systems Engineering and Engineering Management, The Chinese University of Hong Kong, Hong Kong.
E-mail: yu@se.cuhk.edu.hk

1 Introduction

Graph query processing [13,20,5,10,16,21,3,23,2,24,12,22] has attracted much attention in recent years thanks to the increasing popularity of graph databases in various application domains. Existing research on graph query processing is conducted mainly on two types of graph databases. The first one is large graphs such as social networks [11,14]. The second one is *transaction graph databases* that consist of a set of relatively smaller graphs. Transaction graph databases are prevalently used in scientific domains such as chemistry [9], bioinformatics [6], etc.

We focus on query processing in transaction graph databases. There are two types of queries commonly studied in the literature. One is *subgraph query* [13, 20,5,10,16,21,3,23,24,12], which is to retrieve all the graphs in the database such that a given query graph is a *subgraph* of them. The other one is *supergraph query* [2,22], which is to retrieve all the graphs in the database such that the query graph is a *supergraph* of them.

This paper focuses on supergraph query, which has a wide range of applications in chemistry informatics (super-structure search), computer vision (object recognition and shape matching), social science (insider threat detection), etc. For example, in chemical super-structure search, the database contains a set of compound structures with known chemical/biological properties or functionalities. The queries are compound structures with larger graph size. For each query, the system returns a set of compound structures that comprise or make up the query, which are further used to determine possible

chemical properties of the compound query. Some other detailed examples of supergraph query, such as examples in computer vision applications, are also given in [2].

The problem has three challenges. First, *the database can be very large*. For example, there are currently over 26 million chemical compound structures in PubChem Compounds Database[1], which is maintained by the U.S. government. Second, *the database can be highly dynamic*. In PubChem Compounds Database, over 6 million new compounds have been added in the previous year and the database is continuously growing. Third, *the query workload can be bulky and come in a high speed*. For example, PubChem implements a search engine for the super-structure search[2] and queries may come in as a fast-speed stream.

Since processing supergraph queries involves the NP-complete subgraph isomorphism test, sequential scan of the database is prohibited and existing solutions use indexes. However, to address the three challenges highlighted above, we need an index that is: *(1) efficient to construct for a large database, (2) efficient to maintain over dynamic updates, and (3) efficient to process bulky query workloads.* The existing indexes are inadequate due to the following two deficiencies.

First, the existing works utilize commonality (e.g., frequent subgraphs, paths, or trees) among the database graphs to construct their index. To extract common substructures from the database graphs, they usually apply data mining techniques such as frequent subgraph mining [8,18] or its variants [19,7,20,3,2]. These mining operations are expensive and incremental update on the mining results is difficult. Thus, significant overheads are imposed on both index construction and maintenance.

Second, the existing works adopt the *filtering-and-verification* approach. They use the index to first *filter* part of the false answers to produce a *candidate set*, and then *verify* each candidate to see whether it is indeed a subgraph of the query. However, the *optimal* filtering approach suffers from a *bottleneck* that the size of the candidate set is *at least* the size of the final answer set, which results in a high verification cost because each verification is a subgraph isomorphism test.

We address the above two deficiencies by designing an index that is both efficient to construct and update, and devising an efficient algorithm for query processing using the index. Our index also exploits the benefit of *batch processing* to further improve query processing, which is especially efficient when the query graphs share much commonality, a likely condition since queries usually come from the same application domain. Thus, the index effectively avoids common parts among the queries being processed duplicately.

Batch processing is also inherent in many applications especially those in which queries arrive as a high-speed stream [15,1]. A typical example is the search engine designed for chemical molecules. As early as Dec 2004, a chemical software named JChem Base[3] has already implemented the super-structure search mode for chemical compounds. Later in 2005, a chemical structure search engine that supports super-structure search was launched by PubChem. In these applications, often the query processing capability is not able to keep up with the speed of the query stream; in such situations, batch processing is particularly useful since we can answer a batch of queries coming in about the same time and give timely answers without sacrificing any accuracy of the answers.

The design of our index relies on a fast graph commonality extraction method based on simple statistics of graphs. We integrate all the data graphs in the database into a single graph, namely the *integrated graph* (*IG*), by simply following the frequency of the edges. The IG is a compact representation of a set of graphs and has a number of good properties. First, the IG can be constructed in linear time without involving any expensive graph operations such as subgraph isomorphism testing. Second, it is easy and fast to maintain the IG when the database is updated. Third, the commonality, including common subgraphs and supergraphs, of the data graphs can be extracted efficiently without performing any costly operations such as graph mining [8, 18]. These common subgraphs and supergraphs are employed to process queries. Fourth, the graph integration can be applied to both data graphs and query graphs in a unified way. Thus, we similarly construct an IG on the set of query graphs for batch query processing.

In the case when little commonality exists among the data graphs or query graphs, our IG is still effective, because if a graph does not share commonality with others, it only has one place to go in the IG and hence can be located instantly. On the contrary, existing methods that utilize commonality for indexing do not have such a property and are likely not efficient for handling graphs with little commonality.

Based on the concept of IG, we develop a graph query processing system, called *IGquery*. We propose two new techniques, namely *direct inclusion of answers* and *projected-database filtering*.

First, direct inclusion breaks the bottleneck on the candidate set size of the filtering approach by directly

[1] http://www.ncbi.nlm.nih.gov/sites/entrez?cmd=search&db=pccompound&term=all[filt]

[2] http://pubchem.ncbi.nlm.nih.gov/search/search.cgi

[3] http://www.chemaxon.com/product/jc_base.html

obtaining a large subset of the answer set. Direct inclusion utilizes the subgraph-supergraph relationship to allow *groups* of data graphs to be directly included into the answer sets of *groups* of query graphs, thereby dramatically reducing the verification cost. More importantly, direct inclusion does not involve any subgraph isomorphism test. It simply follows the frequency clue in the IG of the data graphs and that of the query graphs to check the inclusion condition, which takes only linear time.

Second, projected-database filtering further produces a small candidate set for the remaining answers not found by direct inclusion. Existing work filters false results from the entire database, which results in a relatively large candidate set. Our projected-database filtering generates the candidate set from a much smaller projected database instead of from the entire database.

To the best of our knowledge, our work is the first that is able to process graph queries in a batch manner and to perform direct inclusion of answers for supergraph queries. We demonstrate by experiments on both real and synthetic datasets that our index construction is up to orders of magnitude more efficient than cIndex [2] and GPTree [22]. The results also show that our query processing is up to two orders of magnitude faster than cIndex and GPTree, and is more scalable, for graphs sharing both much and little commonality. We also show that the update maintenance of our index is efficient.

Paper organization. We give preliminaries in Section 2. We define the supergraph query problem in Section 3. We introduce the concept of integrated graph in Section 4. We present our index construction and query processing system, IGquery, in Sections 5 and 6. We evaluate the performance of IGquery in Section 7. Finally, we review related work in Section 8 and conclude the paper in Section 9.

2 Preliminaries

For simplicity of presentation, we restrict our discussion to *undirected, labelled connected graphs*. We also assume that a graph has at least one edge. However, our method can be applied to directed graphs with minor changes.

A graph g is defined as a triple (V, E, l), where V is the set of vertices, E is the set of edges and l is a labeling function that assigns a label to each vertex and edge. We define the *size* of a graph g, denoted as $|g|$, as the number of edges in g, that is, $|g| = |E(g)|$.

A *distinct edge* in a graph g is defined as a triple, (l_u, l_e, l_v), where l_e is the label of an edge (u, v) in g, and l_u and l_v are the labels of vertices u and v in g. A distinct edge, e_d, may appear multiple times in a graph g and we call each occurrence of e_d an *instance* of e_d in g.

Table 1 Notations Used Throughout

Symbol	Description
$\|g\|$	the size of a graph g, defined as $\|g\| = \|E(g)\|$
$\mathcal{D}$	a graph database
$\mathcal{Q}$	a set of query graphs
$\mathcal{A}_{q_i}$	the answer set of a query q_i
$\mathcal{A}^s_{q_i}$	a subset of the answer set of a query q_i
$\mathcal{C}_{q_i}$	the candidate set of a query q_i
$\mathcal{G}_{\mathcal{D}}$	an integrated graph built on $\mathcal{D}$
$\mathcal{G}_{\mathcal{Q}}$	an integrated graph built on $\mathcal{Q}$
$host(e)$	the set of graphs that currently share e in $\mathcal{G}$
$freq(e)$	the cumulative # of graphs that share e in $\mathcal{G}$
F	the set of discriminative subgraphs as features
F_{sub}/F_{sup}	the set of discriminative subgraphs/supergraphs
$Sup(g, S)$	all supergraphs of g in S
$Sub(g, S)$	all subgraphs of g in S
$Sup(g, \mathcal{G}_S)$	an approximation of $Sup(g, S)$ computed from $\mathcal{G}_S$
$Sub(g, \mathcal{G}_S)$	an approximation of $Sub(g, S)$ computed from $\mathcal{G}_S$

Let g and g' be two graphs. We call that g is a *subgraph* of g' (or g' is a *supergraph* of g), denoted as $g \subseteq g'$ (or $g' \supseteq g$), if there exists an injective function f: $V(g) \to V(g')$, such that for every edge $(u, v) \in E(g)$, we have $(f(u), f(v)) \in E(g')$, $l_g(u) = l_{g'}(f(u))$, $l_g(v) = l_{g'}(f(v))$, and $l_g(u, v) = l_{g'}(f(u), f(v))$, where l_g and $l_{g'}$ are the respective labeling functions of g and g'. The injective function f is called a *subgraph isomorphism* from g to g'.

Table 1 lists the notations used throughout the paper.

3 Problem Definition

The *supergraph query processing problem* we tackle in this paper is given as follows:

– *Input*: A graph database $\mathcal{D} = \{g_1, \ldots, g_n\}$ and a set of queries $\mathcal{Q} = \{q_1, \ldots, q_m\}$, where $m \geq 1$.
– *Output*: $\mathcal{A}_{\mathcal{Q}} = \{\mathcal{A}_{q_1}, \ldots, \mathcal{A}_{q_m}\}$, where $\mathcal{A}_{q_i} = \{g_j : g_j \in \mathcal{D}, g_j \subseteq q_i\}$, i.e., each $\mathcal{A}_{q_i}$ contains the set of data graphs in $\mathcal{D}$ that are subgraphs of q_i.

Different from the existing graph query processing problems, we define our problem to process a batch of queries at a time for the following two reasons. First, there is a need for processing queries that come in as a *high-speed stream*, which is useful in many applications (see Section 1) that require prompt query response. Second, batch query processing enables us to eliminate the repeated processing of common parts among queries so as to obtain a higher throughput.

Our goal in this paper is to develop an efficient system for processing supergraph queries (possibly with a bulky and streaming query workload), with a low-cost index that is easy to construct and maintain.

4 Graph Integration

In this section, we discuss the approach of graph integration, which will be used for both index construction and query processing in subsequent sections.

Given a set of graphs G, the concept of graph integration is to merge all the graphs in G into a single *compact* graph $\mathcal{G}$, whereby the repeated common substructures of the graphs are eliminated in $\mathcal{G}$ as much as possible.

A straightforward approach of graph integration is to first mine frequent subgraphs from G and then merge the graphs in G by sharing their frequent subgraphs in the descending order of their frequency. However, as mentioned earlier, frequent subgraph mining is too costly to be applied, especially when database update is frequent or queries come as a stream. Therefore, we need to find a new way of solving this problem, which we discuss as follows.

We propose a simple but effective scheme to merge a set of graphs into a *compact* graph by utilizing the statistics of the edge frequency of the graphs. Let G be a set of graphs and $\mathcal{G}$ be the compact graph of G, or called the *integrated graph* (*IG*). We keep the information of the graphs in G at the edges of $\mathcal{G}$, while eliminating duplicate edges shared among the graphs in G. We first define the *frequency* of an edge e in $\mathcal{G}$, denoted as $freq(e)$, as the number of graphs in G that share e in $\mathcal{G}$. For the purpose of query processing, we also associate with each edge e in $\mathcal{G}$ the set of graphs (IDs) in G that share e, denoted as $host(e)$.

The basic idea of graph integration is to use the frequency of the edges in the *current* $\mathcal{G}$ to guide the merging of an incoming graph into $\mathcal{G}$. More specifically, when merging a graph g into $\mathcal{G}$, we find all the edges in g that are also in $\mathcal{G}$, and pick the one edge that has the highest frequency in $\mathcal{G}$ (If there are more than one edge having the highest frequency, we simply break the tie by the lexicographic order of the edge labels). Then, using this edge as a starting edge in both g and $\mathcal{G}$, we perform a simultaneous depth-first traversal of both g and $\mathcal{G}$ to find their common subgraph. Let e_0 be the starting edge and e_1 be the next edge to visit in the depth-first traversal of g. Let E_1 be the set of edges that we can choose to visit next to e_0 in the depth-first traversal of $\mathcal{G}$. We find an edge in E_1 that matches e_1 to visit. If there are multiple edges in E_1 matching e_1, we choose the one with the highest frequency to visit. This process continues until we meet an edge in g that cannot be matched in $\mathcal{G}$. The matched edges in the simultaneous depth-first traversal form a common subgraph of g and $\mathcal{G}$. We merge g into $\mathcal{G}$ by sharing this common subgraph, while we create new edges in $\mathcal{G}$ for those edges in g that have not been matched.

Note that we match edges by (l_u, l_e, l_v), i.e., the definition of a distinct edge. There may be multiple instances of the distinct edge e_0 in g. In this case, we run the simultaneous depth-first traversal multiple times starting at each instance of e_0 in g. Among the multiple traversals, we pick up the largest common subgraph of g and $\mathcal{G}$, and we merge g into $\mathcal{G}$ by sharing this subgraph.

To find the edge that has the highest frequency in $\mathcal{G}$ as a starting edge for the simultaneous depth-first traversal, we construct a *header table* to keep the set of distinct edges in $\mathcal{G}$. Each distinct edge e_d in the header table has a pointer to the instance of e_d in $\mathcal{G}$ that has the highest frequency.

Algorithm 1 presents our algorithm for *fast graph integration* (*FGI*). For each incoming graph g_i, FGI first finds the frequency of each distinct edge of g_i from the header table and then picks up the edge e_0 that has the highest frequency (Lines 3-4), where e_0 points to its instance e in $\mathcal{G}$. Then, for each instance e' of e_0 in g_i, FGI finds the largest common subgraph of g_i and $\mathcal{G}$ that can be obtained by Lines 6-9. For each run of Lines 6-9, we obtain a matching subgraph of g_i and $\mathcal{G}$. We then pick the largest matching subgraph, g, and merge g_i into $\mathcal{G}$ by sharing g. Then, a corresponding new edge is created in $\mathcal{G}$ for each edge in g_i but not in g. During the merge, for each edge in g_i, we also increment the frequency and update the host of its matching edge in $\mathcal{G}$ to assist future integration. Note that the graph IDs in each $host(e)$ is automatically sorted since the graphs are merged into $\mathcal{G}$ in the ascending order of their IDs. Finally, $\mathcal{G}$ is outputted when all the graphs in G are merged.

The merge of each g_i into $\mathcal{G}$ takes only linear time in the size of g_i, assuming the number of instances of a distinct edge in g_i is a constant, which is true for most datasets. Thus, the total complexity of Algorithm 1 is $\mathcal{O}(s|G|)$, where s is the average size of the graphs in G. In the worst case, when every graph in G consists of only one distinct edge (i.e., all edges are identical), the complexity is $\mathcal{O}(s^2|G|)$. However, even s^2 is small for graphs in a transaction graph database.

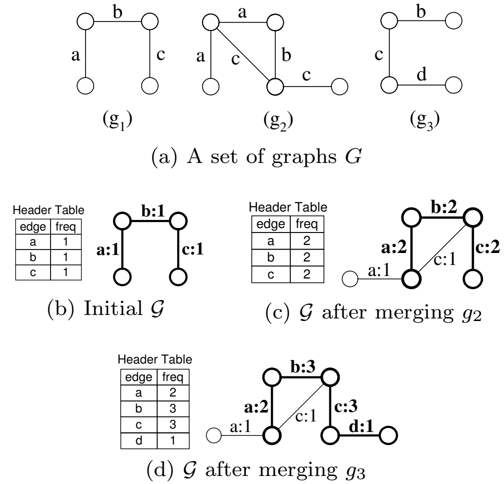
The following example illustrates how FGI works.

Example 1 Figure 1(a) shows a set of graphs G that consists of three graphs g_1, g_2 and g_3. For clarity of presentation, we only show the distinct edges a, b, c, and d in the graphs and assume that all the nodes are

Algorithm 1 *FastGraphIntegration* (*FGI*)

Input: A set of graphs, $G = \{g_1, g_2, \ldots, g_n\}$.
Output: The integrated graph $\mathcal{G}$.

1. $\mathcal{G} \leftarrow g_1$;
2. **for each** $i = 2, \ldots, n$ **do**
3. Find from the header table the distinct edge in g_i that has the highest frequency in $\mathcal{G}$;
4. Let e_0 be this edge and it points to e in $\mathcal{G}$;
5. **for each** instance e' of e_0 in g_i **do**
6. Match g_i with $\mathcal{G}$ by a depth first traversal, starting from e' in g_i and e in $\mathcal{G}$;
7. **for each** edge e'' in g_i in depth first traversal **do**
8. **if**(more than one edge in $\mathcal{G}$ match e'')
9. Choose the edge with the highest frequency;
10. Let g be the largest matching subgraph of g_i and $\mathcal{G}$ obtained in Lines 5-9;
11. Merge g_i into $\mathcal{G}$ by sharing g;
12. **for each** edge instance e of g_i merged into $\mathcal{G}$ **do**
13. Increment *freq*(e);
14. Add graph ID i to *host*(e);
15. Return $\mathcal{G}$;



(a) A set of graphs G

(b) Initial $\mathcal{G}$

(c) $\mathcal{G}$ after merging g_2

(d) $\mathcal{G}$ after merging g_3

Fig. 1 An Example of FGI Algorithm

of the same label. Initially, the integrated graph $\mathcal{G} = g_1$, which is given in Figure 1(b). For clarity, we omit the host of each edge in $\mathcal{G}$ and show the edge frequency after its label. For example, **a:1** means that the edge has a label **a** and frequency 1. We also omit the pointers in the header table, but the edge instances in $\mathcal{G}$ that are pointed by the header table are shown in **bold**.

The FGI algorithm first integrates graph g_2 into the initial $\mathcal{G}$. FGI first checks the frequency of all distinct edges in g_2 in the header table and picks the distinct edge **a** as the starting edge for the subgraph matching with $\mathcal{G}$ (Line 3). Note that there are more than one distinct edge in g_2 having the highest frequency (**a**, **b**, and **c** all have the highest frequency of 1). In this case, we choose **a** as the starting edge by the lexicographic order. Then, FGI matches g_2 with $\mathcal{G}$ by depth first traversal (Lines 5-9). There are two instances of edge **a** in g_2. Starting from the first instance (the one on the left in g_2), the obtained matching subgraph has only one edge (**a** itself); while for the second instance (the one on the top in g_2), the matching subgraph has three edges **a**, **b** and **c**. FGI (Lines 10-14) then merges g_2 into $\mathcal{G}$ by sharing the larger matching subgraph. The resultant $\mathcal{G}$ after merging g_2 is given in Figure 1(c).

Then, FGI further integrates g_3 to the $\mathcal{G}$ in Figure 1(c). The chosen starting edge for g_3 is **b**. When performing the depth first matching of g_3 and $\mathcal{G}$, there are two instances of **c** next to **b** in $\mathcal{G}$ that can be matched with the edge **c** in g_3. FGI (Lines 8-9) chooses **c:2**, which has higher frequency, and obtains the final $\mathcal{G}$ as given in Figure 1(d). □

This graph integration method is simple but has several remarkable advantages. First, by following the descending order of edge frequency when merging the graphs, we are able to integrate the graphs to the position that many other graphs are integrated into. This approach shares the same principle of using frequent subgraphs as the integration guidance to extract the common subgraphs of many graphs. Second, our graph integration approach is very fast since it does not involve any expensive operation such as frequent subgraph mining or subgraph isomorphism test. The most costly step is to perform the depth first traversal that is linear in the size of the graph g. The edge frequency used to guide the integration can be easily collected and maintained during the integration process. Third, the integrated graph keeps all neighborhood information of the graphs. Therefore, by utilizing the integrated graph, we are able to extract not only common subgraphs but also common supergraphs for efficient query processing, which we discuss in Section 6.

5 Integrated Graph as Index

We now discuss how we use integrated graph as an index and how we maintain the index in case of database updates.

5.1 Index Construction

Given the graph database $\mathcal{D}$, our index contains two parts: (1) An integrated graph built on $\mathcal{D}$, denoted as $\mathcal{G}_\mathcal{D}$; (2) A set of feature graphs, F, extracted from $\mathcal{G}_\mathcal{D}$ for the purpose of filtering. We construct $\mathcal{G}_\mathcal{D}$ by Algorithm 1 and discuss how to extract features from $\mathcal{G}_\mathcal{D}$ as follows.

Filtering for processing supergraph queries is performed based on the following *exclusive logic* [2]: let f be a feature graph and q a query, if $f \nsubseteq q$, then $\forall g_i \in \mathcal{D}$ and $g_i \supseteq f$, we have $g_i \notin \mathcal{A}_q$. Let $Sup(f, \mathcal{D}) = \{g_i : g_i \in \mathcal{D}, g_i \supseteq f\}$; that is, $Sup(f, \mathcal{D})$ is the set of graphs in $\mathcal{D}$ that are ***Sup***ergraphs of f. Then, if f is not a subgraph of q, all data graphs in $Sup(f, \mathcal{D})$ can be filtered from the answer set of q. Existing work, cIndex [2], selects a set of features F from frequent subgraphs mined from $\mathcal{D}$. The features are selected to be distinctive and have the maximum coverage of the data graphs. However, their feature selection is costly since it needs to mine the frequent subgraphs first.

We propose to generate a set of *discriminative subgraphs* from $\mathcal{G}_{\mathcal{D}}$ and use them as features. The main idea is to traverse $\mathcal{G}_{\mathcal{D}}$ by the descending order of the host size of the edges in a depth-first manner. At each depth-first step, we grow the current subgraph f to obtain f' by adding one edge (i.e., $f \subseteq f'$). We then define a graph f to be *discriminative* if $|Sup(f, \mathcal{D})|$ is significantly larger than $|Sup(f', \mathcal{D})|$. Formally, given a threshold δ ($0 \leq \delta \leq 1$), if $\frac{|Sup(f', \mathcal{D})|}{|Sup(f, \mathcal{D})|} < \delta$, then graph f is defined to be discriminative. This method has been shown to be very effective in selecting a set of representative patterns (so that other redundant patterns are removed) in [17]. We approximate $Sup(f, \mathcal{D})$ by intersecting the graph IDs in $host(e)$ of all edges in f. We denote this approximate $Sup(f, \mathcal{D})$ as $Sup(f, \mathcal{G}_{\mathcal{D}})$ since it is computed from $\mathcal{G}_{\mathcal{D}}$. Although $Sup(f, \mathcal{G}_{\mathcal{D}})$ is an approximation, it is sufficient for the purpose of filtering (as also verified by our experiments). More importantly, by using the approximation we do not need to perform any subgraph isomorphism testing.

The size of the IG structure and that of the feature set do not increase with the size of the database since common structures are shared. In most cases, the IG and the feature set are small and can be kept in the memory. However, the total size of $host(e)$ for all edges in the IG, as well as that of $Sup(f, \mathcal{G}_{\mathcal{D}})$ for the features, increases linearly in the size of database, but they can be easily stored on and retrieved from the disk. When the structure of the IG is also large, we can keep those edges with a smaller $freq(e)$ on the disk, because these edges are not frequently accessed.

5.2 Index Maintenance

We consider two types of updates in $\mathcal{D}$: *insertion* and *deletion*.

The maintenance of $\mathcal{G}_{\mathcal{D}}$ on data graph insertion is straightforward, as we can simply apply Algorithm 1 to merge the new graph into $\mathcal{G}_{\mathcal{D}}$. Insertion takes linear time in the size of the new graph. Deletion is also simple and efficient with $\mathcal{G}_{\mathcal{D}}$. We keep with each graph $g \in \mathcal{D}$ a set of pointers to the set of edges E in $\mathcal{G}_{\mathcal{D}}$ to which g is a host, i.e., the ID of g is in $host(e)$ for each $e \in E$. When we delete a graph g from $\mathcal{D}$, we simply delete the ID of g from $host(e)$ for each $e \in E$. We store $host(e)$ as a binary tree and thus the total deletion time is $\mathcal{O}(|E| \log |host(e)|)$. When $|host(e)| = 0$, we consider the edge e as *obsolete* and remove it from $\mathcal{G}_{\mathcal{D}}$.

Note that when $\mathcal{G}_{\mathcal{D}}$ is constructed on the initial database, we have $freq(e) = |host(e)|$. Later on when $\mathcal{D}$ is dynamically updated, $freq(e)$ keeps on increasing so as to keep the *cumulative* number of graphs that ever share edge e in $\mathcal{G}_{\mathcal{D}}$, while $host(e)$ is updated as the set of graphs that *currently* share e. Collecting the cumulative statistics in $freq(e)$ helps obtain a more compact $\mathcal{G}_{\mathcal{D}}$, while $host(e)$ should be kept up to date to ensure the correctness of query processing.

For the maintenance of the feature set F, we keep the edges in $\mathcal{G}_{\mathcal{D}}$ that are used to extract each feature. When the ID of a graph g is added to $host(e)$ for all edges e in a feature f, we also add g to $Sup(f, \mathcal{G}_{\mathcal{D}})$. When the ID of a graph g is deleted from $host(e)$ of any edge e in a feature f, we also delete g from $Sup(f, \mathcal{G}_{\mathcal{D}})$. Let $\alpha = (|Sup(f, \mathcal{G}_{\mathcal{D}})|/|\mathcal{D}|)$, where $f \in F$ and $|Sup(f, \mathcal{G}_{\mathcal{D}})| \leq |Sup(f', \mathcal{G}_{\mathcal{D}})|$ for all $f' \in F$, and F is the set of features computed from the database $\mathcal{D}$. If $|Sup(f, \mathcal{G}_{\mathcal{D}})|$ becomes smaller than $\alpha|\mathcal{D}|$, we delete f from F because f is not effective for filtering if $|Sup(f, \mathcal{G}_{\mathcal{D}})|$ becomes small [2]. Furthermore, if an edge e in $\mathcal{G}_{\mathcal{D}}$ is not in any feature and $|host(e)|$ grows as large as $\alpha|\mathcal{D}|$, we run the depth-first search of $\mathcal{G}_{\mathcal{D}}$ to extract new features starting from the edge e. However, since the entire feature selection process is efficient, periodically we will discard all features and select them from scratch. Note that the value of α is determined by the feature selection process, i.e., the value of $(|Sup(f, \mathcal{G}_{\mathcal{D}})|/|\mathcal{D}|)$ for the least effective feature $f \in F$. In this way, when the size of $\mathcal{D}$ changes due to database updates, the value of α is still relative to the new $|\mathcal{D}|$.

6 Query Processing

We now discuss how we apply the concept of IG for query processing. We first give the overall framework and then present the details of each step.

The framework of our query processing system, namely *IGquery*, consists of three major steps as follows.

1. *Query Integration*: Construct an IG $\mathcal{G}_{\mathcal{Q}}$ for the set of input queries $\mathcal{Q}$ in order to extract the commonality among the queries and process the common parts

of the queries once rather than repeatedly for each single query.

2. *Direct inclusion of answers*: Use $\mathcal{G}_{\mathcal{Q}}$, as well as the indexed $\mathcal{G}_{\mathcal{D}}$, to compute a subset of the answer set for each query $q_i \in \mathcal{Q}$.
3. *Projected-database filtering*: Use $\mathcal{G}_{\mathcal{Q}}$, as well as $\mathcal{G}_{\mathcal{D}}$, to compute a small candidate set for the rest of the answer set for each $q_i \in \mathcal{Q}$. The candidates are then verified by subgraph isomorphism to give the final answer set.

6.1 Query Integration

In addition to indexing, the IG is also designed to explore the commonality among the queries in order to eliminate repeated processing of some common part of the queries as much as possible. Since the IG can collapse the common substructures of the queries into a single substructure, we can utilize these commonality to speed up the batch query processing. More specifically, we can extract common substructures and superstructures of the queries from the IG. We then process these common parts *only once*, whose results can serve as partial results for *many queries* (the queries that share these common structures). Without such a query integration, the queries can only be processed one by one and their common parts are processed repeatedly for each individual query, which is a waste of computation. Therefore, the usage of the IG enables the elimination of these repeated (partial) query processing, thereby significantly improving the query efficiency.

For this purpose, we construct an IG for the set of queries, $\mathcal{Q}$, and denote it as $\mathcal{G}_{\mathcal{Q}}$. We first discuss how to construct $\mathcal{G}_{\mathcal{Q}}$. Then in Sections 6.2 and 6.3, we discuss how the common portions among the queries are extracted from $\mathcal{G}_{\mathcal{Q}}$ and processed together.

$\mathcal{G}_{\mathcal{Q}}$ is constructed in the same way as $\mathcal{G}_{\mathcal{D}}$ by applying Algorithm 1. Since the set of queries $\mathcal{Q}$ comes in as a stream, we process $\mathcal{Q}$ in batches and compute $\mathcal{G}_{\mathcal{Q}}$ for each batch of queries. The size of a batch can be either count-based or time-based [4] (similar to the size of a window unit in a sliding window), depending on different applications.

In constructing $\mathcal{G}_{\mathcal{Q}}$, we also use $freq(e)$ to keep the cumulative edge frequency so as to guide the graph integration. Since the nature of supergraph queries implies that the query graphs should have subgraph-supergraph relationship with the data graphs (i.e., sharing some common structures), we use $\mathcal{G}_{\mathcal{D}}$ as a *template* for constructing $\mathcal{G}_{\mathcal{Q}}$. That is, we set $\mathcal{G}_{\mathcal{Q}}$ initially as $\mathcal{G}_{\mathcal{D}}$ except that we initialize $host(e) = \emptyset$ for all e in $\mathcal{G}_{\mathcal{Q}}$. Then for each batch of queries in $\mathcal{Q}$, we update $\mathcal{G}_{\mathcal{Q}}$ by applying Algorithm 1. In this way, we utilize the statistical information already collected in $freq(e)$ of $\mathcal{G}_{\mathcal{D}}$ to construct a high-quality $\mathcal{G}_{\mathcal{Q}}$, and hence a stable performance even at the initial stage of a stream.

Deletion in $\mathcal{G}_{\mathcal{Q}}$ is much simpler than in $\mathcal{G}_{\mathcal{D}}$. After we process a batch of queries, we simply re-initialize $host(e) = \emptyset$ for all e in $\mathcal{G}_{\mathcal{Q}}$ (Of course we may also choose to keep part of it for caching but this is a separate issue and we do not discuss in this paper). In order to maintain the size of $\mathcal{G}_{\mathcal{Q}}$, we remove the edges in $\mathcal{G}_{\mathcal{Q}}$ with the lowest $freq(e)$ when $|\mathcal{G}_{\mathcal{Q}}|$ is larger than the available memory.

6.2 Direct Inclusion of Answers

Existing work [2] on supergraph query processing suffers from a bottleneck that the size of the candidate set is at least that of the answer set. We propose a new approach that can obtain a subset of the answer set directly without costly candidate verification. Thus, together with an effective projected-database filtering algorithm (see Section 6.3), our method effectively overcomes this bottleneck.

6.2.1 Direct Inclusion for a Single Query

Let us first consider the simple case with a single query q. The idea of direct inclusion of answers is based on the following *inclusion lemma*.

Lemma 1 (Inclusion Lemma) *Let q_{sub} be any subgraph of q, i.e., $q_{sub} \subseteq q$, then $\forall g_i \in \mathcal{D}$ and $g_i \subseteq q_{sub}$, we have $g_i \in \mathcal{A}_q$ (i.e., $\mathcal{A}_{q_{sub}} \subseteq \mathcal{A}_q$).*

It is straightforward to prove the correctness of the inclusion lemma: $q_{sub} \subseteq q$ and $g_i \subseteq q_{sub}$ implies that $g_i \subseteq q$, which means that g_i is an answer of the query q.

According to the inclusion lemma, we can maximize the effect of direct inclusion by finding a subgraph q_{sub} of q, such that q_{sub} is a supergraph of as many data graphs in $\mathcal{D}$ as possible, i.e., $|\mathcal{A}_{q_{sub}}|$ is maximized. That means q_{sub} is *a subgraph of a query*, while q_{sub} is *a common supergraph of many data graphs*. Note that, all the data graphs are now integrated into a compact graph $\mathcal{G}_{\mathcal{D}}$, thus $\mathcal{G}_{\mathcal{D}}$ is a common supergraph of all data graphs. Of course, $\mathcal{G}_{\mathcal{D}}$ itself may be too big to be a subgraph of any query, but the subgraphs of $\mathcal{G}_{\mathcal{D}}$ can be.

Therefore, we are inspired to find the common subgraphs of $\mathcal{G}_{\mathcal{D}}$ and q. In particular, if we map q into $\mathcal{G}_{\mathcal{D}}$ in the same way as we merge a data graph into $\mathcal{G}_{\mathcal{D}}$, then according to the concept of graph integration, the

largest matching subgraph of q and $\mathcal{G}_{\mathcal{D}}$ obtained by Lines 4-10 of Algorithm 1 is a good choice of q_{sub} and has a strong *inclusion power* (i.e., q_{sub} is a common supergraph of many data graphs in $\mathcal{D}$, and these data graphs can be directly included into $\mathcal{A}_q$).

Algorithm 2 shows how direct inclusion works. We use the same procedure in Algorithm 1 to find the matching subgraph of q and $\mathcal{G}_{\mathcal{D}}$ (Lines 3-4). We repeat this procedure (Lines 3-9) for each distinct edge e_d of q, so that we can take the full advantage of Algorithm 1 to obtain a larger subset, $\mathcal{A}^s_q$, of the answer set of q, without any expensive operations such as subgraph isomorphism testing.

Algorithm 2 *DirectInclusion*

Input: $\mathcal{G}_{\mathcal{D}}$ and a query q.
Output: A subset of the answer set of q, $\mathcal{A}^s_q$.

1. $\mathcal{A}^s_q \leftarrow \emptyset$;
2. **for each** distinct edge e_d in q **do**
3. Process Lines 4-10 of Algorithm 1 by putting q in place of g_i and $\mathcal{G}_{\mathcal{D}}$ in place of $\mathcal{G}$;
4. Let q_{sub} be the matching subgraph of q and $\mathcal{G}_{\mathcal{D}}$ obtained;
5. **for each** edge e in q_{sub} **do**
6. **for each** g_i in $host(e)$ **do**
7. $++count(g_i)$; // $count(g_i)$ is initially set to 0
8. $Sub(q_{sub}, \mathcal{G}_{\mathcal{D}}) \leftarrow \{g_i : count(g_i) = |g_i|\}$;
9. $\mathcal{A}^s_q \leftarrow \mathcal{A}^s_q \cup Sub(q_{sub}, \mathcal{G}_{\mathcal{D}})$;
10. Return $\mathcal{A}^s_q$;

After finding q_{sub} (Line 4), which is the largest matching subgraph of q and $\mathcal{G}_{\mathcal{D}}$, we use $host(e)$ to find the data graphs that are merged into the same place as q_{sub} (Lines 5-7). For each data graph g_i in $host(e)$ of an edge e in q_{sub}, we use a counter $count(g_i)$ to count the number of edges that g_i shares with q_{sub}. If a data graph g_i is indeed a subgraph of q_{sub}, then $count(g_i)$ must be equal to the number of edges in g_i (Line 8). This is because $count(g_i)$ is incremented for each edge e in q_{sub} only if $host(e)$ contains g_i (Lines 5-7), which means that g_i shares the edge e with q_{sub} when integrated into $\mathcal{G}_{\mathcal{D}}$. Thus, $count(g_i) = |g_i|$ only if $count(g_i)$ is incremented once for every edge in g_i, which means that all edges in g_i ($|g_i|$ number of them) are contained in q_{sub} (ensured by the host lists) and thus g_i is a subgraph of q_{sub}. Therefore, we can directly include g_i in $Sub(q_{sub}, \mathcal{G}_{\mathcal{D}})$. Similar to the definitions of $Sup(f, \mathcal{D})$ and $Sup(f, \mathcal{G}_{\mathcal{D}})$ in Section 5.1, $Sub(q_{sub}, \mathcal{G}_{\mathcal{D}})$ is an approximation of $Sub(q_{sub}, \mathcal{D})$ computed from $\mathcal{G}_{\mathcal{D}}$, where $Sub(q_{sub}, \mathcal{D})$ is the set of graphs in $\mathcal{D}$ that are ***Sub***graphs of q_{sub}. Thus, by Lemma 1, we directly include $Sub(q_{sub}, \mathcal{G}_{\mathcal{D}})$ into $\mathcal{A}^s_q$ (Line 9).

6.2.2 Direct Inclusion for a Set of Queries

The efficiency of direct inclusion can be further improved by processing a set of queries simultaneously. We use the IG for queries, $\mathcal{G}_{\mathcal{Q}}$, to explore the commonality among a batch of queries Q so that these common parts can be processed together for direct inclusion.

The idea of direct inclusion for a single query can be extended for multiple queries by the following lemma.

Lemma 2 (Inclusion for Multiple Queries) *Let q_{sub} be a* common subgraph *of k queries, $q_1, \ldots, q_k$, then $\forall g_i \in \mathcal{D}$ and $g_i \subseteq q_{sub}$, we have $g_i \in \mathcal{A}_{q_j}$, $\forall j \in \{1, \ldots, k\}$ (i.e., $\mathcal{A}_{q_{sub}} \subseteq \mathcal{A}_{q_j}$).*

According to Lemma 2, our task now is to find a graph q_{sub} which is a common subgraph of as many queries as possible; that is, we want to find q_{sub} such that $Sup(q_{sub}, \mathcal{Q})$ is maximized. Since we capture the commonality by graph integration, we use $\mathcal{G}_{\mathcal{Q}}$ to find q_{sub}.

However, there are many common subgraphs in $\mathcal{G}_{\mathcal{Q}}$ for different groups of queries and we cannot exhaustively use all of them to perform the inclusion. In fact, there is a tradeoff when choosing a good q_{sub}. If q_{sub} is of small size, then q_{sub} is a common subgraph of a larger set of queries (i.e., $|Sup(q_{sub}, \mathcal{G}_{\mathcal{Q}})|$ is large). But on the database side, a small-sized q_{sub} tends to have only a small number of data graphs as its subgraph, i.e., $|\mathcal{A}_{q_{sub}}|$ is small. In this case, we are including too few data graphs into the answer sets of many queries. On the other hand, if q_{sub} is large, then $|Sup(q_{sub}, \mathcal{G}_{\mathcal{Q}})|$ is small though $|\mathcal{A}_{q_{sub}}|$ is large. Thus, we are including many data graphs into the answer sets of just a few queries. Therefore, it is challenging to choose a good q_{sub} for more effective direct inclusion.

Inspired by the feature selection in index construction, we propose to select a set of *discriminative subgraphs* of $\mathcal{G}_{\mathcal{Q}}$ for performing inclusion in Lemma 2. This process is similar to the feature selection discussed in Section 5.1 except that $\mathcal{G}_{\mathcal{Q}}$ is used in place of $\mathcal{G}_{\mathcal{D}}$.

Let F_{sub} be the set of discriminative subgraphs selected for direction inclusion. For each $q_{sub} \in F_{sub}$, in order to apply Lemma 2, we need to find $\mathcal{A}_{q_{sub}}$ and include it to the answer sets of each query in $Sup(q_{sub}, \mathcal{G}_{\mathcal{Q}})$. Computing $\mathcal{A}_{q_{sub}}$ may still be expensive; however, we only obtain a partial $\mathcal{A}_{q_{sub}}$, i.e., $\mathcal{A}^s_{q_{sub}}$, by Algorithm 2 and then perform the inclusion of $\mathcal{A}^s_{q_{sub}}$.

Algorithm 3 outlines how we perform direct inclusion for multiple queries. As discussed above, we first select the set of discriminative subgraphs F_{sub} from $\mathcal{G}_{\mathcal{Q}}$. We then obtain $\mathcal{A}^s_{q_{sub}}$ by Algorithm 2 for each $q_{sub} \in F_{sub}$, and directly include $\mathcal{A}^s_{q_{sub}}$ as a partial answer set for each $q_j \in Sup(q_{sub}, \mathcal{G}_{\mathcal{Q}})$. Note that $Sup(q_{sub}, \mathcal{G}_{\mathcal{Q}})$ is

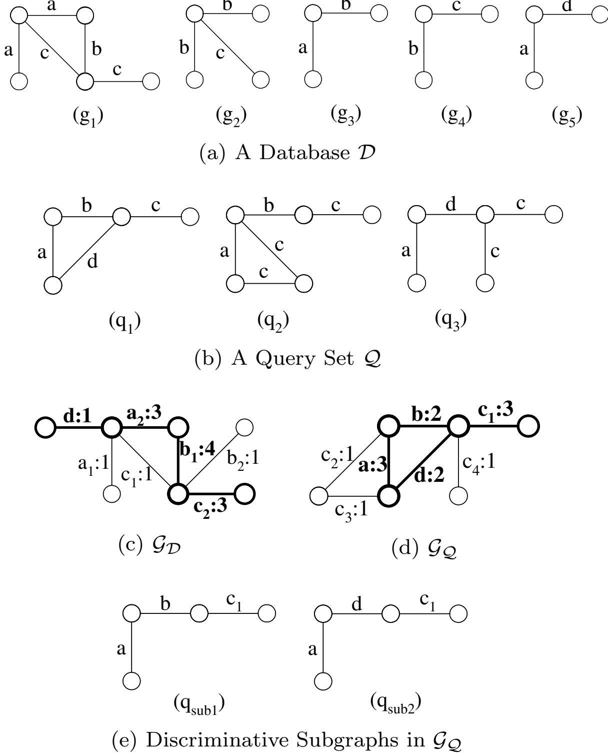
obtained together with F_{sub} (see Section 5.1). Finally in Lines 6-7, for each query q_j that is not a supergraph of any $q_{sub} \in F_{sub}$, we simply obtain $\mathcal{A}^s_{q_j}$ by Algorithm 2.

Algorithm 3 *MultiDirectInclusion*

Input: $\mathcal{G}_\mathcal{D}$, $\mathcal{G}_\mathcal{Q}$, and a batch of queries $Q = \{q_1, \ldots, q_m\}$.
Output: $\mathcal{A}^s_Q = \{\mathcal{A}^s_{q_1}, \ldots, \mathcal{A}^s_{q_m}\}$.

1. $\mathcal{A}^s_{q_j} \leftarrow \emptyset$, $\forall q_j \in Q$;
2. Find the set of discriminative subgraphs F_{sub} from $\mathcal{G}_\mathcal{Q}$;
3. **for each** $q_{sub} \in F_{sub}$ **do**
4. Obtain $\mathcal{A}^s_{q_{sub}}$ by Algorithm 2;
5. $\mathcal{A}^s_{q_j} \leftarrow \mathcal{A}^s_{q_j} \cup \mathcal{A}^s_{q_{sub}}$, $\forall q_j \in Sup(q_{sub}, \mathcal{G}_\mathcal{Q})$;
6. **for each** $q_j \in Q$, where $\mathcal{A}^s_{q_j} = \emptyset$, **do**
7. Obtain $\mathcal{A}^s_{q_j}$ by Algorithm 2;
8. Return $\mathcal{A}^s_Q$;

The following example illustrates how direct inclusion for multiple queries works.



(a) A Database $\mathcal{D}$

(b) A Query Set $\mathcal{Q}$

(c) $\mathcal{G}_\mathcal{D}$

(d) $\mathcal{G}_\mathcal{Q}$

(e) Discriminative Subgraphs in $\mathcal{G}_\mathcal{Q}$

Fig. 2 Direct Inclusion for Multiple Queries

Example 2 Figures 2(a) and (b) give a graph database $\mathcal{D}$ and a set of queries $\mathcal{Q}$, respectively. By applying Algorithm 1, we obtain two IGs, $\mathcal{G}_\mathcal{D}$ and $\mathcal{G}_\mathcal{Q}$, as shown in Figures 2(c) and (d). The header tables are omitted and

Table 2 Hosts for $\mathcal{G}_\mathcal{D}$ and $\mathcal{G}_\mathcal{Q}$

IG	Edge	*host*	IG	Edge	*host*
$\mathcal{G}_\mathcal{D}$	a_1	$\{g_1\}$	$\mathcal{G}_\mathcal{Q}$	a	$\{q_1, q_2, q_3\}$
	a_2	$\{g_1, g_3, g_5\}$		b	$\{q_1, q_2\}$
	b_1	$\{g_1, g_2, g_3, g_4\}$		c_1	$\{q_1, q_2, q_3\}$
	b_2	$\{g_2\}$		c_2	$\{q_2\}$
	c_1	$\{g_1\}$		c_3	$\{q_2\}$
	c_2	$\{g_1, g_2, g_4\}$		c_4	$\{q_3\}$
	d	$\{g_5\}$		d	$\{q_1, q_3\}$

the host of each edge is given in Table 2. For the easy reference of the edges in each IG, we give a subscript for the distinct edge that has more than one instance in the IG.

For simplicity, we assume that we find two discriminative subgraphs q_{sub1} and q_{sub2} as shown in Figure 2(e). By intersecting $host(e)$ of the edges, we obtain $Sup(q_{sub1}, \mathcal{G}_\mathcal{Q}) = \{q_1, q_2\}$ and $Sup(q_{sub2}, \mathcal{G}_\mathcal{Q}) = \{q_1, q_3\}$.

Then, Line 4 of Algorithm 3 invokes Algorithm 2 to compute $\mathcal{A}^s_{q_{sub1}}$. The largest common subgraph of q_{sub1} and $\mathcal{G}_\mathcal{D}$ (Lines 3-4 of Algorithm 2) is q_{sub1} itself: the edges a, b, c_1 in q_{sub1} matches the edges a_2, b_1 and c_2 in $\mathcal{G}_\mathcal{D}$, respectively. Then, Lines 5-9 of Algorithm 2 check the hosts of a_2, b_1 and c_2 in $\mathcal{G}_\mathcal{D}$ (Table 2) to find the data graphs that are subgraphs of q_{sub1} and we obtain $\mathcal{A}^s_{q_{sub1}} = \{g_3, g_4\}$. The procedure returns to Algorithm 3 and Line 5 directly includes $\mathcal{A}^s_{q_{sub1}}$ to the answer sets of the queries in $Sup(q_{sub1}, \mathcal{G}_\mathcal{Q})$, i.e., include g_3 and g_4 to $\mathcal{A}_{q_1}$ and $\mathcal{A}_{q_2}$.

Similarly, for q_{sub2}, Lines 4-5 also directly include $\mathcal{A}^s_{q_{sub2}} = \{g_5\}$ to the answer sets of q_1 and q_3 in $Sup(q_{sub2}, \mathcal{G}_\mathcal{Q})$. □

The above example clearly shows the advantages of direct inclusion of answers. First, we do the inclusion in a "*many-to-many*" manner: including *many* data graphs into the answer sets of *many* queries. Second, each query graph can benefit from different common subgraphs with different groups of queries (e.g., q_1 benefits from both q_{sub1} and q_{sub2} by direct inclusion).

6.3 Filtering

With direct inclusion, we overcome the bottleneck on the size of candidate set in existing work. However, we still need an effective filtering algorithm in order to produce a small candidate set for the remaining answers not found by direct inclusion. In this section, we first present a filtering algorithm for multiple queries. Then, we further improve by designing a novel *projected-database filtering* algorithm.

6.3.1 A Multi-Query Filtering Approach

Let $\mathcal{C}_q$ be the candidate set of q. Given a set of features F, the existing filtering approach computes $\mathcal{C}_q$ based on the exclusive logic, that is, to filter the data graphs in $Sup(f, \mathcal{D})$, $\forall f \in F$ and $f \nsubseteq q$. Therefore, we have $\mathcal{C}_q = (\mathcal{D} - \cup_{f \in F, f \nsubseteq q} Sup(f, \mathcal{D}))$.

However, this filtering approach only processes a single query at a time. Similar to what we do for direct inclusion, we also propose to perform the filtering for a batch of queries Q together, based on the following lemma.

Lemma 3 (Filtering for Multiple Queries) *Let q_{sup} be any common supergraph of k queries, $q_1, \ldots, q_k$, then $\mathcal{C}_{q_j} \subseteq \mathcal{C}_{q_{sup}}$, $\forall j \in \{1, \ldots, k\}$.*

Proof Note that $\mathcal{C}_q = (\mathcal{D} - \cup_{f \in F,\ f \nsubseteq q} Sup(f, \mathcal{D}))$ for a query q. Since $q_{sup} \supseteq q_j$, if $f \nsubseteq q_{sup}$, then $f \nsubseteq q_j$, but not vice versa. Therefore, $(\cup_{f \in F,\ f \nsubseteq q_j} Sup(f, \mathcal{D})) \supseteq (\cup_{f \in F,\ f \nsubseteq q_{sup}} Sup(f, \mathcal{D}))$ and hence $\mathcal{C}_{q_j} \subseteq \mathcal{C}_{q_{sup}}$ holds.

Lemma 3 suggests that we should find the *common supergraph of queries*, q_{sup}, in order to perform batch filtering; while on the database side, q_{sup} is compared with a set of features F which are the *common subgraphs of data graphs*. Note that this is just the reverse way of extracting commonality as in direct inclusion. In direct inclusion, the common subgraphs of queries and the common supergraphs of data graphs are needed.

Let F_{sup} be a set of common supergraphs of queries. The extraction of F_{sup} from $\mathcal{G}_\mathcal{Q}$ for filtering is similar to the extraction of F_{sub} for direct inclusion discussed in Section 6.2.2. The only difference is that we use Lines 5-8 of Algorithm 2 to find $Sub(q_{sup}, \mathcal{G}_\mathcal{Q})$ of each $q_{sup} \in F_{sup}$, instead of intersecting $host(e)$ of the edges.

We apply Lemma 3 in Algorithm 4 to perform filtering for a batch of queries. First, Line 2 generates a set of discriminative common supergraphs of the queries from $\mathcal{G}_\mathcal{Q}$. For each $q_{sup} \in F_{sup}$, Line 4 obtains the candidate set of q_{sup} using the features in F. Then, Line 5 refines the candidate sets of all queries in $Sub(q_{sup}, \mathcal{G}_\mathcal{Q})$ by Lemma 3.

Algorithm 4 *MultiFiltering*

Input: $\mathcal{G}_\mathcal{Q}$, a feature set F, and a batch of queries $Q = \{q_1, \ldots, q_m\}$.
Output: $\mathcal{C}_Q = \{\mathcal{C}_{q_1}, \ldots, \mathcal{C}_{q_m}\}$.

1. $\mathcal{C}_{q_j} \leftarrow \mathcal{D}, \forall q_j \in Q$;
2. Find the set of common supergraphs of queries, F_{sup};
3. **for each** $q_{sup} \in F_{sup}$ **do**
4. $\quad \mathcal{C}_{q_{sup}} \leftarrow (\mathcal{D} - \cup_{f \in F, f \nsubseteq q_{sup}} Sup(f, \mathcal{G}_\mathcal{D}))$;
5. $\quad \mathcal{C}_{q_j} \leftarrow \mathcal{C}_{q_j} \cap \mathcal{C}_{q_{sup}}, \forall q_j \in Sub(q_{sup}, \mathcal{G}_\mathcal{Q})$;
6. Return $\mathcal{C}_Q$;

6.3.2 A Projected-Database Filtering Approach

The filtering algorithm discussed in Section 6.3.1 relies heavily on the features selected. Ideally, we want a feature f to have a large $Sup(f, \mathcal{D})$ so that we can obtain a smaller $\mathcal{C}_q = (\mathcal{D} - Sup(f, \mathcal{D}))$. However, a large $Sup(f, \mathcal{D})$ means that f is a common subgraph of many graphs in $\mathcal{D}$, which further implies that f is likely a common subgraph of even more queries (by the nature of supergraph queries). But for filtering, we require $f \nsubseteq q$. Therefore, we have a dilemma here and $(\mathcal{D} - Sup(f, \mathcal{D}))$ may be large in many cases.

In Algorithm 4, we have partially addressed this problem by refining the candidate sets for a set of queries through their common supergraphs q_{sup} (Line 5). However, the problem of a large $\mathcal{C}_q$ is because $\mathcal{D}$ is large but $Sup(f, \mathcal{D})$ is relatively much smaller. Thus, we do not really solve the problem by Algorithm 4.

As we have just discussed, $Sup(f, \mathcal{D})$ cannot be too large; otherwise, f is only useful for a few queries but not useful for most of the queries. Thus, it leads us to seek to reduce "$\mathcal{D}$" in $(\mathcal{D} - Sup(f, \mathcal{D}))$. Clearly, we cannot actually reduce $\mathcal{D}$ because $\mathcal{D}$ is the database. However, the concept of processing a set of queries together enables us to find a projected database. Our idea is based on the following Lemma.

Lemma 4 (Projected-Database Filtering) *Let q_{sup} be a common supergraph of k queries, $q_1, \ldots, q_k$, then $\mathcal{A}_{q_j} \subseteq \mathcal{A}_{q_{sup}}$, $\forall j \in \{1, \ldots, k\}$.*

Lemma 4 is correct by the definition of supergraph query. Based on the lemma, we can obtain $\mathcal{A}_{q_j}$ from $\mathcal{A}_{q_{sup}}$; thus, we can use $\mathcal{A}_{q_{sup}}$ as the candidate set of the whole set of queries $q_j \in Sub(q_{sup}, \mathcal{G}_\mathcal{Q})$. In most cases, $\mathcal{A}_{q_{sup}}$ is significantly smaller than $\mathcal{D}$ and more importantly, $|\mathcal{A}_{q_{sup}}|$ is close to $|\mathcal{A}_{q_j}|$ so that we can obtain $|\mathcal{C}_{q_j}| \simeq |\mathcal{A}_{q_j}|$. We call $\mathcal{A}_{q_{sup}}$ the *projected database* of q_{sup} and name the filtering approach that uses $\mathcal{A}_{q_{sup}}$ for candidate generation as *projected-database filtering*.

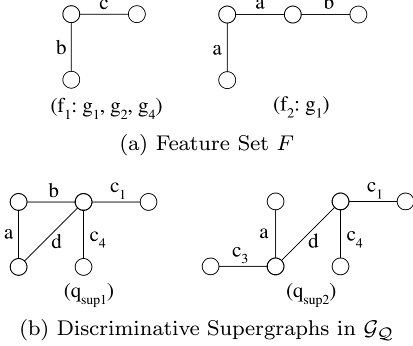
To apply the projected-database filtering, we insert Line 5 of Algorithm 5 into Algorithm 4 to first obtain $\mathcal{A}_{q_{sup}}$. Then, we use $\mathcal{A}_{q_{sup}}$ instead of $\mathcal{C}_{q_{sup}}$ to obtain $\mathcal{C}_{q_j}$ in Line 6 of Algorithm 5.

Algorithm 5 *ProjDBfiltering*

1-4. Same as Lines 1-4 of Algorithm 4;
5. $\quad \mathcal{A}_{q_{sup}} \leftarrow \{g_i : g_i \in \mathcal{C}_{q_{sup}}, g_i \subseteq q_{sup}\}$;
6. $\quad \mathcal{C}_{q_j} \leftarrow \mathcal{C}_{q_j} \cap \mathcal{A}_{q_{sup}}, \forall q_j \in Sub(q_{sup}, \mathcal{G}_\mathcal{Q})$;
7. Return $\mathcal{C}_Q$;

The following example illustrates how filtering works.

Example 3 Consider the database and the set of queries in Figures 2(a-b), we select two features f_1 and f_2 from $\mathcal{G}_\mathcal{D}$. As shown in Figure 3(a), the set of graphs in $Sup(f, \mathcal{G}_\mathcal{D})$ for each feature f is also given after the colon ":". Assume that $F_{sup} = \{q_{sup1}, q_{sup2}\}$, as given in Figure 3(b). We compute $Sub(q_{sup1}, \mathcal{G}_\mathcal{Q}) = \{q_1, q_3\}$ and $Sub(q_{sup2}, \mathcal{G}_\mathcal{Q}) = \{q_3\}$ by Lines 5-8 of Algorithm 2.



(a) Feature Set F

(b) Discriminative Supergraphs in $\mathcal{G}_\mathcal{Q}$

Fig. 3 Filtering for Multiple Queries

Line 4 of Algorithm 4 computes $\mathcal{C}_{q_{sup1}}$ by filtering using the features. Since $f_2 \nsubseteq q_{sup1}$, the data graphs in $Sup(f_2, \mathcal{G}_\mathcal{D}) = \{g_1\}$ can be filtered from $\mathcal{C}_{q_{sup1}}$, as well as from the candidate sets of q_1 and q_3 in $Sub(q_{sup1}, \mathcal{G}_\mathcal{Q})$. Therefore, $\mathcal{C}_{q_1} = \mathcal{C}_{q_3} = \mathcal{C}_{q_{sup1}} = \{g_2, g_3, g_4, g_5\}$.

We now further apply the projected-database filtering. We first obtain $\mathcal{A}_{q_{sup1}} = \{g_3, g_4, g_5\}$ (Line 5 of Algorithm 5) and then assign $\mathcal{C}_{q_1} = \mathcal{C}_{q_3} = \mathcal{A}_{q_{sup1}} = \{g_3, g_4, g_5\}$, which thus further filters g_2 from the candidate sets of both q_1 and q_3 simultaneously.

We then process q_{sup2}. Since $Sub(q_{sup2}, \mathcal{G}_\mathcal{Q}) = \{q_3\}$ and $f_1 \nsubseteq q_{sup2}$, the data graphs in $Sup(f_1, \mathcal{G}_\mathcal{D}) = \{g_1, g_2, g_4\}$ can be further filtered from $\mathcal{C}_{q_3}$. Thus, $\mathcal{C}_{q_3}$ is reduced to be $\{g_3, g_5\}$. □

6.4 The Overall Query Algorithm

We present our overall algorithm, *IGquery*, for query processing over a stream of queries in Algorithm 6. Each time we process a batch of queries, either count-based or time-based, that arrives in the stream (Line 1). We first construct $\mathcal{G}_\mathcal{Q}$ in real-time (Line 2). Then, we obtain a subset of the answer sets for the queries by direct inclusion (Line 3). For the remaining part of the answer set, we use projected-database filtering to obtain a candidate set (Lines 4-6), which is then verified to return the final answer set (Lines 7-8).

The complexity of Algorithm 6 consists of three parts. The first part is constructing the IG on Q, which

Algorithm 6 *IGquery*

Input: $\mathcal{D}$, $\mathcal{G}_\mathcal{D}$, a feature set F, and a stream of queries $\mathcal{Q} = \{q_1, q_2, \ldots\}$.
Output: $\{\mathcal{A}_{q_1}, \mathcal{A}_{q_2}, \ldots\}$.

1. **for each** batch of queries, Q, arrived in $\mathcal{Q}$ **do**
2. Construct $\mathcal{G}_\mathcal{Q}$ as discussed in Section 6.1;
3. Obtain subsets of answers $\mathcal{A}^s_{q_j}$, $\forall q_j \in Q$, by Algorithm 3;
4. Obtain $\mathcal{C}_Q$ by Algorithm 5;
5. **for each** query $q_j \in Q$ **do**
6. $\mathcal{C}_{q_j} \leftarrow (\mathcal{C}_{q_j} - \mathcal{A}^s_{q_j})$; // Direct inclusion of answers
7. $\mathcal{A}_{q_j} \leftarrow (\mathcal{A}^s_{q_j} \cup \{g_i : g_i \in \mathcal{C}_{q_j}, g_i \subseteq q_j\})$; // Verification
8. Output $\mathcal{A}_{q_j}$;

is $\mathcal{O}(s|Q|)$ as given by the analysis of Algorithm 1, where s is the average size of the query graphs in Q. The second part is direct inclusion. In Algorithm 3, computing F_{sub} from $\mathcal{G}_\mathcal{Q}$ takes $\mathcal{O}(s|Q|+|Q|^2)$ time. The depth-first traversal of $\mathcal{G}_\mathcal{Q}$ to compute F_{sub} takes $\mathcal{O}(s|Q|)$ time since the traversal follows the size of $host(e)$ in descending order. Since F_{sub} is a set of discriminative subgraphs, we have $|F_{sub}| = \mathcal{O}(|Q|)$. Since the intersection of the "$host(e)$"s for each $q_{sub} \in F_{sub}$ takes $\mathcal{O}(|Q|)$, the total time for all intersections takes $\mathcal{O}(|Q|^2)$. The third part is filtering, which also takes $\mathcal{O}(s|Q| + |Q|^2)$ time for computing F_{sup}.

Therefore, the total complexity of Algorithm 6 is $(\mathcal{O}(s|Q| + |Q|^2) + X + Y + Z)$ for processing the set of queries Q, where X is the cost for the union and intersection operations on the graph IDs to compute the candidate sets and partial answer sets, Y is the cost of subgraph isomorphism tests to examine the exclusive logic in Line 4 of Algorithm 4 and to obtain the projected database for filtering in Line 5 of Algorithm 5, and Z is the cost of subgraph isomorphism tests to verify the candidates in Line 7 of Algorithm 6. The complexity of $(X + Y + Z)$ depends on the sizes of the candidate set and answer set, which vary for different queries. Alternatively, we can give the query response time of IGquery as follows.

$$T_{response} = (T_{search} + \sum_{q \in Q}(|\mathcal{C}_q| \times T_{I/O} + |\mathcal{C}_q| \times T_{verify}))\ .\quad (1)$$

In Equation (1), $T_{search} = (\mathcal{O}(s|Q| + |Q|^2) + X + Y)$ is the index search time using the IGs, while $T_{I/O}$ is the disk I/O time for fetching each candidate graph from the disk and T_{verify} is the time for verifying the candidates. In general, $Z = \sum_{q \in Q}(|\mathcal{C}_q| \times (T_{I/O} + T_{verify}))$ dominates the cost and hence all existing work seeks to minimize $|\mathcal{C}_q|$. However, $|\mathcal{C}_q| \geq |\mathcal{A}_q|$ for the existing filtering approaches, while in our work $|\mathcal{C}_q|$ can be even much smaller than $|\mathcal{A}_q|$.

7 Performance Evaluation

We evaluate the performance of our algorithm by comparing with cIndex [2] and GPTree [22]. We run all experiments on a machine with a 3.0GHz Pentium 4 CPU and 1GB RAM, running Windows XP Professional Version 2002 SP 3. We evaluate our algorithm extensively by a comprehensive set of metrics:

- The effect of the discriminative threshold δ on the performance of index construction and query processing (Section 7.1).
- The efficiency of IG construction and the effectiveness of IG for query processing (Sections 7.2 and 7.3).
- The effectiveness of direct inclusion and projected-database filtering (Section 7.3).
- The effect of query batch size (both count-based and time-based) on the performance of query processing (Section 7.3).
- The effect of query graph size on the performance of query processing (Section 7.4).
- The effect of query answer-set size on the performance of query processing (Section 7.5).
- The effect of database size on the performance of index construction and query processing (Section 7.6).
- The effect of commonality on the performance of index construction and query processing (Section 7.7).
- The performance of index maintenance and query processing on database updates (Section 7.8).

Query Sets and Graph Databases. We use two real datasets: *AIDS* and *NCI*. *AIDS* is the *AIDS antiviral screen dataset*, which contains 10K graphs. *NCI* is a dataset with 250K graphs, which we obtain from the National Cancer Institute database. Table 3 lists some characteristics of the datasets, where the density of a graph $g = (V, E)$ is defined as $\frac{2|E|}{|V|(|V|-1)}$. More details can be found in their webpages[4].

Table 3 Characteristics of Datasets (Query Sets)

	Range of graph size	Average graph size	Range of density	Average density
AIDS	$1-217$	27.40	$0.009-1.0$	0.10
NCI	$1-252$	19.95	$0.008-1.0$	0.14

We prepare the query sets and graph databases in a way similar to [2, 22] for fair comparison. We first use the two real datasets as the *query sets*. To prepare the *graph databases*, we randomly generate a set of 10K subgraphs of the graphs in the *AIDS* dataset at minimum support threshold 0.001. We also select 10K-100K subgraphs from the *NCI* dataset for a scalability test. We list some characteristics of the graph databases in Table 4.

Table 4 Characteristics of Graph Databases

	Range of graph size	Average graph size	Range of density	Average density
AIDS	$1-58$	15.53	$0.04-1.0$	0.22
NCI	$1-16$	10.25	$0.12-1.0$	0.24

In Section 7.7, we also use synthetic datasets to test the effect of commonality. We give the details of the synthetic datasets in Section 7.7.

We also prepare the query log and the feature base which are required for cIndex as described in [2]. The settings of GPTree are as its default [22]. We tested both the exact and approximate indexes of GPTree, for both indexing and querying; but we found that the difference between the two is small. Thus, we only report the results of the exact index.

7.1 Effect of Discriminative Threshold

We first test the effect of the discriminative threshold δ on feature selection, i.e., the selection of the set of discriminative subgraphs, and how the performance of index construction and query processing varies for different δ. We use the graph database prepared from the *AIDS* dataset in this experiment.

Table 5 reports the results of index construction and query processing with different values of δ. For index construction, the indexing time decreases when δ decreases from 1 to 0, which can be explained by the decrease in the number of features as δ decreases. Overall, the index construction is very efficient as it takes only about 1 second for all values of δ except the very restrictive case when $\delta = 1$.

For query processing time, it is affected by two factors, the index probing time T_{search} and the verification time, as shown in Equation (1). The number of features and the number of candidates recorded in Rows 2 and 4 of Table 5 are indicators of these two factors, respectively. In general, more features may generate a smaller number of candidates (with shorter verification time) at a cost of longer index probing time.

Therefore, the choice of δ is a tradeoff. Fortunately, we can easily pick up a sub-optimal δ that achieves a small overall query processing time. As shown in Table

[4] *AIDS*: http://dtp.nci.nih.gov/
NCI: http://cactus.nci.nih. gov/ncidb2/download.html

Table 5 Results on the Effect of Discriminative Threshold ($1 \geq \delta \geq 0$)

	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0
Indexing time (ms)	13937	1969	1469	1235	1031	844	766	750	734	594	453
Feature #	2988	168	113	91	73	53	50	46	37	27	24
Query processing time (ms)	43.83	19.00	18.65	**18.45**	19.51	19.56	20.55	20.14	20.13	20.96	21.67
Candidate #	386	478	547	572	707	776	929	884	947	1044	1090

5, the best query processing time is achieved at $\delta = 0.7$, i.e., 18.45 milliseconds per query on average. However, for a wider range of δ ($0.6 \leq \delta \leq 0.8$), the query processing time as well as the index construction time do not vary too much. This is because the set of features remains relatively stable within this range of δ. The same findings are also observed for other datasets and we omit the details for clarity. This result demonstrates that our discriminative feature selection is efficient and effective.

We choose $\delta = 0.7$ as a default value for δ for the remaining experiments.

7.2 Indexing Performance: IG and Other Indexes

We compare the performance of our index with cIndex and GPTree. We first report the results of index construction in this subsection. We use the graph database prepared from the *AIDS* dataset in this experiment.

Table 6 reports the overall time for index construction, peak memory consumption, the size of IG and that of the database, and the number of features obtained by each index.

Table 6 Performance of Indexing on *AIDS*

	Time (sec)	Memory (MB)	IG/GDB size (# of edges)	Feature Number
cIndex	3631	531	NA / 186883	34
GPTree	16,549	408	NA / 186883	159
IG	1.24	18	2195 / 186883	91

The results show that our index construction is over three orders of magnitude faster and consumes at least 20 times less memory than both cIndex and GPTree. Our indexing time includes the time for constructing the IG $\mathcal{G}_{\mathcal{D}}$ and for selecting the discriminative features from $\mathcal{G}_{\mathcal{D}}$. The remarkable indexing time is because our algorithm runs in linear time.

The results also show that the concept of IG is indeed able to extract the commonality among the graphs, as the number of edges in the IG is only 1.2% of that in the graph database. This result is also consistent with the low memory consumption of our index.

In the subsequent three subsections, we show that our index is not only compact and efficient to construct, but also very effective for query processing.

7.3 Query Performance: Effects of Individual Components and Batch Processing

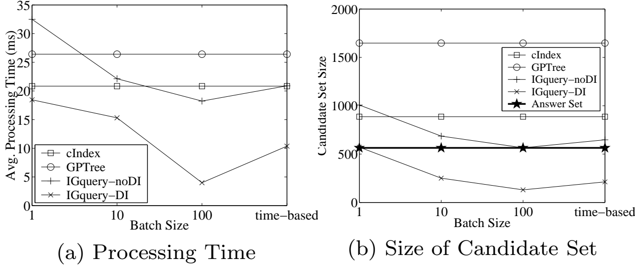
In this experiment, we show the effects of each of the individual components on query performance. There are two key components in IGquery, direct inclusion and projected-database filtering, both of them utilize the IG. Since both direct inclusion and projected-database filtering can operate in batch mode, we also assess the effect of batch processing on query performance. We use the indexes constructed in Section 7.2 for this experiment and use the AIDS query set.

Direct inclusion and projected-database filtering are two key components in IGquery. However, they are different that filtering is essential for query processing, while direct inclusion is dispensable. Filtering is essential because otherwise the candidate set is too large even if we can find the majority of the answer set by direct inclusion, unless the database is small. Although our projected-database filtering is different from the existing filtering algorithms, taking out filtering from IGquery essentially means to match the database graphs one by one. On the other hand, direct inclusion is dispensable because without direct inclusion our algorithm is essentially another filtering-based algorithm. However, direct inclusion is the key to overcome the bottleneck (on the candidate set size) of any existing filtering-based approaches and one of the main contributions of our paper. For this reason, we always retain projected-database filtering and discuss "with and without direct inclusion" in this experiment, while we assess the effect of projected-database filtering by using different batch sizes and compare with the existing filtering algorithms.

We test two versions of our algorithms: (1) with direct inclusion, denoted as IGquery-DI; and (2) without direct inclusion, denoted as IGquery-noDI. Figure 4 reports the average processing time per query and the average number of candidates obtained by each algorithm. In Figure 4(b), we also show the size of the answer set (in **bold** line) as a reference point, which is

the lower bound on the candidate set size of all existing filtering approaches.

We report the results for both count-based and time-based batches. For the count-based batches, we test three fixed sizes: 1, 10, and 100. For the time-based batch, a variable number of queries may come in for each batch, for which we randomly select x number of queries ($x \in [1, 100]$) for each incoming batch.



(a) Processing Time (b) Size of Candidate Set

Fig. 4 Query Performance on *AIDS*: Effects of Individual Components and Batch Processing

We first discuss the effects of direct inclusion and projected-database filtering on query processing. As shown in Figure 4(a), when we take away direct inclusion from IGquery, with only projected-database filtering our algorithm (IGquery-noDI) is faster than cIndex and GPTree only when the batch size increases to around 10. This result can be explained by Figure 4(b) which shows that the number of candidates obtained by IGquery-noDI decreases when the batch size increases.

If we do not take into account batch processing, i.e., when batch size is equal to 1, IGquery-noDI is worse than cIndex and GPTree, since projected-database filtering favors batch processing than single-query processing. However, when direct inclusion is applied, our algorithm (IGquery-DI) improves significantly and is considerably faster than both cIndex and GPTree even when batch size is 1.

We remark that when batch size is 1, our algorithm (denoted as IGquery-1 in subsequent experiments) do not utilize the commonality among the query graphs for query processing; in other words, the queries are processed one by one as in cIndex and GPTree.

When batch processing is enabled, the advantage of IGquery over cIndex and GPTree is immediately seen even for a batch size as small as 10, and the improvement is substantial when the batch size increases to 100. The reason for the improvement is because for a larger batch, more commonality among the queries is being shared. These common structures are processed only once but their results can serve as partial results for many queries, thereby significantly improving the query performance. The performance of the time-based batch is better than that of the batch size 10 but worse than that of the batch size 100, because the average number of queries in a time-based batch is around 50. Therefore, no matter in which batch mode (count-based or time-based), the (average) batch size affects the query performance.

From Figure 4(b), we see that with batch processing, even IGquery-noDI achieves better filtering than cIndex and GPTree. This is because with batch processing, we devise the concept of projected-database filtering. By using the projected database of the supergraphs of the queries, we can generate less candidates than filtering by the query graph alone as in cIndex and GPTree, which is evidenced by IGquery-noDI when the batch size is 10, 100, or time-based in Figure 4(b).

More importantly, the size of candidate set obtained by IGquery-DI is even much smaller than the size of the answer set, which is a bottleneck of all existing filtering approaches. This is because without direct inclusion, the candidate set must include at least all the answer graphs and therefore cannot be smaller than the answer set (as is the case in cIndex and GPTree). By direct inclusion, part of the answer set is directly included and needs not be recorded in the candidate set for further verification; hence we can generate a candidate set that is smaller than the answer set. Note that the difference between the number of candidates obtained by IGquery-noDI and that by IGquery-DI is the number of answer graphs obtained by direct inclusion, which is close to the number of total answer graphs

Finally, the peak memory consumption of IGquery-noDI and IGquery-DI is approximately 21 MB in all cases, which is essentially the size of the IG in memory. The peak memory consumption of cIndex and GPTree is 30 MB and 114 MB, respectively. Thus, our algorithm also consumes less memory than the existing approaches, which demonstrates the compactness of the IG.

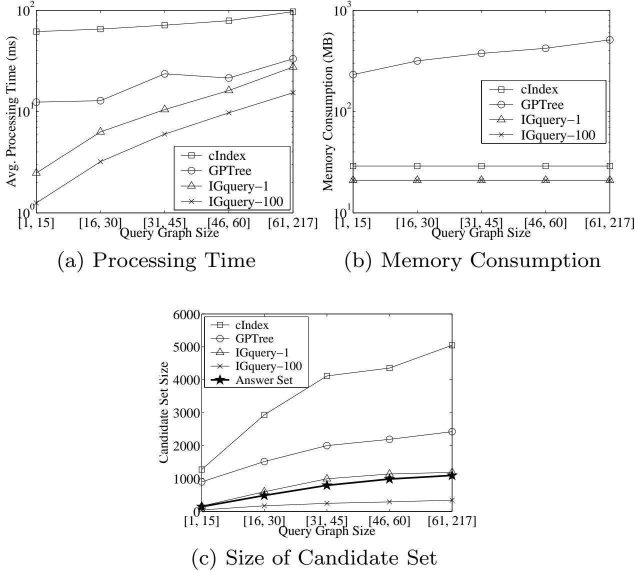
7.4 Query Performance: Effect of Query Graph Size

In this experiment, we assess the effect of query graph size on the performance of IGquery. We use five query sets, with the following graph size ranges: $[1, 15]$, $[16, 30]$, $[31, 45]$, $[46, 60]$, and $[61, 217]$, where the largest query graph has 217 edges.

Figure 5 reports the average processing time per query, the peak memory consumption, and the average number of candidates obtained by each algorithm. We report the results for two batch sizes, 1 and 100, represented by IGquery-1 and IGquery-100 in the figures.

Note that IGquery-1 means that the algorithm processes the queries one by one as in cIndex and GPTree

and the commonality among the queries is not utilized for query processing. Therefore, IGquery-1 is essentially a special case of batch processing when the batch size is equal to 1; that is, IGquery-1 still performs direct inclusion and project-database filtering. In this special case, the common subgraph used in direct inclusion and the common supergraph used in filtering are essentially the query graph itself.



(a) Processing Time (b) Memory Consumption

(c) Size of Candidate Set

Fig. 5 Query Performance on *AIDS*: Effect of Query Graph Size

Figure 5(a) shows that the query processing time of all algorithms increases when the query size increases, which is also consistent with the number of candidates as shown in Figure 5(c). For all query sizes, both IGquery-1 and IGquery-100 are over an order of magnitude faster than cIndex. Compared with GPTree, IGquery-1 is considerably faster and IGquery-100 is up to an order of magnitude faster. The results also show that our algorithm can handle large query graphs very efficiently. The average processing time for queries with size [61, 217] is only 15ms. IGquery-1 and IGquery-100 also use substantially less memory than GPTree and are stable in memory consumption. The memory consumption of IGquery-1 and IGquery-100 are almost the same since the majority of the memory is used to keep the IG.

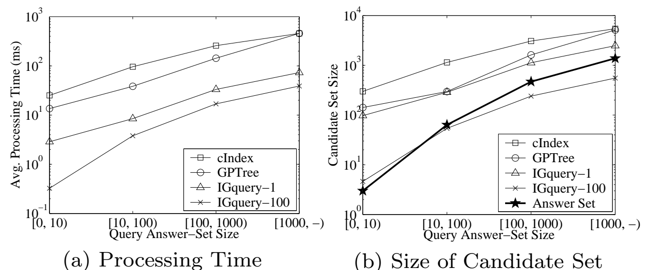
Our projected-database filtering employs common supergraphs of the queries to perform filtering; however, filtering by supergraph in general generates more candidates than filtering by the query graph itself. But in IGquery-1 this supergraph is essentially the query graph itself (i.e., the same as cIndex and GPTree), and thus the filtering effect of IGquery-1 mainly comes from the features being selected, as in cIndex and GPTree. The filtering effect of IGquery-1 (without direct inclusion) is comparable to that of the pure-filtering approaches, cIndex and GPTree (see Figure 4(b)). But IGquery-1 also uses direct inclusion that enables part of the answers to be directly included without being recorded in the candidate set, which makes a major difference from the filtering approaches. As a result, with effective filtering and the help of direct inclusion, IGquery-1 is able to obtain fewer candidates than both cIndex and GPTree as shown in Figure 5(c).

Finally, we note that for supergraph query processing with an index, the processing time depends largely on the candidate set size. As the query graph size increases, the answer set size and so the candidate set size would likely increase too. Both cIndex and GPTree need to verify the entire candidate set which is at least as large as the answer set, while our method has direct inclusion to overcome this bottleneck. Therefore, we expect that our algorithm is able to handle larger query graphs than cIndex and GPTree.

7.5 Query Performance: Effect of Query Answer-Set Size

In this experiment, we assess the effect of query answer-set size on the performance of IGquery. We divide the queries into four bins: $[0, 10)$, $[10, 100)$, $[100, 1000)$, and $[1000, \infty)$, according to the size of the query answer sets.

Figure 6 reports the average processing time per query and the average number of candidates obtained by each algorithm. We do not show the peak memory consumption in the figures, which is 21 MB for both IGquery-1 and IGquery-100, approximately 25 MB for cIndex, and 111 MB for GPTree.



(a) Processing Time (b) Size of Candidate Set

Fig. 6 Query Performance on *AIDS*: Effect of Query Answer-set Size

Figure 6(a) shows that for all sizes of query answer set, IGquery-1 is almost an order of magnitude faster than cIndex and about five times faster than GPTree. When the batch size increases, the improvement further enlarges. The speed-up of IGquery-100 over IGquery-1 is an order of magnitude for queries with a smaller

answer set size, and more than twice for queries with other answer set sizes. We emphasize that the speed-up is very significant considering that IGquery-1 is already very fast.

The query performance can be explained by the candidate set size shown in Figure 6(b). The results show that our direct inclusion and projected-database filtering techniques are more effective than cIndex and GPTree. In Table 7 we also report the number of answers obtained by direct inclusion (DI), which is very close to the number of all query answers.

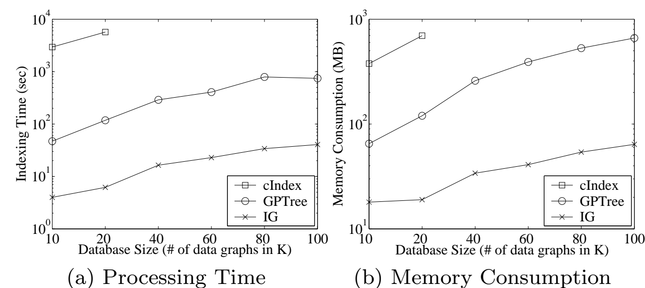
Table 7 Average Number of Answers by DI

	$[0, 10)$	$[10, 100)$	$[100, 1000)$	$[1000, \infty)$
# of ans by DI	2	60	342	909
# of exact ans	3	64	469	1383

7.6 Effect of Database Size

We also evaluate whether the performance of IGquery is affected by the database size. We use the six databases prepared from the *NCI* dataset for this experiment.

Indexing Performance. Figure 7 reports the time and peak memory consumption of index construction for each index. We are not able to obtain cIndex for databases with 40K graphs or more due to its higher memory consumption.



(a) Processing Time (b) Memory Consumption

Fig. 7 Indexing Performance on *NCI*: Effect of Database Size

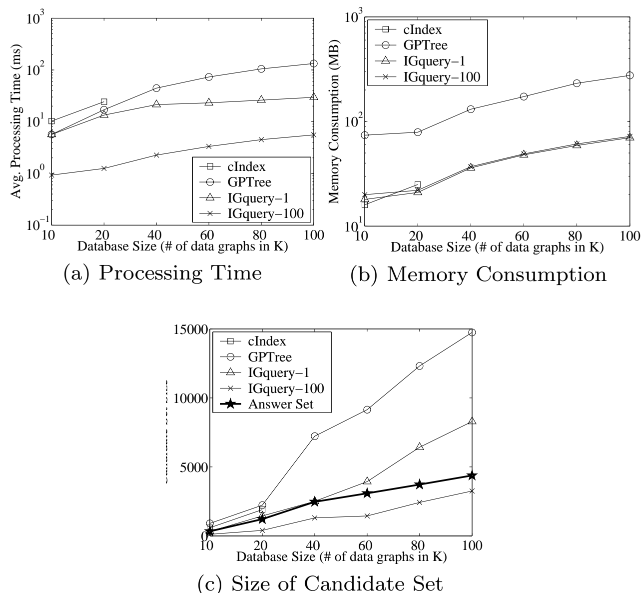
Similar to the previous experiment, our index construction is several orders of magnitude faster than cIndex and over an order of magnitude faster than GPTree, using significantly less memory. Both indexing time and memory usage of all the three indexes increase with database size.

An important observation of this experiment is that when the database size increases, the size of the IG (in terms of # of edges) does not increase, as shown in Table 8. This result reflects that real datasets from the same source/application share much commonality, and the IG can indeed effectively capture the commonality in the data.

Table 8 IG/GDB size (# of edges) for *NCI*

10K	20K	40K	60K	80K	100K
$\frac{568}{100634}$	$\frac{552}{187097}$	$\frac{557}{401917}$	$\frac{553}{608984}$	$\frac{564}{819324}$	$\frac{568}{1025636}$

Query Performance. Figure 8 reports the average processing time per query, the peak memory consumption, and the average number of candidates obtained by each algorithm.



(a) Processing Time (b) Memory Consumption

(c) Size of Candidate Set

Fig. 8 Query Performance on *NCI*: Effect of Database Size

Figure 8(a) shows that the processing time grows steadily when the database becomes larger, which is unavoidable since the answer set size also increases accordingly. However, compared with cIndex, IGquery is more scalable. We note that cIndex is not scalable mainly because its feature selection is too expensive. Compared with GPTree, IGquery is from several times to over an order of magnitude more efficient, depending on the batch size. Figure 8(b) shows that the memory consumption of cIndex and IGquery is comparable, and is much less than that of GPTree. For both IGquery and GPTree, the querying time and memory usage increase only linearly with the database size.

The processing time in Figure 8(a) can be clearly explained by the size of the candidate set reported in Figure 8(c). Due to the application of direct inclusion, IGquery is able to obtain a candidate set much smaller than cIndex and GPTree. When a larger batch size is used, the candidate set obtained by IGquery can be even significantly smaller than the answer set.

7.7 Effect of Commonality

In this subsection, we assess the effect of different degree of commonality on the performance of our index. We find that all the real datasets used in the literature for testing subgraph/supergraph queries share much commonality among the data graphs; thus, we use synthetic datasets to tune the degree of commonality. Most existing work uses frequent subgraphs as commonality; thus, we also use frequent subgraphs to indicate the degree of commonality existing in a dataset.

We generate four datasets with four different degrees of commonality as follows: $C1$:(0, -); $C2$:(718, 101); $C3$:(959391, 286); $C4$:($>$ 1 billion, 1363). Take $C2$:(718, 101) as an example, it means that at a minimum support threshold of 0.01, the dataset $C2$ has 718 frequent subgraphs and the average frequency of these frequent subgraphs is 101. Thus, the graphs in $C1$ share the least commonality (in fact, a large number of the graphs do no share any commonality at all), while those in $C4$ share the most commonality.

We prepare the query sets and the graph databases from each synthetic dataset in a similar way as we do for the real datasets. Each query set has 10K graphs and an average size of 20 edges. The average density of the graphs in the query sets varies from 0.05 to 0.5 for $C1$ to $C4$. Each graph database has 10K graphs and an average size of 10 edges. The average density of the graphs in the graph databases varies from 0.06 to 0.67 for $C1$ to $C4$.

Indexing Performance. Table 9 shows that our index construction is up to many orders of magnitude more efficient than cIndex. We are only able to obtain the result of GPTree for $C3$ (perhaps some special cases are not handled in GPTree). For $C3$, our performance is about twice better than GPTree.

The results also indicate that it takes longer time to construct our index on datasets with little commonality. This is because more new edges need to be created for the IG when the data graphs share little commonality, which is also reflected by the number of edges in the IG. The ratio of the IG size to the database size also correctly reveals the degree of commonality of the datasets.

Table 9 Performance of Indexing on Synthetic Data: Effect of Commonality

	$C1$	$C2$	$C3$	$C4$
cIndex (time: *sec*)	61	6429	482	376
GPTree (time: *sec*)	-	-	1	-
IG (time: *sec*)	3	0.3	0.5	0.8
cIndex (memory: *MB*)	425	420	382	427
GPTree (memory: *MB*)	-	-	47	-
IG (memory: *MB*)	24	24	23	22
cIndex (feature #)	11	94	583	2
GPTree (feature #)	-	-	4	-
IG (feature #)	328	131	870	581
$\frac{\text{IG size (\# of edges)}}{\text{GDB size (\# of edges)}}$	$\frac{58686}{81384}$	$\frac{21862}{163021}$	$\frac{4183}{165326}$	$\frac{578}{170639}$

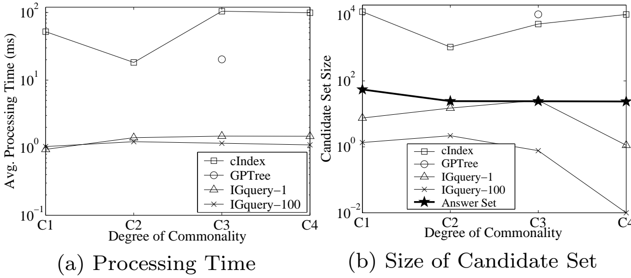
On the contrary, the performance of cIndex is rather unstable. This is perhaps due to the large variation in the set of frequent subgraphs used for their feature selection. For example, there are too few frequent subgraphs for $C1$, while there are too many frequent subgraphs for $C4$ even at a high frequency threshold but they are not discriminative enough to be used as good features. This also explains why so few features are selected by cIndex for $C1$ and $C4$. Our feature selection, however, does not require to first mine the frequent subgraphs.

Query Performance. Figure 9 reports the average processing time per query and the number of candidates (averaged over all queries). The peak memory consumption is 21-24 MB for IGquery, 16-24 MB for cIndex, and 61MB for GPTree (for $C3$ only).

Figure 9(a) shows that compared with GPTree for $C3$, IGquery is over an order of magnitude faster. Compared with cIndex, IGquery is up to two orders of magnitude faster. The results are also reflected by the size of candidate set shown in Figure 9(b). The fact that the candidate set size of IGquery is significantly smaller than the answer set size also reveals that direct inclusion is effective. The performance of cIndex is the best for $C2$; however, indexing $C2$ is also substantially more costly with cIndex as shown in Table 9.

Another interesting observation from Figure 9(a) is that IGquery-1 is the first time more efficient than IGquery-100 for $C1$. The reason for this uncommon result is that the graphs in the dataset $C1$ share very little commonality such that batch processing no longer has the advantage, but rather has the disadvantage as it needs to process more for the batching.

We also notice that the efficiency improvement of IGquery-100 over IGquery-1 is not large for this experiment. This is mainly because the sizes of the answer sets on the synthetic data are only a few dozens.



(a) Processing Time (b) Size of Candidate Set

Fig. 9 Query Performance on Synthetic Data: Effect of Commonality

Given the small number of candidates in both IGquery-1 and IGquery-100, the index probing time T_{search} dominates the overall query processing time, while T_{search} is roughly the same for IGquery-1 and IGquery-100.

Finally, Figure 9(a) also shows that the performance of IGquery is quite stable over the different degrees of commonality. We explain this result by considering the index probing time. The number of matches for a query in a lower-commonality dataset is smaller; thus, although the corresponding IG is larger, index probing only searches a few places in the IG for matching the query. On the other hand, the number of matches for a query in a higher-commonality dataset is larger, but the corresponding IG is smaller and hence index probing also only searches a few places in the IG for matching the query. Therefore, the index probing time is roughly the same for datasets of different commonality. Since the index probing time dominates the overall query time for processing the synthetic datasets, the query time is stable with different degrees of commonality.

7.8 Evaluation on Database Updates

In this experiment, we show that in addition to efficient index construction and fast query processing, our index also has a very low maintenance cost.

We consider three different scenarios of updates: insertion only, deletion only, and a random mix of both, which are denoted as *insert*, *delete*, and *mix* in the discussion. We use the database graphs prepared from the *NCI* dataset and perform updates of 10K graphs as follows. For insertion only, we start with a database of 10K graphs and insert another 10K graphs. For deletion only, we start from a database with 30K graphs and randomly delete 10K graphs from it. For a mix of both, we start with a database of 20K graphs and randomly choose to insert graphs into it from another 10K graphs or delete graphs from it. The final database size we obtain at the end of all updates in each case is 20K.

Table 10 reports the total time and peak memory taken to update the 10K graphs, including graph insertion/deletion and updates of features. Clearly, the index update is very efficient as it takes only about 0.1 second to update 10K graphs (on average it takes about 0.01 millisecond to insert/delete a graph). The *mix* update scenario takes longer time due to the more frequent increase and decrease in the size of $host(e)$ for an edge e that causes more frequent re-ordering of the $host(e)$. As for the memory consumption, the *delete* takes more memory only because we start with a larger database (30K graphs initially) and we record the peak memory consumption.

Table 10 Performance on Index Maintenance

	insert	*delete*	*mix*
Total update time (msec)	99.12	76.90	117.95
Memory consumption (MB)	21	30	22

Table 11 shows the size of the IG (in terms of number of edges) of the 20K database at the end of each update scenario. We also compare it with the size of the IG constructed from-scratch for the 20K database, denoted as *rebuild* in the table. It is shown that the size of the IG obtained by incremental update is almost the same as that obtained by rebuilding from-scratch.

Table 11 Size Ratio of IGs and Query Processing Time

	insert/rebuild	*delete/rebuild*	*mix/rebuild*
IG size ratio	554 / 552	557 / 552	554 / 553
Query time (ms)	4.03 / 3.99	3.14 / 3.01	3.69 / 3.58

Finally, we also show in Table 11 that the query performance on the incrementally updated database does not degrade (or only very slightly) when compared with the query performance on the database rebuilt from-scratch. The memory consumption for query processing on the incrementally updated database is the same as that on the database rebuilt from-scratch, because their IGs have essentially the same size as shown in Table 11.

8 Related Work

Chen et al. propose cIndex [2] for processing supergraph queries. They model the feature selection as the problem of maximum coverage with cost. Therefore, filtering with their features is effective. Zhang et al. propose GPTree [22]. They organize the database graphs

with a tree structure to allow testing subgraph isomorphism from multiple graphs to one graph. A set of significant frequent subgraphs with high filtering power is selected as features. However, the effectiveness of the features in both cIndex and GPTree comes with a tradeoff: the feature generation and selection process involves frequent subgraph mining, which is slow and hard to maintain for database updates. On the contrary, our feature selection process is very efficient and we are able to achieve higher filtering power by our projected-database filtering approach. More importantly, we propose a new technique of direct inclusion, which enables us to outperform cIndex and GPTree. We also further improve our work by exploring the commonality among the queries.

A number of indexes have been proposed for processing subgraph queries [13,20,5,10,16,21,3,23,24,12]. Most of these methods are filtering-and-verification approaches. The exceptions [16,3] cannot be adopted to process supergraph queries because the number of supergraphs of the database graphs is infinite and thus cannot be indexed. A number of these indexes are also extended to handle similarity matching. Similarity search has not been studied for supergraph queries and may be considered as a future work.

9 Conclusions

We propose a query system, IGquery, for processing supergraph queries on transaction graph databases. IGquery constructs an index by extracting commonality among the graphs. When little commonality exists, IGquery can also efficiently locate the matching graphs in the index. IGquery has the following distinguished features: (1) low index construction cost and low index maintenance cost (suitable for dynamic databases); (2) fast query processing by the dual operations of direct inclusion and filtering; (3) capable of batch processing and handling high-speed query streams.

Experimental results verify that: (1) On index construction, IGquery is orders of magnitude more efficient than cIndex [2] and GPTree [22]. The index maintenance cost is also shown to be indeed very small. (2) On query processing, IGquery is up to two orders of magnitude faster and is also more stable, even when the queries are processed one by one (i.e., IGquery-1). The query performance improves further by at least several times when queries are processed as batches. (3) Experiments also show that IGquery is efficient on datasets with both low or high degree of commonality.

Acknowledgements We would like to thank Mr. Chen Chen for providing us cIndex and Mr. Shuo Zhang for providing us GPTree.

References

1. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
2. C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu. Towards graph containment search and indexing. In *VLDB*, pages 926–937, 2007.
3. J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD Conference*, pages 857–872, 2007.
4. L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.
5. H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.
6. J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining protein family specific residue packing patterns from protein structure graphs. In *RECOMB*, pages 308–315, 2004.
7. J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.
8. A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.
9. C. A. James, D. Weininger, and J. Delany. Daylight theory manual daylight version 4.82. *Daylight Chemical Information Systems, Inc.*, 2003.
10. H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, pages 566–575, 2007.
11. Y. Koren, S. C. North, and C. Volinsky. Measuring and extracting proximity in networks. In *KDD*, pages 245–255, 2006.
12. H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *In VLDB*, 2008.
13. D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.
14. H. Tong and C. Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD*, pages 404–413, 2006.
15. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
16. D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985, 2007.
17. D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.
18. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
19. X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD*, pages 286–295, 2003.
20. X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.
21. S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, pages 966–975, 2007.
22. S. Zhang, J. Li, H. Gao, and Z. Zou. A novel approach for efficient supergraph query processing on graph databases. In *EDBT*, pages 204–215, 2009.

23. P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.
24. L. Zou, L. C. 0002, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.