Efficient Query Processing on Graph Databases

JAMES CHENG
The Hong Kong University of Science and Technology
YIPING KE
The Chinese University of Hong Kong
and
WILFRED NG
The Hong Kong University of Science and Technology

We study the problem of processing *subgraph queries* on a database that consists of a set of graphs. The answer to a subgraph query is the set of graphs in the database that are supergraphs of the query. In this article, we propose an efficient index, *FG*-index*, to solve this problem.

The cost of processing a subgraph query using most existing indexes mainly consists of two parts, the *index probing* cost and the *candidate verification* cost. Index probing is to find the query in the index, or to find the graphs from which we can generate a candidate answer set for the query. Candidate verification is to test whether each graph in the candidate set is indeed a supergraph of the query. We design FG*-index to minimize these two costs as follows.

FG*-index consists of three components: the *FG-index*, the *feature-index*, and the *FAQ-index*. First, the FG-index employs the concept of *Frequent subGraph* (*FG*) to allow the set of queries that are FGs to be answered without candidate verification. We call this set of queries *FG-queries*. We can enlarge the set of FG-queries so that more queries can be answered without candidate verification; however, a larger set of FG-queries implies a larger FG-index and hence the index probing cost also increases. We propose the feature-index to reduce the index probing cost. The feature-index uses features to filter false results that are matched in the FG-index, so that we can quickly find the truly matching graphs for a query. For processing non-FG-queries, we propose the FAQ-index, which is dynamically constructed from the set of *Frequently Asked non-FG-Queries* (*FAQs*). Using the FAQ-index, verification is not required for processing FAQs and only a small number of candidates needs to be verified for processing non-FG-queries that are *not frequently asked*. Finally, a comprehensive set of experiments verifies that query processing using FG*-index is up to orders of magnitude more efficient than state-of-the-art indexes and it is also more scalable.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems - Query processing

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Graph Databases, Graph Indexing, Graph Query Processing, Frequent Subgraphs

Author's address: James Cheng, Department of Computer Science and Engineering, HKUST, Clear Water Bay, Hong Kong; email: james.cheng@family.ust.hk.
Author's address: Yiping Ke, Department of Systems Engineering and Engineering Management, CUHK, New Territories, Hong Kong; emails: ypke@se.cuhk.edu.hk.
Author's address: Wilfred Ng, Department of Computer Science and Engineering, HKUST, Clear Water Bay, Hong Kong; emails: wilfred@cse.ust.hk.

1. INTRODUCTION

Graph is a powerful tool for representing and understanding objects and their relationships in various application domains. In recent years, graph databases have become more in use and the volume of graph data increases rapidly. However, the performance of query processing on graph databases is still inadequate due to the high complexity of processing graph data. As a result, it is important to develop efficient algorithms for processing queries on graph databases.

Existing research has been conducted mainly on two types of graph databases. The first type consists of very large graphs, such as the Web graph and social networks. Typical querying tasks for such graph databases include finding the best connection between a given set of query nodes [Faloutsos et al. 2004; Koren et al. 2006; Tong and Faloutsos 2006] and finding subgraphs that match a given query pattern [Güting 1994; Holder et al. 1994; Tong et al. 2007].

The second type is a database that consists of a large set of small graphs such as chemical compounds. This type of databases is especially popular in scientific domains such as chemistry [James et al. 2003] and bio-informatics [Huan et al. 2004]. Typical queries for this type of databases include *subgraph queries* and *similarity queries*. A subgraph query retrieves all the graphs in the database that are supergraphs of a given query graph [Shasha et al. 2002; Yan et al. 2005a; He and Singh 2006; Jiang et al. 2007; Williams et al. 2007; Zhang et al. 2007; Cheng et al. 2007; Zhao et al. 2007], while a similarity query retrieves all the graphs that are structurally similar to a given query graph [Yan et al. 2005b; He and Singh 2006; Jiang et al. 2007; Williams et al. 2007]. These two types of queries have a wide range of applications such as motif discovery in 3D protein structures, pathway discovery in protein interaction graphs, drug design, schema matching, correlation discovery in graph databases [Ke et al. 2007; 2008], and many more.

In this article, we propose an efficient index to process subgraph queries in a database that consists of a set of small graphs. In most of the existing work, a similarity query is processed by evaluating the set of relaxed graphs of the query graph using the index for processing subgraph queries. Thus, our work can also be extended to handle similarity queries in a similar way.

Let $\mathcal{D}$ be a graph database that consists of a set of graphs. The processing of a subgraph query is described as follows: *given a graph q, retrieve all graphs $g \in \mathcal{D}$ such that g is a supergraph of q*. Processing a subgraph query is a fundamental operation for querying graph databases. However, due to the diversity of graphs, a subgraph query is in general very complex, since any part (i.e., any subgraph) of the query graph is a predicate that needs to be satisfied in the query evaluation. Processing the query by a sequential scan on $\mathcal{D}$ to check whether q is a subgraph of each $g \in \mathcal{D}$ is infeasible, since *subgraph isomorphism* testing is known as an *NP-complete* problem [Cook 1971].

In recent years, many efficient indexes [Shasha et al. 2002; Yan et al. 2005a; He and Singh 2006; Jiang et al. 2007; Zhang et al. 2007; Zhao et al. 2007] have been proposed to process subgraph queries on graph databases. Query processing using

these indexes has the following two main steps: *filtering* and *candidate verification*. First, filtering uses the index to eliminate part of the false results and to produce a candidate answer set. Then, candidate verification tests whether each candidate is indeed a supergraph of the query. Since the candidate answer set is in general much smaller than the entire graph database, query processing using the indexes is significantly more efficient than the sequential scan approach.

As pointed out by the authors of the above-mentioned indexes, the cost of candidate verification is the dominating factor in the cost of processing a subgraph query. Therefore, the indexes aim at reducing the candidate set as much as possible. However, due to the high complexity of subgraph isomorphism testing, candidate verification is still the most expensive part in processing a subgraph query since the size of the candidate answer set is at least that of the exact answer set.

Among the existing indexes, *GDIndex* [Williams et al. 2007] is an exception that does not require candidate verification. However, GDIndex is designed mainly for processing databases that contain relatively smaller graphs.

The recently proposed *FG-index* [Cheng et al. 2007] makes an advance in handling the expensive candidate verification process. FG-index is an index defined based on the concept of *Frequent subGraphs* (*FGs*) [Inokuchi et al. 2000]. An FG is a graph that is a subgraph of at least $(\sigma \cdot |\mathcal{D}|)$ graphs in $\mathcal{D}$, where σ $(0 \leq \sigma \leq 1)$ is a predefined threshold. Since the set of FGs can be large, Cheng et al. define the notion of *δ-Tolerance Closed FGs* (*δ-TCFGs*) to cluster the FGs and to organize them into levels. FG-index is then built as a tree-structured index, so that the search space of the index probing is effectively reduced. The parameter δ determines the size of a cluster and hence controls the size of the index at each level.

FG-index classifies queries into two categories: *FG-queries* and *non-FG-queries*, which are queries that are FGs and not FGs, respectively. The main advantage of FG-index over other existing indexes is that no candidate verification is needed for processing FG-queries. For processing non-FG-queries, FG-index is also able to obtain a small candidate answer set to reduce the cost of candidate verification.

There is a problem in FG-index inherited from the concept of FGs. In order to answer a larger set of queries without candidate verification, a small σ should be used to build the index. However, a smaller σ implies a larger index and hence a higher index probing cost. In the index probing process, we need to match the query with the indexed graphs and each matching involves a subgraph isomorphism test. Thus, although the candidate verification cost is lowered, the index probing cost may become too high.

In this article, we address this problem by incorporating a *feature-based search strategy* into FG-index. We compute a set of *features* and build an index, called the *feature-index*, on the features. Since features possess the structural information of the indexed graphs, the feature-index can find a matching graph quickly without processing many false results, thereby effectively reducing the number of subgraph isomorphism tests performed in the index probing process. As a result, we are able to use a small σ to process a large set of queries without candidate verification and with a low index probing cost.

In addition to the improvement in the index probing efficiency, our work makes another advance over FG-index. Using FG-index, the size of the candidate set for a

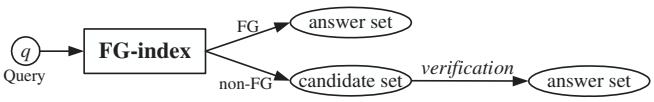


Fig. 1. The Underlying Principle of Query Processing using FG-Index

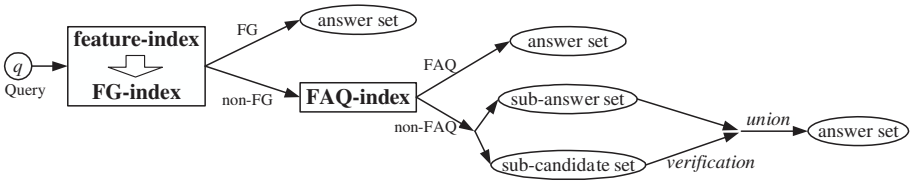


Fig. 2. The Underlying Principle of Query Processing using FG*-Index

non-FG-query is at best close to $(\sigma \cdot |\mathcal{D}|)$, since the candidate set is generated from the indexed graphs, which are FGs. We eliminate this bound on the candidate set size for processing non-FG-queries. The candidate set size can now be smaller than the answer set size and even zero (i.e., no candidate verification).

We achieve this by proposing a new index, called the *FAQ-index*. We model the set of all queries as a stream and define the notion of *Frequently Asked non-FG-Queries* (*FAQs*) within a sliding window. Then, the FAQ-index is constructed on the set of FAQs and is dynamically updated for each window slide. When a query is an FAQ, the FAQ-index answers the query without any candidate verification. When the query is not an FAQ, the FAQ-index is able to obtain a subset of the answer set and to generate only a small number of candidates for verification.

We incorporate the FG-index, the feature-index and the FAQ-index coherently into a unified index framework, called *FG*-index*. Figure 2 depicts the underlying principle of processing a query q using FG*-index, where the counterpart using FG-index is shown in Figure 1. Using FG-index, non-FG-queries must go through candidate verification. In FG*-index, we first employ the feature-index to improve the efficiency of the index probing process. Then, non-FG-queries are answered efficiently using the FAQ-index.

We verify the performance of FG*-index with a comprehensive set of experiments. We demonstrate that the use of the feature-index significantly improves the index probing efficiency, while the use of the FAQ-index significantly reduces the cost of candidate verification. We also show that FG*-index significantly outperforms FG-index with a series of comparisons between the two indexes. In addition, we compare FG*-index with two other state-of-the-art graph indexes, *gIndex* [Yan et al. 2005a] and *C-tree* [He and Singh 2006]. The results show that FG*-index is up to orders of magnitude faster and consumes significantly less memory than gIndex and C-tree. We further show that FG*-index is much more scalable than gIndex and C-tree when we increase the database size as well as the graph density.

Although efficient query processing is the primary objective of this work, efficient index construction and update maintenance are also important concerns. We show that the index construction cost depends mainly on the parameter σ, or equivalently

the number of FGs. Smaller σ results in faster query processing, but higher index construction cost. However, we find that when σ increases, query performance degrades only slightly but the performance of the index construction improves exponentially. Thus, we can build the index with a smaller σ if the index can be built during system idle time or if query performance is critical. Otherwise, we can build the index with a relatively larger σ, which still achieves very impressive query performance according to our extensive experimental results, especially compared with the other indexes. We also suggest guidelines to set σ, as well as δ and other parameters used in the index, based on the experimental results.

For index maintenance, we propose a batch-update strategy that builds an auxiliary FG*-index to process queries on the set of updated graphs and rebuilds the entire index only when the update overhead becomes more expensive than rebuilding the index. This method is simple and can handle frequent database updates. Finally, we verify the efficiency of our update strategy with a set of experiments.

Organization. The rest of the article is organized as follows. Section 2 defines the preliminary concepts. Section 3 presents FG-index. Section 4 conducts a detailed analysis on FG-index, identifying its merits as well as its limitations. Section 5 proposes FG*-index, discussing in detail both the feature-index and the FAQ-index. Section 6 discusses the update of FG*-index. Section 7 reports the experimental results. Section 8 discusses the related work and Section 9 gives the conclusion.

2. PRELIMINARIES

For simplicity in presentation, we restrict our discussion to *undirected, labelled connected graphs*. However, our index and query processing algorithms apply in the same way to directed graphs. In this article, we simply call an undirected, labelled connected graph a graph.

A graph g is defined as a 4-tuple (V, E, L, l), where V is the set of vertices, E is the set of edges, L is the set of labels and l is a labelling function that maps each vertex or edge to a label in L. We define the *size* of a graph g as the number of edges in g, denoted as $size(g) = |E(g)|$.

Given a set of graphs G, a *distinct edge* in G is defined as a 3-tuple, (l_u, l_e, l_v), where l_e is the label of an edge (u, v) in a graph $g \in G$, and l_u and l_v are the labels of vertices u and v in g. Given a distinct edge e and a graph g in G, we define the *count* of e in g, denoted as $count(e, g)$, as the number of occurrences of e in g.

Given two graphs, $g = (V, E, L, l)$ and $g' = (V', E', L', l')$, a *subgraph isomorphism* from g to g' is an injective function h: $V \rightarrow V'$, such that $\forall (u, v) \in E$, $(h(u), h(v)) \in E'$, $l(u) = l'(h(u))$, $l(v) = l'(h(v))$, and $l(u, v) = l'(h(u), h(v))$.

A graph g is called a *subgraph* of another graph g' (or g' is a *supergraph* of g), denoted as $g \subseteq g'$ (or $g' \supseteq g$), if there exists a subgraph isomorphism from g to g'. We call g a *proper subgraph* of g', denoted as $g \subset g'$, if $g \subseteq g'$ and $g \not\supseteq g'$.

2.1 Frequent Subgraphs

Let $\mathcal{D}$ be a graph database that consists of a set of graphs. Given a graph f, the *frequency* of f in $\mathcal{D}$, denoted as $freq(f)$, is defined as the number of graphs in $\mathcal{D}$ that are supergraphs of f; that is, $freq(f) = |\{g : g \in \mathcal{D}, g \supseteq f\}|$. A graph f is called a *Frequent subGraph* (FG) [Inokuchi et al. 2000] if $freq(f) \geq (\sigma \cdot |\mathcal{D}|)$, where

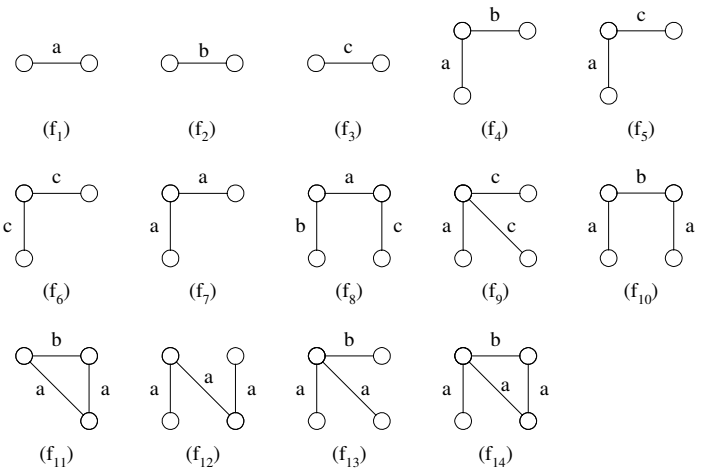


Fig. 3. Frequent Subgraphs

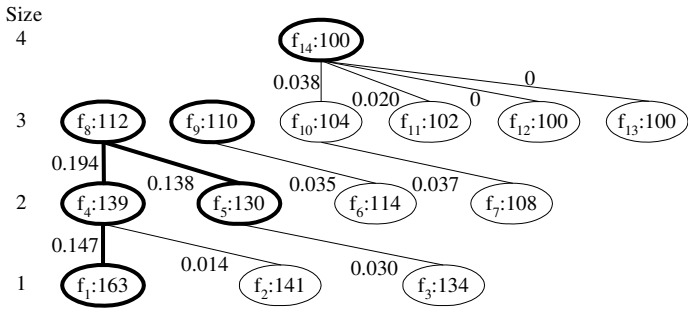


Fig. 4. Frequent Subgraphs and Their Frequency

σ $(0 \leq \sigma \leq 1)$ is a pre-defined *minimum frequency threshold*.

Let $\mathcal{F}$ be the set of all FGs that are mined from $\mathcal{D}$. A graph f is called a *Maximal Frequent subGraph* (*MFG*) [Huan et al. 2004] if $f \in \mathcal{F}$ and $\nexists f' \in \mathcal{F}$ such that $f' \supset f$. A graph f is called a *Closed Frequent subGraph* (*CFG*) [Yan and Han 2003] if $f \in \mathcal{F}$ and $\nexists f' \in \mathcal{F}$ such that $f' \supset f$ and $freq(f') = freq(f)$.

Example 2.1. Figure 3 shows 14 FGs, $f_1, \ldots, f_{14}$, mined from a graph database, where a, b, c represent three distinct edges (note that a distinct edge represents a unique tuple consisting of the labels of an edge and its incident vertices). Figure 4 organizes the FGs according to their size and represents each FG as a node, where the number following ":" is the frequency of the FG. (The number on an edge in Figure 4 is to be introduced later in other examples in this article.)

Among the FGs, f_8, f_9 and f_{14} are MFGs since they have no proper supergraphs. All the FGs, except f_{12} and f_{13}, are CFGs. The FGs f_{12} and f_{13} are not CFGs because they have a supergraph f_{14} which has the same frequency.

Table I. Notations Used Throughout

Symbol	Description
$\mathcal{D}$	the graph database
$size(g)$	the number of edges in a graph g
$count(e, g)$	the number of occurrences of a distinct edge e in g
$freq(f)$	the number of graphs in $\mathcal{D}$ that are supergraphs of a graph f
σ	the minimum frequency threshold ($0 \leq \sigma \leq 1$)
$\mathcal{F}$	the set of all FGs mined from $\mathcal{D}$ w.r.t. σ
q	a query graph
$\mathcal{D}_q$	the answer set of a query q (the set of supergraphs of q in $\mathcal{D}$)
δ	the frequency tolerance factor ($0 \leq \delta \leq 1$)
$\mathcal{T}$	the set of δ-TCFGs mined from $\mathcal{D}$
$\preceq$	a total order defined on $\mathcal{F}$ by Definition 3.5
$CLOS(f_t)$	the closure of a δ-TCFG f_t
GA[i]	the i-th entry of the GA of an IGI or an IFI
$IDA(e, n, m)$	the m-edge ID-array in the size-n ID-entry of a distinct edge e
$\mathcal{C}_q$	the candidate answer set of a query q
$\mathcal{F}_l^u$	the feature set that contains all FGs of size between $[l, u]$
w	the number of time units in a sliding window W
N_{FAQ}	the number of FAQs in a sliding window
M	the size of the available memory
$\mathcal{D}_{del}$	the set of deleted graphs from $\mathcal{D}$
$\mathcal{D}_{new}$	the set of new graphs added to $\mathcal{D}$

2.2 Subgraph Queries

A *subgraph query*, or simply a *query*, is a graph that has at least one edge. Processing a query with a single vertex is trivial and thus not discussed.

The *query processing problem* we tackle in this article is stated as follows. *Given a graph database $\mathcal{D}$ and a query q, retrieve all $g \in \mathcal{D}$ such that g is a supergraph of q.* The *answer set* of a query q is denoted as $\mathcal{D}_q = \{g\colon g \in \mathcal{D}, g \supseteq q\}$.

Given a minimum frequency threshold σ, a query q is called an *FG-query* with respect to σ if $|\mathcal{D}_q| \geq (\sigma \cdot |\mathcal{D}|)$, since $freq(q) = |\mathcal{D}_q| \geq (\sigma \cdot |\mathcal{D}|)$ and hence q is an FG. If $|\mathcal{D}_q| < (\sigma \cdot |\mathcal{D}|)$, then q is a *non-FG-query* with respect to σ.

Table 2.2 gives the notations used throughout this article.

3. FG-INDEX

In this section, we present *FG-index* [Cheng et al. 2007]. We first define the notion of *δ-tolerance CFGs*, which forms the core of FG-index. Then, we discuss the construction of FG-index and query processing using FG-index.

3.1 δ-Tolerance Closed Frequent Subgraphs

FG-index is a tree-structured index built on the set of FGs, $\mathcal{F}$. We define the notion of δ-tolerance CFGs (δ-TCFGs) to cluster the FGs in $\mathcal{F}$ so that they can be organized into levels. The notion of δ-TCFGs also allows us to flexibly tune the size of the index at each level by adjusting the value of δ.

We define the notion of δ-TCFG as follows.

Definition 3.1. (*δ-Tolerance CFG*) A graph, $f \in \mathcal{F}$, is a *δ-Tolerance CFG* (δ-

TCFG) if and only if $\nexists f' \in \mathcal{F}$ such that $f' \supset f$ and $freq(f') \geq ((1-\delta) \cdot freq(f))$, where δ $(0 \leq \delta \leq 1)$ is a pre-defined *frequency tolerance factor*.

We can define CFGs and MFGs by δ-TCFGs as follows.

LEMMA 3.2. *A graph f is a CFG if and only if f is a 0-TCFG. A graph f is an MFG if and only if f is a 1-TCFG.*

COROLLARY 3.3. *Let $\mathcal{T}_\delta$ be the set of δ-TCFGs, $\mathcal{F}_C$ be the set of CFGs, and $\mathcal{F}_M$ be the set of MFGs. Then, $\mathcal{F}_M \subseteq \mathcal{T}_\delta \subseteq \mathcal{F}_C$.*

Corollary 3.3 gives the upper bound and the lower bound on the size of $\mathcal{T}_\delta$. The following example illustrates the concept of δ-TCFGs.

Example 3.4. Consider the 14 FGs in Figure 4. The number on each edge is computed as $d_e = (1 - freq(f_i)/freq(f_j))$, where f_i is the smallest proper supergraph of f_j that has the greatest frequency. According to Definition 3.1, f_j is a δ-TCFG iff $d_e > \delta$. Let $\delta = 0.04$. Then, the set of 0.04-TCFGs is $\{f_1, f_4, f_5, f_8, f_9, f_{14}\}$, i.e., the set of bold nodes in Figure 4. For example, f_1 is a 0.04-TCFG since f_1 does not have a proper supergraph that has a frequency greater than $((1-0.04) \times 163) \approx 156$. The FG f_6 is not a 0.04-TCFG since we have $freq(f_9) > ((1 - 0.04) \times freq(f_6))$.

The set of 1-TCFGs, i.e., the set of MFGs, is $\{f_8, f_9, f_{14}\}$; while the set of 0-TCFGs, i.e., the set of CFGs, contains all FGs except f_{12} and f_{13}.

From now on, we use the lighter notation $\mathcal{T}$ to represent $\mathcal{T}_\delta$ when δ is clear in the context. To create clusters from $\mathcal{F}$ based on $\mathcal{T}$, we need to find the connection between the graphs in $\mathcal{T}$ and those in $(\mathcal{F} - \mathcal{T})$. We establish this connection by defining the closure of a δ-TCFG.

To define the closure of a δ-TCFG, we need to first define a total order on $\mathcal{F}$. We assign a unique number, $num(f)$, to each graph $f \in \mathcal{F}$. Then, we define the total order on $\mathcal{F}$ as follows.

Definition 3.5. (*Graph Set Order*) A *graph set order* $\preceq$ on $\mathcal{F}$ is a total order defined as follows. Let $f_1, f_2 \in \mathcal{F}$, $f_1 \preceq f_2$ if one of the following statements is true.

(1) $size(f_1) < size(f_2)$.
(2) $size(f_1) = size(f_2)$ and $freq(f_1) > freq(f_2)$.
(3) $size(f_1) = size(f_2)$, $freq(f_1) = freq(f_2)$, and $num(f_1) \leq num(f_2)$.

We further define $f_1 \prec f_2$ if $f_1 \preceq f_2$ and $f_1 \neq f_2$.

Based on the graph set order, we now define the notion of the closest δ-TCFG supergraph to build the connection of the graphs in $\mathcal{T}$ and those in $(\mathcal{F} - \mathcal{T})$.

Definition 3.6. (*Closest δ-TCFG Supergraph*) Given $f_t \in \mathcal{T}$ and $f \in (\mathcal{F} - \mathcal{T})$, f_t is called the *closest δ-TCFG supergraph* of f if $f_t \supset f$ and $\nexists f'_t \in \mathcal{T}$ such that $f'_t \supset f$ and $f'_t \prec f_t$.

LEMMA 3.7. *For each $f \in (\mathcal{F} - \mathcal{T})$, the closest δ-TCFG supergraph of f exists and is unique.*

PROOF. We first prove the existence of f's closest δ-TCFG supergraph. If $f \in (\mathcal{F} - \mathcal{T})$ does not have any supergraph in $\mathcal{T}$, then f itself must be an MFG.

According to Corollary 3.3, all MFGs are δ-TCFGs. Therefore, we have $f \in \mathcal{T}$, which contradicts the assumption that $f \in (\mathcal{F} - \mathcal{T})$.

The uniqueness of f's closest δ-TCFG supergraph follows directly from Definitions 3.5 and 3.6. □

Based on Definition 3.6 and Lemma 3.7, we can assign to each δ-TCFG, $f_t \in \mathcal{T}$, with a cluster of FGs whose closest δ-TCFG supergraph is f_t. We define this cluster of FGs as the closure of a δ-TCFG as follows.

Definition 3.8. (*Closure of a δ-TCFG*) Given $f_t \in \mathcal{T}$, the *closure* of f_t is defined as $CLOS(f_t) = \{f : f_t \text{ is the closest } \delta\text{-TCFG supergraph of } f\}$.

Based on the graph set order, Lemma 3.7 ensures that a query $q \in (\mathcal{F} - \mathcal{T})$ must have a unique closest δ-TCFG supergraph, f_t, and q can be located in the closure of f_t. We illustrate the concept of closure by the following example.

Example 3.9. Referring to Figures 3 and 4, the set of FGs is ordered according to the graph set order $\preceq$, where $num(f_i) = i$. We have $f_1 \prec f_4$ since $size(f_1) < size(f_4)$; while for f_1 and f_2 which are of the same size, $f_1 \prec f_2$ since $freq(f_1) > freq(f_2)$. When $\delta = 0.04$, f_{14} is the closest δ-TCFG supergraph of $f_7, f_{10}, f_{11}, f_{12}$ and f_{13}; in other words, $CLOS(f_{14}) = \{f_7, f_{10}, f_{11}, f_{12}, f_{13}\}$.

3.2 Framework of FG-Index

Before we present FG-index, we first give the framework of FG-index as follows.

(1) *Index Construction.* The construction of FG-index consists of two major steps. The first step is to mine $\mathcal{F}$, which can be done by using an existing FG-mining algorithm [Inokuchi et al. 2000; Yan and Han 2002]. Note that $\mathcal{D}_f$ of each $f \in \mathcal{F}$ is also obtained by the FG-mining process. The second step is to compute $\mathcal{T}$ from $\mathcal{F}$ and then build the index based on $\mathcal{T}$.

FG-index consists of two parts: the *core FG-index* and the *edge-index*. We briefly describe each of them as follows.

The core FG-index is a tree-structured index. The root of the tree is an inverted-index constructed on the set $\mathcal{T}$. Then, a child inverted-index is built on the closure of each $f \in \mathcal{T}$. If a closure is too large, a local set of δ-TCFGs is computed from the closure. In this way, we construct the tree recursively. We keep the root of the core FG-index in the main memory and other nodes on the disk. We also associate $\mathcal{D}_f$ with each indexed FG f.

The core FG-index is built on the set of FGs and hence is not able to answer those non-FG-queries that do not have an edge in any of the FGs. To process these queries, we build another index, called the *edge-index*, on the set of infrequent distinct edges[1] in $\mathcal{D}$. For each infrequent distinct edge e in the edge-index, we also associate $\mathcal{D}_e$ with e.

(2) *Query Processing.* Given a query q, we first search q in the core FG-index. If q is a δ-TCFG, we directly retrieve q and $\mathcal{D}_q$ from the inverted-index at the root of the core FG-index. Otherwise, we first find q's closest δ-TCFG supergraph,

[1]A distinct edge can be regarded as a graph with only one edge.

f. Then, the index constructed on the closure of f is loaded from the disk to locate q and $\mathcal{D}_q$.
If q cannot be found in the core FG-index, then q is a non-FG-query. In this case, we use the core FG-index to find a set of q's subgraphs and retrieve $\mathcal{D}_f$ for each of these subgraphs f. If q consists of any infrequent distinct edges, we also retrieve $\mathcal{D}_e$ from the edge-index for each infrequent distinct edge e in q. Then, we compute C_q as the intersection of all the "$\mathcal{D}_f$"s and all the "$\mathcal{D}_e$"s. Finally, we obtain $\mathcal{D}_q$ by verifying whether each $g \in C_q$ is a supergraph of q.

3.3 Index Construction

We now present the structure of FG-index and algorithm for constructing FG-index.

3.3.1 *Structure of FG-Index.* We first define the structure of FG-index. The edge-index is a simple hashtable that keeps the set of infrequent distinct edges. For the edge e in each non-empty hash slot, we also link $\mathcal{D}_e$ to the slot.

The core FG-index is a multi-level index tree, where a node in the tree is an *Inverted-Graph-Index* (*IGI*) constructed on a cluster of FGs. The structure of an IGI is formally defined as follows.

Definition 3.10. (*Inverted-Graph-Index*) Given a set of graphs G, an *Inverted-Graph-Index* (*IGI*) constructed on G consists of the following components:

—An array, called the *Graph Array* (*GA*), stores G.
—An array, called the *Edge Array* (*EA*), stores the set of distinct edges in G.
—Each distinct edge e in the EA is associated with a set of IDs of the graphs that contain e. The set of IDs is organized as follows.
  —The IDs are first organized by the size of the graphs. The IDs of the graphs that are of size n are grouped together in an entry, called a *size-n ID-entry*.
  —Within each size-n ID-entry, the IDs are further organized by the number of occurrences of e in each of the graphs. The IDs of the graphs that have m occurrences of e are grouped together in an array, called an *m-edge ID-array*.

We assign the ID of a graph as the position that the graph is stored in the GA. For example, the graph stored in the i-th entry of the GA, denoted as GA[i], is given the ID "i". We also denote *the m-edge ID-array in the size-n ID-entry* of a distinct edge e in the EA as $IDA(e, n, m)$. We use the following example to illustrate the structure of an IGI.

Example 3.11. Referring to the FGs in Figures 3 and 4, let $\delta = 0.04$, then $\mathcal{T} = \{f_1, f_4, f_5, f_8, f_9, f_{14}\}$. Figure 5 shows the corresponding IGI constructed on $\mathcal{T}$. For example, the size-3 ID-entry of the distinct edge c has two ID-arrays: the 1-edge ID-array, denoted as $IDA(c, 3, 1)$, containing one ID "4", and the 2-edge ID-array, denoted as $IDA(c, 3, 2)$, containing one ID "5". The two IDs correspond to f_8 and f_9 in GA[4] and GA[5], respectively.

As shown in Figure 3, f_9 is of size 3, $count(a, f_9) = 1$ and $count(c, f_9) = 2$. In the IGI shown in Figure 5, f_9 is stored in GA[5]. Thus, we have the ID "5" in $IDA(a, 3, 1)$ and $IDA(c, 3, 2)$.

We now describe the structure of the core FG-index. A conceptual view of the core FG-index is shown in Figure 6. The root of the core FG-index, or simply called

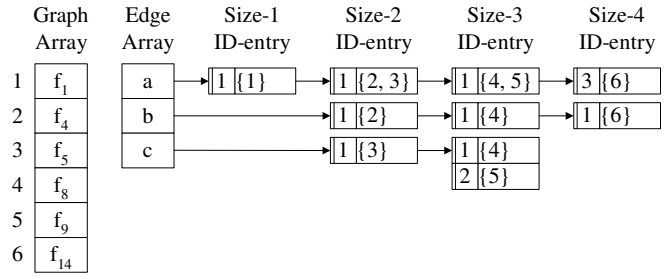


Fig. 5. Inverted-Graph-Index of Example 3.11

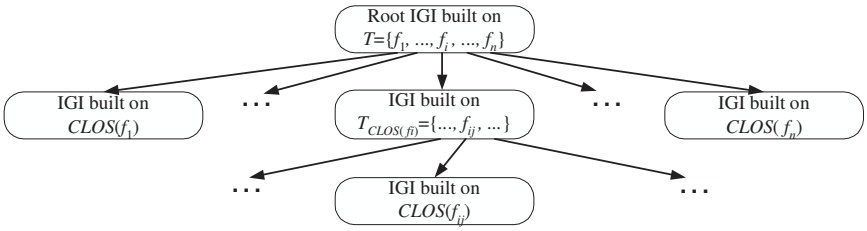


Fig. 6. The Conceptual View of the Core FG-Index

Algorithm 1 BuildIndex

Input: $\mathcal{D}$, σ and δ.
Output: FG-index.

1. $\mathcal{F} \leftarrow$ **MineFG**$(\mathcal{D}, \sigma)$;
2. $\mathcal{T} \leftarrow$ **ComputeTCFG**$(\mathcal{F}, \delta)$;
3. **BuildCoreFGindex**$(\mathcal{T}, \mathcal{F}, |\mathcal{T}|)$;
4. **BuildEdgeIndex**$(\mathcal{D}, \sigma)$;

the *root IGI*, is an IGI constructed on $\mathcal{T}$. Then, for each $f_i \in \mathcal{T}$, if $CLOS(f_i) \neq \emptyset$, a *child IGI* is constructed on $CLOS(f_i)$. However, if the size of $CLOS(f_i)$ is larger than the size of $\mathcal{T}$, a *local* set of δ-TCFGs, $\mathcal{T}_{CLOS(f_i)}$, is computed from $CLOS(f_i)$ (using a smaller δ). The child IGI is then constructed on $\mathcal{T}_{CLOS(f_i)}$ instead of $CLOS(f_i)$. Thus, when the child IGI is loaded into the main memory, the memory consumption is guaranteed to at most double the size of the root IGI. Then, for each $f_{ij} \in \mathcal{T}_{CLOS(f_i)}$, a child IGI is constructed on $CLOS(f_{ij})$ and so on recursively.

3.3.2 *Constructing FG-Index.* The algorithm for constructing FG-index, *BuildIndex*, is presented as Algorithm 1. The algorithm first invokes *MineFG* to mine $\mathcal{F}$ from $\mathcal{D}$ with respect to σ. Then, *ComputeTCFG* is invoked to compute $\mathcal{T}$ from $\mathcal{F}$ with respect to δ. Finally, *BuildCoreFGindex* and *BuildEdgeIndex* are invoked to construct the core FG-index and the edge-index. We omit the details of the procedure MineFG, which can be any existing FG-mining algorithm [Inokuchi et al. 2000; Yan and Han 2002], but we discuss the other three procedures as follows.

Procedure 2 **ComputeTCFG**($\mathcal{F}_{this}, \delta$)

1. Sort $\mathcal{F}_{this}$ s.t. $\forall f_1, f_2 \in \mathcal{F}_{this}$, f_1 is ordered before f_2 if $f_1 \prec f_2$;
2. $\mathcal{T}_{this} \leftarrow \mathcal{F}_{this}$;
3. Let T_i be the set of FGs in $\mathcal{T}_{this}$ that consist of i edges;
4. **for each** $i = 1, 2, \ldots$ **do**
5. **for each** $f \in T_i$ **do**
6. **for each** $f' \in T_{i+1}$ **do**
7. **if** ($f \subset f'$)
8. **if** ($freq(f') \geq (1-\delta) \cdot freq(f)$)
9. $\mathcal{T}_{this} \leftarrow \mathcal{T}_{this} - \{f\}$;
10. **break**; */* go to Line 5 */*
11. Return $\mathcal{T}_{this}$;

Procedure 3 **BuildCoreFGindex**($\mathcal{T}_{this}, \mathcal{F}_{this}, N$)

1. Create an empty IGI, with an empty GA and an empty EA;
2. **for each** $f_t \in \mathcal{T}_{this}$ **do**
3. Store f_t in the first free entry in the GA;
4. **for each** distinct edge e in f_t **do**
5. **if** (e is not in the EA)
6. Add e to the EA;
7. Add the ID of f_t to $IDA(e, size(f_t), count(e, f_t))$;
8. **if**($\mathcal{F}_{this} \neq \mathcal{T}_{this}$)
9. **for each** $f \in (\mathcal{F}_{this} - \mathcal{T}_{this})$ **do**
10. Find f's closest δ-TCFG supergraph, f_t;
11. Add f to $CLOS(f_t)$;
12. **for each** $f_t \in \mathcal{T}_{this}$ **do**
13. **if**($CLOS(f_t) \neq \emptyset$)
14. **if**($|CLOS(f_t)| \geq N$)
15. $\mathcal{T}_{CLOS(f_t)} \leftarrow$ **ComputeTCFG**($CLOS(f_t), \delta$);
16. **BuildCoreFGindex**($\mathcal{T}_{CLOS(f_t)}, CLOS(f_t), N$);
17. **else**
18. **BuildCoreFGindex**($CLOS(f_t), CLOS(f_t), N$);

Procedure 4 **BuildEdgeIndex**($\mathcal{D}, \sigma$)

1. Create an empty edge-index;
2. **for each** distinct edge e that appears in less than $\sigma|\mathcal{D}|$ graphs **do**
3. Add e and $\mathcal{D}_e$ to the edge-index;

ComputeTCFG, as shown in Procedure 2, first sorts the set of input FGs $\mathcal{F}_{this}$ (note that the sorting does not involve any expensive graph operation). Based on the order defined by $\prec$, the first supergraph f' of a graph f found in $\mathcal{F}_{this}$ has the greatest frequency among all other supergraphs of f. Thus, if $freq(f') < ((1-\delta) \cdot freq(f))$, then $\forall f'' \supset f$, $freq(f'') \leq freq(f') < ((1-\delta) \cdot freq(f))$. This implies that, to check whether f is a δ-TCFG, we only need to find the first supergraph of f that has one more edge than f (Lines 6-10). If the first supergraph of f, f', is found and $freq(f') \geq ((1-\delta) \cdot freq(f))$ (Lines 7-8), then f is not a δ-TCFG by

Definition 3.1. Thus, f is removed from $\mathcal{T}_{this}$ (Line 9). Otherwise, f is a δ-TCFG.

BuildCoreFGindex, as shown in Procedure 3, recursively constructs the core FG-index as follows. We first build an IGI on $\mathcal{T}_{this}$ (Lines 2-7). For each $f_t \in \mathcal{T}_{this}$, we store f_t in the GA in the order that f_t is processed. For each distinct edge e in f_t, if e is not in the EA, we add e to the EA. Then, the ID of f_t is added to the end of the array $IDA(e, size(f_t), count(e, f_t))$. Since the ID of f_t is assigned as the position in the GA where f_t is stored, the IDs in each ID-array are automatically sorted. We use a hashtable to access each distinct edge in the EA.

After we build the IGI on $\mathcal{T}_{this}$, Line 8 checks if $\mathcal{T}_{this}$ is the closure of a δ-TCFG. If $\mathcal{T}_{this}$ is the closure of a δ-TCFG, i.e., $\mathcal{T}_{this} = \mathcal{F}_{this}$, then we do not need to construct any child IGI. Otherwise, $\mathcal{T}_{this}$ is a set of δ-TCFGs and Lines 9-11 compute the closure for each $f_t \in \mathcal{T}_{this}$. We can use the IGI built on $\mathcal{T}_{this}$ to find the closest δ-TCFG supergraph of a graph (Line 10) efficiently, which will be discussed in Algorithm 5 when we process a query using the IGI. Finally, Lines 12-18 recursively call BuildCoreFGindex to construct the child IGI on the closure of each $f_t \in \mathcal{T}_{this}$. If the closure of some f_t is still too large, ComputeTCFG is first called to construct a nested set of δ-TCFGs on $CLOS(f_t)$, using a smaller δ.

BuildEdgeIndex, as shown in Procedure 4, adds each infrequent distinct edge in $\mathcal{D}$ to the edge-index. The set of infrequent distinct edges can be obtained freely from the FG-mining process. These edges are also accessed via the same hashtable used for the EA, where a flag is used to indicate whether or not an edge is frequent.

3.3.3 *Memory and Disk Residence.* We keep the root IGI and the edge-index in the main memory, and we store all the other parts of FG-index on the disk.

The $\mathcal{D}_f$ for each indexed graph f is stored on the disk. Given two graphs f and f', if $f \supset f'$, then $(\mathcal{D}_f \cap \mathcal{D}_{f'}) = \mathcal{D}_f$. Thus, we do not want to store the duplicate graphs in $\mathcal{D}_f$ and $\mathcal{D}_{f'}$. We remove the duplicates as follows. For each $f \in \mathcal{T}$, we organize the FGs in $CLOS(f)$ as a tree. The root of the tree is f and the parent of a node in the tree is the first supergraph (as ordered by $\prec$) of the node. This supergraph can be found efficiently using the child IGI that is built on $CLOS(f)$. Then, we only keep $(\mathcal{D}_{f_c} - \mathcal{D}_{f_p})$ at each node f_c, where f_p is the parent of f_c. Thus, only $\mathcal{D}_f$ is exact, while the duplicate graphs in $\mathcal{D}_{f'}$, where $f' \in CLOS(f)$, are removed and can be recovered by traversing from f' up to the root f.

3.4 Query Processing using FG-index

The processing of a query q using FG-index is classified into two cases: when q is an FG-query and when q is a non-FG-query.

3.4.1 *Processing FG-Queries.* When q is an FG-query, Algorithm 5 invokes Procedure 6 to process q recursively, starting at the root IGI. Let E be the set of distinct edges in q. *ProcFGQbyIGI* checks only those graphs that contain all edges in E. It starts with the graphs that have the same size as q (Line 2) until a supergraph of q is found (Lines 9-14).

Let i be the size of the graphs that are indexed in *thisIGI*. For each $e \in E$, ProcFGQbyIGI first obtains $K(e)$, which is the set of IDs of the graphs that are of size i and have at least $count(e, q)$ occurrences of e (Lines 5-6). The IDs in each $K(e)$ are sorted in ascending order. Then, the "$K(e)$"s for all $e \in E$ are intersected to find a supergraph for q. Let f be the first supergraph of q, whose ID is obtained by

Algorithm 5 ProcFGQ

Input: The core FG-index and a query q.
Output: $\mathcal{D}_q$.

1. Return **ProcFGQbyIGI**(The root IGI, q);

Procedure 6 ProcFGQbyIGI($thisIGI, q$)

1. Let E be the set of distinct edges in q;
2. **for each** $i = size(q), size(q)+1, \ldots$ **do**
3. **for each** $e \in E$ **do**
4. Create an empty set, $K(e)$;
5. **for each** $j \geq count(e, q)$ **do**
6. Access $IDA(e, i, j)$ in *thisIGI* and copy the IDs in $IDA(e, i, j)$ to $K(e)$;
7. Sort $K(e)$ in ascending order;
8. Intersect $K(e)$, $\forall e \in E$, until an ID, k, is obtained;
9. **if**(f in GA[k] of *thisIGI* is a supergraph of q)
10. **if**($f = q$)
11. Return $\mathcal{D}_f$;
12. **else**
13. Load f's child IGI, *childIGI*, into the main memory;
14. Return **ProcFGQbyIGI**($childIGI, q$);
15. **else**
16. Go to *Line* 8 and continue the intersection;
17. Return $\emptyset$;

the intersection (Lines 8-9). According to the order in which each graph is added to the GA, f must be either q or the closest δ-TCFG supergraph of q. Thus, we either output $\mathcal{D}_q$ (Line 11), or continue to find q by recursively invoking ProcFGQbyIGI to process on f's child IGI (Lines 13-14). If the intersection of the "$K(e)$" is an empty set or if no supergraph of q is found for the current i, we increment i (Line 2) and continue a new round of iteration to search for a supergraph of q. Finally, Line 17 shows a terminating condition, which indicates that q is not an FG-query and we process q by Algorithm 7 in Section 3.4.2.

The efficiency of the intersection of the "$K(e)$"s depends on the size of each $K(e)$. The IDs in each $K(e)$ belong to a local set of δ-TCFGs that are of a specific size and contain at least $count(e, q)$ occurrences of e. Thus, the size of $K(e)$ is small, because the size of the whole set of δ-TCFGs is small as controlled by δ.

The following example illustrates the processing of an FG-query by Algorithm 5.

Example 3.12. Referring to the IGI in Example 3.11, let $q = f_{11}$. We demonstrate how $\mathcal{D}_q$ is obtained by ProcFGQbyIGI. Since $size(f_{11}) = 3$, we start from the Size-3 ID-entries, that is, $i = 3$. Since $IDA(a, 3, j)$ is empty, for $j \geq count(a, f_{11}) = 2$, we have $K(a) = \emptyset$, which implies that the intersection of $K(a)$ and $K(b)$ will also be an empty set. Therefore, ProcFGQbyIGI directly proceeds to $i = 4$ in Line 2. We first copy the ID "6" from $IDA(a, 4, 3)$ to $K(a)$. Since $count(b, f_{11}) = 1$, we also copy the ID "6" from $IDA(b, 4, 1)$ to $K(b)$. Then, intersecting $K(a)$ and $K(b)$ gives "6". Since f_{14} in GA[6] is a supergraph of f_{11} (in fact, the closest δ-TCFG

Algorithm 7 ProcNonFGQ

Input: FG-index, and a query q.
Output: $\mathcal{D}_q$.

```
1.  Create an empty set, S;
2.  Let E be the set of frequent distinct edges of q;
3.  for each i = size(q) − 1, size(q) − 2, ..., 1 do
4.      Create an empty set, K;
5.      for each e ∈ E do
6.          for each j = 1, ..., count(e, q) do
7.              if(IDA(e, i, j) is not empty)
8.                  Copy the IDs in IDA(e, i, j) to K;
9.      Sort K in descending order and remove the duplicate IDs;
10.     for each ID, k, in K do
11.         if(f in GA[k] has edges in E and f ⊂ q)
12.             S ← (S ∪ {D_f});
13.             Remove all distinct edges of f from E;
14.             Go to Line 15 if E becomes empty;
15. for each infrequent distinct edge, e, in q do
16.     S ← (S ∪ {D_e});
17. C_q ← (⋂_{s∈S} s);
18. Return D_q ← {g : g ∈ C_q, g ⊇ q};
```

supergraph of f_{11}), Line 14 invokes ProcFGQbyIGI to process on the child IGI of f_{14}. The recursive call finally returns $\mathcal{D}_{f_{11}}$ (details omitted).

3.4.2 *Processing Non-FG-Queries.* When ProcFGQ returns an empty set, then q is a non-FG-query. In this case, Algorithm 7 is used to obtain $\mathcal{D}_q$. The algorithm ProcNonFGQ consists of two parts: Lines 2-14 check the set of frequent distinct edges E (if any) in q, while Lines 15-16 handle the set of infrequent distinct edges (if any). In Line 1, ProcNonFGQ assigns an empty set S, which is used to hold the answer sets of q's subgraphs. Intersecting the answer sets in S then gives the candidate set of q, $\mathcal{C}_q$, in Line 17.

First, in Lines 2-14, ProcNonFGQ uses the core FG-index to find a set of subgraphs of q that are indexed. Then, for each subgraph f found, $\mathcal{D}_f$ is retrieved and included into S. Then, in Lines 15-16, ProcNonFGQ retrieves $\mathcal{D}_e$ for each infrequent distinct edge e of q from the edge-index and includes $\mathcal{D}_e$ into S. Finally, in Lines 17-18, ProcNonFGQ generates $\mathcal{C}_q$ by intersecting all ID sets (i.e., $\mathcal{D}_f$ or $\mathcal{D}_e$) in S, and verifies each candidate in $\mathcal{C}_q$ to give $\mathcal{D}_q$.

We now explain how to search for the subgraphs of q that are indexed in FG-index. Unlike the search for the supergraph of q in Procedure 6, the search for subgraphs moves in the reverse direction starting with the graphs that have one fewer edge than q (Line 3). Then, the IDs of the graphs are copied to a set K and sorted in descending order (Lines 4-9), since for graphs of the same size, a graph f with a larger ID implies that f has a smaller frequency and hence a smaller $\mathcal{D}_f$.

For each ID in K, Lines 10-11 perform a subgraph isomorphism test between f and q to ensure that f is a subgraph of q before using $\mathcal{D}_f$ to produce $\mathcal{D}_q$. This process can be costly since K contains all potential subgraphs of q. To reduce the

number of subgraph isomorphism tests in this step, we obtain only a small set of *maximal FG subgraphs* of q. Here, f is a maximal FG subgraph of q iff $f \in \mathcal{F}$ and $\nexists f' \supset f$ such that $f' \subset q$. Using the maximal FG subgraphs of q is more effective in reducing the size of $\mathcal{C}_q$ because the answer set of a maximal FG subgraph is smaller than that of a non-maximal FG subgraph of q. However, we do not obtain all maximal FG subgraphs of q in the index but stop the search when all edges in E are covered (Lines 11-14), since obtaining all those missing maximal FG subgraphs does not further reduce the size of $\mathcal{C}_q$ substantially.

4. THE ANATOMY OF FG-INDEX: MERITS AND LIMITATIONS

In this section, we present a detailed analysis of the efficiency of query processing using FG-index. We identify the merits of using FG-index and also discuss the limitations. Then, in Section 5, we propose *FG*-index* to address the limitations.

Let $\mathcal{C}_q$ be the *candidate answer set* of processing a query q. Let T_{search} be the index probing time, $T_{I/O}$ be the disk I/O time of fetching each candidate graph from the disk, and T_{verify} be the candidate verification time.

The response time of processing q using a graph index is given as follows:

$$T_{response} = (\, T_{search} + |\mathcal{C}_q| \times T_{I/O} + |\mathcal{C}_q| \times T_{verify}) \; . \tag{1}$$

Since candidate verification involves the expensive operation of subgraph isomorphism testing, $(|\mathcal{C}_q| \times T_{verify})$ usually dominates $T_{response}$. Most existing indexes [Shasha et al. 2002; Yan et al. 2005a; He and Singh 2006; Jiang et al. 2007; Zhang et al. 2007] aim to reduce $|\mathcal{C}_q|$ as much as possible. Thus, the *optimal* $T_{response}$ of using these indexes is when $\mathcal{C}_q = \mathcal{D}_q$:

$$T_{response} = (\, T_{search} + |\mathcal{D}_q| \times T_{I/O} + |\mathcal{D}_q| \times T_{verify}) \; . \tag{2}$$

4.1 Merits of FG-Index

The major advantage of using FG-index over existing indexes is its efficiency for processing FG-queries. When q is an FG, using FG-index obtains $\mathcal{D}_q$ directly without any candidate verification. Thus, the response time is given as follows:

$$T_{response} = (\, T_{search} + |\mathcal{D}_q| \times T_{I/O}) \; . \tag{3}$$

Equation (3) is a significant reduction from Equation (2), because we completely remove the dominating factor, $(|\mathcal{C}_q| \times T_{verify})$, from $T_{response}$. We remark that the cost of retrieving the answer set from the disk, i.e., $(|\mathcal{D}_q| \times T_{I/O})$, is inevitable unless the main memory is large enough to store the whole database.

4.2 Limitations of FG-Index

Although FG-index is a significant improvement over existing indexes, there is a condition that must be satisfied in order to achieve the response time defined by Equation (3); that is, the queries must be FGs with respect to σ. This becomes a limitation in using FG-index.

To include more queries into the category of FGs, FG-index should use a small σ. However, a small σ produces a large number of FGs, which in turn gives rise

to a large index. Although the concept of δ-TCFG partitions the large indexing space into many smaller spaces at different levels, the search space can still be large when the number of FGs is large. The large search space leads to more subgraph isomorphism tests (Line 9 of Procedure 6) performed in the process of finding q's closest δ-TCFG supergraph. As a result, T_{search} is substantially increased.

Thus, the setting of σ becomes a limitation of FG-index in achieving the best query performance, as stated below:

—When σ is small, more queries can be answered by FG-index without candidate verification. The response time for processing the FG-queries is given by Equation (3), but T_{search} can be large.

—When σ is large, T_{search} is small but the best response time for processing most queries is given by Equation (2).

Another limitation of FG-index is on the processing of non-FG-queries. In order to reduce the candidate verification cost, FG-index generates $\mathcal{C}_q$ by intersecting the answer sets of the maximal FG subgraphs of q. Thus, $\mathcal{C}_q$ is close to $(\sigma \cdot |\mathcal{D}|)$ since the frequency of the maximal FG subgraphs is at least $(\sigma \cdot |\mathcal{D}|)$. However, $\mathcal{D}_q$ can be much smaller than $(\sigma \cdot |\mathcal{D}|)$ since q is a non-FG-query. Therefore, for processing non-FG-queries, the $\mathcal{C}_q$ obtained by FG-index may be much larger than $\mathcal{D}_q$.

5. FG*-INDEX

In Section 4, we identify two limitations of FG-index, one related to the index probing cost and another related to the candidate verification cost. In this section, we propose our solution, *FG*-index*, to both of the limitations. FG*-index consists of FG-index as well as two new indexes: the *feature-index* and the *FAQ-index*.

The feature-index is used to lower the index probing cost by reducing the number of subgraph isomorphism tests performed in the index probing process, even when the number of FGs is large.

The FAQ-index totally avoids candidate verification for processing *frequently asked non-FG-queries*. For processing those non-FG-queries that are not frequently asked, the FAQ-index improves the query performance in either of the following two ways: (1) the FAQ-index obtains a large subset of the answer set and thus only a small number of candidates need to be verified; or (2) the FAQ-index generates a small candidate set that is close to the answer set.

5.1 The Feature-Index

We process a query q using FG-index by intersecting the IDs of the size-i ($i \geq size(q)$) graphs that contain the edges in q. When the number of indexed graphs is large or the database is dense, this edge-based intersection may return a large number of matches, because edges lose most of the structural information of the graphs. Since each match needs to be verified by a subgraph isomorphism test, T_{search} is significantly increased as a result.

Our solution to this problem is to adopt a *feature-based search strategy*. We first define a set of features and then build an index on the features.

5.1.1 *Feature Selection.* The selection of features can rely on domain expert knowledge. However, considering that domain expert knowledge may not always be

available, we provide a way to select the features without domain expert knowledge.

To facilitate the index probing process, a desirable set of features should satisfy the following criteria. First, the features should possess the structural information of their supergraphs that are indexed. Second, the number of features cannot be large; otherwise, the search for the features of a query is itself too expensive. Third, it should be efficient to compute the features.

A suitable feature that satisfies the first criterion is the subgraphs of the graphs indexed in FG-index. However, it cannot meet the second criterion if all the subgraphs are used as features. Thus, we use only part of the subgraphs and define the feature set as $\mathcal{F}_l^u = \{f_e : f_e \in \mathcal{F}, l \leq size(f_e) \leq u\}$. We set $l = 2$ in $\mathcal{F}_l^u$, since the size-1 FGs are just frequent edges. The choice of u determines the first two criteria but also presents a dilemma: u should be set larger so that the features can possess more structural information of their supergraphs, but a larger u also means a larger number of features. However, u can be easily determined by running a few test sets. Our experiments show that, compared with the edge-based index probing, the search efficiency is already significantly improved for a value of u as small as 4. The last criterion for selecting the features is also satisfied since $\mathcal{F}_l^u$ is obtained freely from $\mathcal{F}$, which is used to build FG-index.

Another benefit of selecting $\mathcal{F}_l^u$ as the feature set is that it makes the index particularly efficient for answering queries that are small structures, which are commonly found in many applications [Williams et al. 2007].

5.1.2 *The Structure of the Feature-Index.* The feature-index consists of the following two components: the *Feature Hash Index* (*FHI*) and a set of *Inverted-Feature-Indexes* (*IFIs*).

The FHI is a simple hashtable that keeps the set of features. The hash key of a feature is computed from the canonical label [Williams et al. 2007] of the feature. If there are more than one feature being hashed into the same hashtable slot, the collision is handled by chaining. We compute the canonical label of a feature from the adjacency list of the feature.

Each feature f_e in the FHI is also associated with $\mathcal{D}_{f_e}$. Therefore, if a query q is a feature, its answer set can be directly retrieved using the FHI.

The structure of an IFI is defined as follows.

Definition 5.1. (*Inverted-Feature-Index*) Given a set of graphs G and a set of features F_e, an *Inverted-Feature-Index* (*IFI*) on G and F_e is defined as follows:

—An array, called the *Graph Array* (*GA*), stores G.

—An array, called the *Feature Array* (*FA*), stores F_e.

—Each feature f_e in the FA is associated with a set of IDs of the graphs in G that are supergraphs of f_e. The IDs are organized by the size of the graphs. The IDs of the graphs that are of size n are stored together in an array, called the *size-n ID-array* of f_e.

Recall from Definition 3.10 that each IGI in FG-index is built on a set of graphs G. Thus, we construct an IFI on G and $\mathcal{F}_l^u$ to improve the index probing efficiency. We keep $\mathcal{F}_l^u$ in the FHI and store it in the main memory, while each IFI is resident in the memory or on the disk according to their respective IGI. The construction

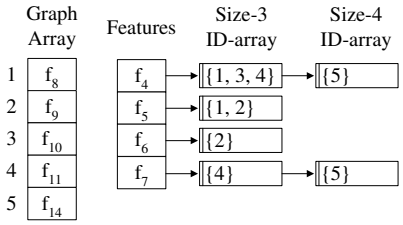


Fig. 7. The Inverted-Feature-Index of Example 5.2

of an IFI is similar to that of an IGI as shown in Lines 1-7 of Procedure 3; thus, we omit the details here. An example of an IFI is shown as follows.

Example 5.2. Referring to the FGs in Figures 3 and 4, for the purpose of illustration, we choose the feature set $\mathcal{F}_2^2 = \{f_4, f_5, f_6, f_7\}$ and we set $\delta = 0$, i.e., $\mathcal{T} = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{14}\}$. Figure 7 shows an IFI constructed on $\mathcal{T}$ and $\mathcal{F}_2^2$, where we omit $\{f_1, f_2, f_3, f_4, f_5, f_6, f_7\}$ from the GA for clarity. Note that the IFI and the IGI of FG-index share the same GA.

In Figure 7, the size-3 ID-array of the feature f_4 has three IDs, $\{1, 3, 4\}$, which correspond to the three size-3 supergraphs of f_4, $\{f_8, f_{10}, f_{11}\}$, in GA[1], GA[3] and GA[4], respectively.

We do not build an IFI for every IGI in FG-index, since an IFI is used to reduce the number of subgraph isomorphism tests performed in the index probing process when the IGI is large. Thus, we only build an IFI for the IGI at an intermediate node in the core FG-index, since these IGIs are usually large (otherwise they will not have child IGIs). The IGIs at a leaf node of the core FG-index are small; thus, no IFI is needed for the IGI at a leaf node.

5.1.3 *Query Processing Using the Feature-Index.* We now discuss query processing using the feature-index, for both FG-queries and non-FG-queries. As shown in Algorithm 8, we process a query according to its size. If $size(q) < l$, we process q using FG-index since the size of the features is at least l. If the size of q falls within the size range of a feature, i.e., $[l, u]$, we can directly retrieve $\mathcal{D}_q$ using the FHI if q is an FG. But if q is not an FG, FG-index is used to answer q. We will discuss how to improve the efficiency of processing non-FG-queries later using the FAQ-index.

When the size of q is greater than u, we first find a set of features that are subgraphs of q. For the purpose of facilitating index probing, we only need the set of *maximal features* of q, defined as $F_q = \{f_e : f_e \subset q, f_e \in \mathcal{F}_l^u, \nexists f'_e \in \mathcal{F}_l^u \; s.t. \; f'_e \supset f_e \; and \; f'_e \subset q\}$, since the maximal features contain the structural information of their subgraph features. Thus, we can enumerate the size-u subgraphs of q and then the size-$(u-1)$ subgraphs, and so on until we find all the maximal features.

However, the number of maximal features can still be large, especially because all the size-u subgraphs of q are maximal features of q. Therefore, we find a representative set of maximal features that contain all distinct edges of q. We compute this representative set by enumerating the features of q starting from the size-u features. Let E be the set of distinct edges in q. For each feature f_e enumerated, we remove all distinct edges in f_e from E. We add f_e to F_q and continue the

Algorithm 8 FProcQ

Input: FG-index, the feature-index, and a query q.
Output: $\mathcal{D}_q$.

1. **if**($size(q) < l$)
2. Process q using FG-index;
3. **else if**($l \leq size(q) \leq u$)
4. Search q in the FHI;
5. Return $\mathcal{D}_q$ if q is found, otherwise process q using FG-index;
6. **else if**($size(q) > u$)
7. Find a set of *maximal features* of q, F_q, using the FHI;
8. $\mathcal{D}_q \leftarrow$ **FProcFGQ**(q, F_q);
9. **if**($\mathcal{D}_q \neq \emptyset$)
10. Return $\mathcal{D}_q$;
11. **else**
12. Return **FProcNonFGQ**(q, F_q);

Procedure 9 FProcFGQ(q, F_q)

1. **for each** $i = size(q), size(q)+1, \ldots,$ **do**
2. Intersect the size-i ID-arrays of all $f_e \in F_q$ until an ID, k, is obtained;
3. **if**(f in GA[k] is a supergraph of q)
4. Return $\mathcal{D}_f$ if $f = q$, otherwise search q using f's child IGI and IFI (if any);
5. **else**
6. Go to *Line* 2 and continue the intersection;
7. Return $\emptyset$;

Procedure 10 FProcNonFGQ(q, F_q)

1. Create an empty set, S;
2. **for each** $i = size(q)-1, size(q)-2, \ldots, u+1$ **do**
3. **for each** unique ID, k, in the size-i ID-arrays of all $f_e \in F_q$ **do**
4. **if**(f in GA[k] is a subgraph of q)
5. $S \leftarrow (S \cup \{\mathcal{D}_f\})$;
6. Remove any f_e, whose size-i ID-array contains k, from F_q;
7. Go to *Line* 11 if F_q becomes empty;
8. **if**($F_q \neq \emptyset$)
9. **for each** $f_e \in F_q$ **do**
10. $S \leftarrow (S \cup \{\mathcal{D}_{f_e}\})$;
11. **for each** infrequent distinct edge, e, in q **do**
12. Obtain $\mathcal{D}_e$ from the edge-index in FG-index;
13. $S \leftarrow (S \cup \{\mathcal{D}_e\})$;
14. $\mathcal{C}_q \leftarrow (\bigcap_{s \in S} s)$;
15. Return $\mathcal{D}_q \leftarrow \{g : g \in \mathcal{C}_q, g \supseteq q\}$;

enumeration, until E becomes empty. Thus, we stop when all edges in E (i.e., q) are covered, since obtaining all those missing maximal features does not further improve the filtering power much (Line 3 of Procedure 9) but increases the cost of the intersection since we need to process more features (Line 2 of Procedure 9).

After we obtain F_q, we invoke *FProcFGQ*, as shown in Procedure 9, that uses the IFI to process q. We intersect the size-i ID-array of each feature $f_e \in F_q$, starting from $i = size(q)$ and upwards. The first supergraph of q obtained by the intersection is either q, in which case we return $\mathcal{D}_q$ directly, or q's closest δ-TCFG supergraph, in which case we recursively invoke FProcFGQ, or ProcFGQbyIGI (as in Procedure 6), to process q.

If q is not found by FProcFGQ, Line 7 of Procedure 9 returns an empty set to Algorithm 8, which then invokes *FProcNonFGQ* in Procedure 10 to process q. The algorithm is very similar to ProcNonFGQ in Algorithm 7; thus, we omit the detailed description here due to the space limit. However, FProcNonFGQ is far more efficient than ProcNonFGQ since we are now using features rather than simple edges. In addition, the ID-arrays of the features are also more efficient to access than are the *IDA*s of the edges.

Example 5.3. Referring to the IFI in Example 5.2, let $q = f_{11}$. We demonstrate how the use of the feature-index can improve the index probing efficiency.

According to the settings of Example 5.2, $\mathcal{T} = \{f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{14}\}$. If we use the IGI, we will use the ID-entries of the edges a and b to search for q. Thus, we will first check whether f_{10} is q and then whether f_{11} is q, since both f_{10} and f_{11} contain the edges a and b.

However, if we use the IFI, we intersect the ID-arrays of the features f_4 and f_7, which gives f_{11} directly. Thus, f_{10} is skipped since its ID is not in the ID-array of f_7, i.e., f_{10} is not a supergraph of f_7. In reality, when the index is much bigger, a significantly larger number of false results can be pruned using the IFI.

5.1.4 *The Advantages of Using the Feature-Index.* The feature-index improves the index probing of FG-index, or reduces T_{search}, in the following two ways.

First, the size-i ID-array of a maximal feature of q is much smaller than the total size of $IDA(e, i, j)$, $\forall j \geq count(e, q)$, of an edge e. This is apparent since an edge has far more supergraphs than a feature. Thus, the intersection using the IFI is more efficient than using the IGI. Moreover, using the IGI requires us to first collect the set of IDs from the "$IDA(e, i, j)$"s and then sort the IDs, while the ID-arrays in the IFI are sorted already.

Second, features possess much more structural information about q and its supergraphs than do the simple edges of q. There can be a large number of graphs that contain the edges of q but are not the supergraphs of q. In contrast, the number of graphs that contain the set of maximal features of q but are not the supergraphs of q is much smaller. Therefore, using the IFI significantly reduces the number of subgraph isomorphism tests required in the index probing process.

5.2 The FAQ-Index

The use of the feature-index significantly reduces the index probing cost; however, the dominating factor in the cost of processing non-FG-queries is the candidate verification cost. We propose an index built on a set of *Frequently Asked non-FG-Queries* (*FAQs*), called the *FAQ-index*, to improve the performance of processing non-FG-queries (i.e., both FAQs and non-FAQs). We remark that the FAQ-index is not simply caching for handling exact-matching queries, but is an index that is mainly designed for processing non-exact-matching queries.

5.2.1 *Definition of FAQ.* Before we define the notion of FAQs, we first define a *sliding window* model [Golab and Özsu 2003] in a *stream* of queries. We need the sliding window model because the set of all queries asked in the whole history is too large for building an index. Thus, the sliding window model allows us to control the size of the index to be built. In addition, the model also enables us to index the more recently asked queries, which are more likely to be asked again according to the principle of *temporal locality*. We define the sliding window as follows.

Definition 5.4. (*Time Unit and Sliding Window*) Let $\mathcal{S}$ be a *stream* of non-FG-queries. A *time unit*, t_i, is an excerpt of $\mathcal{S}$. A *sliding window* is a fixed number of successive time units in $\mathcal{S}$, where the window slides forward for every incoming time unit. Let t_τ be the *current time unit*. The *current window* is $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$, where w is the number of time units in W.

In the real case, both non-FG-queries and FG-queries come together in the stream. Since FG-queries can be answered without candidate verification, we only focus on non-FG-queries. In the rest of Section 5.2, all queries refer to non-FG-queries. We now define FAQ.

Definition 5.5. (*Frequently Asked Queries*) Let $freq(q, t)$ be the *frequency* of a query q within a time unit t, i.e., the number of times q is asked within t. Let $T^k = \langle t_{\tau-k+1}, \ldots, t_\tau \rangle$ be the k *most recent* time units in $W = \langle t_{\tau-w+1}, \ldots, t_\tau \rangle$, where $1 \leq k \leq w$. The *average frequency* of q in T^k is defined as follows:

$$avgFreq(q, T^k) = \frac{\sum_{i=\tau-k+1}^{\tau} freq(q, t_i)}{k}.$$

We define the *maximum average frequency (maxAvgFreq)* of q in W as follows:

$$maxAvgFreq(q, W) = MAX\{avgFreq(q, T^k) : 1 \leq k \leq w\}.$$

Let $Q(W)$ be the set of all queries in W. The set of *Frequently Asked Queries (FAQs)* in W is defined as the first N_{FAQ} queries in $Q(W)$ that have the highest values of maxAvgFreq, where N_{FAQ} $(0 \leq N_{FAQ} \leq |Q(W)|)$ is a pre-defined threshold.

We define the average frequency for a query in the window, since the query may have low frequency in some time units but high frequency in others. In addition, we favor the more recent time units since the older units are expiring. We compute the average frequency for a query over each k most recent time units, for $1 \leq k \leq w$, and take the maximum, which is then used to determine whether the query is an FAQ. We discuss how we set the threshold N_{FAQ} later when we construct the FAQ-index.

Definition 5.5 works well when the frequency at which queries of each size are asked is roughly equal. However, when the queries of a certain size are asked much less frequently than the queries of other sizes, the queries of that size may mostly be non-FAQs and discarded. Discarding these queries, especially the largest and smallest queries, is not desirable according to the following two lemmas.

LEMMA 5.6. *Given a query q and two FAQs q_1 and q_2, where $q_1 \subset q_2 \subset q$, then $\mathcal{D}_q \subseteq \mathcal{D}_{q_2} \subseteq \mathcal{D}_{q_1}$.*

LEMMA 5.7. *Given a query q and two FAQs q_3 and q_4, where $q_3 \supset q_4 \supset q$, then $\mathcal{D}_q \supseteq \mathcal{D}_{q_4} \supseteq \mathcal{D}_{q_3}$.*

Lemma 5.6 implies that we can estimate $\mathcal{C}_q$ by either $\mathcal{D}_{q_1}$ or $\mathcal{D}_{q_2}$. However, if both q_1 and q_2 are indexed, then we take $\mathcal{C}_q = \mathcal{D}_{q_2}$ since $\mathcal{D}_{q_2}$ is smaller and closer to $\mathcal{D}_q$.

Now, suppose that we also have q_3 indexed. Then, we can obtain $\mathcal{C}_q = (\mathcal{D}_{q_2} - \mathcal{D}_{q_3})$, since $\mathcal{D}_{q_3} \subseteq \mathcal{D}_q$. However, if we have a smaller supergraph of q, say q_4, we can obtain an even smaller $\mathcal{C}_q = (\mathcal{D}_{q_2} - \mathcal{D}_{q_4})$. Therefore, keeping FAQs of every size can better improve the query performance since a query can be of any size.

Since we determine the FAQs by ranking the maxAvgFreq values, we propose a normalization on the maxAvgFreq of the queries, as given by Definition 5.8.

Definition 5.8. (*Normalized maxAvgFreq*) Let *AVG-maxAvgFreq*(i) be the *average* maxAvgFreq of all queries in W that are of size i. The *normalized maxAvgFreq* of q in W is defined as follows:

$$maxAvgFreq^*(q, W) = maxAvgFreq(q, W) * \frac{MAX\{AVG\text{-}maxAvgFreq(i) : \forall i\}}{AVG\text{-}maxAvgFreq(size(q))}.$$

By substituting the normalized maxAvgFreq into Definition 5.5, queries of each size now have an equal probability to be selected as FAQs or discarded. We will further demonstrate the effect of taking this normalization by our experiments.

5.2.2 *Construction and Query Processing of the FAQ-Index.* The *FAQ-index* consists of the following two components: the *FAQ Hash Index* (*QHI*) and the *Inverted-FAQ-Index* (*IQI*).

The QHI is the same as the FHI except that the QHI is built on the set of FAQs. The IQI is an IGI (see Definition 3.10) defined on a set of FAQs. However, we do not include all the FAQs in the IQI, since the IQI needs to be updated incrementally for each window slide and hence updating the IQI for all FAQs can be expensive. Note that unlike the update on the graph database, the update on a stream of queries is much more frequent and the amount of changes to the FAQ-index is much greater than that to FG-index due to a database update. Therefore, we only build a single IQI for the FAQs. We limit the number of FAQs to be the number of graphs indexed by the largest IGI at any leaf node of the core FG-index, so that searching a query in the IQI can be as efficient as in an IGI. When the number of FAQs exceeds this limit, we discard the FAQs that have smaller maxAvgFreq values.

The FAQ-index is used to improve the efficiency of processing non-FG queries as follows. First, an incoming query in the stream is hashed to match with the FAQs in the QHI. If the query is found in the QHI, the answer set is retrieved directly without any candidate verification. If the query is not an FAQ, we use the IQI to find its subgraphs and supergraphs that are FAQs. Then, Lemmas 5.6 and 5.7 are applied to obtain the candidate set. The algorithm of query processing using the FAQ-index is shown in Algorithm 11. Since the IQI shares the same structure as the IGI, Lines 5-6 of Algorithm 11 are processed in a similar way as we process the IGI in Procedure 6 and Algorithm 7. We omit the details but point out the difference as follows. Line 5 of Algorithm 11 is processed in the same way as Algorithm 7 except that we skip Lines 15-16 and Line 18 of Algorithm 7, and from Line 14 we go to Line 17 instead of to Line 15. Line 6 of Algorithm 11 is processed as Procedure 6 except that we replace Lines 10-14 of Procedure 6 by computing the union of $\mathcal{D}_f$ for each supergraph f of q obtained by the intersection in Line 8 of Procedure 6.

Algorithm 11 QProcNonFGQ

Input: The FAQ-index, and a query q.
Output: $\mathcal{D}_q$.

1. Search q in the QHI;
2. **if**(q is in the QHI)
3. Return $\mathcal{D}_q$;
4. **else**
5. Use the IQI to generate $\mathcal{C}_q$ from q's subgraphs that are FAQs;
6. Use the IQI to obtain $\mathcal{D}'_q$, which is the union of the answer sets of q's supergraphs that are FAQs;
7. Return $\mathcal{D}_q \leftarrow \mathcal{D}'_q \cup \{g : g \in (\mathcal{C}_q - \mathcal{D}'_q), g \supseteq q\}$;

5.2.3 *Parameter Settings and Maintenance of the FAQ-Index.* Before discussing the maintenance of the FAQ-index, we first need to determine the number of time units w in the window and the *length* of each time unit.

Let M be the size of the available memory. We use $(wM/(w+1))$ memory for the sliding window and the remaining $M/(w+1)$ memory as a buffer to keep the incoming queries from the stream. The length of a time unit is defined as the length of time that is needed to fill the $M/(w+1)$ memory with the incoming queries. The threshold N_{FAQ} in Definition 5.5 is set as the total number of FAQs that the $(wM/(w+1))$ memory is able to hold.

The following example illustrates how we set the parameters.

Example 5.9. Suppose that we have $M = 110$ MB of available memory and the number of time units in the window is 10, i.e., $w = 10$. Then, we have 100 MB of memory for the sliding window and 10 MB for the buffer to keep the incoming queries. The length of a time unit is the time needed to fill the 10 MB buffer with the incoming queries. Assume that the 100 MB of memory is able to hold 1000 FAQs. Then, $N_{FAQ} = 1000$.

Finally, we discuss the maintenance of the FAQ-index. For each distinct incoming query in the stream, we keep the query in the QHI, where the memory to hold the query is assigned from the buffer of $M/(w+1)$ memory. When the $M/(w+1)$ memory is used up, we re-compute the FAQs as defined in Definition 5.5. Those queries that are not FAQs are deleted from the QHI until $M/(w+1)$ memory is released for the buffer to hold the incoming queries. Then, the old IQI is deleted and a new IQI is constructed from the set of FAQs in the current window.

5.3 Query Processing using FG*-Index

FG-index* consists of the following three components: FG-index, the feature-index and the FAQ-index. We have discussed how to use each of the components to process a query. Now, we combine the three components to process a query, as shown in Algorithm 12.

The algorithm *ProcessQuery* processes q according to the size of q. First, if the size of q is smaller than that of the smallest feature, we use FG-index to process q. However, if q is not an FG, then we use the FAQ-index instead of FG-index to process q, since FG-index generates a large candidate set for non-FG-queries.

Algorithm 12 ProcessQuery

Input: FG*-index, and a query q.
Output: $\mathcal{D}_q$.

```
1. if(size(q) < l)
2.     Invoke ProcFGQ to use FG-index to process q;
3.     if(q is not found by ProcFGQ)
4.         Invoke QProcNonFGQ to use the FAQ-index to process q;
5. else if(l ≤ size(q) ≤ u)
6.     if(q is a feature in the FHI)
7.         Return D_q;
8.     else   /* q is not an FG */
9.         Invoke QProcNonFGQ to use the FAQ-index to process q;
10. else if(size(q) > u)
11.    if(q is an FAQ in the QHI)
12.        Return D_q;
13.    else
14.        Invoke FProcFGQ to use the feature-index to process q;
15.        if(q is not found by FProcFGQ)
16.            Invoke QProcNonFGQ to use the FAQ-index to process q,
                 and invoke FProcNonFGQ to use the feature-index to refine C_q;
```

If the size of q is within the size range of features, we first check if q is a feature using the FHI. If q is not a feature, then it must be a non-FG-query. Thus, the FAQ-index is used to process q.

If the size of q is greater than that of the largest feature, then we first check whether q is an FAQ using the QHI, since large-sized queries are more likely to be non-FG-queries. If q is not an FAQ, then we process q using the feature-index. If q is not found by the feature-index, then q must be a non-FG-query and we use the IQI to answer q. We also invoke FProcNonFGQ to refine $\mathcal{C}_q$ using the feature-index.

5.4 Query Performance Improvement of FG*-Index

In Section 4, we give the query response time of FG-index in Equation (3) for processing FG-queries and in Equation (1) for non-FG-queries. We now give the response time of FG*-index by comparing with that of FG-index.

For processing FG-queries using FG*-index, if q is a feature, then the response time is given as follows:

$$T_{response} = (|\mathcal{D}_q| \times T_{I/O}) \ . \tag{4}$$

We do not include the index probing time in Equation (4) because the time taken to find q in the FHI by hashing q is negligible.

If q is not a feature, then the response time of FG*-index is given as follows:

$$T_{response} = (T^*_{search} + |\mathcal{D}_q| \times T_{I/O}) \ . \tag{5}$$

T^*_{search} in Equation (5) is significantly smaller than T_{search} in Equation (3), because using the feature-index significantly reduces the number of subgraph isomorphism tests in the index probing process.

For processing non-FG-queries, if q is an FAQ, then the response time of FG*-index is given by Equation (4), because q is answered using the QHI and the QHI has the same structure as the FHI.

If q is not an FAQ, then the response time of FG*-index is given as follows:

$$T_{response} = (T^*_{search} + |\mathcal{D}_q| \times T_{I/O} + |\mathcal{C}^*_q| \times T_{verify}) \ . \quad (6)$$

If we have indexed the supergraphs of q in the IQI, then we can obtain a subset $\mathcal{D}'_q \subset \mathcal{D}_q$. Thus, $|\mathcal{C}^*_q| = |(\mathcal{C}_q - \mathcal{D}'_q)|$, which is usually very small. Otherwise, the $\mathcal{C}^*_q$ obtained using the IQI and the IFI is also much smaller than the $\mathcal{C}_q$ obtained by FG-index as given in Equation (1).

Overall, the use of the feature-index and the FAQ-index in FG*-index improves the performance of FG-index for processing both FG-queries and non-FG-queries, which we verify by extensive experiments in Section 7.

6. UPDATE OF FG*-INDEX

In this section, we propose an efficient algorithm for updating FG*-index when the graph database is updated.

[Cheng et al. 2007] briefly discusses how to update FG-index incrementally for each graph added to or deleted from the database, which can be extended to update FG*-index. However, this update algorithm is not efficient for the following reasons. Let g be the graph to be updated. First, this algorithm requires the enumeration of every subgraph $g' \subseteq g$, which can be costly especially when g is large, even though some pruning can be performed. Second, the update requires the ID of g to be inserted into or removed from $\mathcal{D}_f$ for every subgraph f of g in the index, which involves many disk I/Os since $\mathcal{D}_f$ is stored on the disk. Third, processing update on one graph at a time is inefficient and may severely slow down query processing, especially when updates are frequent and queries are asked frequently.

We propose a different strategy for updating the index. Instead of updating the index for each graph each time, we devise an algorithm that updates the index for a batch of graphs each time. The update is divided into two parts: *handling deleted graphs* and *handling new graphs*.

We first discuss the handling of deleted graphs. We do not update FG*-index for each deleted graph but keep all currently deleted graphs in a set, $\mathcal{D}_{del}$. Then, for each query q, after we obtain $\mathcal{D}_q$ using FG*-index, we compute $(\mathcal{D}_q - \mathcal{D}_{del})$ as the final answer set. The set subtraction is efficient since it is operated on graph IDs; but the question is: *when do we update the index for the deleted graphs?* We use a simple mechanism here: when the set subtraction time is longer than the query processing time using FG*-index, we rebuild the index from scratch on $(\mathcal{D} - \mathcal{D}_{del})$.

Next, we discuss the handling of new graphs that are added to the database. Again, we do not update FG*-index for each new graph but keep all new graphs in a set, $\mathcal{D}_{new}$. For each query q, we first obtain $\mathcal{D}_q$ using FG*-index. Then, we perform candidate verification for each graph in $\mathcal{D}_{new}$ against q. Finally, $(\mathcal{D}_q \cup \{g \in \mathcal{D}_{new} : g \supseteq q\})$ is returned as the answer set.

The question again is: *when do we update the index for the newly added graphs?* We cannot simply rebuild FG*-index when verifying the graphs in $\mathcal{D}_{new}$ is more costly than query processing using FG*-index, because candidate verification is far

more costly than set subtraction. Obviously, we do not want to rebuild the entire FG*-index too frequently because it is costly. We devise a solution as follows.

When the time for candidate verification on $\mathcal{D}_{new}$ is longer than the query processing time using FG*-index, we build a new FG*-index on $\mathcal{D}_{new}$ with the same setting of the parameters, except that we disable the FAQ-index to avoid duplicate processing of the same query. We call this new FG*-index the *auxiliary FG*-index*. If the auxiliary FG*-index already exists, we delete it and build a new one on $\mathcal{D}_{new}$ and the set of graphs on which the old auxiliary FG*-index was built. We then empty $\mathcal{D}_{new}$ to keep new added graphs. Now, we process each query with both FG*-index and the auxiliary FG*-index, as well as performing candidate verification on $\mathcal{D}_{new}$ if more new graphs are just added. The answer set for a query q is $(\mathcal{D}_q \cup \mathcal{D}'_q \cup \{g \in \mathcal{D}_{new} : g \supseteq q\})$, where $\mathcal{D}_q$ and $\mathcal{D}'_q$ are the answer sets returned by FG*-index and the auxiliary FG*-index, respectively.

We adopt the following strategies for handling newly added graphs:

(1) When the time for candidate verification on $\mathcal{D}_{new}$ is longer than the query processing time using FG*-index, we rebuild the auxiliary FG*-index.
(2) When the accumulated time of constructing the auxiliary FG*-indexes is longer than the construction time of the current FG*-index, we rebuild the FG*-index.

We set the first condition based on the reason that rebuilding the auxiliary FG*-index on the newly added graphs is much more efficient than rebuilding the FG*-index from scratch on the entire database. This is simply because the number of newly added graphs is far smaller than the number of graphs in the original graph database $\mathcal{D}$; otherwise, the database should have been updated as triggered by the second condition.

The second condition is met when the overall overheads on building all the auxiliary FG*-indexes become greater than rebuilding the index from scratch. Note that the total update cost should be counted into the query processing cost, since if update can be done off-line, we can simply rebuild the index for every database update. Therefore, we need to control the overall cost spent on the update. Note that rebuilding FG*-index from the new database is of approximately the same cost as building the FG*-index from the old database, because the number of updated graphs is relatively small compared with the size of the database. Thus, the second condition triggers the index to be rebuilt when the overall time spent on building the auxiliary FG*-indexes becomes greater than rebuilding the index.

Note that the above two types of database update are not processed separately. Rather, at any time we may have both deleted graphs and new graphs. Thus, the answer set of a query q is $(\mathcal{D}_q \cup \mathcal{D}'_q \cup \{g \in \mathcal{D}_{new} : g \supseteq q\} - \mathcal{D}_{del})$.

The efficiency of our batch-update algorithm depends on two factors: the efficiency of mining the set of FGs and that of building the index structures from the FGs. The latter is efficient since our index does not require a set of FGs of low support, for which the index construction cost is very efficient as verified by our experiments. This is true even when the database becomes large, because the number of FGs remains roughly the same for the same σ, which is evidenced from frequent pattern mining from data streams [Manku and Motwani 2002; Yu et al. 2004; Cheng et al. 2008b; 2008c]. The former, i.e., mining FGs, is also efficient when the database size is small to moderate, because we do not require a set of

Table II. Characteristics of Datasets and Query Sets

	Number of graphs	Range of graph size	Average graph size	Range of density	Average density	Num. of distinct edges	Num. of distinct nodes
$\mathcal{D}_{AIDS}$	10K	1−217	27.40	0.009−1.0	0.10	221	51
$\mathcal{D}_{cancer}$	10K−100K	1−252	19.95	0.008−1.0	0.14	303	63
$\mathcal{D}_{den0.1}$	10K	31−68	50.49	0.06−0.15	0.10	220	20
$\mathcal{D}_{den0.2}$	10K	31−68	50.49	0.17−0.26	0.20	220	20
$\mathcal{D}_{den0.3}$	10K	31−68	50.49	0.27−0.38	0.32	220	20
$\mathcal{D}_{den0.4}$	10K	31−68	50.49	0.37−0.50	0.43	220	20
$\mathcal{D}_{den0.5}$	10K	31−68	50.49	0.48−0.62	0.54	220	20
$Q_{AIDS\text{-}1}$	100K	1−24	14.16	0.08−1.0	0.15	221	51
$Q_{AIDS\text{-}2}$	100K	1−24	14.77	0.08−1.0	0.14	221	51
Q_{FG}	100K	1−21	13.47	0.09−1.0	0.15	303	63
$Q_{non\text{-}FG}$	100K	2−23	15.67	0.08−1.0	0.13	303	63
Q_{mixed}	100K	1−23	15.12	0.08−1.0	0.14	303	63
$Q_{den0.x}$	100K	1−8	3.80	0.29−1.0	0.59	220	20

FGs of low support. However, when the database becomes very large, then the cost of mining FGs from scratch can be costly. In this case, we need incrementally update the set of FGs.

The problem of incrementally updating the set of FGs is very similar to the problem of incrementally maintaining the set of FGs in a data stream (we can use a sliding window with variable size when there is deletion). We can apply the concepts for incrementally maintaining frequent itemsets in a data stream to design an efficient algorithm for updating the FGs, which is our on-going work. At the current stage, however, when we do not have an algorithm for incrementally maintaining the set of FGs, our index is more suitable for static environments, or for the dynamic environments in which the database size is small to moderate.

7. PERFORMANCE EVALUATION

We evaluate the query performance using FG*-index by comparing with FG-index [Cheng et al. 2007], as well as two other state-of-the-art graph indexes, *gIndex* [Yan et al. 2005a] and *C-tree* [He and Singh 2006]. We run all experiments on an AMD Opteron 248 with 2GB RAM, running Linux 64-bit.

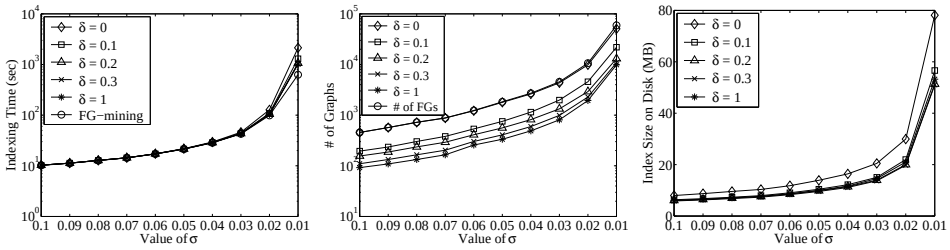
7.1 Datasets and Query Sets

We use the following datasets and query sets as shown in Table II, where datasets are represented as $\mathcal{D}_x$ and query sets as Q_y.

Among the datasets, $\mathcal{D}_{AIDS}$ and $\mathcal{D}_{cancer}$ are real datasets. $\mathcal{D}_{AIDS}$ is the *AIDS antiviral screen dataset*, which is provided by [Yan et al. 2005a]. Since $\mathcal{D}_{AIDS}$ has only 10K graphs, we use six $\mathcal{D}_{cancer}$ datasets of size from 10K to 100K to perform a scalability test. We obtain the $\mathcal{D}_{cancer}$ datasets from the National Cancer Institute database[2], where more detailed characteristics of the data can be found.

We notice that the density of most graphs in the real datasets is relatively low;

[2]http://cactus.nci.nih.gov/ncidb2/download.html.



(a) Index Construction Time (b) Number of δ-TCFGs/FGs (c) Size of Index on Disk

Fig. 8. The Effects of σ and δ on the Index Construction

thus, we use the synthetic graph data generator[3] [Cheng et al. 2007] to generate five datasets $\mathcal{D}_{den0.x}$, by varying the average graph density from 0.1 to 0.5.

The queries in $Q_{AIDS\text{-}1}$ and $Q_{AIDS\text{-}2}$ are randomly selected from 400K subgraphs of the graphs in the dataset $\mathcal{D}_{AIDS}$. The queries in Q_{FG}, $Q_{non\text{-}FG}$ and Q_{mixed} are randomly selected from 430K subgraphs of the graphs in $\mathcal{D}_{cancer}$. $Q_{den0.x}$ represents five sets of query sets, in which the queries are randomly selected from up to 4.8M subgraphs of the graphs in the corresponding $\mathcal{D}_{den0.x}$. The query sets are also further classified into FG-, non-FG-, and mixed-type- queries; we give this detail until we use the respective query sets.

7.2 Sensitivity Analysis on the Parameters of FG*-Index

We first test the effects of the parameters, σ and δ, as well as the feature-index and the FAQ-index, on the performance of FG*-index. We also provide guidelines on how to set the parameters in FG*-index. We use the dataset $\mathcal{D}_{AIDS}$.

7.2.1 *The Effects of σ and δ.* We test σ from 0.1 to 0.01 and δ from 0 to 1. We use *gSpan* [Yan and Han 2002] to mine the set of FGs for each σ. We construct FG*-index, where the feature set used to construct the feature-index is $\mathcal{F}_2^7$. We disable the FAQ-index in FG*-index so that the effect of σ can be clearly revealed. We will test the effect of σ using the FAQ-index in Section 7.3.1. We also note that δ is automatically adjusted in the FAQ-index.

Figure 8 shows the index construction time and the number of δ-TCFGs. The construction time includes the time taken by gSpan to mine the FGs; but we also report the FG-mining time in Figure 8(a), which dominates the total index construction time in most cases. We omit δ between 0.3 and 1 because the number of 0.3-TCFGs is very close to that of 1-TCFGs, as shown in Figure 8(b). Figure 8(b) also shows that the number of δ-TCFGs at a value of δ as small as 0.1 is already significantly smaller than the number of FGs, which is the top line in Figure 8(b). Figure 8(a) shows that it is very efficient to construct the index except for σ smaller than 0.03, after which the indexing time increases almost exponentially. The increase in the indexing time is mainly due to the rapid increase in the number of FGs when σ becomes smaller than 0.03, as shown in Figure 8(b).

Figure 8(c) reports the size of the FG*-indexes on the disk, for each σ and each δ. The raw dataset $\mathcal{D}_{AIDS}$ requires 4.8 MB of space on the disk. The indexes

[3]http://www.cse.ust.hk/graphgen/.

Table III. Peak Memory Consumption (MB) of the Index Construction

σ	0.1	0.09	0.08	0.07	0.06	0.05	0.04	0.03	0.02	0.01
$0 \leq \delta \leq 1$	8	8	8	9	9	10	11	16	29	108

at $\sigma \geq 0.03$ are about 2-3 times larger than the raw dataset, but the indexes at $\sigma < 0.03$ are much larger. We emphasize that the index size depends mainly on the number of FGs as shown in Figure 8(b), rather than on the size of the raw dataset. For different values of δ, the index size is the largest when $\delta = 0$, because most of the FGs indexed are 0-TCFGs and hence the duplicate graphs in the answer sets of most FGs are not removed since they are δ-TCFGs (see details in Section 3.3.3).

Finally, Table III lists the *peak* memory consumption of constructing the indexes for each σ. The increase in the memory consumption is due to the increase in the number of FGs when σ becomes smaller. However, for the different values of δ, the memory consumption remains unchanged, because all FGs are loaded into the main memory for constructing FG*-index.

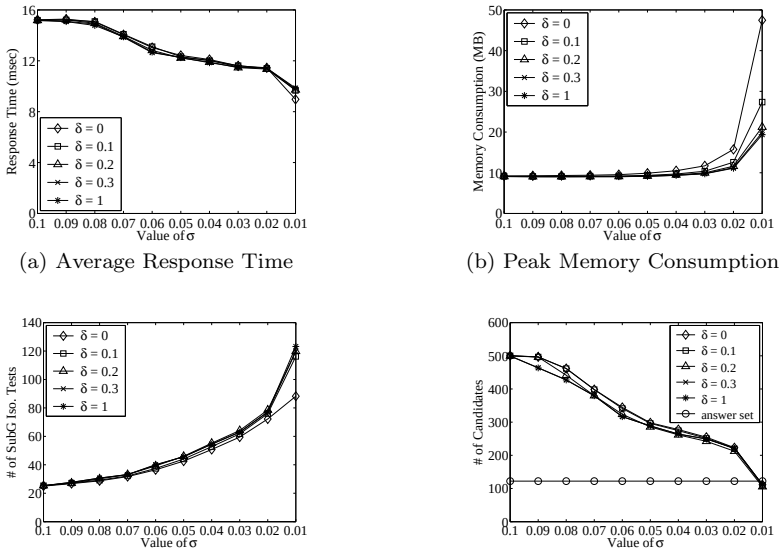
For query processing, we aim to test the performance of processing both FG-queries and non-FG-queries. We use $Q_{AIDS\text{-}1}$ because the answer set of the queries in $Q_{AIDS\text{-}1}$ has a size ranging from 50 to 8222; thus, a query in $Q_{AIDS\text{-}1}$ can be either an FG-query or a non-FG-query with respect to σ ($0.01 \leq \sigma \leq 0.1$).

Figure 9(a) reports the average response time of processing a query for $Q_{AIDS\text{-}1}$. In contrast to the index construction, the result shows that a smaller σ gives a shorter response time in query processing. The decrease in the response time can be explained by the decrease in the size of the candidate set reported in Figure 9(d). Although Figure 9(c) reveals an increase in the number of subgraph isomorphism tests performed in the index probing process, the combined cost of index probing and candidate verification still decreases when σ is smaller.

Figure 9(a) shows that the variation in δ does not have much effect on the query response time. However, Figure 9(b) shows that the memory consumption is significantly increased when $\delta = 0$ and σ is small. The increase in the memory consumption can be explained by Figure 8(b), which shows that the set of 0-TCFGs is much larger than that of the other δ-TCFGs. As a result, the root IGI that is built on the set of 0-TCFGs is also larger.

Overall, the query performance of FG*-index is only slightly degraded when σ becomes larger and still very impressive even for the largest σ. Considering the index construction cost, a moderate σ seems to be the best choice. For example, when $\sigma = 0.05$, the index construction cost is only slightly higher than that of $\sigma = 0.1$, while the query response time is only slightly longer than that of $\sigma = 0.01$ but the memory consumption is much lower. Therefore, we can build FG*-index at a moderate σ in most cases and at a small σ only when query response time is critical. On the other hand, the value of δ does not have a significant effect on index construction and query processing, except that the memory consumption increases considerably when $\delta = 0$. We further discuss how to find the optimal values of the two parameters in Section 7.2.4.

7.2.2 *The Effect of the Feature-Index.* We now show how the use of the feature-index improves the index probing efficiency, by comparing with FG-index. We



(a) Average Response Time (b) Peak Memory Consumption

(c) Average Index Probing Cost (d) Average Candidate Set Size

Fig. 9. The Effects of σ and δ on Query Performance

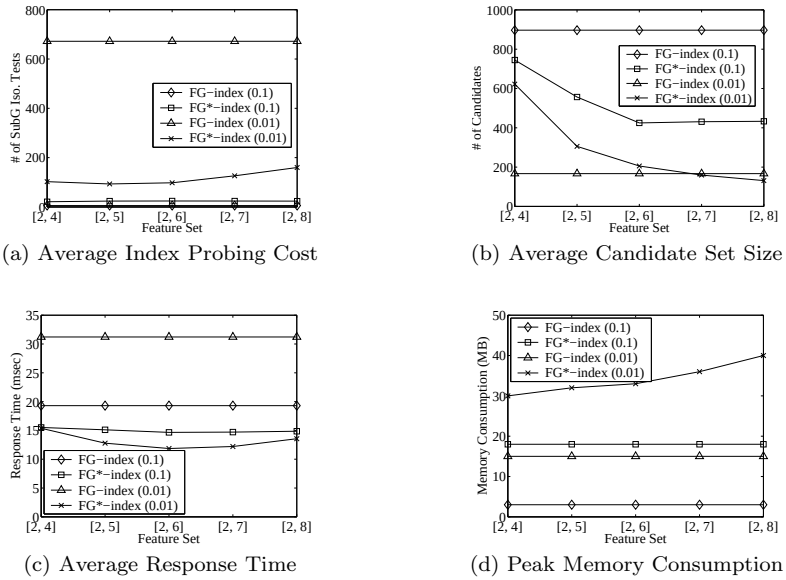
Table IV. The Effect of the Feature-Index on the Index Construction

	FG-index (0.1)	FG*-index (0.1)	FG-index (0.01)	FG*-index (0.01)
Indexing time (sec)	10.20	10.21 − 10.27	1108	1111 − 1430
Memory consumption (MB)	9	10 − 11	103	104 − 110
Index size on disk (MB)	2	5 − 7	40	44 − 64

disable the FAQ-index in FG*-index, so that the improvement comes only from the feature-index. We set $\delta = 0.1$ and report the two extreme values of σ tested in Section 7.2.1, i.e., $\sigma = 0.1$ and $\sigma = 0.01$. For each σ, we construct five feature-indexes from the feature sets $\mathcal{F}_2^4$, $\mathcal{F}_2^5$, $\mathcal{F}_2^6$, $\mathcal{F}_2^7$, and $\mathcal{F}_2^8$, which are represented in Figure 10 as $[2,4]$, $[2,5]$, $[2,6]$, $[2,7]$, and $[2,8]$, respectively.

Table IV reports the index construction time (including FG-mining time, which is 10.19 and 628.1 sec for $\sigma = 0.1$ and 0.01), the peak memory consumption, and the size of the indexes on the disk. To save space, we report the results of FG*-index as a range, since the range is small and the increase is linear when u increases from 4 to 8, where u is the upper bound of the size of a feature. The results show that constructing FG*-index is almost as efficient as constructing FG-index. The indexing time of FG*-index is almost not changed for $\sigma = 0.1$, because we have only 455 FGs and hence the size of the feature sets is also small. The memory consumption is increased only slightly. However, there is a greater increase in both the indexing time and memory consumption for $\sigma = 0.01$, because about 60K FGs are used to build FG*-index. The size of the index on the disk increases by at most 20 MB due to the feature-index.

For evaluation of the query performance, we use $Q_{AIDS\text{-}1}$ for the same reason as



(a) Average Index Probing Cost

(b) Average Candidate Set Size

(c) Average Response Time

(d) Peak Memory Consumption

Fig. 10. Query Performance of the Feature-Index

we give in Section 7.2.1. We record the following four metrics: the average number of subgraph isomorphism tests performed in the index probing process per query, the average size of the candidate set per query, the average response time per query, and the peak memory consumption. The results are reported in Figures 10(a-d).

Figure 10(a) shows that, for $\sigma = 0.01$, the number of subgraph isomorphism tests performed in the index probing process using FG-index is very large, while that using FG*-index is significantly reduced due to the use of the feature-index.

For $\sigma = 0.1$, the number of subgraph isomorphism tests using FG-index is the smallest since only 455 FGs are indexed. However, Figure 10(b) reveals that, for $\sigma = 0.1$, the size of the candidate set using FG-index is very large, because σ is large and hence most queries are non-FG-queries. Thus, index probing using FG-index at $\sigma = 0.1$ is fast but the candidate verification is very costly.

From Figure 10(b), we see that using the feature-index also reduces the candidate set size. However, for $\sigma = 0.01$, the candidate set size of FG*-index is smaller than that of FG-index only when $\mathcal{F}_2^7$ and $\mathcal{F}_2^8$ are used to build the feature-index. This is because, in our implementation, we simply use the feature subgraphs of a query, rather than the maximal FG subgraphs, to obtain the candidate set; while we mainly rely on the FAQ-index to obtain a small candidate set.

Figure 10(c) verifies that using the feature-index indeed speeds up the query processing significantly. We also find that using FG-index at $\sigma = 0.01$ is slower than that at $\sigma = 0.1$, which can be explained by the high index probing cost when $\sigma = 0.01$. However, when the feature-index is used, we can use a smaller σ to build FG*-index to reduce the index probing cost and at the same time obtain a small candidate set. The figure also shows that the feature-indexes built on $\mathcal{F}_2^6$ and $\mathcal{F}_2^7$ achieve the best response time. Thus, it shows that when too few features are used, the index probing performance is not improved; but when too many features are

Table V. The Effect of Normalization of maxAvgFreq

	Index probing cost	Candidate set size	Total response time	Total update time	Memory consumption
Normalized	155	181	1379 sec	9 sec	36 MB
Non-normalized	156	198	1606 sec	22 sec	44 MB

used, finding the features themselves becomes too costly.

Figure 10(d) shows that FG*-index consumes about 15 to 25 MB more memory than FG-index, as u increases from 4 to 8. The increase in the memory consumption is due to the use of the feature-index. However, we emphasize that the increase in memory consumption is not relative, but solely depends on the number of features, which should not be too large as too many features will have a counter effect on the index probing performance as verified by both Figures 10 (a) and (c).

7.2.3 *The Effect of the FAQ-Index.* We now test how the use of the FAQ-index improves the performance of processing non-FG-queries. We set $\sigma = 0.01$ and $\delta = 0.1$ for both FG*-index and FG-index, and the feature-index of FG*-index is constructed on $\mathcal{F}_2^7$.

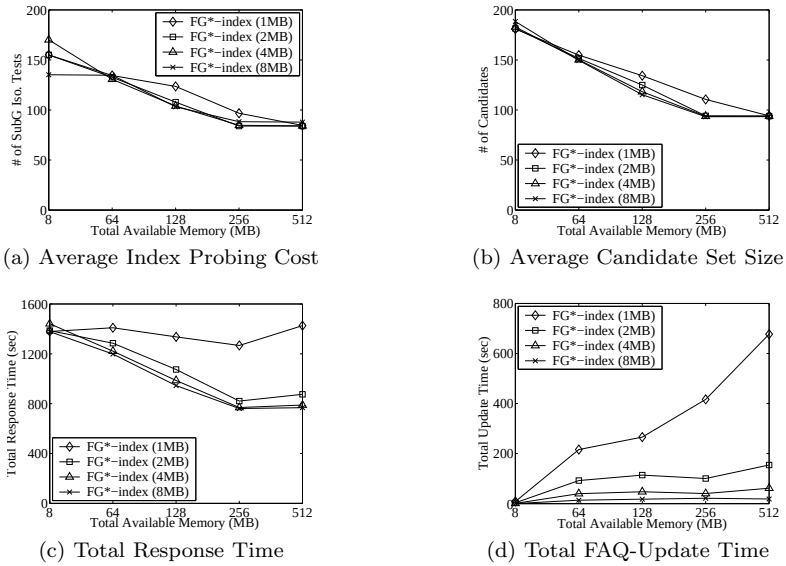
Since the FAQ-index is constructed on the set of non-FG-queries, we use the query set $Q_{AIDS\text{-}2}$, which consists of only queries with an answer set size at most $99 < (\sigma \times 10K) = 100$. The query set $Q_{AIDS\text{-}2}$ is modeled as a stream prepared as follows. The stream is made up of 100 blocks and each block consists of 1K queries. Each block has some queries that are repeated from the previous block. The number of repetitions follows a Poisson distribution with 100 as the mean.

We test the effects of the normalization of maxAvgFreq defined in Definition 5.8, the total available memory for the FAQ-index, and the length of a time unit in a sliding window in terms of memory size (or simply the *unit length*). If we set the unit length to be 1 MB, the length of the time unit is the time needed to fill 1 MB of memory with incoming queries.

We first examine the effect of the normalization. The total available memory for the FAQ-index is set to 8 MB and the unit length to 1 MB. We record the following five metrics of query performance: the average number of subgraph isomorphism tests performed in the index probing process per query, the average size of the candidate set per query, the total elapsed time for processing all 100K queries (including dynamically updating the FAQ-index), the total time for dynamically updating the FAQ-index, and the peak memory consumption.

Table V shows that the normalization indeed improves the query performance. In particular, the candidate set size is reduced, which verifies Lemmas 5.6 and 5.7 and, as a result, the query response time is also reduced by about 16.5%.

We now test the available memory for the FAQ-index from 8 MB to 512 MB. At the same time, we test four unit lengths, 1 MB, 2 MB, 4 MB and 8 MB, which are represented as "FG*-index (i MB)" in this experiment, where $i = 1, 2, 4$ and 8. Equivalently, we also test the effect of the number of units in a sliding window (i.e., w), since the number of units is equal to ("*available memory*"/"*unit length*"). We report the results in Figures 11(a-d). Since the memory consumption is consistently 35-40 MB greater than the available memory for the FAQ-index, we omit the details.



(a) Average Index Probing Cost

(b) Average Candidate Set Size

(c) Total Response Time

(d) Total FAQ-Update Time

Fig. 11. Performance of the FAQ-Index

Figures 11 (a) and (b) show that both the index probing cost and the candidate set size are significantly reduced when more memory is available for the FAQ-index, except when the available memory increases from 256 MB to 512 MB. The results explain the speed-up of query processing when the available memory increases from 8 MB to 256 MB, as shown in Figure 11(c).

We observe from Figures 11(a-c) that the results remain almost unchanged when the available memory increases from 256 MB to 512 MB. This is because all the queries are kept in the QHI when the available memory is slightly larger than 256 MB. Although more available memory allows us to keep more FAQs and answer queries that are FAQs directly using the QHI, we remark that not all queries are FAQs and the performance improvement shown in Figure 11 does not come only from the use of the QHI. When a query is asked the first time, it is not an FAQ and cannot be answered using the QHI. For processing such queries, the performance improvement comes from the use of the IQI. In the query set tested, we allow 10% of the queries to be repeated in each block of the query stream as to test the effect of the QHI, while the rest of the queries are processed using the IQI.

Although the query response time is not improved further when the available memory increases from 256 MB to 512 MB, the time taken to update the FAQ-index increases. This is particularly obvious for FG*-index (1 MB), since the update becomes more frequent when the unit length decreases. Thus, the result also explains why FG*-index (1 MB) is the slowest in Figure 11(c).

Table VI gives the performance comparison of FG-index, FG*-index without the FAQ-index, and the full FG*-index. We report the results of FG*-index (4 MB) with 256 MB of available memory for the FAQ-index.

The results show that using the FAQ-index even further improves the index probing efficiency. The size of the candidate set is also significantly reduced. On average,

Table VI. Performance Improvement made by the FAQ-Index

	Index probing cost	Candidate set size	Response time per query	Memory consumption
FG-index	726	196	34.83 msec	15 MB
FG*-index (no FAQ)	148	210	15.94 msec	27 MB
FG*-index	85	94	7.69 msec	291 MB

query processing using FG*-index is five times faster than using FG-index, and two times faster than using FG*-index without the FAQ-index. Note that the larger memory consumption of FG*-index does not imply that FG*-index is not scalable, because the memory consumption is *not relative* but depends on the *absolute* available memory assigned for the FAQ-index. In addition, it is acceptable to use (291-27)=264 MB of memory to double the speed of query processing, because 264 MB of memory is commonly affordable today. Moreover, we will show in Section 7.3 that other indexes consume significantly more memory than FG*-index.

7.2.4 *Guidelines on Setting the Parameters of FG*-Index.* We have tested the effects of the following parameters on the performance of FG*-index: (1) σ; (2) δ; (3) u; (4) the available memory; and (5) the unit length. Based on the results obtained, we provide the following guidelines on setting the parameters.

First, the results show that the smaller the value of σ, the shorter the query response time (Figure 9(a)), but the longer the index construction time (Figure 8(a)). However, it is not entirely true that when the value of σ is smaller, the query response time will always be shorter. The advantage of using a smaller σ is to reduce the size of the candidate set as shown in Figure 9(d); however, Figure 9(c) shows that the index probing cost increases when σ decreases. Therefore, there is a point at which the response time will become longer, when the increase in the index probing cost is greater than the reduction in the candidate verification cost. Thus, the optimal value of σ can be found if query response time is critical. However, if we take into account both the index construction cost and the query performance, Figures 8 and 9 show that a moderate value of σ is actually a better choice. We further demonstrate that FG*-index at a moderate σ achieves orders of magnitude better query performance than other indexes in Section 7.3.

Second, Figures 8 and 9 show that the values of δ do not significantly affect the index construction cost and the query response time, but the memory consumption is doubled when δ decreases from 0.1 to 0. Therefore, the choice of δ is not very critical as far as δ is not too close to 0 and a recommendation based on the experimental results is to set $\delta \geq 0.1$.

Third, Figure 10(a) shows that the index probing cost first decreases and then increases when u increases, implying that there is an optimal u in reducing the index probing cost. However, Figure 10(b) shows that the candidate verification cost decreases when u increases. Thus, we need to consider both the index probing cost and the candidate verification cost in setting u. We can set u slightly larger than 2 and increase u until the index probing cost starts to increase. Then, we start to consider the candidate verification cost as well when we further increase u. Since the number of FGs increases quickly when u becomes larger, usually there are

only a few tests needed. More importantly, the results in Figure 10 show that the FG*-indexes built on the different feature-sets are all very efficient, which means that these values of u are all sub-optimal.

Lastly, the available memory for the FAQ-index depends on the memory available in the system in which the queries are evaluated. However, we remark that an amount of memory as small as 8 MB can already improve the response time from 15.94 msec to 13.78 msec (about 16%), and 256 MB of memory can reduce the response time to only 7.69 msec (two times). The unit length affects the update cost of the FAQ-index and hence it should not be too small. However, the optimal length of a unit depends on the query workload. From our experimental results as shown in Figures 11(c) and 11(d), setting the length of a unit to be at least 4 MB only incurs a small update cost on the total response time.

In conclusion, our results show that the query performance of FG*-index is very impressive for a wide range of parameters tested and the index construction is also very efficient except for $\sigma \leq 0.02$. More importantly, we show in the following experiments that, compared with other state-of-the-art indexes, FG*-index is significantly more robust and scalable.

7.3 Scalability Tests

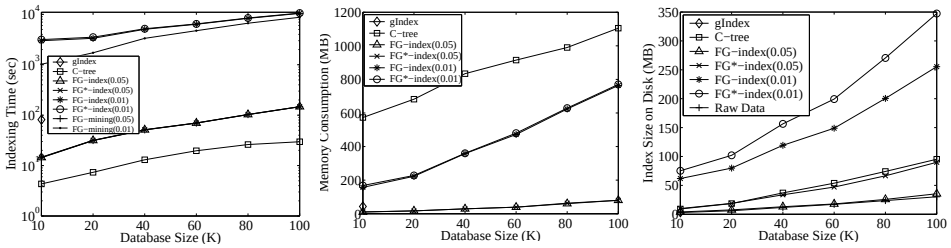
We now compare FG*-index with gIndex and C-tree, as well as FG-index, through two scalability experiments by varying the database size and the graph density.

7.3.1 *Scalability Test on Database Size.* We first assess the performance of the indexes at different database sizes. We use the dataset $\mathcal{D}_{cancer}$ by varying the size from 10K graphs to 100K graphs.

For both FG*-index and FG-index, we set $\delta = 0.1$ and test two values of σ, $\sigma = 0.05$ and $\sigma = 0.01$. For FG*-index, we use $\mathcal{F}_2^7$ to construct the feature-index, and set the available memory for the FAQ-index to be 256MB and the unit length to be 4MB. The settings of gIndex and C-tree are the default values suggested in their papers.

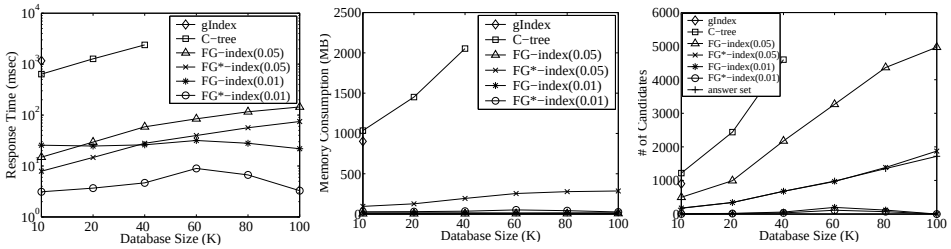
Figure 12 reports the experimental results of constructing each of the indexes. Figure 12(a) shows that the indexing time of FG*-index and FG-index at $\sigma = 0.01$ is much longer (due to the large number of FGs) than that of the others. However, constructing both FG*-index and FG-index at $\sigma = 0.05$ is very quick. Constructing C-tree is the quickest but the construction uses much more memory as shown in Figure 12(b), while gIndex cannot be built for databases that have more than 10K graphs. Thus, taking both the indexing time and the memory consumption into account, FG*-index and FG-index at $\sigma = 0.05$ are the most efficient to construct and the figures also show that their construction costs increase only slightly as the database size increases.

Figure 12(c) shows that the size of the indexes on the disk increases linearly when the database size increases. The result also shows that FG*-index and FG-index are the largest when $\sigma = 0.01$ but the smallest when $\sigma = 0.05$. The result thus shows that, with the increase in the database size, the index size of FG*-index and FG-index also depends mainly on the value of σ, or more precisely, on the number of FGs. When the database size increases, the index size increases linearly because the answer set size of the FGs increases.



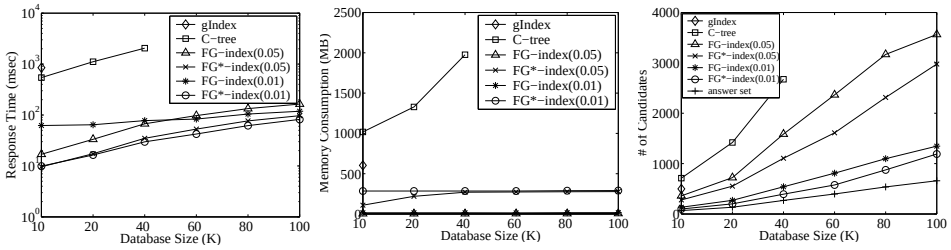
(a) Index Construction Time (b) Peak Mem. Consumption (c) Size of Index on Disk

Fig. 12. The Effect of Database Size on Index Construction



(a) Average Response Time (b) Peak Mem. Consumption (c) Average Candidate Set Size

Fig. 13. Performance of Processing FG-Queries

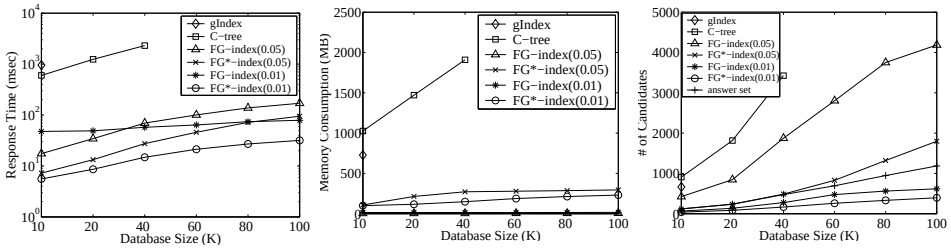


(a) Average Response Time (b) Peak Mem. Consumption (c) Average Candidate Set Size

Fig. 14. Performance of Processing non-FG-Queries

To test the query performance, we prepare three sets of queries, Q_{FG}, $Q_{non\text{-}FG}$ and Q_{mixed}, as shown in Table II. With respect to $\sigma = 0.01$, Q_{FG} consists of only FG-queries, $Q_{non\text{-}FG}$ consists of only non-FG-queries, and Q_{mixed} consists of a mixture of FG-queries and non-FG-queries. The purpose for using the three types of queries is to test whether FG*-index is efficient for all types of queries. In Figures 13 to 15, we report the following three metrics: the average response time per query, the peak memory consumption, and the average size of the candidate set of a query.

The results are very clear: FG*-index at both $\sigma = 0.05$ and $\sigma = 0.01$ achieves remarkable performance improvement as compared with the other indexes. The figures show that FG*-index is more scalable than gIndex and C-tree. Compared with gIndex, FG*-index is over two orders of magnitude faster and also consumes signif-



(a) Average Response Time (b) Peak Mem. Consumption (c) Average Candidate Set Size

Fig. 15. Performance of Processing Both FG and Non-FG Queries

icantly less memory, for all types of queries. Compared with C-tree, FG*-index is two orders of magnitude faster for processing FG-queries and mixed-type queries and, on average, 60 times faster for processing non-FG-queries. Furthermore, the difference becomes greater when the database size increases. The memory consumption of C-tree increases quickly and the memory is used up when the database size is greater than 40K. Compared with FG-index (with respect to the same σ), FG*-index is also significantly faster, although the improvement is not as obvious as compared with gIndex and C-tree. On average, FG*-index is from two times to an order of magnitude faster than FG-index.

The performance improvement of FG*-index can be explained by the size of the candidate sets obtained by FG*-index and the other indexes, as shown in Figures 13(c), 14(c) and 15(c), in which we also give the average size of the answer set of a query as a reference.

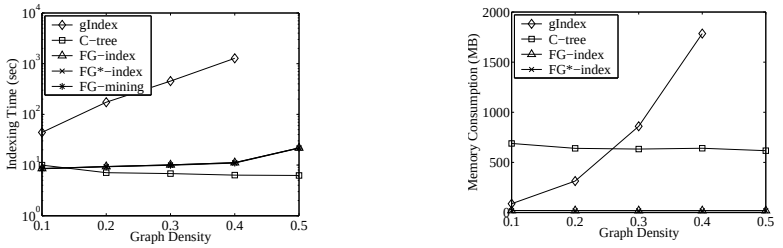
From the results of this experiment, we conclude that FG*-index is significantly more efficient than the other indexes for processing both FG-queries and non-FG-queries, i.e., all types of queries.

Figures 13(b) and 15(b) show that the memory consumption of FG*-index at $\sigma = 0.05$ is slightly higher than that of the other FG*-index and FG-index. This is because Q_{FG} consists of only FG-queries with respect to $\sigma = 0.01$, but those FG-queries that have frequency greater than $0.01|\mathcal{D}|$ but smaller than $0.05|\mathcal{D}|$ are non-FG-queries with respect to $\sigma = 0.05$. Thus, there are both FG-queries and non-FG-queries in Q_{FG} with respect to $\sigma = 0.05$. For the same reason, there are more non-FG-queries in Q_{mixed} with respect to $\sigma = 0.05$ than with respect to $\sigma = 0.01$. As a result, more memory is used for the FAQ-index in FG*-index at $\sigma = 0.05$ for processing Q_{FG} and Q_{mixed}.

Figures 13 to 15 show that the query performance of FG*-index at $\sigma = 0.05$ is also very impressive and close to that of FG*-index at $\sigma = 0.01$. In addition, Figure 12 shows that FG*-index at $\sigma = 0.05$ is the most efficient to construct. Therefore, this set of experiments verifies that the following strategy is not affected by the change in the database size: we can use FG*-index at a larger σ if it is too costly to construct FG*-index at a smaller σ.

7.3.2 *Scalability Test on Graph Density.* As shown in Table II, the average density of the graphs in both $\mathcal{D}_{AIDS}$ and $\mathcal{D}_{cancer}$ is relatively low. Thus, we use the five datasets, $\mathcal{D}_{den0.x}$, by varying the average graph density from 0.1 to 0.5.

We set $\delta = 0.1$ for both FG*-index and FG-index. And we set $\sigma = 0.05$ in this



(a) Index Construction Time (b) Peak Memory Consumption

Fig. 16. The Effect of Graph Density on Index Construction

experiment so that the query processing efficiency does not come from a high index construction cost. For FG*-index, we use $\mathcal{F}_2^3$ to construct the feature-index since the FGs are smaller graphs, and set the available memory for the FAQ-index to 256 MB and the unit length to 4 MB. The settings of gIndex and C-tree are the default values suggested in their papers.

Figure 16(a) shows that the index construction time of FG*-index is comparable to those of FG-index and C-tree, but significantly shorter than that of gIndex. Figure 16(b) shows that constructing FG*-index consumes significantly less memory than constructing both gIndex and C-tree. The construction costs of FG*-index, FG-index and C-tree remain stable over different graph densities. The increase in the cost of constructing gIndex is due to the rapid increase in the number of graphs to be indexed, because a graph with a higher density has more subgraphs.

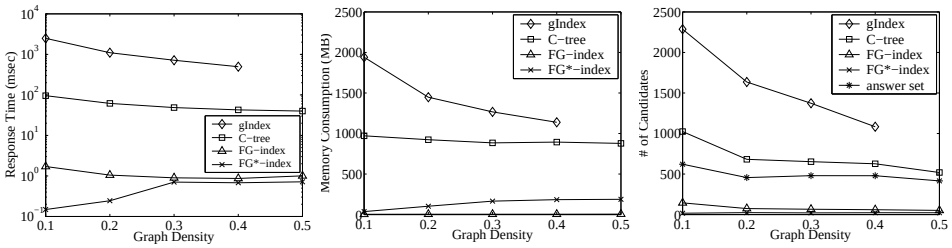
Figure 16(a) shows that it takes slightly longer time to build FG*-index and FG-index at the density of 0.5. This is because considerably more FGs are generated at the density of 0.5. However, the peak memory consumption is not increased because building the index consumes less memory than mining the FGs; thus, the peak memory consumption is taken from mining the FGs, which is relative to the database size and hence remains stable over different densities.

The size of the indexes on the disk is at most 6 MB larger than that of the respective database size on the disk, except that of gIndex which grows from 14 to 74 MB when the density increases from 0.1 to 0.4. This result also confirms the results reported in Figures 16 (a) and (b). We omit the details due to space limits.

To test the query performance, we prepare a set of queries for each of the five datasets, shown as $Q_{den0.x}$ in Table II. We randomly select the queries and do not classify them as FG-queries or non-FG-queries, since gIndex and C-tree do not distinguish between the two types of queries and we have tested the performance of FG*-index on different types of queries in Section 7.3.1.

Figures 17(a) and 17(b) show that FG*-index can process a query orders of magnitude faster than both gIndex and C-tree, and FG*-index also consumes significantly less memory. This result can be explained by the size of the candidate set as shown in Figure 17(c). Note that the candidate set obtained by FG*-index is even significantly smaller than the answer set because candidate verification is only needed for non-FG-queries that are not frequently asked.

The figures show that the query performance of the indexes is not degraded when the density increases. We explain this result as follows. The two main factors that



(a) Average Response Time (b) Peak Mem. Consumption (c) Average Candidate Set Size

Fig. 17. Query Performance on Different Graph Densities

determine the query performance are the index probing cost and the candidate set size. The index probing cost mainly depends on the number of graphs that are indexed, which does not change significantly for different densities. The candidate set size is determined by the answer set of the subgraphs (and also that of the supergraphs for the FAQ-index in FG*-index) of the query. Since a graph with a high density has more subgraphs, more subgraphs of the query can be found to give a small candidate set. This explains why the query performance of the indexes is actually improved slightly when the density increases.

The performance of FG*-index is the best when the density is 0.1 and 0.2. This is different from the other indexes because the query sets of these two densities contain many small-size FG-queries, which can be directly answered by the FHI in the feature-index. This result highlights another advantage of FG*-index: it is efficient in processing small-size FG-queries.

7.4 The Effect of Database Updates on FG*-index

Finally, we assess the performance of our update algorithm using the dataset $\mathcal{D}_{cancer}$ that consists of 20K graphs. We divide the 20K graphs into two databases, namely the *current database* and the *source database*, where each database initially contains 10K graphs. We first build FG*-index on the current database, with $\sigma = 0.05$ and other settings being the same as in Section 7.3.1. This FG*-index is called the *current FG*-index*. Then, at each step, we randomly select 10 graphs from the current database and the source database. The graphs are deleted from the database from which they are selected; however, if the graphs are from the source database, then they are also added to the current database. In this way, we model both the insertion and deletion for the current database.

We randomly select 10K queries from Q_{mixed}. After each update of the current database, we process the 10K queries using the current FG*-index and the *auxiliary structures*, which include the auxiliary FG*-index, $\mathcal{D}_{del}$ and $\mathcal{D}_{new}$. This process continues until we need to rebuild FG*-index from scratch.

At the point when we rebuild a new FG*-index from the new database, about 3.3K new graphs are added and 3K graphs are deleted. We report the experimental results as follows. During the entire update process, 11 auxiliary FG*-indexes are built. The time taken to construct each of them is shown in Table VII. The total time of building these indexes is 15.33 sec, which is longer than the time taken to build the current FG*-index, which takes 14.52 sec, thereby satisfying Condition

Table VII. Construction Time (sec) of the 11 Auxiliary FG*-Indexes

1	2	3	4	5	6	7	8	9	10	11
0.81	0.52	0.71	0.95	1.17	1.33	1.49	1.74	1.99	2.19	2.43

(2) in Section 6. The memory required to build these indexes is 2-4 MB.

The total index construction time for the entire update process is 29.43 sec, which includes the 15.33 sec for building the 11 auxiliary FG*-indexes and 14.10 sec for rebuilding the new FG*-index from the new database at the end of the update.

The average response time of processing a query using the current FG*-index and the auxiliary structures is 18.23 msec, including the total index construction time. However, the result does not show how much the query performance is degraded compared with using the FG*-index that is updated whenever the database is updated. Since it takes too long if we build a new FG*-index for each of the 6.3K graphs that are updated, we use an approximation as follows. We build a new FG*-index at each of the 11 points when an auxiliary FG*-index is built and use this FG*-index to process the 10K queries. The response time per query averaged over the 11 points is 13.75 msec, not including the time to build the FG*-indexes. Thus, the response time of the batch-update strategy is only $(18.23/13.75) = 1.33$ times longer than the (approximated) optimal time. We note that the optimal time does not include the time taken to update the index whenever a graph is inserted or deleted, which can be expensive especially if the index is large.

Finally, we note that we do not draw comparisons with gIndex, C-tree and FG-index, because the update is not implemented in the package provided to us by the authors. Although we are not able to compare with the update-per-graph strategy, we are convinced that our batch-update strategy is more efficient and practical for the following reason. In practice, there can be in fact three operations that we need to process at any time: a query to be processed, a graph to be deleted, and a graph to be inserted. Our update strategy allows us to first place the graphs to be updated into $\mathcal{D}_{del}$ and $\mathcal{D}_{new}$, and continue the query processing instantly, rather than waiting for the update to be completed. This is particularly advantageous when the update is frequent. On the contrary, if the update is not frequent, we can even build FG*-index at a smaller σ to optimize the query performance and perform the update when the system is idle.

8. RELATED WORK

A number of indexes have been proposed for processing subgraph queries. Among them, *GraphGrep* [Shasha et al. 2002] is a path-based approach to indexing graph databases. However, the set of paths in a graph database is huge and hence may affect the performance of the index. [Yan et al. 2005a] propose *gIndex*, which indexes the subgraphs in a database. Since the number of all subgraphs is too large, a set of *discriminative FGs*, $\mathcal{F}_d$, is defined and gIndex is then constructed on $\mathcal{F}_d$. A query q is processed by first generating $\mathcal{C}_q = (\bigcap_{f \in \mathcal{F}_d \wedge f \subseteq q} \mathcal{D}_f)$ and then $\mathcal{D}_q$ is obtained by verifying $\mathcal{C}_q$. Another graph-based approach is *C-tree* [He and Singh 2006] defined on the notion of *graph closure*. Each internal node in C-tree is a closure of its children and each leaf node is a graph in the database. Thus, a closure

is similar to a minimum bounding rectangle in an R-tree. Searching in C-tree is also analogous to that of an R-tree, except that the matching is between graphs. A faster approximate subgraph isomorphism testing is performed between a query and every internal node; however, the exact subgraph isomorphism testing is still required for matching a query with every leaf node (i.e., candidate verification).

We are also aware of a number of recent developments in indexing graph databases. *TreePi* [Zhang et al. 2007] is an index constructed on a set of discriminative features selected from a set of frequent subtrees. A query is first partitioned into a set of features and then matched with the set of indexing features to obtain a candidate set. TreePi also utilizes the location information of the features in the database graphs to further refine the candidate set as well as to facilitate the subgraph isomorphism testing in the verification step. *GString* [Jiang et al. 2007] considers the semantics of the graph structures in the database. A set of basic structures in the specific domain is selected. Both the graphs and the query are transformed into strings in terms of the basic structures. Then, an index is built on the strings of the graphs and query processing is performed as string matching. The use of the basic structures, instead of using individual nodes and edges, not only improves the searching efficiency, but also reduces the candidate set size. For both TreePi and GString, candidate verification is required for processing all queries.

GDIndex [Williams et al. 2007] is an index constructed based on graph decomposition. No candidate verification is needed using GDIndex. However, the index is designed for databases that consist of relatively smaller graphs and do not have a large number of distinct graphs.

[Zhao et al. 2007] use frequent trees as features to build an index. Trees, instead of graphs, are used as features because they achieve a good tradeoff between feature size, feature selection cost and pruning power. The features are used for filtering and to produce a candidate set. Thus, candidate verification is required.

In addition to the above indexes, *Daylight* [James et al. 2003] and *AnMol* [Srinivasa and Kumar 2003] are indexes for processing molecular structures. *DataGuides* [Goldman and Widom 1997], *T-index* [Milo and Suciu 1999], *F&B-index* [Kaushik et al. 2002], *D(k)-index* [Chen et al. 2003], SIT [Cheng and Ng 2004] and SIT-Lattice [Ng and Cheng 2007], and *FIX* [Zhang et al. 2006] are indexes for query processing on semi-structured data and XML. Most of these indexes are based on path or subtree structures.

The concept of δ-tolerance is also used in [Cheng et al. 2006; 2008a] to define concise representations for the sets of frequent patterns and association rules. The goal of their work is for redundancy removal, while we apply it for graph indexing.

9. CONCLUSIONS

We propose an efficient index, FG*-index, for processing subgraph queries on graph databases. FG*-index consists of the following three components: FG-index, the feature-index, and the FAQ-index.

First, FG-index adopts the concept of FGs to classify a large set of queries as FG-queries, which are answered without candidate verification. As shown by our experiments, FG-queries are the most expensive queries to process using other existing indexes due to the large size of the candidate sets.

Second, the feature-index is employed to reduce the high index probing cost, so that more FGs can be indexed to allow more queries to be answered without candidate verification. In addition, queries that are features can be answered instantly using the feature-index.

Lastly, the FAQ-index is used to answer frequently asked non-FG-queries without candidate verification and with negligible index probing cost. If the query is not frequently asked and not an FG, using the FAQ-index allows us to obtain part of the answer set and verify only a small number of candidates.

We evaluate the performance of FG*-index with extensive experiments. The results show that using FG*-index is up to orders of magnitude faster than using the state-of-the-art indexes, including gIndex [Yan et al. 2005a], C-tree [He and Singh 2006] and FG-index [Cheng et al. 2007], for processing both FG-queries and non-FG-queries. FG*-index is also much more scalable than the other indexes.

Finally, we propose a batch-update strategy that enables FG*-index to keep its query processing efficiency while at the same time handling frequent updates. The experimental results show that our update strategy achieves query performance (including the update cost) that is only slightly worse than the optimal query performance (not including the update cost).

Acknowledgements. We thank the anonymous reviewers for their valuable comments, which have helped to improve the quality of the article significantly. We also thank Dr. Xifeng Yan for providing us gIndex, and Mr. Huahai He and Dr. Ambuj K. Singh for providing us C-tree. This work is partially supported by RGC GRF under project number 617808.

REFERENCES

Chen, Q., Lim, A., and Ong, K. W. 2003. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD Conference*. 134–144.

Cheng, J., Ke, Y., and Ng, W. 2006. delta-tolerance closed frequent itemsets. In *ICDM*. 139–148.

Cheng, J., Ke, Y., and Ng, W. 2008a. Effective elimination of redundant association rules. *Data Min. Knowl. Discov. 16,* 2, 221–249.

Cheng, J., Ke, Y., and Ng, W. 2008b. Maintaining frequent closed itemsets over a sliding window. *To appear in Journal of Intelligent Information Systems*.

Cheng, J., Ke, Y., and Ng, W. 2008c. A survey on algorithms for mining frequent patterns over data streams. *Knowledge and Information Systems Journal 16,* 1, 1–27.

Cheng, J., Ke, Y., Ng, W., and Lu, A. 2007. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD Conference*. 857–872.

Cheng, J. and Ng, W. 2004. Xqzip: Querying compressed xml using structural indexing. In *EDBT*. 219–236.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *STOC*. 151–158.

Faloutsos, C., McCurley, K. S., and Tomkins, A. 2004. Fast discovery of connection subgraphs. In *KDD*. 118–127.

Golab, L. and Özsu, M. T. 2003. Issues in data stream management. *SIGMOD Record 32,* 2, 5–14.

Goldman, R. and Widom, J. 1997. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*. 436–445.

Güting, R. H. 1994. Graphdb: Modeling and querying graphs in databases. In *VLDB*. 297–308.

He, H. and Singh, A. K. 2006. Closure-tree: An index structure for graph queries. In *ICDE*. 38.

Holder, L. B., Cook, D. J., and Djoko, S. 1994. Substucture discovery in the subdue system. In *KDD Workshop*. 169–180.

Huan, J., Wang, W., Bandyopadhyay, D., Snoeyink, J., Prins, J., and Tropsha, A. 2004. Mining protein family specific residue packing patterns from protein structure graphs. In *RECOMB*. 308–315.

Huan, J., Wang, W., Prins, J., and Yang, J. 2004. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*. 581–586.

Inokuchi, A., Washio, T., and Motoda, H. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*. 13–23.

James, C. A., Weininger, D., and Delany, J. 2003. Daylight theory manual daylight version 4.82. *Daylight Chemical Information Systems, Inc.*.

Jiang, H., Wang, H., Yu, P. S., and Zhou, S. 2007. Gstring: A novel approach for efficient search in graph databases. In *ICDE*. 566–575.

Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. 2002. Covering indexes for branching path queries. In *SIGMOD Conference*. 133–144.

Ke, Y., Cheng, J., and Ng, W. 2007. Correlation search in graph databases. In *KDD*. 390–399.

Ke, Y., Cheng, J., and Ng, W. 2008. Efficient correlation search from graph databases. *To appear in IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

Koren, Y., North, S. C., and Volinsky, C. 2006. Measuring and extracting proximity in networks. In *KDD*. 245–255.

Manku, G. S. and Motwani, R. 2002. Approximate frequency counts over data streams. In *VLDB*. 346–357.

Milo, T. and Suciu, D. 1999. Index structures for path expressions. In *ICDT*. 277–295.

Ng, W. and Cheng, J. 2007. An efficient index lattice for xml query evaluation. In *DASFAA*. 753–767.

Shasha, D., Wang, J. T.-L., and Giugno, R. 2002. Algorithmics and applications of tree and graph searching. In *PODS*. 39–52.

Srinivasa, S. and Kumar, S. 2003. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. In *VLDB*. 975–986.

Tong, H. and Faloutsos, C. 2006. Center-piece subgraphs: problem definition and fast solutions. In *KDD*. 404–413.

Tong, H., Faloutsos, C., Gallagher, B., and Eliassi-Rad, T. 2007. Fast best-effort pattern matching in large attributed graphs. In *KDD*. 737–746.

Williams, D. W., Huan, J., and Wang, W. 2007. Graph database indexing using structured graph decomposition. In *ICDE*. 976–985.

Yan, X. and Han, J. 2002. gspan: Graph-based substructure pattern mining. In *ICDM*. 721–724.

Yan, X. and Han, J. 2003. Closegraph: mining closed frequent graph patterns. In *KDD*. 286–295.

Yan, X., Yu, P. S., and Han, J. 2005a. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst. 30,* 4, 960–993.

Yan, X., Yu, P. S., and Han, J. 2005b. Substructure similarity search in graph databases. In *SIGMOD Conference*. 766–777.

Yu, J. X., Chong, Z., Lu, H., and Zhou, A. 2004. False positive or false negative: Mining frequent itemsets from high speed transactional data streams. In *VLDB*. 204–215.

Zhang, N., Özsu, M. T., Ilyas, I. F., and Aboulnaga, A. 2006. Fix: Feature-based indexing technique for xml documents. In *VLDB*. 259–270.

Zhang, S., Hu, M., and Yang, J. 2007. Treepi: A novel graph indexing method. In *ICDE*. 966–975.

Zhao, P., Yu, J. X., and Yu, P. S. 2007. Graph indexing: Tree + delta >= graph. In *VLDB*. 938–949.

...