iGraph: A Framework for Comparisons of Disk-Based Graph Indexing Techniques

Wook-Shin Han
Department of Computer Engineering
Kyungpook National University, Korea
wshan@knu.ac.kr

Jinsoo Lee
Department of Computer Engineering
Kyungpook National University, Korea
jslee@www-db.knu.ac.kr

Minh-Duc Pham
Department of Computer Engineering
Kyungpook National University, Korea
duc@www-db.knu.ac.kr

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

ABSTRACT

Graphs are of growing importance in modeling complex structures such as chemical compounds, proteins, images, and program dependence. Given a query graph Q, the *subgraph isomorphism* problem is to find a set of graphs containing Q from a graph database, which is NP-complete. Recently, there have been a lot of research efforts to solve the subgraph isomorphism problem for a large graph database by utilizing graph indexes. By using a graph index as a filter, we prune graphs that are not real answers at an inexpensive cost. Then, we need to use expensive subgraph isomorphism tests to verify filtered candidates only. This way, the number of disk I/Os and subgraph isomorphism tests can be significantly minimized. The current practice for experiments in graph indexing techniques is that the author of a newly proposed technique does not implement existing indexes on his own code base, but instead uses the original authors' binary executables and reports only the wall clock time. However, we observe this practice may result in several problems. In order to address these problems, we have made significant efforts in implementing all representative indexing methods on a common framework called iGraph. Unlike existing implementations which either use (full or partial) in-memory representations or rely on OS file system cache without guaranteeing real disk I/Os, we have implemented these indexes on top of a storage engine that guarantees real disk I/Os. Through extensive experiments using many synthetic and real datasets, we also provide new empirical findings in the performance of the full disk-based implementations of these methods.

1. INTRODUCTION

Proceedings of the VLDB Endowment, Vol. 3, No. 1

Graphs are of growing importance in modeling complicated structures such as chemical compounds [7, 8], proteins [11], images [17], and program dependence [16]. The number of graphs in a database can grow up to tens of millions. For example, PubChem [8] contains more than 30 million chemical compounds in its database.

One of most important graph queries is the *graph containment*. That is, given a query graph Q and a database D of graphs, find all graphs in D that contain Q. This problem is also called the *subgraph isomorphism* problem, which belongs to NP-complete [23].

Recently, there have been a lot of research efforts to solve the subgraph isomorphism problem by utilizing graph indexes and a filter-and-refinement strategy. Thus, using a graph index as a filter, we first prune graphs that are not real answers at a low cost. Then, we need to use expensive subgraph isomorphism tests to verify filtered candidates only. This way, the number of disk I/Os and subgraph isomorphism tests can be significantly minimized.

Note that there are two different research streams processing graph queries. One stream handles a large number of small graphs such as chemical compounds using subgraph isomorphism. Most existing approaches [12, 20, 21, 26, 27, 29, 30, 31] belong to this stream. The other stream handles a small number of large graphs using *approximate* graph search such as for a large ontology and for protein structures [22, 28]. The former is the focus of our study.

The current practice for experiments in graph indexing techniques is that the author of a newly proposed technique does not implement existing indexes on his own code base, but instead uses the original authors' binary executables and reports only the wall clock time. However, we observe this practice may result in several problems as follows:

- The elapsed time can be a very brittle measure unless all implementations are done on the same code base.
- The number of disk I/Os, a very robust measure, is not used.
- Existing experiments use small data and index files (less than 20 MBytes in most cases), and thus, all files are already cached in the OS file system during experiments. Thus, real disk I/Os are not properly obtained. This can be very prob-

lematic when we deal with large data and index files. Furthermore, some index techniques use only in-memory representations, i.e., all graphs and indexes are resident in memory.

- Thus, instead of minimizing disk I/Os, recent algorithms have focused on minimizing CPU costs by using different data structures and algorithms (such as subgraph isomorphism algorithms) optimized for their environments.
- Only small numbers of graphs are tested. Most of the existing executables fail with a large number of graphs. The maximum number of graphs tested in previous experiments is 100,000.
- Deep analysis of experiments is very hard; it is difficult to see why one method is better than another.

In order to address the aforementioned problems, we have made significant efforts in implementing all representative indexing methods on a common framework called iGraph. Since we were unable to acquire the source code of any technique from its authors, we use best-effort re-implementations based on the original papers. Especially, for all indexing methods, we use the same subgraph isomorphism algorithm and a common storage engine that guarantees real disk I/Os by bypassing the OS file system cache. Note that all current DBMSs support this function by using either a raw device or a special OS API. Our storage engine supports standard disk-based structures and algorithms including heap files, B+-trees, inverted indexes, disk-based prefix trees, binary large object (BLOB) files, an LRU buffer manager, m-way posting list intersection, and external sorting. By using iGraph, the effects of using different structures and algorithms can be minimized. In addition, we can easily see the effect of the buffer size. The source code of the graph indexes that we implement will be released following publication of this paper.

Our goal is to provide a fair comparison of disk-based graph indexing techniques by using a common framework and full disk-based implementations rather than (full or partial) in-memory based implementations. We also perform a thorough experimental evaluation of the existing graph indexing methods using many small and large datasets and present a detailed analysis.

The remainder of this paper is organized as follows. Section 2 reviews the background information as well as existing work on graph indexing. Section 3 presents the details of our implementations. Section 4 presents the results of performance evaluation. Section 5 summarizes and concludes our paper.

2. BACKGROUND

2.1 Problem Definition

In this paper, we use an *undirected labeled graph* for modeling complicated structures. A graph g is defined as a quadruple (V, E, L_v, L_e) where V is the set of vertices, E ($\subseteq V \times V$) is the set of undirected edges, and L_v and L_e are label functions that map a vertex or an edge to a label, respectively.

Now, we formally define our problem as follows:

Definition 1. [26] A graph $G_1 = (V_1, E_1, L_{1v}, L_{1e})$ is subgraph isomorphic to $G_2 = (V_2, E_2, L_{2v}, L_{2e})$ if there is an injective function $f : V_1 \rightarrow V_2$ such that (1) $\forall v \in V_1, L_{1v}(u) = L_{2v}(f(u))$, and (2) $\forall (v, u) \in E_1, (f(u), f(v)) \in E_2$ and $L_{1e}(u, v) = L_{2e}(f(u), f(v))$. Here, we can say G_1 is a subgraph of G_2 or G_2 is a supergraph of G_1. □

2.2 Related Work

We can classify existing algorithms into two categories depending on whether or they use frequent graph mining.

2.2.1 Mining Based Approaches

Indexing: Representative indexing techniques in this category are gIndex [26, 27], FG-Index [12], Tree+Δ [30], and SwiftIndex [20]. In their indexing algorithms, they first extract subgraphs as features using (modified) frequent graph mining algorithms. Thus, we can obtain a set of features F where each f in F is associated with a posting list of graph IDs for the graphs containing f. We say a feature f is *frequent* if $|D_f| \geq minSup$ where D_f is a set of graphs in a given graph database D such that f is a subgraph of every graph in D_f. $|D_f|$ is called the *support* of f.

The gIndex and FG-Index exploit gSpan, a frequent graph mining algorithm. gIndex generates all frequent subgraphs of size up to $maxL$ as well as a subset of infrequent subgraphs of size up to $maxL$ using the so-called *size-increasing support function* while FG-Index generates all frequent subgraphs regardless of their size. With the size-increasing support function, the smaller the size of features, the more infrequent subgraphs are included. FG-Index additionally includes all infrequent edges (of size 1) for completeness of its algorithm. After generating all features using the mining algorithm, gIndex then removes all non-discriminative features. A feature f is *discriminative* with respect to F if $\frac{|\bigcap_{f' \in \mathsf{F} \wedge f' \subsetneq f} D_{f'}|}{|D_f|} \geq \gamma_{min}$.

The Tree+Δ and SwiftIndex perform their own tree mining algorithms to extract trees from graphs since the most frequent features are trees, and the cost of tree mining from graphs may be cheaper than that of graph mining. Tree+Δ generates all frequent trees of size up to $maxL - 1$[1], while SwiftIndex generates all frequent and discriminative trees of size up to $maxL$. Both also generate all infrequent edges as features for the completeness of their algorithms.

In addition to tree features, Tree+Δ generates graph features on the fly while processing queries in order to improve the pruning power. The process of choosing graph features on the fly is as follows: Tree+Δ chooses all simple cycles C from a query graph without checking whether or not they are discriminative, and then, for each cycle c in C, Tree+Δ extends c by growing one vertex and checks whether the extended graph g is discriminative with respect to c only, which does not guarantee that g is a discriminative feature. If g is qualified, g is added to Δ. We repeat the enlargement of g until it reaches $maxL$. Thus, Tree+Δ may generate a lot of features in Δ if the queries have many cycles. In the worst case, the size of Δ may exceed the size of all graph features in gIndex as we will see in our experiments with dense datasets.

Query Processing: Algorithm 1 shows a framework for processing a subgraph containment query. First, it finds features associated with their posting lists. Next, it intersects all posting lists to obtain candidate graphs. Finally, it executes a subgraph isomorphism test for each candidate for refinement.

In FindFeatures, gIndex and Tree+Δ enumerate all subgraphs or subtrees of size up to $maxL$ for a given query q, respectively, and check whether such features are in their indexes. To eliminate unnecessary features, they remove a feature f' if f exists in their indexes such that $f' \subsetneq f$, since the graph IDs in the posting list of f is a subset of the graph IDs in the posting list of f'. To minimize the enumeration cost, we don't enumerate any supergraph of f' if f' is not in the index. Tree+Δ further finds graph features in its Δ.

[1]In [30], the size of a graph pattern is defined as the number of vertices rather than the number of edges.

Algorithm 1 QueryProcessing(q, I, D)

- q: an input graph
- I: a graph index
- D: a graph database

```
1: F ← FindFeatures(q, I);
2: C ← IntersectPostingLists(F); /*|F|-way merge join*/
3: R ← ∅;
4: for each candidate c in C
5:    if SubgraphIsomorphism(c, q) = true
6:       R ← R ∪ {c};
7: return R;
```

In order to minimize the enumeration cost, FindFeatures of FG-Index does not enumerate all possible features. Our new finding is that this strategy reduces the enumeration cost but could result in poor pruning power, which might result in poor performance especially for graph databases having a large number of graphs as we will see in the experiment. FG-Index also proposes a simple yet effective strategy called *verification-free* strategy. That is, if a feature f corresponding to a query graph g is in the index, we can be sure that all graph IDs in the posting list of f are answers without performing subgraph isomorphism tests with g. We note that this strategy can be readily applied to any feature-based indexing techniques. We also note that, in their implementation, part of the features as well as their posting lists are resident in memory.

SwiftIndex supports a fast, feature finding algorithm called Prefix-QuickSI. Without enumerating all subtrees of a query, it executes subgraph isomorphism tests between a query and all features in the index in "batch fashion." For this purpose, [20] encodes a feature as QI-Sequence and stores all features in a prefix tree (please refer to [20] for details). Their original implementation, however, is in-memory based (that is, all graphs in the database and indexes are resident in memory), and thus, it is not clear whether it would still be fast in a disk-based version.

2.2.2 Non-Mining Based Approaches

The representative indexing techniques in this category are GraphGrep [21], C-Tree [29], and gCode [31]. Since their indexing and query processing methods are very different from each other, we explain each method individually.

The GraphGrep enumerates as features all paths of size up to a threshold length. For query processing, it follows Algorithm 1, which is a common framework for *filtering-and-verification*. As shown in [26, 27], its pruning power is significantly lower than that of gIndex, since paths as features do not preserve structural information of graphs compared with subgraphs and subtrees.

The C-Tree uses the concept of *graph closure*, which is a "bounding graph" that contains a set of graphs. C-Tree builds a hierarchical tree using this concept. In query processing, it traverses from the root node to the leaf node by performing the *pseudo* subgraph isomorphism test for each node. If the graph closure for a node N is disqualified by the pseudo subgraph isomorphism test, C-Tree does not access N, so that all graphs recursively contained by N are pruned. However, our new findings show that there are three serious problems in this framework. 1) If we reach the leaf node, we first access each graph in the node and execute a pseudo subgraph isomorphism test before executing the subgraph isomorphism test algorithm. The original implementation of C-Tree uses the Ullmann's algorithm [23] as a subgraph isomorphism algorithm, which is slower than the pseudo subgraph isomorphism test, which is much slower than the state-of-the art subgraph isomorphism algorithms such as VF2 [9] and QuickSI [20]. 2) Unlike a fixed size bounding box in R*-tree, a graph closure is variable-sized, and the sizes of graph closures in upper-level nodes can be very large (exceeding a page size of 8 KBytes). Thus, the pseudograph isomorphism test for non-leaf nodes is very expensive. 3) The dead space of a graph closure (i.e., space inside a graph closure that contains no graph) is very large so that the pruning power of C-Tree is the lowest among all representative indexing methods we consider. Thus, in some cases, C-Tree is even slower than the naive sequential scan performing a subgraph isomorphism test for each graph in the database as we will see in the experiment.

The gCode proposes a two-step filtering strategy: one at the index-level and the other at the object-level. In indexing, for each graph g in the graph database, gCode first computes a vertex signature for each vertex v which is a combination of neighborhood information around v (that is, v's local structure) and the two largest Eigenvalues for local structures (by mapping neighborhood information to an adjacency matrix and computing Eigenvalues from the matrix). It then generates a graph signature for g by combining all vertex signatures. These graph signatures are indexed in a tree structure called GCode-Tree. gCode also maintains a vertex signature dictionary to store all distinct vertex signatures. In addition, for each graph g, gCode maintains a list of pairs $<sid, cnt>$. Here, sid is a vertex signature ID which is a key to probe the vertex signature dictionary; cnt refers to how many times this signature appears in g. In query processing of a query graph q, gCode extracts vertex signatures and a graph signature from q. As index-level filtering, gCode finds all qualified signatures from GCode-Tree. For object-level filtering, for each qualified graph g, it accesses the list of pairs (vertex signature ID, the frequency of this signature) of g and performs pruning (refer to [31] for details about pruning rules). In this case, we have to probe the vertex signature dictionary to obtain vertex signatures for g. Our new findings show that the index-filtering alone is not powerful for real spare datasets, and thus, the number of candidates after the index-level filtering can be very large. This can result in many disk I/Os to access candidate graphs as we will see in the experiment.

3. IMPLEMENTATION

We first explain how iGraph stores graphs in a database. A graph consists of a list of vertices and a list of edges. A graph g is stored as a tuple in a heap page if the size of g is less than the page size. Otherwise, g is stored as a BLOB. The storage format of g is as follows: 1) For representing each vertex, we store a pair of (vid, $vlabel$). Since we store vertices in vid order, we omit storing vid; 2) For representing each edge, we store a triple (s_{vid}, t_{vid}, $elabel$) where s_{vid} and t_{vid} represent IDs of vertices forming this edge, and $elabel$ represents the label of the edge. Each graph is assigned a unique graph ID. To efficiently find a tuple containing the corresponding graph using a given graph ID gid, we build a B+-tree whose key is a graph ID.

To store graph/tree features, gIndex and Tree+Δ use an inverted index such that the search key is the hash value of a canonical graph code of a graph. A posting list can be stored as a tuple if the size of the posting list is smaller than the page size; otherwise, it is stored as a BLOB. gIndex uses the DFS code as a canonical graph code [25] and Tree+Δ uses the BFS code as a canonical tree code, which is similar to this DFS code. In Tree+Δ, Δ is implemented as an inverted index for graph features where each feature is encoded as the CAM code [15]. We use djb2 [19] as the hash function, which is considered robust.

The gCode uses GCode-Tree as an index-level filter for storing a set of graph signatures. Each graph signature for g contains a hash value and a list of Eigenvalues for all vertices of g. Each leaf

entry of the GCode-Tree points to a list of Eigenvalues, which are stored in a tuple of a heap page. gCode also maintains a vertex signature dictionary, which is implemented as a B+-tree where its key and value are a vertex signature ID and a vertex signature. In addition, for each graph g, gCode maintains a list of pairs (vertex signature ID, the frequency of this signature). To minimize the disk I/O cost, we store this list together with the list of Eigenvalues in a tuple of a heap file so that we don't need any extra I/O.

The SwiftIndex uses a prefix tree to store QI-Sequences of features [20]. In the original implementation of SwiftIndex, all graphs and the prefix tree in the database are first loaded in memory before query execution. For a fair comparison, we instead use a disk-based prefix tree where each node is stored as a mini-page [10] rather than a full page, since each node can be much smaller than the regular page. However, the size of the root node of the prefix tree can be large, and thus, the root node is stored as a full page. In case a node is larger than a mini-page, an overflow page is used. To efficiently mine frequent and discriminative features, we use a tree mining algorithm rather than following the original algorithm which is very slow with VF2.

The FG-Index uses a multi-level index tree called core FG-Index where each node is an *Inverted-Graph-Index* (IGI). FG-Index partitions frequent features into groups, and each group is assigned to an IGI. Note that the size of each IGI is variable and could be very large. Each IGI consists of a list of features and an inverted index for efficiently finding these features in this IGI. The list of features is stored in a list of heap pages since the size of the list can be much larger than one page. In the original implementation of FG-Index, the root IGI is loaded in memory before executing queries. Since this kind of optimization can be applied to any other feature-based indexing techniques, for a fair comparison, we don't make the IGI resident in memory but instead let the buffer manager handle index pages in memory.

Regarding C-Tree, we follow the original implementation exactly using a java bytecode analyzer. We allocate a page for each node in the C-Tree unless the size of a node exceeds the page size. If the size of a node is larger than the page size, an overflow page is used.

To efficiently intersect a set of posting lists, we exploit an m-way, external, merge join for all feature-based indexing techniques. For sorting, we use an m-way, external, merge sort algorithm. In order to minimize the disk I/Os, we don't generate the last run in disk.

4. EXPERIMENTS

We evaluate the performance of all representative graph indexing techniques. The algorithms considered are as follows: gIndex [26, 27], C-Tree [29], FG-Index [12], Tree+Δ [30], gCode [31], and SwiftIndex [20]. We also tested SeqScan (a naive sequential scan) as a sanity check. Since we were unable to acquire the source code of any technique from its authors, we use best-effort re-implementations based on the original papers. However, for C-Tree, we followed the original implementation exactly using a java bytecode analyzer. In order to mine features for gIndex and FG-Index, we used gboost [5], an open source implementation of gSpan[2]. All techniques are implemented with a common storage engine using Microsoft Visual C++.

[2]We can use Gaston for faster tree/graph mining instead of gSpan.

4.1 Experiment Setup

Datasets: For small datasets, we use a real AIDS Antiviral dataset (refereed to here as AIDS) [3] and five synthetic datasets. AIDS consists of 10,000 graphs and is a subset of the AIDS Antiviral dataset[3]. Note that the AIDS Antiviral dataset is the standard graph dataset utilized in virtually all related work [12, 14, 20, 25, 26, 27, 29, 30, 31]. AIDS has 25.42 vertices and 27.40 edges, on average. The numbers of distinct vertex labels and distinct edge labels are 51 and 4, respectively.

In order to generate synthetic datasets, we use GraphGen [6], a synthetic graph generator. GraphGen can generate a collection of graphs with controlling input parameters, such as the number of graphs, the average size of each graph, the number of unique vertex/edge labels, and the average density of each graph. The density d of a graph $g = (V, E)$ is defined as $\frac{\text{\# of edges in } g}{\text{\# of edges in a complete graph}}$. That is, the denominator is $\frac{|V| \times (|V|-1)}{2}$. The density has a normal distribution with the input value as the mean and 0.01 as the variance. We generated five datasets by varying the density (0.3, 0.5, 0.7) and setting the number of vertex/edge labels to 50, and by varying the number of vertex/edge labels (20, 50, 80) and setting the number of density to 0.5. The average size of a graph is 30. For example, a data set Synthetic.10K.E30.D5.L20 means that it contains 10,000 graphs; the average size (the number of edges) of each graph is 30; the density for each graph is 0.5; and the number of distinct vertex/edge labels is 20.

For the large dataset, we use a real chemical compound dataset (referred to here as PubChem). PubChem consists of 1 million graphs and is a subset of the PubChem chemical structure database[4]. PubChem has 23.98 vertices and 25.76 edges, on average. The numbers of distinct vertex labels and distinct edge labels are 81 and 3, respectively.

Since C-Tree does not support edge labels, we instead insert an additional vertex for each edge label as in [31].

Query sets: For AIDS, we use the existing query sets $Q4$, $Q8$, $\cdots$, $Q24$ which can be downloaded from [3]. This query set has been used in [12, 20, 27, 26]. Each query set Qn contains 1000 graphs where each graph size is n. In order to generate query sets for the other datasets, we first randomly select 1000 graphs from each dataset whose size is larger than or equal to 24. Then, for each graph g, we remove edges until g is still connected and contains 24 edges. This query set is called $Q24$. In order to generate $Q20$, we remove edges from each graph in $Q24$ until the remaining graph contains 20 edges. We repeat this process to generate the remaining query sets. For AIDS, we also generate queries using our query generator and perform experiments. Since the trends of experimental results are similar to those using the existing query sets in [3], we omit these experimental results for brevity.

Setup: We conducted all the experiments on a Window Vista machine with Xeon Quad Core 2.27 GHz CPU and 8 GBytes RAM. We used LRU as the buffer replacement algorithm, and set the page size to 8 KBytes. We used a Barracuda SATA 1TB 7,200 RPM hard disk; the average latency is 4.16ms; the maximum random read seek time is 8.5ms; and the maximum random write seek time is 9.5ms [1].

We adopt the default parameter values used in each technique: for all feature-based indexing techniques, the support threshold is set to 10%, and the maximum feature size $maxL$ = 10. For gIndex and SwiftIndex, γ_{min} is set to 2. For FG-Index, δ is set to 0.1. For gIndex, we follow the same size-increasing function as in [26].

[3]http://dtp.nci.nih.gov/

[4]ftp://ftp.ncbi.nlm.nih.gov/pubchem/

We use the average number of candidates, the average number of page accesses, and the average wall clock time as the performance metrics. We use VF2 [13] for the subgraph isomorphism test, which can be downloaded from [9]. To avoid the buffering effect of the OS file system and to guarantee actual disk I/Os, we use the FILE_FLAG_NO_BUFFERING flag [2] when opening data and index files.

Since each query set contains 1000 queries, we can either clean the current buffer for each query execution or reuse the current buffer for the next queries. We name the former *cold run* while the latter is named *hot run*. We performed experiments with both hot run and cold run.

4.2 Results for Small Graph Databases

4.2.1 AIDS dataset

Database construction Cost: Table 1 shows database construction costs for AIDS. Since SeqScan does not have any indexes, its construction cost (i.e., database loading cost only) is the minimum among all techniques. Except for SeqScan, the database construction cost contains 1) the database loading cost and 2) the index construction cost. To calculate the index construction cost of an indexing technique T, we need to deduct the database construction cost of SeqScan from the database construction cost of T. Note that there are no features in C-Tree or gCode.

Table 1: Database construction cost for AIDS.

	Construction time (sec.)	# of features	Size (MByte)
SeqScan	1.59	NA	3.90
C-Tree	3.88	NA	10.51
gIndex	26.57	1220	5.95
FG-Index	19.53	664	10.43
Tree+Δ	15.05	640	7.94
gCode	6.50	NA	9.45
SwiftIndex	16.22	283	4.80

The database construction costs of C-Tree and gCode are the lowest, since they do not perform the expensive mining process. This result is consistent with those reported in [12, 20, 27, 30].

The database construction cost of gIndex is comparable to all feature selection methods such as Tree+Δ, FG-Index, and SwiftIndex. This result differs significantly from those reported in [12, 20, 27, 30]. This is due to the fact that these papers 1) ignore edge labels without any clear reason[5] and 2) do not utilize their own gIndex implementation on the same code base, but instead use the binary executable of the original implementation. The main reason for this discrepancy between our implementation and the original is that the discriminative feature selection algorithm of the original "*might use a slower implementation*" than ours.[6] Note that the major cost in indexing is the mining cost, and there is no significant performance differences between tree mining "from graphs" and graph mining since most features are trees. We confirm this fact with Gaston [4, 24], a state-of-the-art tree/graph mining tool, which supports tree/graph mining "from graphs" within the same framework. However, since gIndex generates all infrequent and discriminative features of small sizes (by up to three), its indexing cost might significantly increase if there are too many small, infrequent features as we will see in experiments with synthetic datasets. We also note that gIndex has the largest number of features but it index size is smaller than FG-Index and Tree+Δ, since its average posting size is much smaller than those of FG-Index and Tree+Δ. That is, non-discriminative features whose posting sizes are large are removed from gIndex.

Query Processing Cost: Figure 1 shows the average number of candidates by varying query sets. Note that the average number of candidates is invariant to the buffer size. Although gCode and C-Tree perform well for most query sets, this information might be misleading, since both methods execute additional object-level filtering before running the subgraph isomorphism test for each candidate; gCode executes vertex signature based filtering for each candidate, while C-Tree runs a pseudo subgraph isomorphism test for each candidate leaf entry (i.e., each candidate graph). Thus, we added two additional curves, gCode (I) (gCode with index-level filtering only) and C-Tree (I) (C-Tree with index-level filtering only), to plot the average number of candidates at the index level. As we see here, C-Tree (I) and gCode (I) perform the worst since their indexing-level pruning power is the lowest. This new finding can explain why C-Tree and gCode perform poorly as we will see.

The gIndex performs the best for most queries since it has the largest number of (frequent and discriminative) features in its index, and thus its pruning power is the best. FG-Index performs the worst among all feature-based indexing techniques in terms of the average number of candidates (7.80 times more candidates generated compared with gIndex). In fact, [12] did not report the number of candidates. This is because FG-Index does not find all features from its index that are subgraphs of a given query, but instead uses a strategy to find a subset of index features to minimize the feature finding cost. However, this strategy may incur a lot of candidates for sparse graph databases, which might lead to poor performance in large databases as we will see in experiments for the one million database, PubChem. SwiftIndex ranks in between Tree+Δ and FG-Index in terms of the average number of candidates, since it uses a subset of features in Tree+Δ but exploits all necessary features in its index for query processing unlike FG-Index.

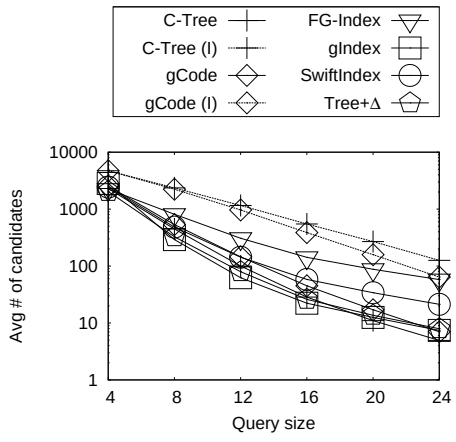
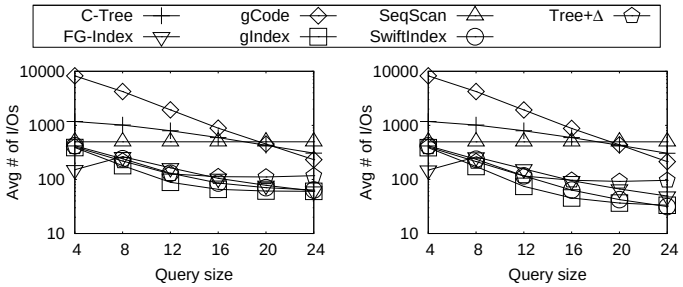


Figure 1: Average number of candidates by varying query sets (AIDS).

[5]We have communicated with Xifeng Yan who first ignored the edge label. He did that simply in order to "make the problem more difficult." Subsequent work imitated his setting without clear reason.

[6]Regarding this fact, we have communicated with Xifeng Yan. He also believes that wall-clock comparisons should be avoided unless the implementations are done on the same code base.

Figure 2 shows the average number of I/Os by varying query sets (from $Q4$ to $Q24$) when the buffer size is set to 1 MByte[7]. gIndex shows the best performance for all query sets except $Q4$. For $Q4$, FG-Index performs the best since it exploits the verification-free strategy. Note that the other feature-based indexing techniques such as gIndex, SwiftIndex, and Tree+Δ can easily support this verification-free strategy as well. Thus, gIndex can perform similarly to FG-Index even for a very small query size. gCode performs the worst. This phenomenon is explained as follows: 1) gCode (I) generates up to 17.45 times more candidates in the index level than gIndex; 2) For a given query $Q = (V, E)$, $|V|$ index lookups over the vertex signature dictionary are required for each candidate. Therefore, if the buffer size is insufficient for buffering all pages necessary for such lookups, gCode performs very poorly. We also notice that, gIndex and SwiftIndex achieve slightly better performance with hot run (see Figure 2(b)) since it accesses the smallest number of pages, and some pages accessed for the previous query execution can reside in the buffer and thus, can be used for the current and next queries. Regardless of query sets, SeqScan accesses a constant number of disk I/Os since it must scan the whole graph database. Unlike the other indexing methods, the number of disk I/Os in Tree+Δ increases as the query size increases. This is because Tree+Δ mines graph features on the fly during query execution, and larger query sizes may have more graph features.

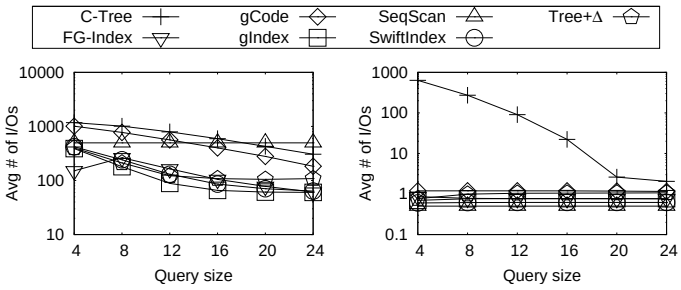


(a) Cold Run. (b) Hot Run.

Figure 2: Number of I/Os by varying query sets (AIDS, buffer size = 1 MByte).

Figure 3 shows the average number of I/Os by varying query sets when the buffer size is set to 10 MBytes. Note that the buffer size (1,200 pages) is larger than the database of each technique except C-Tree (1,345 pages). As shown in Figure 3(a), all techniques except gCode show almost the same performance as Figure 2(a). However, we observe that a drastic improvement occurs with gCode compared with the buffer size being set to 1 MByte, since the cost of index lookups over the vertex signature dictionary can be minimized with a large buffer size. As shown in Figure 3(b), with hot run, all techniques except C-Tree show constant performance regardless of query sizes, since all database pages can reside in the buffer after their initial access. As for C-Tree, the number of disk I/Os is slightly reduced compared with a small buffer size, since the database size of C-Tree is still larger than the buffer size, and tree traversal incurs the *sequential flooding effect* [18] (i.e., pages buffered are replaced before they are reused). We performed additional experiments with the buffer size being set 100 MBytes. C-Tree also showed constant performance. With hot run, SeqScan performs the best in terms of disk I/Os which does not need to access additional index pages.

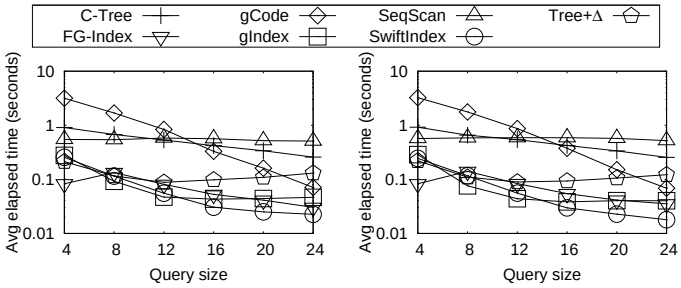
[7]In our storage manager, the number of buffer pages allocated must be a multiple of 100. Thus, 100 pages (not 120 pages) are allocated in this case. This is about 20% of the database size of SeqScan.



(a) Cold Run. (b) Hot Run.

Figure 3: Number of I/Os by varying query sets (AIDS, buffer size = 10 MBytes).

Figure 4 shows the average elapsed time by varying query sets when the buffer size is set to 1 MByte. The trends of all curves are consistent with those for the number of I/Os. This is because the disk I/O cost is dominant in overall performance unless there are significant differences among the numbers of subgraph isomorphism tests. For gIndex, although its cost is lowest in terms of the average numbers of candidates and disk I/Os, it is slightly slower than FG-Index and SwiftIndex in terms of the average elapsed time. This is due to slow subgraph enumeration from a query in gIndex. For $Q24$, the subgraph enumeration cost in gIndex constitutes 57.3% of the total query processing time. We note that there is room to optimize since gIndex enumerates subgraphs using gSpan, which is not optimized for extracting subgraphs from a single graph. We also note that VF2 performs very well for labeled graphs. We may accelerate the subgraph isomorphism test further by using QuickSI [20]. This fact indicates that the I/O cost must be carefully optimized to obtain good performance. With the buffer size set to 10 MBytes, the trends of elapsed times are similar to those in Figure 3.



(a) Cold Run. (b) Hot Run.

Figure 4: Average elapsed time by varying query sets (AIDS, buffer size = 1 MByte).

4.2.2 Synthetic dataset

The default dataset we use is Synthetic.10K.E30.D5.L50. The purpose of this experiment is to see how performance changes with varying parameter values.

Database Construction Cost: Table 2 shows database construction costs for the default synthetic dataset. Compared with Table 1, there exist remarkable changes in the database construction costs. First, the number of frequent features for these dense graphs is only 37. Almost all features in FG-Index, Tree+Δ, and SwiftIndex are infrequent features. Second, a very small number of frequent features enable fast indexing times for FG-Index, Tree+Δ, andSwiftIndex. C-Tree also performs fast indexing since it does not generate features. The indexing cost of gCode is much

higher than those of these indexing methods. Third, the indexing cost of gIndex is much more expensive than other feature-based indexing methods since it mines all infrequent and discriminative features of size up to 3.

Table 2: Database construction cost for the default synthetic dataset.

	Construction time (sec.)	# of features	Size (MBytes)
SeqScan	1.42	NA	3.57
C-Tree	8.92	NA	11.69
gIndex	290.02	44250	12.99
FG-Index	6.55	13616	5.22
Tree+Δ	6.42	13614	5.59
gCode	70.36	NA	29.57
SwiftIndex	8.00	13606	4.77

Query Processing Cost: Figure 5 shows the average number of candidates by varying query sets for Synthetic.10K.E30.D5.L50. Compared with the results in Figure 1, there are drastic changes to gCode (I), C-Tree (I), and Tree+Δ. gCode (I) is much more effective with large query sizes for dense graph databases. C-Tree (I) performs more effective for dense datasets. Tree+Δ performs much better than the other feature based indexing techniques especially for large query sizes. This is explained as follows: we run experiments using $Q4$, $Q8$, $\cdots$, and $Q24$ in this order (i.e., in order of increasing query size). Thus, graph features reclaimed (i.e., mined graphs at query processing time) at small sizes are used for larger query sizes. However, in return, Tree+Δ suffers from very slow performance in terms of disk I/Os and the elapse time as we will see in the next paragraph.

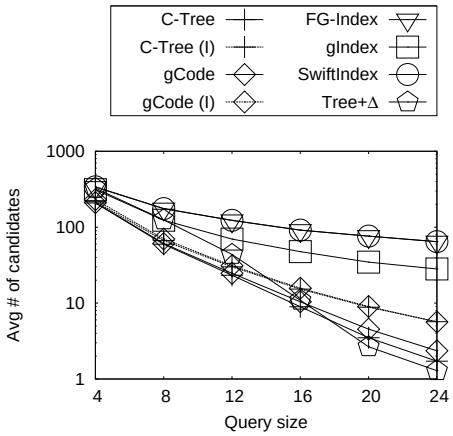
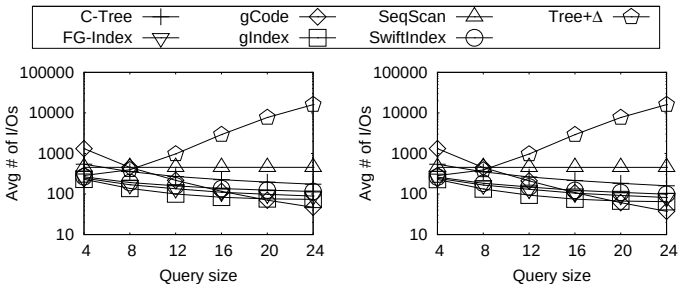


Figure 5: Average number of candidates by varying query sets (synthetic dataset).

Figure 6 shows the average number of I/Os by varying query sets when the buffer size is set to 1 MByte. gIndex shows the best performance for $Q4$ through $Q16$. FG-Index does not outperform gIndex even for $Q4$ since there exist no frequent features of size 4, and thus, the verification-free strategy is of no use in this case. gCode performs the best for $Q20$ and $Q24$ since the graph code based filtering performs very well with large query sizes for denser graph databases. Regarding Tree+Δ, its performance becomes worse with larger query sizes. This is because, queries in this dense synthetic dataset contain many cycles, and thus, the cost of mining graph features on the fly is very high (see Table 3), especially for a small buffer size. Again, with a small buffer size, we notice that, there is little change in the number of disk I/Os with hot run as shown in Figure 6(b).



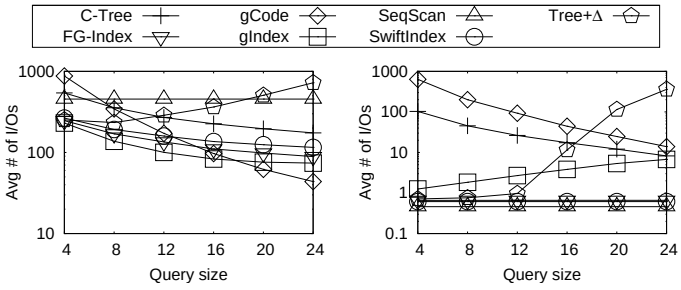
(a) Cold Run. (b) Hot Run.

Figure 6: Number of I/Os by varying query sets (synthetic dataset, buffer size = 1 MByte).

Table 3: # of graph features for each query size mined by Tree+Δ during query processing.

# of graph features		
AIDS	$Q4$	2
	$Q8$	64
	$Q12$	101
	$Q16$	148
	$Q20$	139
	$Q24$	111
Synthetic.10K.E30.D5.L50	$Q4$	165
	$Q8$	1203
	$Q12$	5146
	$Q16$	18976
	$Q20$	57306
	$Q24$	129923

Figure 7 shows the average number of I/Os by varying query sets when the buffer size is set to 10 MBytes. Since Tree+Δ reclaims graph features on the fly, a larger buffer size significantly affects its overall performance. This is why Tree+Δ performs much better when the buffer size is set to 10 MBytes. However, its mining cost during query processing time is still very expensive and performs worse than SeqScan for large query sizes ($Q20$ and $Q24$). With hot run, there is a notable change in gIndex. Since the number of index features used by FG-Index or SwiftIndex is much smaller than gIndex, it is highly likely that the pages accessed by FG-Index or SwiftIndex are reused for hot run. This result indicates that more features in the index simply do not guarantee better performance.

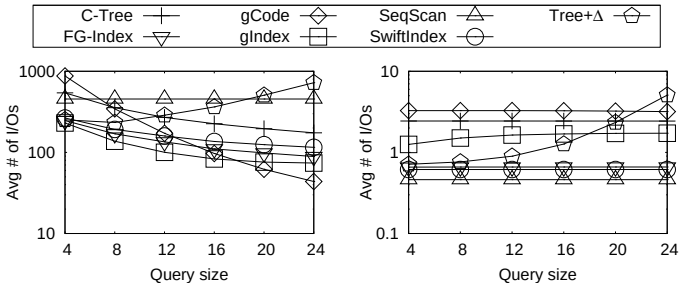


(a) Cold Run. (b) Hot Run.

Figure 7: Number of I/Os by varying query sets (synthetic dataset, buffer size = 10 MBytes).

We also perform experiments (see Figure 8) with the buffer size set to 100 MBytes (much larger than the database size). There is drastic improvement with "hot run" in Tree+Δ since the graph fea-

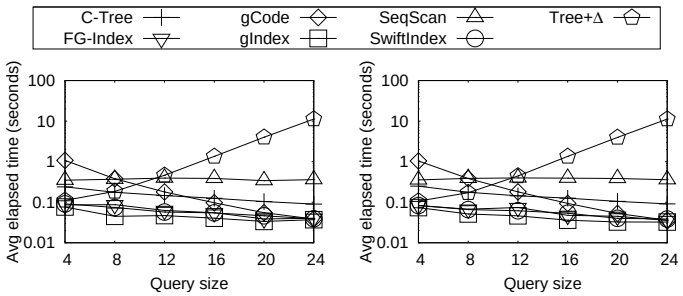
ture reclamation cost is significantly reduced with the very large buffer size. This is because all pages including newly created pages containing graph features can reside in the buffer, and we only need to flush out the newly created pages at the end of all query executions. Regarding all methods except Tree+Δ with hot run, they now show constant performance regardless of query size since most (or all) pages can reside in the buffer.



(a) Cold Run. (b) Hot Run.

Figure 8: Number of I/Os by varying query sets (synthetic dataset, buffer size = 100 MBytes).

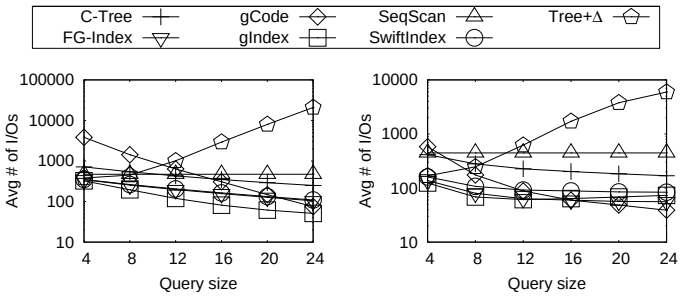
Figure 9 shows the average elapsed time by varying query sets when the buffer size is set to 1 MByte. The trends of all curves are consistent with those for the number of I/Os, since the disk I/O cost is dominant in overall performance. Regarding $Q20$ and $Q24$, although gCode performs slightly better than gIndex and FG-Index in terms of the number of I/Os, gCode is slightly slower than gIndex and FG-Index in terms of elapsed time since 1) the cost of generating vertex signatures (including the Eigenvalue computation for each vertex signature) from a large query size and 2) the object-level filtering cost are significant. gIndex shows the best performance in both cold and hot runs for a moderate dense dataset.



(a) Cold Run. (b) Hot Run.

Figure 9: Average elapsed time by varying query sets (synthetic dataset, buffer size = 1 MByte).

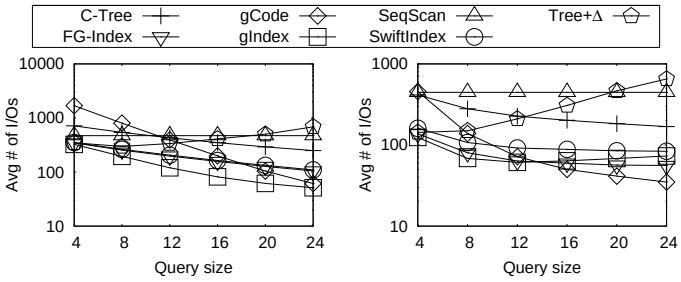
Now, we show the effect of density. For this purpose, we run experiments with two more datasets with density being set to 0.3 and 0.7, respectively. Figure 10 shows experimental results with cold run for varying density when the buffer size is set to 1 MByte. As the density increases, the performance of gIndex becomes worse than FG-Index. This is because gIndex uses a lot more (small-sized) features than FG-Index but the posting-list intersection cost exceeds the benefit of having smaller candidates. This indicates that features must be carefully selected to minimize the total cost of query execution rather than simply minimizing the number of candidates. We omit experimental results with hot run when the buffer size is set to 1 MByte, since there is very little change compared with cold run.



(a) Density=0.3 (b) Density=0.7

Figure 10: Number of I/Os by varying density (synthetic dataset, buffer size = 1 MByte, cold run).

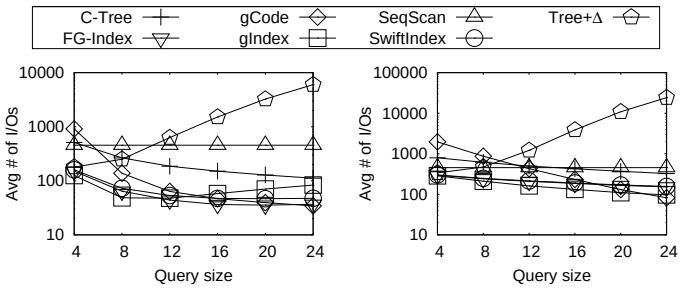
Figure 11 shows experimental results with cold run for varying density when the buffer size is set to 10 MBytes. gIndex performs the best again with density=0.3, while it is slower than other indexing methods with density=0.7. FG-Index performs better than gIndex for large query sizes with density=0.7 since it uses the smallest number of features to intersect the posting lists of those features. gCode performs the best for large query sizes with density=0.7.



(a) Density=0.3 (b) Density=0.7

Figure 11: Number of I/Os by varying density (synthetic dataset, buffer size = 10 MBytes, cold run).

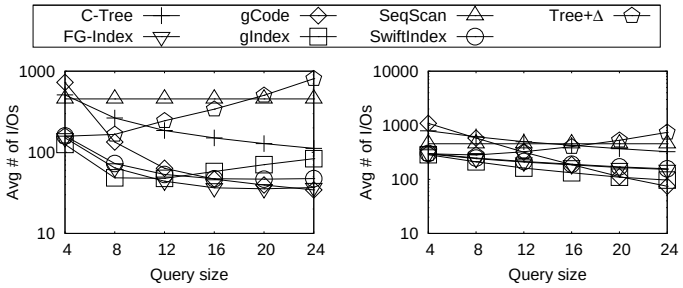
Next, we show the experimental results by varying the number of vertex/edge labels. For this purpose, we run experiments with two more datasets with the number of vertex/edge labels being set to 20 and 80, respectively. Figure 12 shows experimental result with cold run for varying density when the buffer size is set to 1 MByte. gIndex performs comparatively better for a larger number of labels since its pruning cost is relatively more effective, while it performs worse than SwiftIndex, FG-Index, and gCode for large query sizes ($\geq$ 12) when the number of labels is set to 20.



(a) # of labels=20 (b) # of labels=80

Figure 12: Number of I/Os by varying the number of labels (buffer size = 1 MByte, cold run).

Figure 13 shows experimental results with cold run for varying the number of labels when the buffer size is set to 10 MBytes. There is no drastic performance change except for Tree+Δ. With a large buffer size, its graph mining cost at run time can be significantly reduced.



(a) # of labels=20 (b) # of labels=80

Figure 13: Number of I/Os by varying the number of labels (buffer size = 10 MBytes, cold run).

4.3 Results for Large Graph Database

Note that, in our research, a real one million graph dataset is used for the first time. Since both SeqScan and C-Tree require prohibitive times to finish the experiments even with large buffer sizes, we exclude them from a large graph database. As for gCode, we can run experiments with a 1 GByte buffer and hot run; with smaller buffer sizes than 1 GByte and cold run, we are unable to finish the experiments within a week.

Database construction Cost: Table 4 shows database construction costs for PubChem. The trends of the results are similar to those of Table 1 since PubChem also contains sparse graphs. Note that, although gIndex has more features than FG-Index and Tree+Δ, its size is smaller than FG-Index and Tree+Δ. This is because frequent and non-discriminative features that have large posting lists are removed from gIndex.

Table 4: Database construction cost for the large dataset.

	Construction time (sec.)	# of features	Size (MBytes)
SeqScan	94.49	NA	370.06
C-Tree	4575.18	NA	1013.10
gIndex	4074.01	2370	662.27
FG-Index	2375.62	1837	1329.85
Tree+Δ	1613.84	1733	1261.72
gCode	575.69	NA	848.66
SwiftIndex	2112.05	753	490.24

Figure 14 shows the average number of candidates by varying query sets for PubChem. Compared with the results in Figure 1, there are drastic changes to FG-Index. Its pruning power is up to 13.09 times lower than gIndex, since FG-Index uses a strategy to select a subset of features in its index to minimize the filtering cost. However, this strategy incurs a serious performance problem in dealing with a large dataset such as PubChem, as we will see. SwiftIndex ranks between gIndex and FG-Index.

Query Processing Cost: Figure 15 shows the average number of I/Os by varying query sets for PubChem when the buffer size is set to 100 MBytes. Detailed experimental results with cold run are as follows: for $Q4$, FG-Index performs the best due to its verification-free strategy; for $Q8 \sim Q12$, gIndex performs the best since its pruning power is the best; for $Q16 \sim Q24$, either SwiftIndex or FG-Index performs the best since their posting list intersection costs are the least. For $Q24$, gIndex performs worse than SwiftIndex and FG-Index although its pruning power is the highest. This is because the cost of intersecting more posting lists leads to lower performance. SwiftIndex performs the best with hot run for larger query sizes ($Q16 \sim Q24$), since the number of its index pages accessed is the smallest, and these index pages are highly likely to be resident in memory.

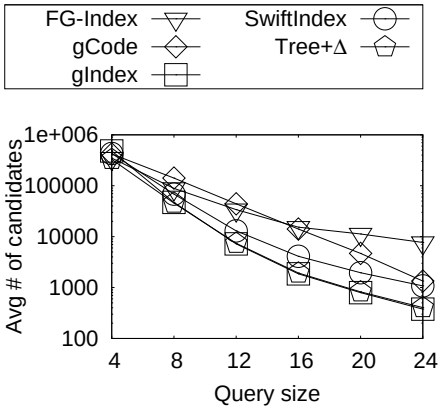
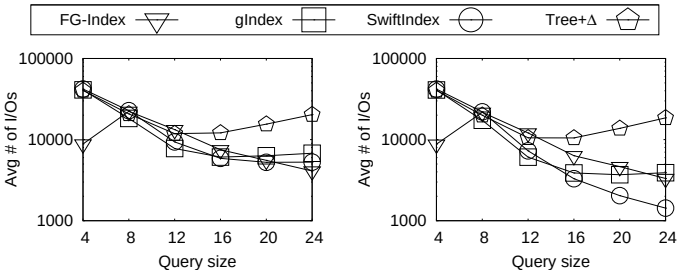


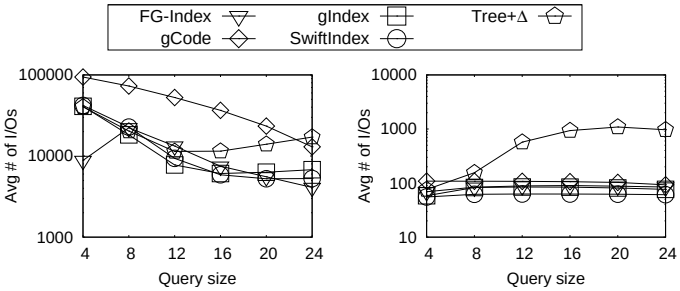
Figure 14: Average number of candidates by varying query sets (PubChem).



(a) Cold Run. (b) Hot Run.

Figure 15: Number of I/Os by varying query sets (PubChem, buffer size = 100 MBytes).

Figure 16 shows the average number of I/Os by varying query sets for PubChem when the buffer size is set to 1 GByte. Compared with Figure 15, all methods except Tree+Δ show constant performance with hot run since most pages accessed are resident in the buffer. Tree+Δ incurs about 10 times more disk I/Os due to run-time graph feature mining.

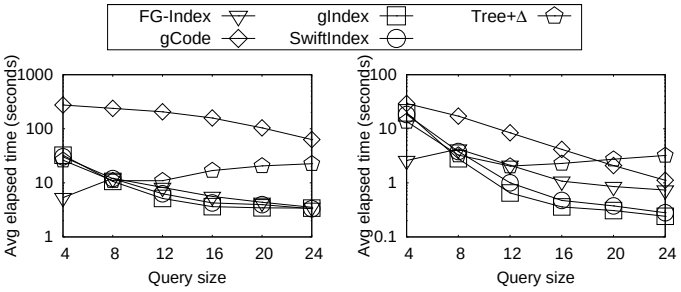


(a) Cold Run. (b) Hot Run.

Figure 16: Number of I/Os by varying query sets (PubChem, buffer size = 1000 MBytes).

Figure 17 shows the average elapsed time by varying query sets when the buffer size is set to 1 GByte. The trends of all curves except the one for gIndex are consistent with those for the number

of I/Os, since the disk I/O cost is dominant in overall performance. Although gIndex performs worse than SwiftIndex and FG-Index in the number of I/Os for large query sizes, it performs the best for all query sizes except $Q4$ due to a good combination of the lowest number of candidates and low disk I/O costs. This is because it significantly outperforms FG-Index and SwiftIndex in terms of the average number of candidates, although gIndex performs slightly worse than them in terms of disk I/Os.



(a) Cold Run. (b) Hot Run.

Figure 17: Average elapsed time by varying query sets (PubChem, buffer size = 1000 MBytes).

4.4 Summary of New Findings

- There is no single winner for all experiments.
- To our surprise, although gIndex is the oldest method among all representative graph indexing techniques we consider, it performs the best for sparse datasets (AIDS and PubChem) since its pruning power is the best, and thus, the I/O cost is usually the lowest. Here, we obtain real disk I/O costs by bypassing the OS filesystem caching. Note that all existing work uses small datasets, and these files are already in OS filesystem cache during query processing. Thus, the disk I/O cost is marginal in existing experiments. However, in reality, with large databases, the disk I/O cost is dominant in query processing.
- gIndex performs slower than FG-Index dense datasets with small labels, since it uses many ineffective features to minimize the number of candidates, and thus the number of "index pages" (i.e., posting lists) accessed is the largest.
- The database construction costs of gIndex and FG-Index are comparable to the tree feature based indexing techniques for sparse datasets (only a couple of times slower). We also confirm this fact with Gaston [4, 24], a state-of-the-art tree/graph mining tool, which supports tree/graph mining "from graphs" within the same framework.
- For a small sparse dataset using large query sizes, SwiftIndex performs the best in terms of elapsed time since it utilizes a fast feature selection algorithm called PrefixQuickSI.
- Tree+Δ performs the best with large query sizes for dense datasets in terms of the number of candidates since the previously reclaimed graph feature sets have good pruning power. However, due to the cost of graph mining on the fly, it performs very poor with small buffer sizes for dense queries and datasets.
- gCode performs the worst with small buffer sizes for sparse graph sets since 1) its index-level pruning power is much lower than other feature-based indexing techniques, and 2) index lookups over the vertex signature dictionary are very expensive with small buffer sizes. Instead, gCode performs the best in cold runs with large query sizes ($Q20$ and $Q24$) for dense graphs in terms of disk I/Os.
- C-Tree performs poor for most cases since its index-level pruning power is the lowest, and the cost of its pseudo graph isomorphism test is more expensive than VF2. It is even slower than SeqScan with small buffer sizes.

5. CONCLUSION

In this paper, we provide a comparison of disk-based graph indexing techniques by using a common framework called iGraph and full disk-based implementations rather than (full or partial) in-memory based implementations. We performed extensive experiments with small and large real datasets by varying parameter values such as the buffer size.

Although there is no single winner for all experiments, to our surprise, gIndex, the oldest method among all representative graph indexing techniques we considered, performs the best for most queries for sparse datasets (AIDS and PubChem) since its pruning power is the best, and thus, the I/O cost is usually the lowest. Tree+Δ performed the best for dense datasets in terms of the number of candidates, although it performed very poor with small buffer sizes for dense queries and datasets. For dense graphs, gCode performed the best with large query sizes ($Q20$ and $Q24$) with cold run in terms of disk I/Os, while it performed the worst with small buffer sizes for sparse datasets, since its index-level pruning power is much lower than other feature-based indexing techniques. C-Tree performs poor for most cases (even slower than SeqScan with small buffer sizes) since its index-level pruning power is the lowest, and the cost of its pseudo graph isomorphism test is more expensive than VF2.

We believe that our community will benefit greatly from our implementations and new findings. The source code of the graph indexes that we implemented will be released at http://www.igraph.or.kr/.

6. ACKNOWLEDGMENTS

This paper is based on research supported by the R&D program of MKE/KEIT (KI10033545). This paper is also supported in part by MEST/KOSEF (R11-2008-007-03003-0).

7. REFERENCES

[1] http://www.seagate.com.
[2] http://msdn.microsoft.com/en-us/library/cc644950(VS.85).aspx.
[3] Aids antiviral dataset used for gindex. http://www.xifengyan.net/software.htm.
[4] Gaston. http://www.liacs.nl/ snijssen/gaston/iccs.html.
[5] gboost. http://www.kyb.mpg.de/bs/people/nowozin/gboost/.
[6] Graphgen — a synthetic graph data generator. http://www.cse.ust.hk/graphgen/.
[7] National cancer institute. http://dtp.nci.nih.gov/.
[8] The pubchem project. pubchem.ncbi.nlm.nih.gov.
[9] Vf2 library. http://amalfi.dis.unina.it/graph/db/vflib-2.0/.
[10] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. *The VLDB Journal*, pages 169–180, 2001.
[11] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Res*, 28:235–242, 2000.

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

[13] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[14] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, pages 38–49, 2006.

[15] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, page 549, 2003.

[16] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE*, pages 286–295, 2005.

[17] E. G. M. Petrakis and C. Faloutsos. Similarity searching in medical image databases. *TKDE*, 9(3), 1997.

[18] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 2003.

[19] V. Roussev, G. G. R. III, and L. Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4(Supplement 1):105 – 113, 2007.

[20] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *PVLDB*, 1(1):364–375, 2008.

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

[22] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[23] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[24] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *PKDD*, pages 392–403, 2005.

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

[26] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.

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

[28] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

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

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

[31] L. Zou, L. Chen, J. X. Yu, and Y. Lu. A novel spectral coding in a large graph database. In *EDBT*, pages 181–192, 2008.