A SCALABLE GRAPH-COARSENING BASED INDEX FOR DYNAMIC GRAPH DATABASES

by

Akshay Kansal

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2017

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Akshay Kansal

Thesis Title: A Scalable Graph-Coarsening Based Index for Dynamic Graph Databases

Date of Final Oral Examination: 11th May 2017

The following individuals read and discussed the thesis submitted by student Akshay Kansal, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Francesca Spezzano, Ph.D.	Chair, Supervisory Committee
Vijay Dialani, Ph.D.	Member, Supervisory Committee
Amit Jain, Ph.D.	Member, Supervisory Committee
Sole Pera, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Francesca Spezzano, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

Dedicated to my parents

ACKNOWLEDGMENTS

The author wishes to express gratitude to Francesca for her valuable advice and direction with this thesis. The author also wishes to thank Vijay for introducing the author to the area of research. Finally, thanks to the authors of `one-pass` algorithm [35] for sharing the implementation of `gIndex`, `FG-Index`, and their algorithm.

ABSTRACT

Graph is a commonly used data structure for modeling complex data such as chemical molecules, images, social networks, and XML documents. This complex data is stored using a set of graphs, known as graph database $\mathcal{D}$. To speed up query answering on graph databases, indexes are commonly used. State-of-the-art graph database indexes do not adapt or scale well to dynamic graph database use; they are static, and their ability to prune possible search responses to meet user needs worsens over time as databases change and grow. Users can re-mine indexes to gain some improvement, but it is time consuming. Users must also tune numerous parameters on an ongoing basis to optimize performance and can inadvertently worsen the query response time if they do not choose parameters wisely. Recently, a `one-pass` algorithm has been developed to enhance the performance of these indexes in part by using the algorithm to update them regularly. However, there are some drawbacks, most notably the need to make updates as the query workload changes.

We propose a new index based on *graph-coarsening* to speed up query answering time in dynamic graph databases. Our index is parameter-free, query-independent, scalable, small enough to store in the main memory, and is simpler and less costly to maintain for database updates.

We conducted an extensive sets of experiments on two types of databases, i.e., chemical and social network databases, to compare our graph-coarsening based index vs. *hybrid*-indexes as follows. First, we considered no database updates or query workload changes (static graph databases) and compared the indexes according to query

answering time and index size for different *minSup* values. Second, we compared the indexes in the case of dynamic graph databases, i.e. when graphs are added to or removed from the database. Third, we compared the indexes with regard to query workload changes. Fourth, we studied the scalability of our index vs. *hybrid*-indexes.

Experimental results show that our index outperforms *hybrid*-indexes (i.e. indexes updated with `one-pass`) for query answering time in the case of social network databases, and is comparable with these indexes for frequent and infrequent queries on chemical databases. Our graph-coarsening index can be updated up to 60 times faster in comparison to `one-pass` on dynamic graph databases. Moreover, our index is independent of the query workload for index update and is up to 15 times better after *hybrid* indexes are attuned to query workload for social network databases.

This work is also published in 26th ACM International Conference on Information and Knowledge Management (CIKM) held in Singapore[18].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

minSup – Minimum support of a graph in a graph database

XML – Extensible Markup Language

NP – Non-polynomial

maxL – Maximum length or size of a graph

δ-TCFG – *delta*-Tolerance Closed Frequent subgraphs

GA – Graph Array

IGI – Inverted Graph Index

GHz – Giga-Hertz

MB – Megabyte

GB – Gigabyte

RAM – Random Access Memory

LIST OF SYMBOLS

$\mathcal{D}$	Graph database
$\mathbf{Q}$	Query graph
$\boldsymbol{G}$	Graph
$\boldsymbol{V}$	Set of vertices
$\boldsymbol{E}$	Set of Edges
$\boldsymbol{VL}$	Set of Vertex Labels
$\boldsymbol{EL}$	Set of Edge Labels
σ	Support of a graph in a graph database
δ	Frequency tolerance factor used in `FG-Index`
$\mathcal{D}_G$	Set of graphs that contain G
α	Swapping parameter used in `one-pass`
$\subseteq$	Subset or equal to
$\subset$	Subset
$\not\subseteq$	Not subset or equal to
$\supseteq$	Super-set or equal to
$\in$	Element of
ν	A function to assign labels to vertices

$\wedge$	Logical AND
$\geq$	Greater or equal
γ	Discriminative ratio
$\cap$	Set intersection
$\setminus$	Set difference
$>$	Greater than
$r(B)$	Coarsening Ratio of a set of edges B
$\omega(e)$	Edge-weighting function to compute edge count of edge e
$\rho(e)$	Edge-weighting function to compute coarsening ratio of edge e
$\mathcal{I}$	Coarsening Index
$\leq$	Less than or equal to
Co_T	Time Complexity of Coarsening
So_T	Space Complexity of Coarsening

CHAPTER 1

INTRODUCTION

Scientists and practitioners commonly use graphs to model social networks, financial transaction networks, chemical compounds, proteins, images, XML documents or other complex data, and typically store them in graph databases [2, 5, 7, 8, 14, 16, 23, 25, 27, 31, 32, 35]. A graph database $\mathcal{D}$ is simply a collection of graphs. A *dynamic* graph database is a database that changes over time. However, graph databases do not always respond quickly to a user, especially when frequently updated.

A graph database query consists of a graph Q, with the answer to Q being the set of all graphs G in the graph database $\mathcal{D}$ such that $Q \subseteq G$. A naïve user or one using a graph database that lacked an index would search the database by attempting one or more queries over the full set of graphs in the database. Of course, this approach is very inefficient, especially when the database is large. Further, testing whether a graph is contained in another one, subgraph isomorphism problem, is NP-complete [9]. Therefore, graph databases incorporate a graph database index and answer queries in two steps. First, *filter* to narrow down the search to a subset of graphs in $\mathcal{D}$, and then *verify*. The filtering step is performed by using a *graph database index* that maps a feature (or subgraph) F as a key to the IDs of database graphs that contain the requested feature as a value. The index enables users to retrieve a candidate answer set that filters out false positive candidates. After filtering, search results are verified

by completing a subgraph isomorphism test on every candidate to ensure the query is contained. Optimally, index size fits in main memory, improving query response time.

The research literature identifies many ways to generate features for indexing. The main approaches rely on *frequent subgraphs*, *paths*, or *trees*, with varying performance results (see [19] for a survey and performances comparison). However, these indexes do not adapt or scale well to *dynamic* graph database use; they are *static*, and as databases change and grow, the indexes become large and outdated, and their ability to reduce the size of a candidate answer set (*pruning power*) worsens over time [35].

Recently, Yuan et al. [35] proposed a `one-pass` algorithm to solve this problem by building a starting index with `gIndex` or `FG-Index` and performing updates based on changes to the graph database and query workload (*hybrid*-index). More specifically, this algorithm keeps the initial number of features constant, and uses the query workload to determine which index features are relevant to the current workload; features more relevant to the current search swap out those that are less relevant. However, this approach assumes that the query workload does not change rapidly. If users do not update the index when query workload starts to change, the query function may not prune a sufficient number of graphs from the search and therefore take longer to deliver results. In real-world applications, databases and queries often change frequently, which would result in the need for frequent index updates. However, attuning the index to the current query workload ignores possible new queries in the future. Consequently, the index will be unable to efficiently answer the full query range. The pruning power for queries not belonging to the query workload used to tune the index will be reduced.

Another drawback of state-of-the-art indexes is that they require users to tune

many parameters. While parameters help to reduce the search scope and improve index pruning power, they can do the opposite if not chosen wisely. Research results from several studies illustrate this by showing how their indexes outperform the competition based on the parameter values chosen [7, 19, 32, 35].

We develop a new *graph-coarsening* based index for graph databases that scales far more effectively to *dynamic* real-world graph database use. Graph coarsening [17] is used to find a more succinct representation of a graph by grouping its vertices together. It preserves basic graph information such as nodes, edges, labels, edge counts, and graph's sub-structures. Since several nodes and edges in the graph database are frequent, index size remains small and can be stored in the main memory. Also, a coarsened graph is easier to index as the information contained in it can be represented by a simple hashmap.

Therefore, we propose a new index that uses a *new* definition of graph coarsening to generate an index that is parameter-free, query-independent, scalable and small enough to be stored in main memory which also performs efficient update operations without reducing the pruning power of the index.

We conduct a detailed experimental comparison of our index vs. state-of-the-art solutions for *dynamic* graph databases on several real-world databases. Experimental results show that: (1) we outperform *hybrid*-indexes for dynamic graph databases for query answering time by up to 3 times in the case of social network databases. (2) We are scalable with a faster construction time and smaller index size. (3) We can update our index up to 60 times faster in comparison to `one-pass`. (4) Our index is independent of the query workload for index update and is up to 15 times better after *hybrid*-indexes are attuned to query workload.

The thesis is organized as follows. Chapter 2 introduces basic notions used in the

rest of the document. Related work is discussed in Chapter 3. Chapter 4 describes the graph database querying framework used for the index. Chapter 5 describes our graph-coarsening based index. Chapter 6 reports on our experiments comparing our index with state-or-the-art approaches. Finally, conclusions are drawn in Section 7. Appendix A is included to enable reproducibility of experimental results by the reader.

CHAPTER 2

PRELIMINARY DEFINITIONS

In this chapter, we introduce all the definitions we will use.

Let VL be a set of vertex labels and EL be a set of edge labels. A *labeled graph* is a 3-tuple $G = (V, E, \nu)$ where

- V is the set of vertices,
- $E \subseteq V \times V \times EL$ is a set of labeled directed edges, and
- $\nu : V \rightarrow VL$ is a function assigning labels to vertices.

We assume labeled graphs to be directed. Whenever we refer to an undirected graph, we assume each undirected edge (u, v, ℓ) *to be represented by both* (u, v, ℓ) *and* (v, u, ℓ) *directed edges.*

We define the size of a graph $G = (V, E, \nu)$ as $|E|$.

A *graph database* $\mathcal{D} = \{G_1, G_2, ..., G_n\}$ is a set of labeled graphs. Each graph $G_i \in D$, $i \in [1, n]$, has a unique identifier denoted by $id(G_i)$.

Let $G = (V, E, \nu)$ be a labeled graph and let u be a node in V. The *degree* of node u w.r.t. edge label ℓ, and destination node v's label $\nu(v)$, denoted by $deg(u, \ell, \nu(v))$, is defined as the size of the set $\{v' | (u, v', \ell) \in E \wedge \nu(v') = \nu(v)\}$.

Definition 1 (Graph Query)**.** *A graph query is defined as a graph that may or may not exist in a graph database D. A graph query can either be a* subgraph *or* supergraph

in a graph database. A subgraph *query may be contained by graphs in D while a* supergraph *query may contain graphs in D.*

We work with *subgraph* queries only in this thesis. Graph querying requires us to understand subgraph isomorphism, which is defined as follows.

Definition 2 (Subgraph Isomorphism)**.** *Let* $G = (V,\ E, \nu)$ *and* $G' = (V', E', \nu')$ *be two labeled graphs. A subgraph isomorphism is an injective function* $f : V \to V'$ *such that*

1. $\forall u \in V$, $\nu(u) = \nu'(f(u))$, *and*

2. $\forall (u, v, \ell) \in E$, $(f(u), f(v), \ell) \in E'$.

A graph G is a *subgraph* of another graph G', denoted by $G \subseteq G'$, if there exists a subgraph isomorphism from G to G'. Conversely, G' is called a supergraph of G.

The problem of deciding whether $G \subseteq G'$ is called *subgraph isomorphism problem* and it is proven to be NP-complete [9].

The following definition defines the answer to a graph query Q in a graph database.

Definition 3 (Subgraph Query Processing)**.**
Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ *and a graph query* Q*, the answer to* Q *w.r.t.* $\mathcal{D}$ *is the set*

$$ans(Q) = \{G \in \mathcal{D} \mid Q \subseteq G\}$$

As the subgraph isomorphism problem is NP-complete, usually graph databases answer subgraph queries in two steps by using the *filter+verify* approach. First, filter to narrow down the search to a subset of graphs in $\mathcal{D}$, and then verify. The filtering step is performed by using a *graph database index* that maps a *feature* (or subgraph)

F as a key to the IDs of database graphs that contain the requested feature as a value. The index enables users to retrieve a candidate answer set that filters out false positive candidates. False positive candidates are filtered out by using the following sufficient condition, called *inclusive logic*. Let F be an index feature, let $G \in D$ be a database graph, and let Q be a graph query. If $F \subseteq Q \wedge F \not\subseteq G$, then $Q \not\subseteq G$.

The *pruning power* of a graph database index is the ability to reduce the size of a candidate answer set.

Given a graph database $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ and a graph G, we denote by $\mathcal{D}_G$ the set of graphs in the database that contains G. The size of the set $\mathcal{D}_G$ is called the *support* of G and is denoted by $supp(G)$.

Definition 4 (Frequent subgraphs)**.** *Let G be a graph and $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ be a graph database. We say that G is a* frequent subgraph *if $supp(G) \geq minSup$, where $minSup$ is a given minimum support threshold.*

Definition 5 (Infrequent subgraphs)**.** *Let G be a graph and $\mathcal{D} = \{G_1, G_2, \ldots, G_n\}$ be a graph database. We say that G is an* infrequent subgraph *if $supp(G) < minSup$, where $minSup$ is a given minimum support threshold.*

CHAPTER 3

RELATED WORK

There are various ways of generating features for indexing graph databases. According to [19], the main approaches rely on (1) *simple paths* [4, 10, 13, 40], (2) *trees* [15, 26, 37], (3) *graphs* [7, 29, 30, 32, 34, 38], and (4) a combination of *trees and graphs/cycles* [20].

Among the works that use graphs as features, there are some that rely on frequent subgraphs [7, 32, 38]. Recently, Katsarou et al. [19] compared the performances of CT-index [20], GCode [40], `gIndex` [32], GRAPES [13], GraphGrepSX [4], and Tree+Δ [38] according to query processing time, index size and index construction time, and scalability. Their experimental results show that GRAPES and GraphGrepSX are the state-of-the-art best performing indexes for graph databases. However, their comparison is based on *static* graph databases only and they did not consider, in their analysis, the case of a database changing over time. When the graph database has significantly changed over time, the index becomes outdated and need to be updated. This operation is time consuming and memory intensive [19, 35].

Even if it has been shown that approaches based on frequent graphs such as `gIndex` *and* `FG-Index` *are usually an order of magnitude slower than Grapes and GraphGrepSX on static databases [19], they are, currently, the only ones that can work with dynamic graph databases as they can be updated by using the* `one-pass` *algorithm. Therefore, since the focus of our paper is to design an index suitable for dynamic*

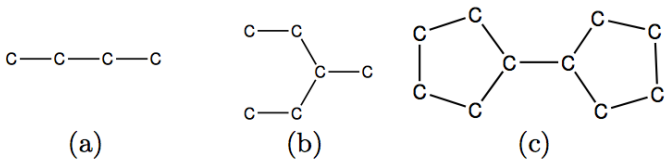


Figure 3.1: A sample graph database [32].

graph databases, we will compare our approach with hybrid-indexes only.

In the following, we give an overview of the above mentioned indexes `gIndex` and `FG-Index`, and index update algorithm, `one-pass`.

3.1 GIndex

`gIndex` [32] introduced the feature-based indexing approach by first mining a set of features or subgraphs $\mathcal{F}$ from graph database $\mathcal{D}$ and then, building a map of feature $F \in \mathcal{F}$ to the set of graph IDs $\mathcal{D}_F$ that contain F. To mine frequent subgraphs, the authors used `gSpan` [31], a depth-first search (DFS) based algorithm. `gSpan` uses minimum DFS code and lexicographic order to traverse the graph database and reduce isomorphic subgraphs generated from depth-first search traversal up to size $maxL$.

The number of fragments, or subgraphs, generated from `gSpan` varies based on minimum support, *minSup*. To ensure all queries can be answered, `gIndex` indexes size-0 (nodes) and size-1 (edges) fragments. A *size-increasing support function* is used to reduce the number of fragments generated as the size of fragments increases. There are still many fragments that do not add to the filtering power of the index and are called *redundant* fragments.

Example 1. *All the graphs in sample graph database from Figure 3.1 contain carbon-*

chains: C, C—C, C—C—C, and C—C—C—C. Fragments C—C, C—C—C, and C—C—C—C do not provide more indexing power than fragment c. Thus, they are redundant for indexing. However, the carbon ring in Figure 3.1 is a discriminative fragment as only graph (c) contains it while graphs (b) and (c) have all of its subgraphs. [32]

Therefore, a feature selection is performed to select *discriminative* fragments that add to the filtering power of the index. These features have support that is much less than the common support between their subgraphs. To identify a discriminative fragment, γ is introduced as the discriminative ratio for a fragment.

Definition 6 (Discriminative Fragment)**.** *A frequent subgraph F is* discriminative *if and only if*

$$|\bigcap_{F' \subset F, F' \in \mathcal{F}} \mathcal{D}_{F'}| \ / \ |\mathcal{D}_F| > \gamma$$

where γ is the discriminative ratio.

Index is constructed by hashing each discriminative fragment and building a key-value pair of fragment hash as the key and list of graph IDs as the value. Once the index is constructed, a query Q can be answered by generating a candidate answer set from the index features $\mathcal{F}$ by intersecting the graph IDs of features $F \in \mathcal{F}$ which are subset of Q. More formally, the candidate answer set is defined as

$$C_Q = \bigcap_{F \subseteq Q \wedge F \in \mathcal{F}} \mathcal{D}_F$$

Verification or subgraph isomorphism tests are performed on graphs in C_Q to ensure Q is contained in each graph. Verification is not required if $Q \in \mathcal{F}$.

3.2 FG-Index

The approach used by `FG-Index` [7] eliminates candidate verification for frequent subgraphs. Let us suppose that an index uses as features the set of all frequent subgraphs. If a query Q is a frequent subgraph, then verification can be avoided as all the graphs containing the query are the values corresponding to the index feature Q. However, the number of all frequent subgraphs may be big and may cause the index not to fit in the main memory. A way to compress (without loosing any information) the set of frequent graphs is to use the notion of Closed Frequent subgraphs (CFGs).

Definition 7 (Closed Frequent subgraph)**.** *A frequent subgraph G is* closed *if there does not exist another frequent subgraph G' such that $G \subset G'$ and $\mathcal{D}_G = \mathcal{D}_{G'}$.*

If the set of features indexed is the set of all CFGs , the answer set of query Q corresponds to

$$ans(Q) = \bigcap_{Q \subseteq F \wedge F \in \mathcal{F}} \mathcal{D}_F$$

As a result, time required to answer frequent queries is the same as generating candidate answer set.

The goal of `FG-Index` is to find a smart way to index all CFGs, as their size may still be too big. Thus, they rely on a relaxed notion of CFGs, namely δ-Tolerance Closed Frequent subgraphs (δ-TCFGs).

Definition 8. (δ-Tolerance Closed Frequent subgraph (δ-TCFG)). *A frequent subgraph G is δ-TCFG if and only if G is a frequent subgraph and there does not exist another frequent subgraph G' such that $G \subset G'$ and $|D_{G'}| \geq (1-\delta)|D_G|$, where $\delta \in [0,1]$ is a given frequency tolerance factor.*

Using δ-TCFGs allows us to cluster frequent graphs and generate indexing features for `FG-Index` which fit in the main memory.

To construct `FG-Index`, the set of frequent subgraphs $\mathcal{F}_\mathcal{G}$ is mined from the graph database for a given support $minSup$. Each graph in $\mathcal{F}_\mathcal{G}$ is assigned with a unique ID. Then, given a specific value of δ, the set $\mathcal{T}$ of δ-TCFGs is computed from $\mathcal{F}_\mathcal{G}$. The graphs in $\mathcal{T}$ are then sorted by increasing size, decreasing frequency, and increasing unique IDs assigned before. These δ-TCFGs are then stored in main memory in a list called Graph Array (GA) and the list index for each δ-TCFG is used as new ID for graphs in $\mathcal{T}$.

An Edge Array (EA) stores distinct edges from graphs in $\mathcal{T}$. Each edge e in EA further groups graphs in T by the size of the graphs and count of e in each graph, and maps it to id of such graphs in GA. This is called the Inverted-Graph Index (IGI). The first level of IGI is built on $\mathcal{T}$. The next level of IGI is constructed from the set of frequent supergraphs corresponding to a δ-TCFG and each IGI stored on the disk. An Egde-index is included which contains the set of infrequent distinct edges in graph database to ensure any query can be answered.

To answer a query Q, group the query by its distinct edges and get edge counts. For each edge e in Q, use frequency of e and find ids from Edge Array that have edge count greater than or equal to frequency of e. The intersection of such ids for all edges in Q will give an id. Use this id to get the δ-TCFG G from Graph Array. If $G = Q$, the answer is $\mathcal{D}_G$. Otherwise, use G's IGI stored on the disk to search for Q. No verification is required if an answer is found. If an answer is not found, it means the query is not frequent. Use Edge-index to generate a candidate answer set and obtain an answer after verification.

3.3 One-pass Algorithm

Index construction is done only once to build `gIndex` and `FG-Index`. When the graph database has significantly changed over time, the index features become outdated and need to be re-mined. This operation is time consuming and memory intensive [35]. The `one-pass` algorithm offers a way to maintain these indexes by applying updates to them. Updates are applied by measuring the goodness of index features known as *pruning power*. For a query Q, an index feature $F \subseteq Q$ *prunes* graphs in $\mathcal{D} \setminus \mathcal{D}_F$. More formally:

Definition 9 (Pruning Power and Cover)**.** *The pruning power of a feature F is the cardinality of F's pruning cover defined as $C(F, \mathcal{D}, \mathcal{Q}) = \{(G, Q) \mid G \in \mathcal{D}$ is filtered out by feature F for $Q \in \mathcal{Q}\}$, where $\mathcal{D}$ is the graph database and $\mathcal{Q}$ is a set of queries to be answered and called query workload.*

The pruning cover of the set of index features $\mathcal{F}$ is defined as $C(\mathcal{F}, \mathcal{D}, \mathcal{Q}) = \bigcup_{F \in \mathcal{F}} C(F, \mathcal{D}, \mathcal{Q})$.

When $\mathcal{D}$ and $\mathcal{Q}$ are clear from the context, $C(F, \mathcal{D}, \mathcal{Q})$ is abbreviated as $C(F)$.

Given a starting index, e.g. `gIndex` or `FG-Index`, `one-pass` computes the *loss score* of each feature in the index and the *benefit score* of each frequent subgraph which can be added to the index as a feature. The loss and benefit scores are defined in the following.

Definition 10 (Loss Score)**.** *The loss score $L(F, \mathcal{F})$ of an index feature F in the set of index features $\mathcal{F}$ is the decrease of the pruning power caused by removing F from $\mathcal{F}$, i.e. $L(F, \mathcal{F}) = |C(F) \setminus C(\mathcal{F} \setminus F)|$.*

Definition 11 (Benefit Score)**.** *The benefit score $B(H, \mathcal{F})$ of a subgraph H is the increase of the pruning caused by adding H to the set of index features $\mathcal{F}$ and it is given by $B(H, \mathcal{F}) = |C(H) \setminus C(\mathcal{F})|$.*

When the benefit of adding a frequent subgraph H outweighs the loss of an indexed feature F, H then replaces F in the index. A swapping criterion, called $swap_\alpha$, helps in this decision making by using the loss and benefit scores defined as follows.

Definition 12 (Swapping Criterion ($swap_\alpha$))**.** *An index feature F is replaced by* `one-pass` *algorithm with a frequent subgraph H if*

$$B(H, \mathcal{F}) > (1 + \alpha)L(F, \mathcal{F}) + (1 - alpha)|C(\mathcal{F})|/k$$

where k is the number of index features and $\alpha \in [0, 1]$.

In practice, the value of $|C(\mathcal{F})|/k$ is two orders of magnitude larger than $L(F, \mathcal{F})$. When $L(F, \mathcal{F}) = 0$, a frequent subgraph H with low benefit score will not be swapped in the index. Therefore, $1 - \alpha$ acts as a normalizing factor and, by setting the value of α between 0.95 and 0.995, $|C(\mathcal{F})|/k$ does not dominate the swapping criterion.

Query workload plays an important role in determining the pruning power of the index. When the pruning power drops below a threshold, an update is triggered to swap features. After the updates have been applied, the pruning power of the index is restored for the current query workload. The algorithm keeps the index size near constant since features can only be swapped in and out of the index.

`One-pass` algorithm can be combined with `gIndex` and `FG-Index` into a *hybrid* index. More specifically, the index is initially constructed by using either `gIndex` or `FG-Index` and, then, `one-pass` algorithm decides which features to swap. The *hybrid*

index outperforms the greedy solution defined in [35] to generate an initial index for `one-pass` algorithm. Therefore, we focus only on the *hybrid* index to compare with `one-pass` algorithm.

3.4 Discussion

In this section, we discuss possible drawbacks of the work described above.

Tuning the Index. Features are selected and/or updated based on criteria which utilize different parameters. `gIndex` uses size-increasing support function and discriminative ratio. `FG-Index` uses δ and minimum support. One-pass algorithm uses minimum support and α used in swapping criterion. While the parameters help reduce the size and improve pruning power of the index, they can do the opposite if not chosen wisely. Results from [7, 32, 35] show that their indexes outperform the competitors for the parameter values chosen. Further analysis showed `gIndex` outperformed others for the same parameter values only in the case of sparse graphs and `FG-Index` for only dense graphs [14]. More analysis is required case by case to be able to tune the indexes which enables `gIndex` to perform optimally for dense graphs and `FG-Index` for sparse graphs.

Index Size. In terms of size, one-pass algorithm does well at maintaining the size of the index as the number of index features remains fixed. On the other hand, `gIndex` and `FG-Index` require a lot of space. The number of features mined increases exponentially as the size of the graph database increases [35]. These features need to be stored in memory as both `gIndex` and `FG-Index` use them more than once. Indexed features also increase exponentially increasing space requirements. Also, the *hybrid* index roughly maintains its initial size as `one-pass` algorithm only allows

features to be swapped. Since `gIndex` and `FG-Index` are not scalable, reconstructing the index can be very time consuming and the size may not fit in the main memory.

Updating the Index with Current Query Workload. `one-pass` algorithm uses the query workload to determine which index features are relevant to the current workload. The features that are less relevant are swapped out by more relevant ones. This assumes that the query workload is not changing rapidly. Implying, index requires updates when query workload starts to change. Until the updates are applied to the index, it behaves sub-optimally increasing query answer time. In real-world applications, the database and queries may change frequently. This will require frequent index updates making it difficult to keep the index attuned to incoming queries for high pruning power. One of the experiments conducted by Yuan et al. [35] showed that when index was updated to perform with query set having minimum support of 0.3%, the pruning power of the index decreased significantly for query set having minimum support of 0.8%. Moreover, pruning candidates for queries never appeared before is equally important since in most real-life applications queries are not known.

CHAPTER 4

GRAPH DATABASE QUERYING FRAMEWORK

In this section, we describe our framework to query evolving graph databases. The framework takes advantage of graph coarsening technique [17] and propose a *new* definition of graph coarsening suitable for graph database indexing.

4.1 Graph Coarsening

We consider labeled graphs in our graph database. The following example shows how a chemical compound is represented by labeled graph.

Example 2. *Consider the Ethene compound showed in Figure 4.1 (a). We represent Ethene with a labeled graph* $G = (V, E, \nu)$ *(see Figure 4.1 (b)), where* $VL = \{C, H\}$, $EL = \{s, d\}$[1], $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, 3, s), (2, 3, s), (5, 4, s), (6, 4, s), (3, 4, d)\}$, $\nu(1) = \nu(2) = \nu(5) = \nu(6) = H$, *and* $\nu(3) = \nu(4) = C$.

Graph-coarsening [17] consists of finding a succinct representation of the graph that also preserves the original graph structure. Usually, a graph G is coarsened by merging together *similar* vertices into a unique super-node and by assigning edges between super-nodes as follows. If there was an edge between two vertices u and v in G and u has been merged into a new vertex u' while v has been merged into a new vertex v', then the coarsened graph G' will contain an edge between u' and v'.

[1]Edge label s (resp. d) denotes a single (resp. double) bond.

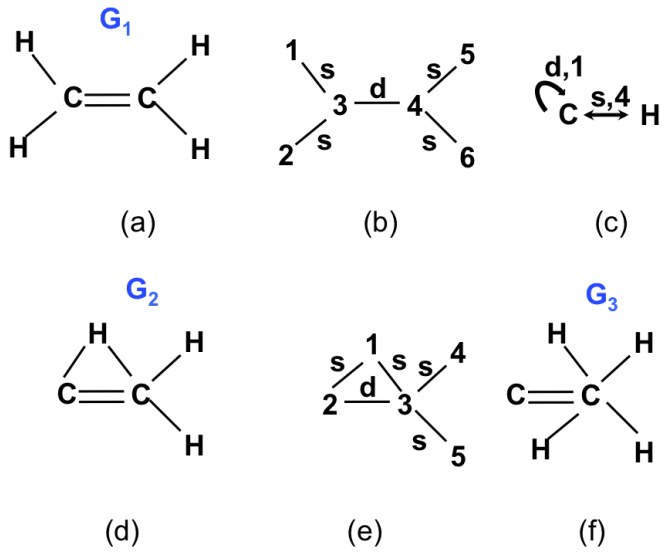


Figure 4.1: (a) Ethene compound G_1, (b) Ethene's representation as a labeled graph, (c) naïve labeled graph coarsening, (d) and (f) two other compounds G_2 and G_3, and (e) G_2's representation as a labeled graph.

Moreover, usually, a weight is added to each edge in the coarsened graph to keep track of the number of edges in the original graph that collapse in a unique edge in the coarsened one. Thus, coarsening is mapping labeled graphs to edge-labeled weighted graphs.

Example 3. *Consider the labeled graph representation of the Ethene compound* $G = (V, E, \nu)$ *from Example 2. Suppose we merge together nodes having the same node label. A possible coarsening of* G *is shown in Figure 4.1 (c) and is given by the edge-labeled weighted graph* $G' = (V', E', \omega)$ *where* $V' = \{C, H\}$ *is the set of nodes,* $E' = \{(H, C, s), (C, C, d)\}$ *is the set of labeled edges, and* ω *is the edge weighting function. Since edges* $(1, 3, s), (2, 3, s), (5, 4, s)$, *and* $(6, 4, s)$ *from* G *collapse into the edge* (H, C, s) *in* G', *we have that* $\omega((H, C, s)) = 4$, *while* $\omega((C, C, d)) = 1$.

However, synthesizing a set of edges by just using the number of edges that collapse, is not so meaningful to express the graph structure. Consider, for instance,

the graphs in Figure 4.1 (d) and (f). The coarsening of these two graphs results to be the same of the one of Ethene in Figure 4.1 (a) as they all have four edges between nodes C and H.

Therefore, in order to better preserve the structure of the original graph in its coarsened version, we introduce the concept of *coarsening ratio*.

Definition 13 (Coarsening ratio)**.** *Let $G = (V, E, \nu)$ be a labeled graph and let $B = \{(u_1, v_1, \ell), ..., (u_n, v_n, \ell)\}$ be the subset of all edges in E such that $\nu(u_1) = ... = \nu(u_n)$ and $\nu(v_1) = ... = \nu(v_n)$. The coarsening ratio $r(B)$ of the set of edges B is defined as*

$$r(B) = \frac{1}{\max(deg(u_1, \ell, \nu(v_1)), ..., deg(u_n, \ell, \nu(v_1)))}$$

The coarsening ratio represents the biggest substructure in the original graph involving nodes labeled as $\nu(u_1)$ and $\nu(v_1)$, and edge label ℓ.

Example 4. *Consider the labeled graph G_1 representing Ethene compound in Figure 4.1 (b). Edges in the set $E_b = \{(3, 1, s), (3, 2, s), (4, 5, s), (4, 6, s)\}$ are all edges representing a single bond between Carbon C and Hydrogen H. We have that $deg(3, s, H) = 2$ and $deg(4, s, H) = 2$, then, the coarsening ratio for E_b, representing the directed edge (C, H, s), is $r(E_b) = \frac{1}{\max(2,2)} = 0.5$. On the other hand, $E'_b = \{(1, 3, s), (2, 3, s), (5, 4, s), (6, 4, s)\}$ is the set of all the edges having a single bond between Hydrogen H and Carbon C. As $deg(1, s, C)) = deg(2, s, C) = deg(5, s, C) = deg(6, s, C) = 1$, the coarsening ratio for E'_b, representing the directed edge (H, C, s), is $r(E'_b) = \frac{1}{\max(1,1,1,1)} = 1$.*

The set of all edges representing a double bond from C to C is $E''_b = \{(3, 4, d), (4, 3, d)\}$ and has a coarsening ratio of $r(E''_b) = 1$ as both nodes 3 and 4 have a degree of 1.

Consider now the compound G_2 in Figure 4.1 (d). It can be represented as the labeled graph in Figure 4.1 (e). In this case, the set of edges $E_e = \{(2,1,s), (3,1,s), (3,4,s), (3,5,6,s)\}$, representing a single bond from C to H, has coarsening ratio $r(E_e) = \frac{1}{\max(1,3)} = 0.33$ as $deg(2,s,H) = 1$ and $deg(3,s,H) = 3$. For the set of edges representing the single bond from H to C, the coarsening ration is 0.5 in G_2.

For the compound in Figure 4.1 (f), the coarsening for the set of edges representing the single bond from C to H is 0.25, while from H to C it is 1.

As we can see, the coarsening ratio allows to distinguish among different graph structures.

It is worth noting that the coarsening ratio is always a value in the interval $(0,1] \cup \{\infty\}$.

In our framework, we coarsen labeled graphs to edge-labeled double-weighted[2] graphs. Specifically, the edge weighting function ω keeps track of the number of collapsing edges, while the edge weighting function ρ assigns the coarsening ratio to each edge in the coarsened graph.

Definition 14 (Coarsening)**.** *Let $G = (V, E, \nu)$ be a labeled graph. A coarsening of G is an edge-labeled double-weighted graph $G' = (V', E', \omega, \rho)$ such that:*

- *$V' = \{\nu(u) | u \in V\}$, i.e. we merge in a unique vertex all vertices in G having the same vertex label (as a consequence, we have $|V'| \leq |V|$),*
- *$E' = \{(\nu(u), \nu(v), ep) | (u, v, ep) \in E\}$,*
- *$\omega : E' \to \mathbb{Z}_{\geq 0}$ is an edge weighting function s.t. for each edge $e = (u', v', ep) \in E'$, $\omega(e) = |A(u', v', ep)|$, where $A(u', v', ep) = \{(u, v, ep) \in E | u \in \nu^{-1}(u') \wedge v \in \nu^{-1}(v')\}$, and*

[2]We have two weighting functions for the edges.

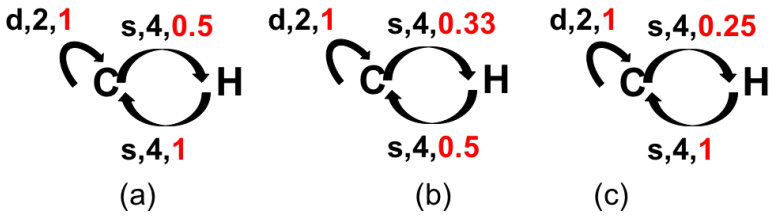


Figure 4.2: (a) (resp. (b), (c)) coarsening of graph G_1 (resp. G_2, G_3) from Figure 4.1 according to Definition 14.

- $\rho : E' \to (0, 1]$ *is an edge weighting function s.t. for each edge* $e = (u', v', ep) \in E'$, $\rho(e) = r(A(u', v', ep))$.

Example 5. *Consider graphs* G_1, G_2, *and* G_3 *from Figure 4.1. The coarsening of labeled graphs representing* G_1, G_2, *and* G_3 *is shown in Figure 4.2 (a),(b), and (c), respectively.*

The main differences between Definition 14 and the graph coarsening proposed in [17] are the introduction of the coarsening ratio and the absence of a contraction factor regulating the number of nodes in the coarsening (hence we are parameter-free).

4.1.1 Algorithm

Algorithm 1 shows how to coarsen graphs. The algorithm takes a graph G as input and returns the coarsening C of G. We use a hashmap to store coarsening for efficiency. For each node in G, lines 6-9 compute the degree of outgoing edges of a node w.r.t edge label and destination node label. Coarsening ratio and edge counts are computed on line 17 and 18 respectively. For completeness, line 19 adds coarsening of nodes.

Algorithm 1 Graph Coarsening

Input: Graph $G = (V, E, \nu)$
Output: Coarsening C

```
1: Let C be a hashmap
2: Let deg be a hashmap
3: for u ∈ V do
4:    if ν(u) ∉ C then
5:       Let C[ν(u)] be a hashmap
6:    for e = (u, v, ℓ) ∈ {e' ∈ E | e' = (u, w, ℓ')} do
7:       if (u, ℓ, ν(v)) ∉ deg then
8:          deg[(u, ℓ, ν(v))] ← 0
9:       Increment deg[(u, ℓ, ν(v))] by 1
10:   C[ν(u)][_][_] ← (0, ∞) (for completeness)
11: for key = (u, ℓ, ν(v)) ∈ deg do
12:   if ν(v) ∉ C[ν(u)] then
13:      Let C[ν(u)][ν(v)] be a hashmap
14:   if ℓ ∉ C[ν(u)][ν(v)] then
15:      C[ν(u)][ν(v)][ℓ] ← (0, ∞)
16:   r = 1/deg[key]
17:   if r < C[ν(u)][ν(v)][ℓ][1] then
18:      C[ν(u)][ν(v)][ℓ][1] ← r (coarsening ratio)
19:   Increment C[ν(u)][ν(v)][ℓ][0] by deg[key] (edge count)
```

4.1.2 Complexity Analysis

The time complexity Co_T of coarsening is $O(|V| + |E|)$. The coarsening algorithm iterates through each node and edge only once. The space complexity So_T of coarsening is $O(|V| + |E|)$ as most space will be utilized when a graph contains edges with distinct labels for nodes and edges.

4.1.3 Coarsening and Graph Containment

Graph coarsening can be used to prune database graphs that do not contain a query. The following proposition states that we can use coarsening ratio (function ρ) and

edge count (function ω) to give sufficient conditions to determine if a labeled graph is not contained in another one.

Proposition 1 (Graph containment)**.** *Let G_1 and G_2 be two labeled graphs and let $G'_1 = (V_1, E_1, \omega_1, \rho_1)$ (resp. $G'_2 = (V_2, E_2, \omega_2, \rho_2)$) be the coarsening of G_1 (resp. G_2). Let $e_1 = (u, v, \ell) \in E_1$ and $e_2 = (u, v, \ell) \in E_2$ be two coarsened edges. If $\omega_1(e_1) > \omega_2(e_2)$ or $\rho_1(e_1) < \rho_2(e_2)$ then $G_1 \not\subseteq G_2$.*

Proof by contradiction. Let us assume that $G_1 \subseteq G_2$ and let $H_1 \subseteq G_1$ (resp. $H_2 \subseteq G_2$) be the subgraph of G_1 (resp. G_2) such that the coarsening of H_1 (resp. H_2) is equal to e_1 (resp. e_2). Since $G_1 \subseteq G_2$, then, by definition of coarsening, also $H_1 \subseteq H_2$. It follows that, as $H_1 \subseteq H_2$, we must have that $\omega_1(e_1) \leq \omega_2(e_2)$ and $\rho_1(e_1) \geq \rho_2(e_2)$, which contradicts the hypothesis. □

Example 6. *Consider graphs G_1, G_2, and G_3 from Figure 4.1 whose coarsening is shown in Figure 4.2. According to Proposition 1 we can say that $G_2 \not\subseteq G_1$, $G_2 \not\subseteq G_3$, $G_3 \not\subseteq G_1$, and $G_3 \not\subseteq G_2$.*

Consider, for instance, the case $G_2 \not\subseteq_? G_1$. Let $e_1 = (C, H, s)$ be the edge from C to H in the coarsening $G'_1 = (V'_1, E'_1, \omega_1, \rho_1)$ of G_1 (shown in Figure 4.2 (a)) and let $e_2 = (C, H, s)$ be the same edge but in the coarsening $G'_2 = (V'_2, E'_2, \omega_2, \rho_2)$ of G_2 (shown in Figure 4.2 (b)). We have that $\rho_2(e_2) > \rho_1(e_1)$ and then, G_2 cannot be contained in G_1. The motivation is that G_2 contains the features $C - H_3$, i.e. a Carbon atom connected with three Hydrogen atoms, that in not present in G_1 where the biggest substructure involving C and H is $C - H_2$, i.e. a Carbon atom connected with two Hydrogen atoms.

Proposition 1 just provides sufficient conditions. We need, in fact, the subgraph isomorphism test to say that $G_1 \not\subseteq G_2$ and $G_1 \not\subseteq G_3$. However, we can simply say

that $G_1 \not\subseteq G_2$ because G_1 has four different nodes labeled as H, while G_2 contains only three of them. Therefore, we introduce this additional candidate pruning step in candidate set generation (see Section 5.1).

CHAPTER 5

GRAPH-COARSENING BASED INDEX

The index we propose uses coarsened edges as features instead of frequent subgraphs.

A coarsened graph is easier to index as the information contained in it can be represented using a hashmap. Graphs in a coarsened graph database can be grouped by distinct edges, edge weights, and coarsening ratio as the key while graph IDs are stored as the value. Given a coarsened graph $G = (V, E, \omega, \rho)$ and an edge $e = (u, v, \ell) \in E$, the *index key* for the edge e is defined as the 5-tuple $key(e) = \langle u, v, \ell, \omega(e), \rho(e) \rangle$. For a graph database $\mathcal{D}$, the value, denoted by $value(e)$, for the key $key(e)$ is given by the set of IDs of graphs in $\mathcal{D}$ whose coarsening contains the edge e. As we are dealing with edge labeled double-weighted graphs, we refer to the following definition of subgraph isomorphism. Let $G = (V, E, \omega, \rho)$ and $G' = (V', E', \omega', \rho')$ be two edge labeled double-weighted graphs. A subgraph isomorphism is an injective function $f : V \to V'$ such that (1) $\forall e = (u, v, \ell) \in E$, $e' = (f(u), f(v), \ell) \in E'$, (2) $\omega(e) = \omega'(e')$, and $\rho(e) = \rho'(e')$.

In addition, we also index each single vertex v appearing in a coarsened graph with key $key(v)$ equal to the 5-tuple $(v, \_, \_, 0, \infty)$ and value $value(v)$ equal to the set of IDs of graphs in $\mathcal{D}$ whose coarsening contains the vertex v.

Our *graph-coarsening based index* $\mathcal{I}$ is then constructed in three steps.

For each graph $G \in$ graph database $\mathcal{D}$,

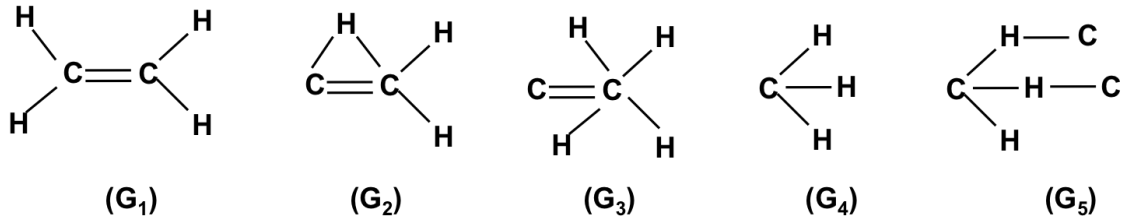


Figure 5.1: A sample graph database $\mathcal{D}$.

1. coarsen G to $G' = (V', E', \omega, \rho)$,

2. for all edges $e \in E'$, compute $key(e)$;
   if $\mathcal{I}$ contains $key(e)$, then add $id(G)$ to $value(key(e))$
   otherwise, insert $key(e)$ in index $\mathcal{I}$ with value $value(key(e)) = \{id(G)\}$;

3. for all vertices $v \in V'$, compute $key(v)$;
   if $\mathcal{I}$ contains $key(v)$, then add $id(G)$ to $value(key(v))$
   otherwise, insert $key(v)$ in index $\mathcal{I}$ with value $value(key(v)) = \{id(G)\}$;

The cost to build our index is $Co_T.|\mathcal{D}|$ or, $O((|V| + |E|) \times |\mathcal{D}|)$. And, the cost to store our index is $So_T.|\mathcal{D}|$ or, $O((|V| + |E|) \times |\mathcal{D}|)$.

Example 7. *Consider the graph database in Figure 5.1. The corresponding graph-coarsening based index $\mathcal{I}$ is shown in Figure 5.2.*

It is worth noting that our proposed index is *parameter-free*, as we are not using any parameter to coarsen a graph. Moreover, the size of the index is linear in the size of the graph database, whereas *hybrid*-indexes index frequent fragments whose size is exponential in the one of the database.

In addition, since several nodes and edges in a graph database are frequent (consider, for instance, the case of chemical compounds databases or social networks where nodes having the same property can be merged together), the size of our

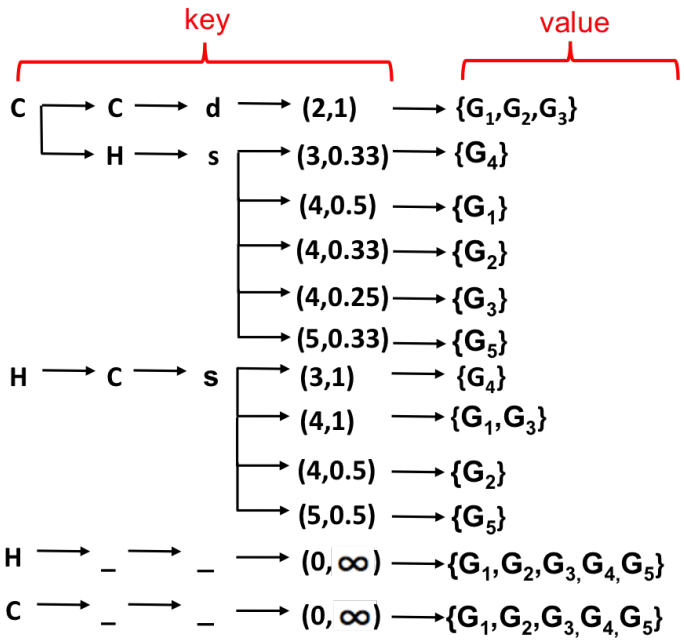


Figure 5.2: Graph-coarsening based index $\mathcal{I}$ for graph database $\mathcal{D}$ in Figure 5.1. The pair (c, r) denotes the collapsing edge count c and the coarsening ratio r.

graph-coarsening index remains smaller than the database size and can be stored in the main memory. This also enhances the scalability of our index.

5.1 Query Processing

After the graph-coarsening based index $\mathcal{I}$ is constructed for graph database $\mathcal{D}$, query processing comprises of (1) generating candidate answer set and (2) candidate verification.

Candidate answer set C_Q for query $Q = (V, E, \nu)$ is generated as follows:

1. Coarsen the graph query Q to $Q' = (V', E', \omega, \rho)$;

2. Let $C_Q = \mathcal{D}$. For each edge $e(u, v, \ell) \in Q'$, intersect C_Q with all value sets retrieved from the index and corresponding to the set of keys $\mathcal{K} = \{(u, v, \ell, p, q) \in$

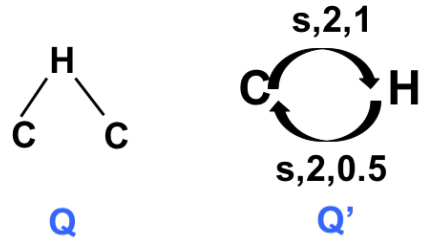


Figure 5.3: A sample graph query Q (left) and its coarsening Q' (right).

$\mathcal{I} \mid p \geq \omega(e) \wedge q \leq \rho(e)\}$:[1]

$$C_Q = C_Q \bigcap_{e(u,v,\ell) \in E'} \left(\bigcap_{k \in \mathcal{K}} value(k) \right)$$

3. Remove from C_Q all graphs $G = (V_G, E_G, \nu_G)$ such that there exists a node $v \in V_G$ such that

$$|\{u \in V_G | \nu_G(u) = \nu_G(v)\}| < |\{u' \in V | \nu(u') = \nu_G(v)\}|$$

i.e. the graph G does not have enough nodes with label $\nu_G(v)$ to contain the query Q.

Example 8. *Consider the graph database $\mathcal{D}$ from Figure 5.1, the corresponding graph-coarsening based index $\mathcal{I}$ shown in Figure 5.2, and the query Q and its coarsening $Q' = (V', E', \omega, \rho)$ from Figure 5.3. E' contains two edges: $e_1 = (C, H, s)$ and $e_2 = (H, C, s)$. By considering edge e_1, we have to retrieve from the index $\mathcal{I}$ and intersect all value sets corresponding to the set of keys $\mathcal{K}_1 = \{(C, H, s, p, q) \in \mathcal{I} \mid p \geq 2 \wedge q \leq 1\}$. Then, we have*

[1] In case Q' is a graph containing a single vertex v and no edges, $C_Q = C_Q \cap value((v, \_, \_, \_, 0, \infty))$.

$$C'_Q = \bigcap_{k \in \mathcal{K}_1} value(k) = \{G_1, G_2, G_3, G_4, G_5\}$$

The set of database graphs that may contain edge e_2 is given, instead, by the intersection of all value sets corresponding to the set of keys $\mathcal{K}_2 = \{(H, C, s, p, q) \in \mathcal{I} \mid p \geq 2 \wedge q \leq 0.5\}$, i.e.

$$C''_Q = \bigcap_{k \in \mathcal{K}_2} value(k) = \{G_2, G_5\}$$

Finally, the candidate answer set C_Q for query Q is

$$C_Q = C'_Q \cap C''_Q = \{G_2, G_5\}$$

The cost of query answering is,

$$Co_{QueryAnswering} = Co_{Coarsening} + Co_{CandidateGeneration} + C_Q * T_{sub}$$

$$= Co_T + O(|V_Q| + |E_Q|) + C_Q * T_{sub}$$

$$= O(|V_Q| + |E_Q|) + C_Q * T_{sub}$$

where, T_{sub} is the time for subgraph isomorphism test. When no graph is pruned by the index (worst case), $C_Q = |\mathcal{D}|$.

5.2 Index Update

The index we are proposing is easy to update as graph database changes. Updates are required only when graphs are added to or removed from the database since the

index is query-independent. Conversely, `one-pass` algorithm needs to update the index even when the query workload is changed.

Adding. When a new graph G is added to graph database, we follow the steps of index construction to add the new graph ID to the graph-coarsening based index $\mathcal{I}$. More specifically, we coarsen G to G' and, for each edge e in G', we add the pair $\langle key(e), \{ID(G)\}\rangle$ to index $\mathcal{I}$ if $key(e)$ is not present in $\mathcal{I}$, otherwise we add $ID(G)$ to $value(key(e))$. A similar index update is done for index entries corresponding to vertices in G'.

Removing. When a graph G is removed from graph database, its graph ID is removed from all sets $value(key(e))$ where e is an edge in coarsened version G' of G. Similarly for entries corresponding to vertices in G'. The entry for an edge (and/or vertex) is removed from the index if only graph G contained it.

It is worth noting that the cost of updating our index is constant in the size of the graph database, while in the case of *hybrid*-indexes is inversely correlated to the "quality" of the index before the update [35].

5.3 Other Indexing Approaches

In this section, we describe the additional approaches we had tried. We compared performance of the approach described previously with these approaches before deciding to keep it.

5.3.1 Graph Reduction Using Query Before Verification

Once we have a candidate answer set C_Q we have to verify whether each graph $G \in C_Q$ actually contains the query or not via subgraph isomorphism test. In order to reduce

the size of subgraph isomorphism test $Q \subseteq_? G$, we reduce the size of graph G by removing from it all nodes v (and edges involving those nodes) such that there does not exist a node v' in Q whose label is the same of v's label.

We ran an experiment on `eMolecules` chemical database to assess improvements of our new approach on different query types, i.e., frequent, infrequent and random (see Section 6.1 for definition). Table 5.1 shows the verification time results for the experiment conducted on chemical database. This approach improved the verification time from 171 ms to 138 ms, for frequent queries, but the reduction time was large enough (38 ms) that there was no noticeable gain in total time taken. Therefore, we decided to keep our original approach.

Table 5.1: Runtime Comparison with and without Graph Reduction Using Query on `eMolecules` for different query types. Rows with Yes for Graph Reduction in the table use the new approach.

Query Type	Graph Reduction	Reduction Time (ms)	Verify Time (ms)	Total Time (ms)
Frequent	Yes	38	131	169
	No	0	170	170
Infrequent	Yes	22	89	111
	No	0	109	109
Random	Yes	1	5	6
	No	0	5	5

5.3.2 Subgraphs And Their Counts

Consider Figure 4.1, the original index cannot answer that $G_1 \not\subseteq G_2$ and $G_1 \not\subseteq G_3$. The reason is that there is no information in G_1 that is not contained in G_2. Let us consider coarsened edge $C - H$ in Figure 4.2 (a), (b) that represents coarsening of G_1 and G_2 respectively. The edge count is the same between coarsened G_1 and G_2 and

the coarsening ratio is also greater for $C - H$ in coarsened G_1, when G_1 is the query, causing G_2 to not be eliminated.

When we reconsider graphs G_1 and G_2, there is only one subgraph with one $C-H_2$ in G_2 but two of such subgraphs in G_1. If we add the count of these subgraphs along with the coarsening ratio of each subgraph, there is only one subgraph $C - H_2$ in G_2 which allows for possibility of only one graph that is one $C - H_2$ in G_1. G_2 can be eliminated with this approach. Same applies for G_3 as there is only one substructure with one C and four H.

The above approach adds more time to candidate generation as we now need to count substructures and reduce the count when a substructure matches. It is easy to reduce the count in the above example, but it becomes difficult when there is more than one substructure connecting source label to destination label. We need to eliminate the largest substructures before smaller substructures can be eliminated due to graph containment problem. Table 5.2 shows run time comparison for experiment conducted on `BlogCatalog3` social network database. The verification time improves by about 20% for frequent queries but there is no gain for infrequent or random queries. However, the candidate generation time or filter time increases by 5 times making the index less desirable. The space usage for the new approach nearly doubled from 4.8 MB to 8.3 MB as well. Therefore, we decided to keep the original index.

Table 5.2: Query Processing Time Comparison with and without subgraph Count. Rows with Yes for subgraph Count use the new approach.

Query Type	subgraph Count	Filter (ms)	Verify (ms)	Total Time (ms)
Frequent	Yes	27	4	31
	No	5	5	9
Infrequent	Yes	27	2	29
	No	5	2	7
Random	Yes	27	3	30
	No	5	3	8

CHAPTER 6

EXPERIMENTS AND RESULTS

As reported in Chapter 3, there are many indexes defined for graph databases. However, the majority of them do not adapt in case of database changes.

As the main goal of our work is to provide a solution for dynamic graph databases, in this section we compare our graph-coarsening index and the state-of-the-art indexes working for dynamic *graph databases, i.e.* hybrid-*indexes.*

6.1 Experiment Setup

We used the implementation of *hybrid*-indexes developed for the paper [35] and kindly provided by the authors. Since their implementation was in Java, we used the same language to implement our index. The verification algorithm used is `VF2` [24].

We ran all the experiments on a 2.8 GHz Intel Xeon E5-1410 processor with 16 GB of RAM and CentOS 7.2.1511. We used two types of datasets in our experiments. First, to be consistent with previous work [7, 19, 32, 35], we used a chemical database for index comparison. We chose `eMolecules` [11] database, which contains 458,835 graphs with mean graph density (MGD) of 0.13. Second, we tested these graph database indexes on the domain of social networks for the first time. Social networks are relevant because they are dynamic graphs and can be used to study social relations. Crawling an entire online social network (OSN) is hard these days because

of the huge size of the network and restrictions on the API. Often, information is gathered for individual nodes, together with their neighbors and neighbors' neighbors, i.e. ego networks, or with all neighbors up to a fixed distance. It is clear that, in this case, the retrieved data is in the form of a graph database. Also, OSNs are dynamic as they continuously chance over time. Since a social network is a single graph, we computed ego networks from nodes in the social network to generate a graph database. We use three social networks for comparison, namely `DBLP` [21], `BlogCatalog3` [36], and `Slashdot` [21]. `Slashdot` is a signed network (i.e. it contains two types of relationships, namely friend and foe relationship) while `BlogCatalog3` and `DBLP` are unsigned. The three social networks have 317,080 (MGD=0.77), 10,312 (MGD=0.89) and 77,357 egos (MGD=0.61) respectively. We labeled nodes in the social networks as follows. First, we computed the page-rank of nodes in a network by using the SNAP library [22]; second, we assigned 10 node labels, 0-9, to the nodes of each social network (there are 7 node labels in `eMolecules`). To assign a label, we computed the page rank of each node, and uniformly distributed the page ranks into 10 buckets between the upper and lower bound of the computed page ranks.

The chemical database is indexed with minimum supports of 10%, 20%, 30% and 40% of the database size. For a fair comparison, we use 2%, 3% and 4% for social networks as there were not many subgraphs at minimum support of 10% or higher. We set the $\delta = 0.1$ to compute δ-TCFGs for `FG-Index`.

We considered three types of queries: *frequent*, *infrequent*, and *random*. *Frequent* and *infrequent* queries are defined in Chapter 2. Random queries are graphs or subgraphs randomly chosen from the graph database. Previous work compared graph database indexes by using frequent queries only. We give a better picture of how indexes behave in the case of infrequent and random queries, also. To generate these

three types of queries we proceeded as follows. We first mined frequent subgraphs with a low minimum support of 1%. To generate frequent queries, we then selected mined subgraphs that have minimum support greater than or equal to the minimum support used to build the index. To generate infrequent queries, we selected subgraphs that have a minimum support less than the one of the index (and greater than 1%). To generate query sets, we randomly sampled a subset of frequent and infrequent queries respectively. We used database graphs to select random queries and generate query sets. Each query set contained 1,000 queries. The size of each graph database used in the experiments is 30,000 graphs. If the number of graphs in the database was less than 30,000 graphs, we randomly chose graphs from the database till we reached 30,000 graphs.

The index construction time for other indexes on ego networks was daunting due to the size of the ego networks. Therefore, we used a maximum graph size of 15 for `DBLP` and `BlogCatalog3` ego networks. The two databases contained sparse graphs. However, ego networks in `Slashdot` were dense and to construct indexes in a reasonable amount of time we changed the maximum size constraint to 10 edges. The graph database for `Slashdot` was still dense.

We tested the indexes for 10 sets of 1,000 queries for each query type and averaged the results for each query set and computed the mean of all query sets. If the number of queries was less than 1,000, queries were randomly repeated to reach desired size. The frequent and infrequent queries were mined with gSpan [31] using $maxL$=10. The index features were also mined with $maxL$=10.

We compared query answering time, index size and/or index construction time for different *minSup* values in each experiment. When considering index size, we also considered the portion of index that was stored on disk as in case of *hybrid-*

`FG-Index`. Since a majority of time is spent on candidate answer set verification during query processing, the size of the candidate answer set is important to speed-up query answering. Therefore, a faster query answering time suggests a candidate set is closer to the actual answer. We define this closeness as *goodness* of candidate answer set.

We conducted four sets of experiments to compare our graph-coarsening based index vs. *hybrid*-indexes. First, we considered no database updates or query workload changes (static graph databases) and compared the indexes according to query answering time and index size for different *minSup* values. Results are reported in Section 6.2. Second, we compared the indexes in the case of dynamic graph databases, i.e. when graphs are added to or removed from the database. Results are reported in Section 6.3. Third, we compared the indexes with regard to query workload changes. Results are shown in Section 6.4. Finally, we studied the scalability of our index vs. *hybrid*-indexes. Results are shown in Section 6.5.

6.2 Our index vs. hybrid-indexes

We compare our graph-coarsening index with *hybrid*-indexes. We built *hybrid*-indexes for different *minSup* values. Our index was constructed only once as we are not dependent on minimum support. The common trend we observe across all databases is that the run time for query answering of our graph-coarsening index is, by definition of the index, independent of the minimum support chosen to build other indexes or to generate the query sets, and therefore near constant, while *hybrid*-indexes start to perform worse as the *minSup* increases because a lesser number of features is indexed. Results are reported in the following subsections.

Table 6.1: Count of index features on Social Network Databases for Standalone Index comparison

Social Network Database	Index	*minSup*		
		2%	3%	4%
BlogCatalog3	*hybrid*-gIndex	9638	9162	9038
	hybrid-FG-Index	8246	2746	1118
	coarsening	907	907	907
DBLP	*hybrid*-gIndex	1583	1487	1453
	hybrid-FG-Index	936	497	308
	coarsening	446	446	446
Slashdot	*hybrid*-gIndex	1439	1436	1435
	hybrid-FG-Index	64	44	33
	coarsening	263	263	263

6.2.1 Social Network Databases

The run time on social network databases of our graph-coarsening index vs. *hybrid*-indexes for different minimum supports is shown in Figures 6.1, 6.2 and 6.3 for answering frequent, infrequent, and random queries, respectively. Table 6.1 shows the number of features indexed for different *minSup* values by database. The number of features for FG-Index are reported for what was stored in main memory and on disk.

In the case of BlogCatalog3 and DBLP databases, our index outperforms other indexes for frequent and random queries, while, for infrequent queries, the results are comparable with *hybrid*-gIndex, but better than *hybrid*-FG-Index. The indexes perform better overall in case of BlogCatalog3 than DBLP because the number of features is higher providing more chances to eliminate candidates, the *pruning power*

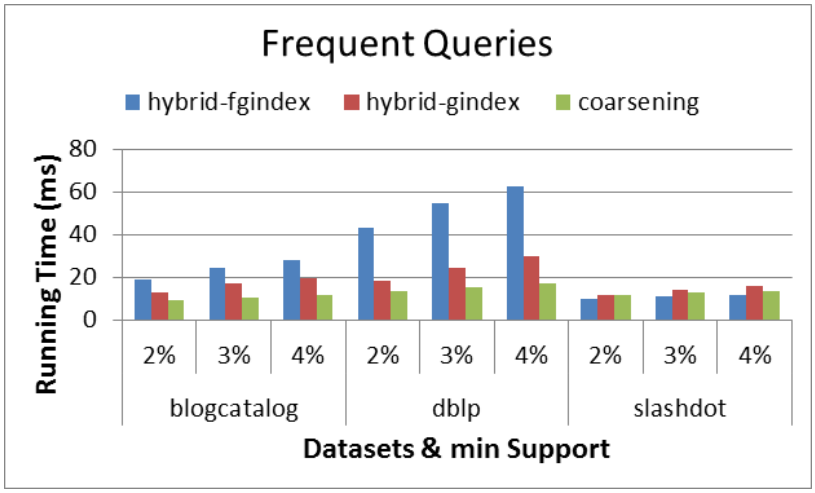


Figure 6.1: Runtime comparison for Frequent queries on Social Network databases.

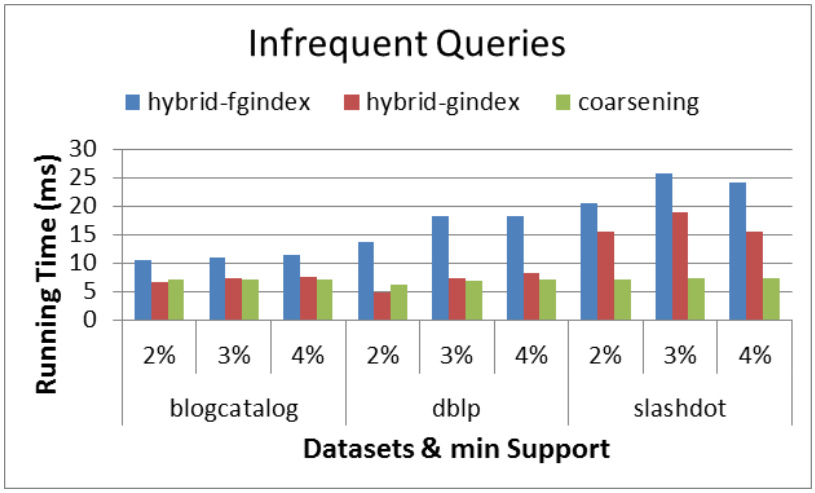


Figure 6.2: Runtime comparison for Infrequent queries on Social Network databases.

per feature is lower however.

Our index contains only 907 features while performing better compared to *hybrid*-`gIndex` and *hybrid*-`FG-Index` that contain 9,638 and 8,246 features, respectively for *minSup*=2%, in case of `BlogCatalog3`. For `DBLP`, our index contains 446 features while *hybrid*-`gIndex` and *hybrid*-`FG-Index` contain 1,583 and 986 features respectively for *minSup*=2%. Better performance with lesser number of features shows that our

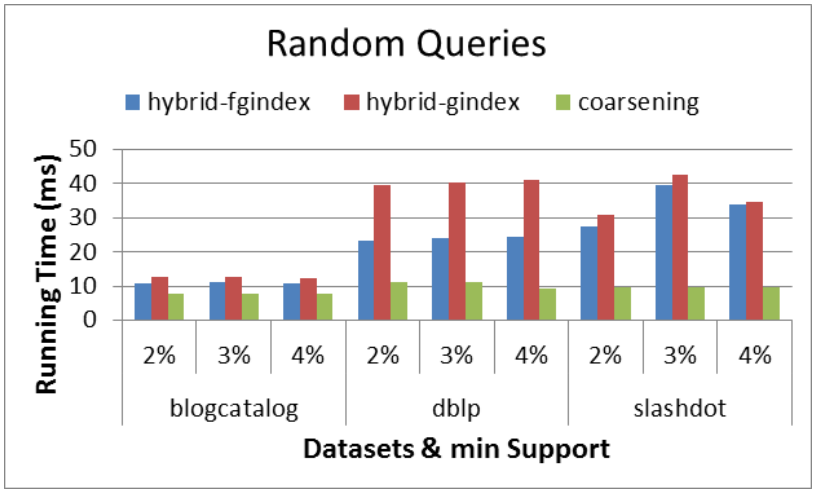


Figure 6.3: Runtime comparison for Random queries on Social Network databases.

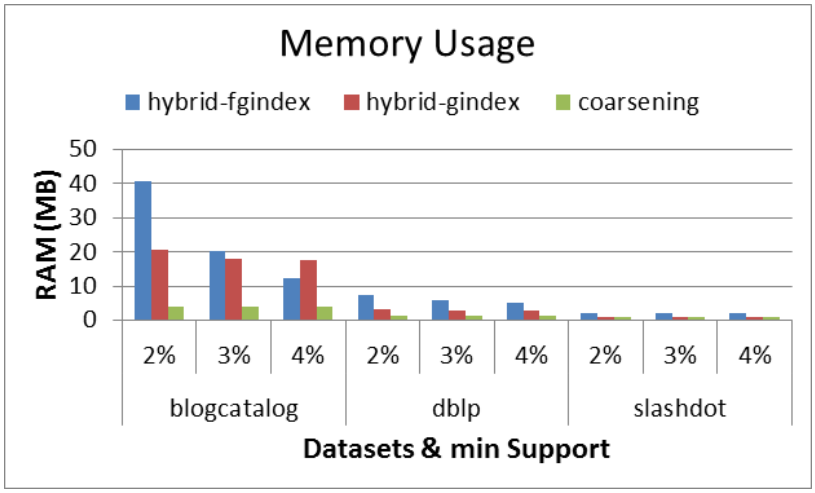


Figure 6.4: Index memory consumption comparison for Social Network databases.

index has a higher quality of features.

For `Slashdot` database, we observe a different pattern. Our index performs comparably to both *hybrid*-`gIndex` and *hybrid*-`FG-Index` for frequent queries, but outperforms them for infrequent and random queries. This shows our index performs well for dense databases as well.

In general, we observe our index to be independent of the database and query

type (and min support as well) and we are always able to answer any query in less than 20 ms for social network databases.

The comparison of memory consumption between our graph-coarsening index and *hybrid*-indexes for different minimum supports is shown in Figure 6.4 for social network databases. We require up to 4 times less space compared to *hybrid*-`gIndex` while up to 10 time less space compared to *hybrid*-`FG-Index` (see `BlogCatalog3` *minSup*=2%). Lesser number of index features in Table 6.1 also supports that our index requires less memory. Higher memory consumption in case of *hybrid*-`FG-Index` with less features is due to index stored on disk compared to our index or *hybrid*-`gIndex` which are stored in main memory only.

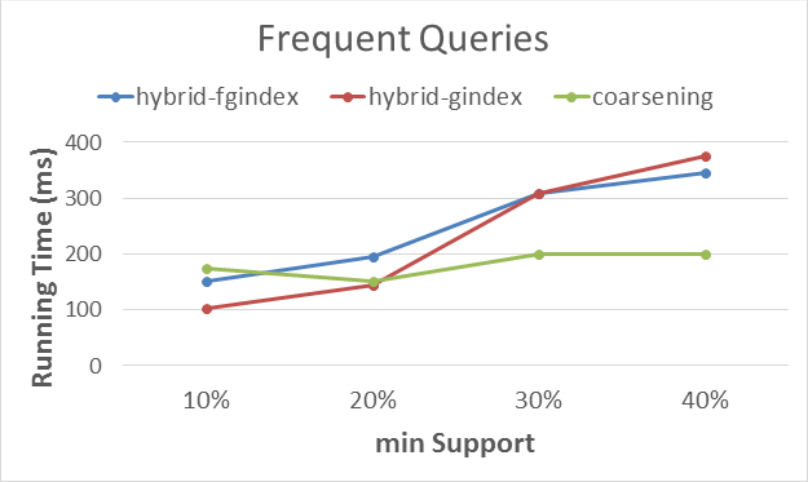


Figure 6.5: Runtime comparison for Frequent queries on eMolecules database.

6.2.2 Chemical Database

Figures 6.5, 6.6 and 6.7 show the running time comparison on chemical database for answering frequent, infrequent, and random queries, respectively. Table 6.2 shows the number of features indexed for different *minSup* values by database. The number of

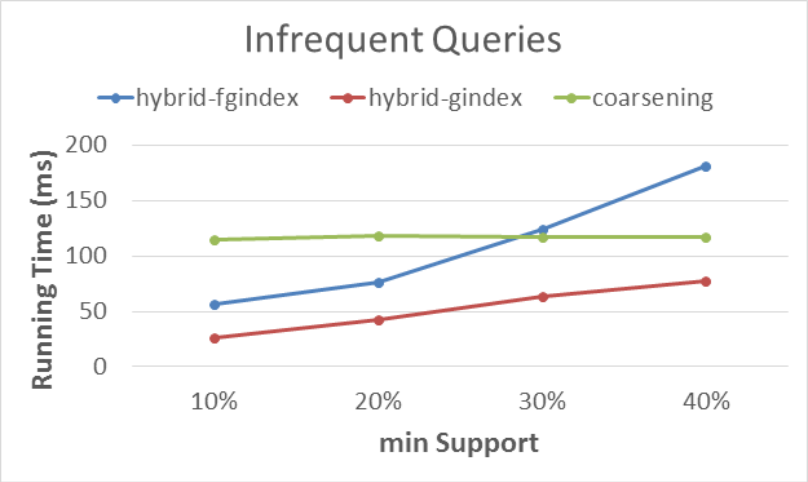


Figure 6.6: Runtime comparison for Infrequent queries on eMolecules database.

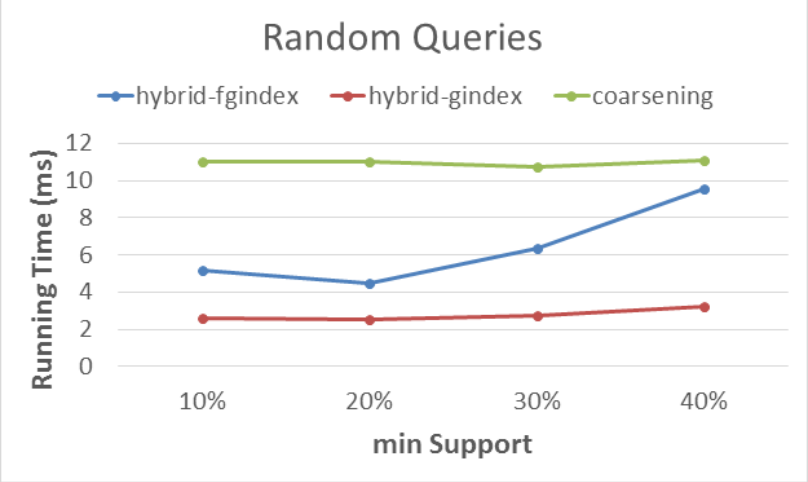


Figure 6.7: Runtime comparison for Random queries on eMolecules database.

features for *hybrid*-`FG-Index` are reported for what was stored in main memory and on disk.

For frequent queries, our index is comparable for 10% and starts to beat competitors from a minimum support of 20%. The *hybrid*-indexes benefit from structure information contained in the index when the *minSup* is low requiring less subgraph isomorphism tests because more queries are answered by the index directly, without

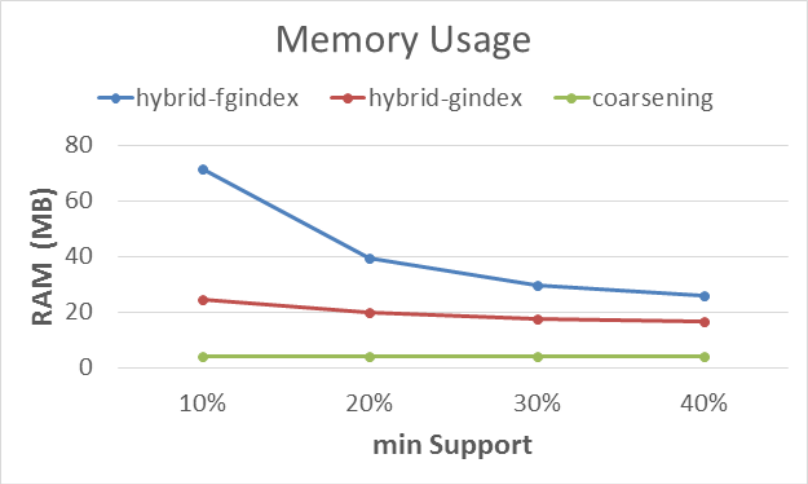


Figure 6.8: Index memory consumption comparison for eMolecules databases.

verification. But as *minSup* increases, lesser number of queries is answered directly that requires index feature joins to generate candidate answer sets. Therefore, more subgraph isomorphism tests need to be performed to get the answer. The number of features in our index is less than *hybrid*-indexes for $minSup \leq 30\%$ but better or comparable performance which shows our index has a higher quality of features.

We beat *hybrid*-`FG-Index` at $minSup{=}40\%$ for infrequent queries. And, we are comparable with *hybrid*-`FG-Index` for random queries on testing with the same minimum support.

The size of the candidate set for queries against graph-coarsening index grows as support of the queries decreases. A less frequent coarsened query is contained in a larger set of graphs because the coarsening is more common in the graphs than the structure of the query itself and requires more time for verification. This is why the query answering time is nearly constant across different minimum supports for our index.

The comparison of memory consumption between our graph-coarsening index and

Table 6.2: Count of index features on Chemical Database for Standalone Index comparison

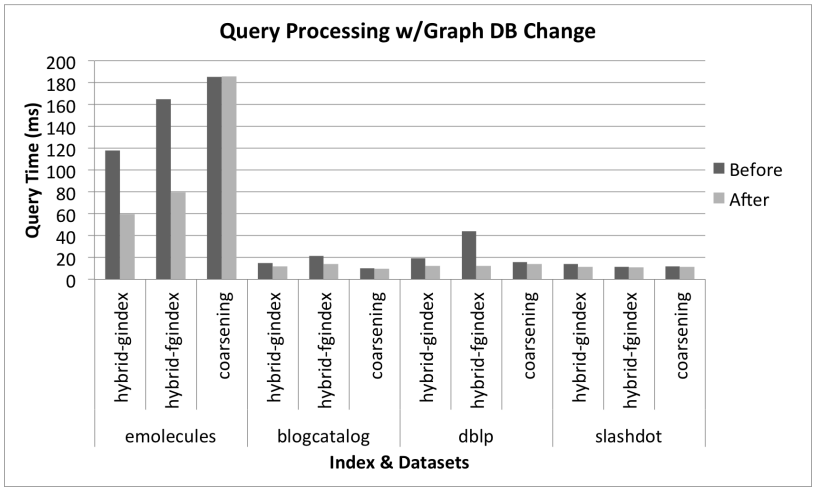
Chemical Database	Index	*minSup*			
		10%	20%	30%	40%
eMolecules	*hybrid*-gIndex	799	596	532	512
	hybrid-FG-Index	2619	769	361	229
	coarsening	329	329	329	329

hybrid-indexes for different minimum supports is shown in Figure 6.8 for the chemical database. In general, our index uses far less memory than competitors.

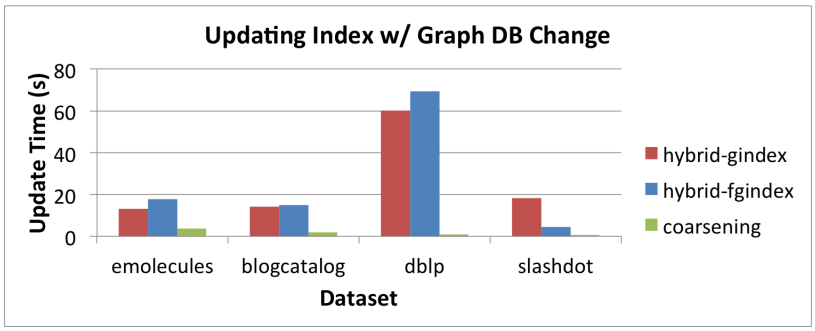
We require 4 to 6 times less space compared to *hybrid*-gIndex while 6 times to 17 times less space compared to *hybrid*-FG-Index to store the index when going from 40% minimum support to 10% minimum support. Lesser number of index features in Table 6.2 also supports that our index requires less memory.

6.2.3 Discussion

The main difference between chemical and social network databases is that a chemical database is rich in frequent subgraphs as the number of atoms (node labels) is fixed and they follow precise rules to form bonds among themselves making a higher number of subgraphs redundant. Ego networks present more variety than chemical compounds. In numbers, for a *minSup* equal to 4% of database size, we have 1,270 discriminative fragments in *hybrid*-gIndex for eMolecules. But we have 1,453 and 9,038 discriminative fragments in case of DBLP and BlogCatalog3. Higher number of discriminative fragments suggests that the *pruning power* of *hybrid*-gIndex for social network databases is lesser because fewer index features aid in pruning out graphs for a frequent query.



(a) Query time.



(b) Update time.

Figure 6.9: (a) Runtime comparison for graph DB change between different datasets for frequent queries. (b) Index update time comparison.

6.3 Comparison on Dynamic Graph Databases

To simulate a dynamic graph database, we replace 40% of the graphs in each of our four databases with new graphs. We use only frequent queries for comparison as the change affects them the most. We first built the index with the original database and ran a set of frequent queries. We used the same set of queries as the query workload for index update. Then, we used `one-pass` to update *hybrid*-indexes for the changed database. To construct the *hybrid*-index, we used *minSup* equal to 10% of database size for `eMolecules` and *minSup*=2% for social network databases.The α parameter

in the swapping criterion for updating the *hybrid*-indexes with `one-pass` algorithm was set to 0.99.

Figure 6.9 (a) shows the query answering time before and after the index update. We observe that *hybrid*-indexes benefit from the update made by `one-pass` algorithm, especially in the case of chemical database. Our index, instead, is up to date at any time and maintains a constant pruning power, independently of the database update. Our index is comparable for social network databases in terms of query processing before and after the update. Regarding the time for updating the index, as shown in Figure 6.9 (b), we are up to 60 times faster than `one-pass` algorithm.

6.4 Query Workload Changes

In this section, we compare the querying time when the query workload type changes over time. The `one-pass` uses query workload changes to detect and update the index. It uses ADWIN [3] for change detection and if the query workload has changed significantly, update is applied to the index. Our index does not depend on query workload and remains current with regards to it. Moreover, we do not have any cost in updating the index as we are query independent.

We set the size of each query workload type to 10,000 queries and ran queries in batches of 100. We changed the workload from frequent queries to random or infrequent queries, and back to frequent queries. Since the index performs differently with different workloads, we started with frequent queries for optimal index performance. To increase feature swaps, we tried four approaches before reverting back to frequent queries. We either changed the query workload to contain infrequent or random queries only, or we changed the query workload to be infrequent then random

or random then infrequent. For ease of understanding, the query workload changes we used are as follows.

1. Frequent → Infrequent → Frequent queries
2. Frequent → Random → Frequent queries
3. Frequent → Infrequent → Random → Frequent queries
4. Frequent → Random → Infrequent → Frequent queries

Change in query workload was detected by ADWIN which triggered index update and features were swapped. While the index was updating, we used the current index until the updated index was available for use. We ran *hybrid*-indexes with $minSup =$ 2% (resp. 10%) in case of social networks (resp. `eMolecules`).

Figure 6.10 and 6.11 show the query answering time for this experiment on `DBLP` and `eMolecules`, and Figure 6.12 and 6.13 show the query answering time on `BlogCatalog3` and `Slashdot`. The results are plotted using a line graph for each batch of 100 queries and workload change can be noticed as querying time changes for every 10,000 queries. While running the code for *hybrid*-indexes, the code to update the index threw exception. The runtime after exception is reported as dropping to zero. It is unclear why the code threw exception. Since we were using previous implementation, we did not modify it.

The *hybrid*-indexes benefit from update when query workload changes from frequent to infrequent. There is no improvement in overall performance otherwise. *hybrid*-`FG-Index` benefits from update when the query workload changes back to frequent queries from infrequent or random queries while *hybrid*-`gIndex` performs about the same regardless of the update. Our index outperforms *hybrid*-indexes

on social network databases and results independent of the query workload. For frequent queries, we are up to 15 times better than *hybrid*-`FG-Index` and up to 7 times better than *hybrid*-`gIndex`. In the case of chemical database, we are comparable, and sometimes better, than competitors on frequent queries.

6.5 Scalability

In this section, we compare the scalability of our graph-coarsening index with *hybrid*-indexes. For this experiment, we followed the same approach as `one-pass` paper [35], i.e., we randomly sample five graph databases from `eMolecules` database. These databases have a size from 2^{16} to 2^{20} graphs.

Figures 6.14 (a) and (b) show the index construction time and memory usage, respectively, for different database sizes. *hybrid*-indexes were built with *minSup*=10%. Our graph-coarsening based index is always faster to construct and requires up to 6 (resp. 5) times less space than *hybrid*-`FG-Index` (resp. *hybrid*-`gIndex`). Results show our index grows linearly space-wise, while other indexes grow exponentially, with increase in database size, making our index desirable for larger databases.

Moreover, we also compare the construction time and memory usage with different minimum supports in the case of a database containing 2^{16} graphs. Results are shown in Figures 6.14 (c) and (d), respectively. The running time to build the index and the memory usage are represented by a flat line in the case of our graph-coarsening index as we are independent of the minimum support. As expected, both construction time and memory usage decrease for *hybrid*-indexes as the value of *minSup* increases.

6.6 Results Summary

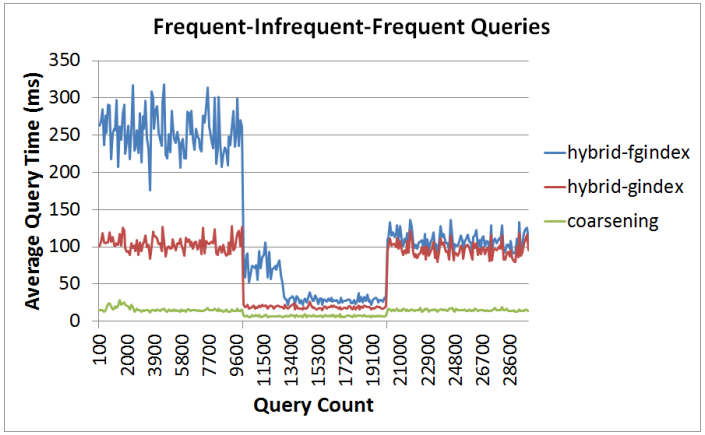
In summary, *our index outperforms competitors on denser graph databases (e.g. social networks) and performs comparably for sparse ones (e.g. chemical databases).* Experimental results show that:

(1) We outperform state-of-the-art indexes for query answering time by up to 3 times in the case of social network databases. In the case of chemical database, we are comparable with *hybrid*-indexes for frequent and infrequent queries.

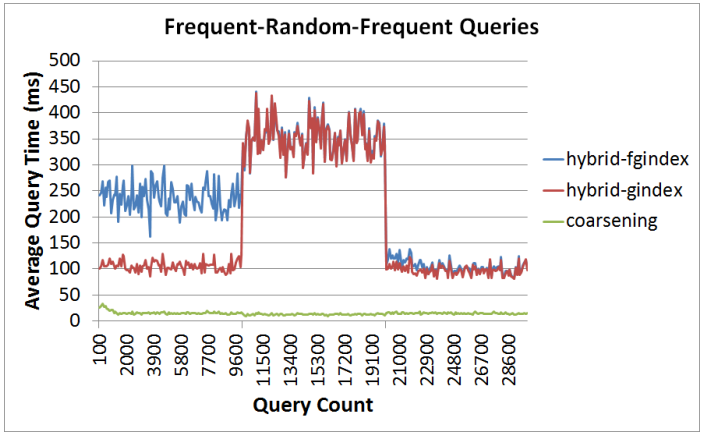
(2) We are scalable with a faster construction time and smaller index size.

(3) We can update our index up to 60 times faster in comparison to `one-pass` algorithm for dynamic graph databases.

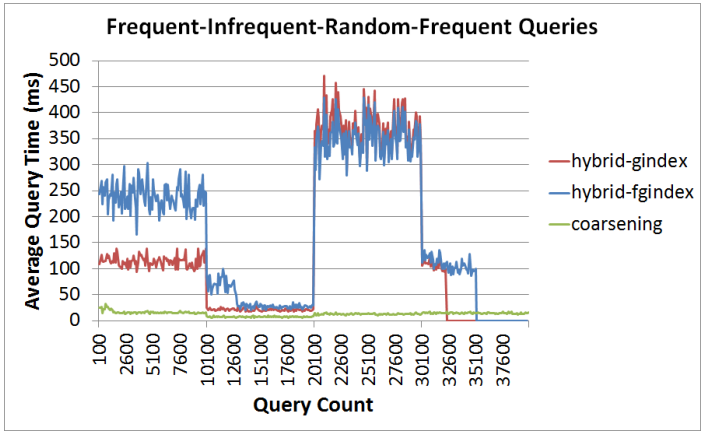
(4) Our index is independent of the query workload for index update and is up to 15 times better after *hybrid* indexes are attuned to query workload.



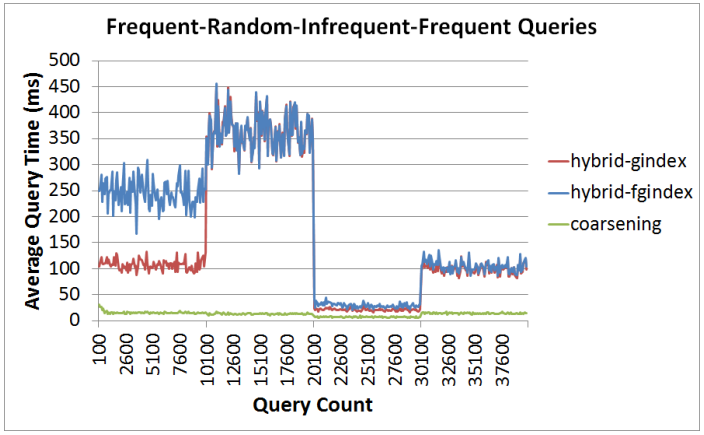
(a) Query workload change from frequent to infrequent queries for DBLP.



(b) Query workload change from frequent to random queries for DBLP.

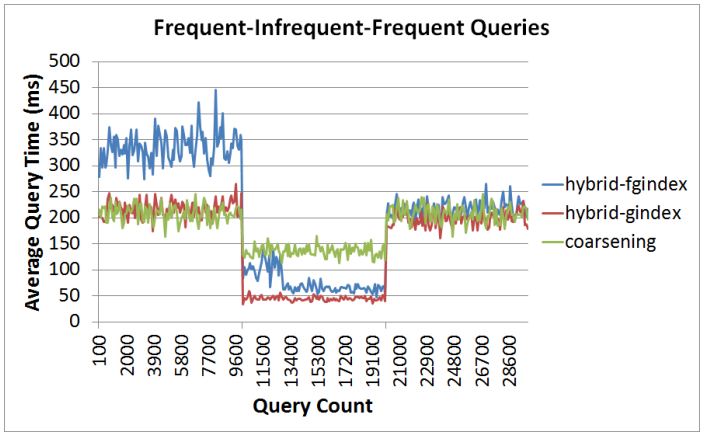


(c) Query workload change from frequent to infrequent queries for DBLP.

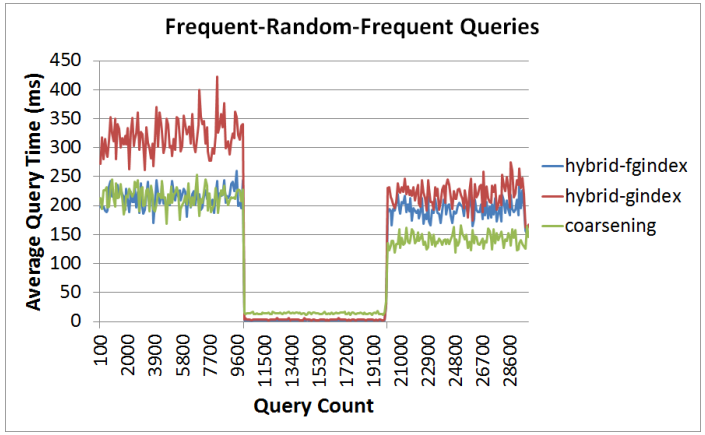


(d) Query workload change from frequent to random queries for DBLP.

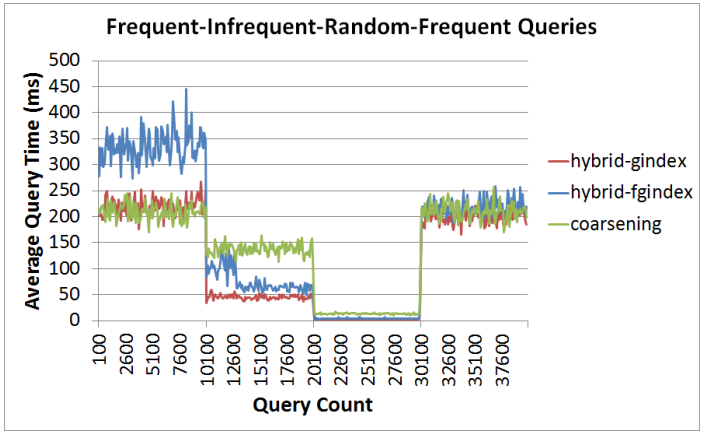
Figure 6.10: Runtime comparison for query workload change for DBLP.



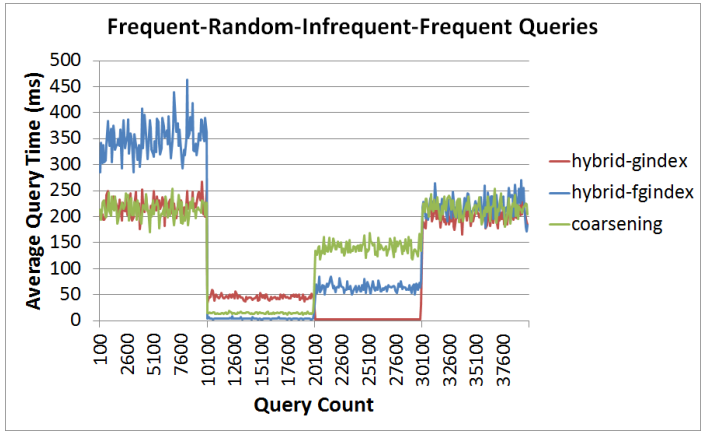
(a) Query workload change from frequent to infrequent queries for `eMolecules`.



(b) Query workload change from frequent to random queries for `eMolecules`.

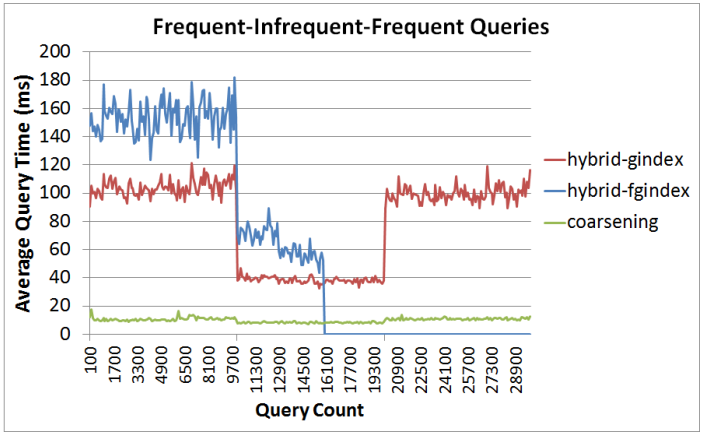


(c) Query workload change from frequent to infrequent queries for `eMolecules`.

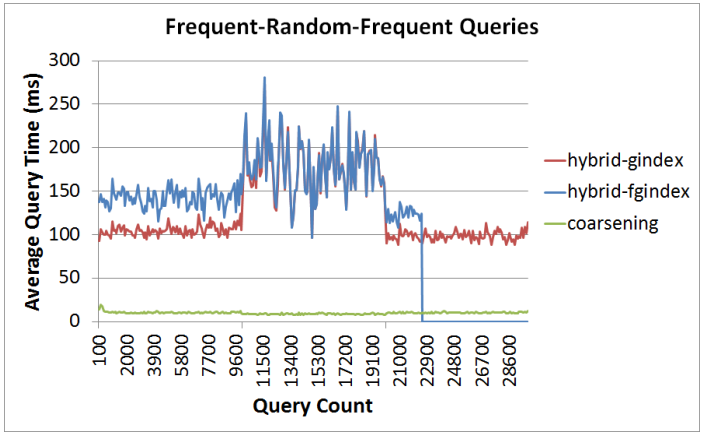


(d) Query workload change from frequent to random queries for `eMolecules`.

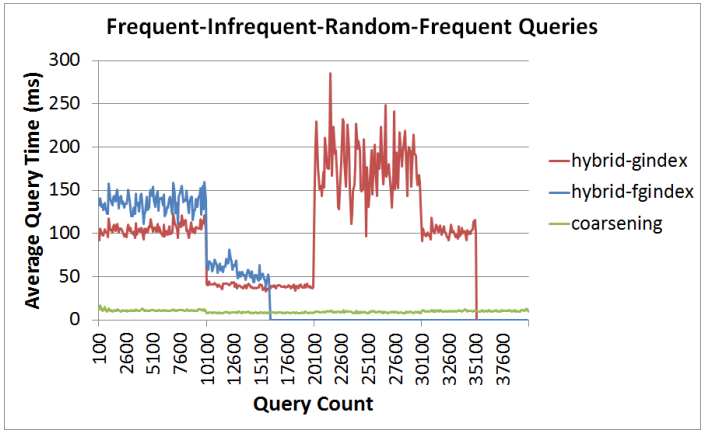
Figure 6.11: Runtime comparison for query workload change for `eMolecules`.



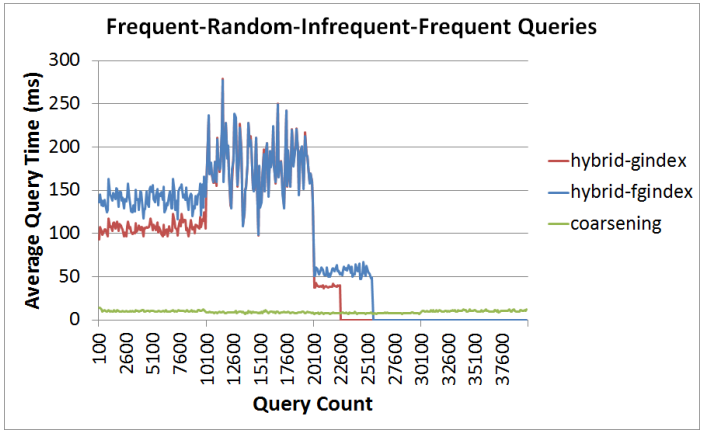
(a) Query workload change from frequent to infrequent queries for `BlogCatalog3`.



(b) Query workload change from frequent to random queries for `BlogCatalog3`.

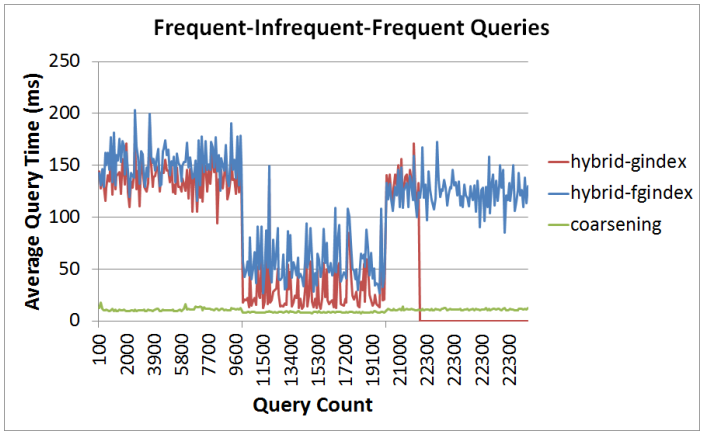


(c) Query workload change from frequent to infrequent queries for `BlogCatalog3`.

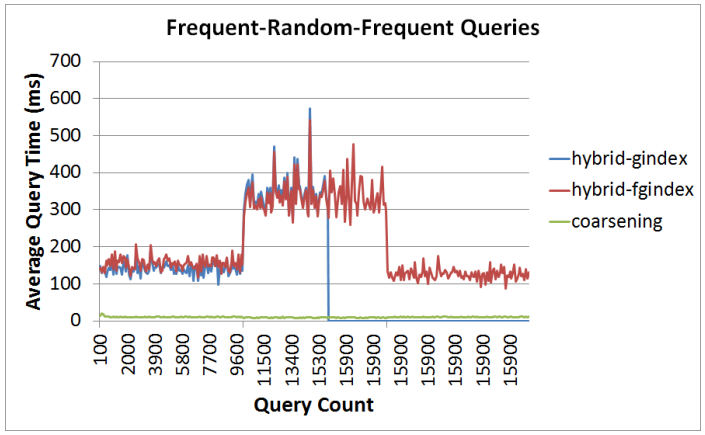


(d) Query workload change from frequent to random queries for `BlogCatalog3`.

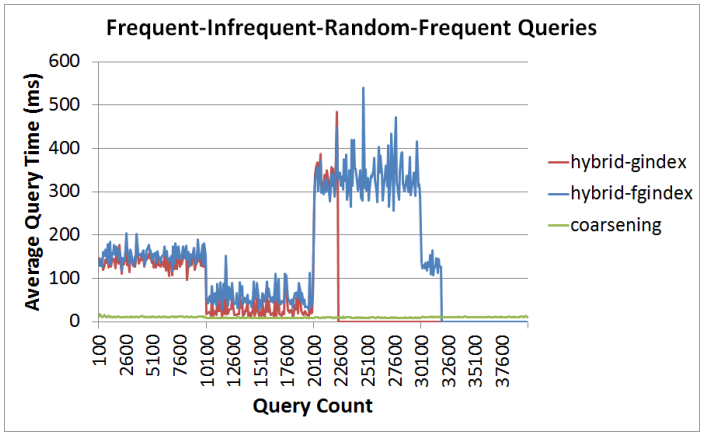
Figure 6.12: Runtime comparison for query workload change for `BlogCatalog3`.



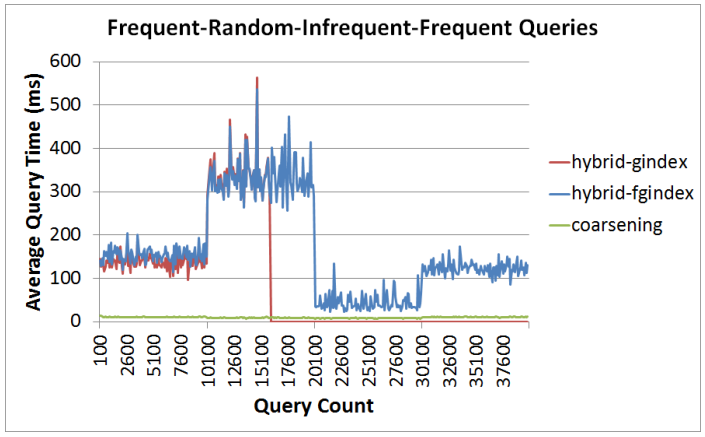
(a) Query workload change from frequent to infrequent queries for `Slashdot`.



(b) Query workload change from frequent to random queries for `Slashdot`.

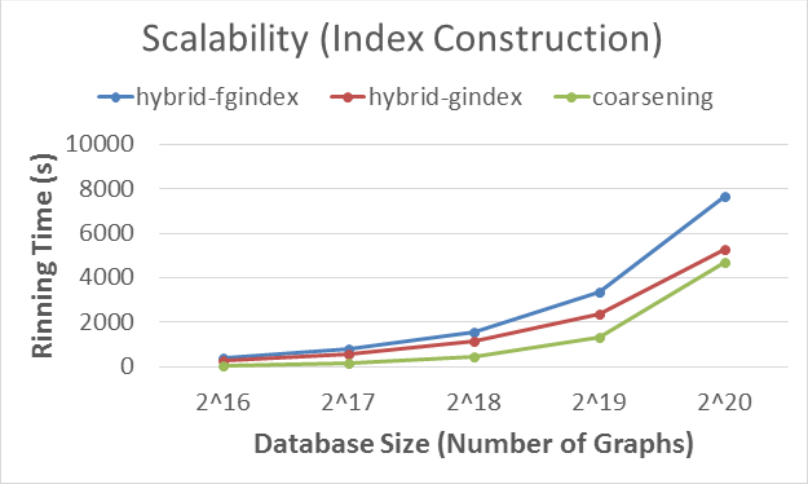


(c) Query workload change from frequent to infrequent queries for `Slashdot`.

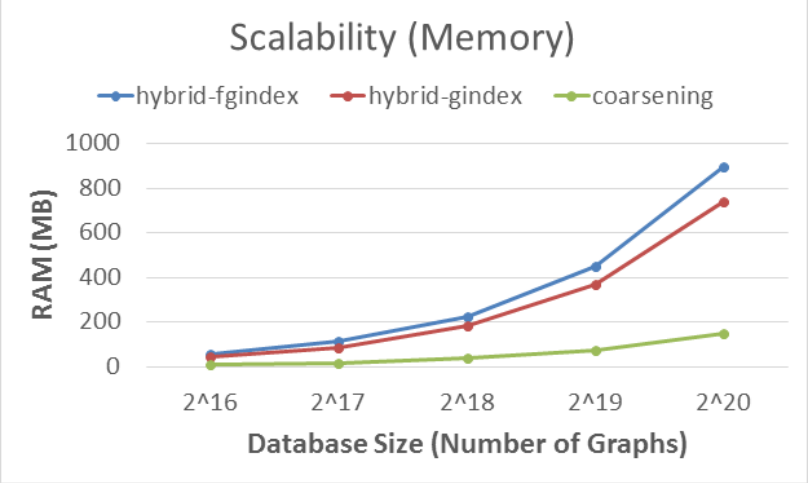


(d) Query workload change from frequent to random queries for `Slashdot`.

Figure 6.13: Runtime comparison for query workload change for `Slashdot`.

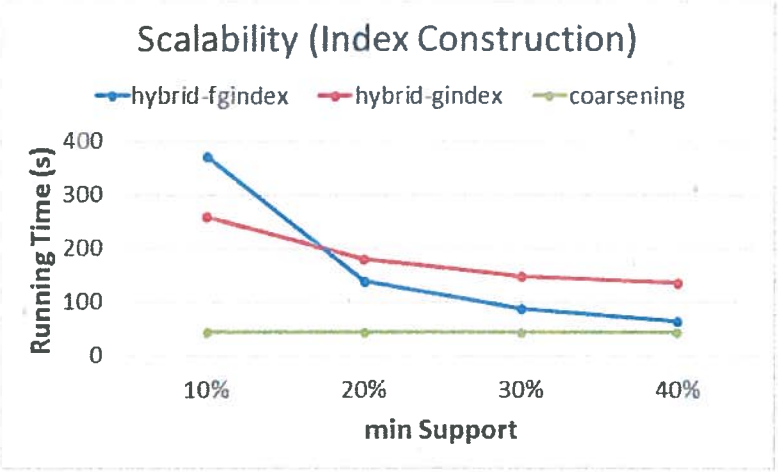


(a) Index construction time for different database sizes.

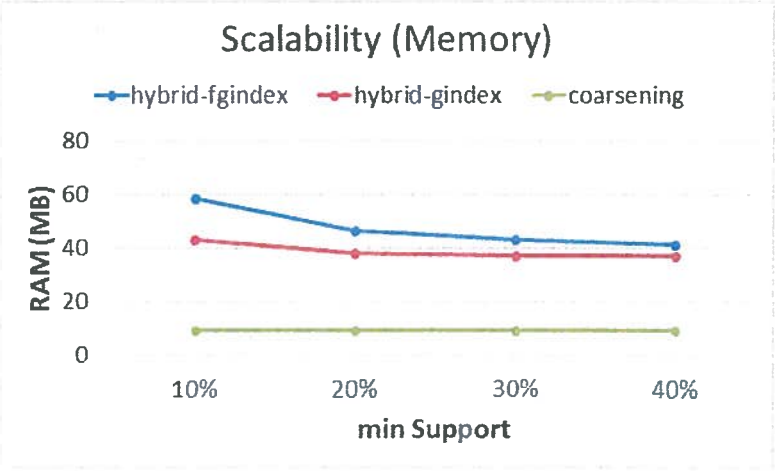


(b) Memory consumption for different database sizes.

Figure 6.14: Scalability comparison: Index construction time and memory consumption.



(c) Index construction time for different min supports for a graph databases containing 2^{16} graphs.



(d) Memory consumption for different min supports for a graph databases containing 2^{16} graphs.

Figure 6.15 Scalability comparison: Index construction time and memory consumption.

CHAPTER 7

CONCLUSIONS

7.1 What have we done so far?

We proposed a new index based on graph-coarsening for speeding up the query answering time in dynamic graph databases. The index is parameter-free, query-independent, scalable, small enough to store in the main memory, and is simpler and less costly to maintain in case of database updates.

Experimental results showed that we outperform *hybrid*-indexes for query answering time in the case of social network databases. In the case of chemical database, we are comparable with competitors for frequent and infrequent queries. We can update our index up to 60 times faster in comparison to `one-pass` for dynamic graph databases. Moreover, our index is independent of the query workload for index update and is up to 15 times better after *hybrid*-indexes are attuned to query workload.

7.2 Future directions

We have tested our index for several real datasets. As future work, we would like to measure performance of our index on synthetic datasets. It will help us understand how the index behaves with different types of data. To generate graph databases, we will vary parameters such as

- Number of nodes
- Density
- Number of distinct node labels
- Number of graphs in the database

Queries will be generated for each parameter after the database has been created. We will compare the results with state-of-the-art *static* indexes such as GRAPES [19] and GraphGrepSX [4]. We will use GraphGen [1] algorithm to generate synthetic data.

In future, we also plan to use our index to study the following.

- Test our graph-coarsening based index for supergraph search, i.e. when the answer to a query Q is the set of all graph databases G s.t. $Q \supseteq G$, and compare our performances with `cIndex` [6], the state-of-the-art index for supergraph query and `cIndex` updated by `one-pass`.
- Compare performance of our index on a cluster for very large graph databases by exploring ways a database can be distributed over a cluster.
- Study substructure similarity search in graph databases, i.e. finding similar structures in a graph database by edge relaxations on a query graph [33]. Basically, edge relaxation allows node and edge labels to be ignored in a query graph while preserving total edge counts. Our index can adapt to these relaxations and can be used to generate a candidate set.
- Adapt our index to solve the top-k problem, i.e. finding top-k graphs in a graph database that are most similar to a query graph [39].

REFERENCES

[1] Graphgen — a synthetic graph data generator. `http://www.cse.ust.hk/graphgen/`.

[2] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1089–1105, 2001.

[3] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Mining frequent closed graphs on evolving data streams. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'11, pages 591–599, 2011.

[4] Vincenzo Bonnici, Alfredo Ferro, Rosalba Giugno, Alfredo Pulvirenti, and Dennis Shasha. Enhancing graph database indexing by suffix tree structure. In *Proceedings of the 5th IAPR International Conference on Pattern Recognition in Bioinformatics*, PRIB'10, pages 195–203, 2010.

[5] Christian Borgelt and Michael R Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM'02, pages 51–58. IEEE, 2002.

[6] Chen Chen, Xifeng Yan, Philip S Yu, Jiawei Han, Dong-Qing Zhang, and Xiaohui Gu. Towards graph containment search and indexing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'07, pages 926–937, 2007.

[7] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: Towards verification-free query processing on graph databases. In *Proceedings of the 2007 International Conference on Management of Data*, SIGMOD'07, pages 857–872, 2007.

[8] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: An adaptive path index for xml data. In *Proceedings of the 2002 International Conference on Management of Data*, SIGMOD'02, pages 121–132, 2002.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC'71, pages 151–158, 1971.

[10] Raffaele Di Natale, Alfredo Ferro, Rosalba Giugno, Misael Mongiovì, Alfredo Pulvirenti, and Dennis Shasha. Sing: Subgraph search in non-homogeneous graphs. *BMC bioinformatics*, 11(1):96, 2010.

[11] eMolecules Database. http://www.emolecules.com.

[12] GitLab. http://www.gitlab.com.

[13] Rosalba Giugno, Vincenzo Bonnici, Nicola Bombieri, Alfredo Pulvirenti, Alfredo Ferro, and Dennis Shasha. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PloS one*, 8(10):e76911, 2013.

[14] Wook-Shin Han, Jinsoo Lee, Minh-Duc Pham, and Jeffrey Xu Yu. igraph: A framework for comparisons of disk-based graph indexing techniques. *Proceedings of the VLDB Endowment*, 3(1):449–459, 2010.

[15] Huahai He and Ambuj K Singh. Closure-tree: An index structure for graph queries. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE'06, pages 38–38, 2006.

[16] CA James, D Weininger, and J Delany. Daylight theory manual daylight version 4.82. daylight chemical information systems, 2003.

[17] Chanhyun Kang, Sarit Kraus, Cristian Molinaro, Francesca Spezzano, and V. S. Subrahmanian. Diffusion centrality: A paradigm to maximize spread in social networks. *Artificial Intelligence*, 239:70–96, 2016.

[18] Akshay Kansal and Francesca Spezzano. A scalable graph-coarsening based index for dynamic graph databases. In *26th ACM International Conference on Information and Knowledge Management (CIKM)*, CIKM'17, 2017.

[19] Foteini Katsarou, Nikos Ntarmos, and Peter Triantafillou. Performance and scalability of indexed subgraph query processing methods. *Proceedings of the VLDB Endowment*, 8(12):1566–1577, 2015.

[20] Karsten Klein, Nils Kriege, and Petra Mutzel. Ct-index: Fingerprint-based graph indexing combining cycles and trees. In *Proceedings of the 27th International Conference on Data Engineering*, ICDE'11, pages 1115–1126, 2011.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[22] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[23] Thomas Madej, Jean-François Gibrat, and Stephen H Bryant. Threading a database of protein cores. *Proteins: Structure, Function, and Bioinformatics*, 23(3):356–369, 1995.

[24] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10), 2004.

[25] Euripides G. M. Petrakis and A Faloutsos. Similarity searching in medical image databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):435–447, 1997.

[26] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375, 2008.

[27] Ali Shokoufandeh, Sven J Dickinson, Kaleem Siddiqi, and Steven W Zucker. Indexing using a spectral encoding of topological structure. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2 of *CVPR'99*, 1999.

[28] SMILES. https://en.wikipedia.org/wiki/Simplified_molecular-input_line-entry_system.

[29] David W Williams, Jun Huan, and Wei Wang. Graph database indexing using structured graph decomposition. In *Proceedings of the 23rd International Conference on Data Engineering*, ICDE'07, pages 976–985, 2007.

[30] Yan Xie and Philip S Yu. Cp-index: on the efficient indexing of large graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM'11, pages 1795–1804, 2011.

[31] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM'02, pages 721–724, 2002.

[32] Xifeng Yan, Philip S Yu, and Jiawei Han. Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 International Conference on Management of Data*, SIGMOD'04, pages 335–346, 2004.

[33] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 766–777, 2005.

[34] Dayu Yuan and Prasenjit Mitra. Lindex: a lattice-based index for graph databases. *The VLDB Journal*, 22(2):229–252, 2013.

[35] Dayu Yuan, Prasenjit Mitra, Huiwen Yu, and C. Lee Giles. Updating graph indices with a one-pass algorithm. In *Proceedings of the 2015 International Conference on Management of Data*, SIGMOD'15, pages 1903–1916, 2015.

[36] Reza Zafarani and Huan Liu. Social computing data repository at ASU. http://socialcomputing.asu.edu, 2009.

[37] Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: A novel graph indexing method. In *Proceedings of the 23rd International Conference on Data Engineering*, ICDE'07, pages 966–975, 2007.

[38] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: Tree + delta >= graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 938–949, 2007.

[39] Yuanyuan Zhu, Lu Qin, Jeffrey Xu Yu, and Hong Cheng. Finding top-k similar graphs in graph databases. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT'12, pages 456–467, 2012.

[40] Lei Zou, Lei Chen, Jeffrey Xu Yu, and Yansheng Lu. A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*, EDBT'08, pages 181–192, 2008.

APPENDIX A

REPRODUCING EXPERIMENTS

A.1 Getting the code

The code can be downloaded from GitLab [12] at the URL below. The repository will be made public after publication of this thesis.

URL: `https://akshaykansal1@gitlab.com/akshaykansal1/graph-index.git`

The repository can be cloned and opened in Eclipse. We used Mars to write the code. The code was written in and is compatible with Java SE 7.

A.2 Data Formats

The implementation mainly deals with two formats of graph data.

1. SMILES [28] - Mainly for chemical data from `eMolecules`.
2. Generic Graph Format

A.2.1 Generic Graph Format

This format was introduced with gSpan [31]. The graph in the example below are undirected.

Example:

t represents the beginning of graph, v represents a node with node_id i and label l

and e represents an edge that connects source node_id i to destination node_id j with label el.

```
t # 0 (graph id)
v 0 0 (v i l)
v 1 0
v 2 0
e 0 1 1 (e i j el)
e 1 2 1
e 0 2 1
```

A.3 Running The Code

This section covers the basic structure of the code base, file naming conventions used, data formatting for social networks and running experiments.

Please Note: All code written in this repository has file names hard coded

A.3.1 Repository Structure

The repository is structured as follows.

1. `./dblp`, `./slashdot`, `./blogcatalog`, `./gindex` contain the data used for the experiments.

2. `./lib` contains `jar` needed to run the code.

3. `./src/graphindex` contains all of our implementations. Other folders under `./src` contain the implementation from `one-pass` [35].

A.3.2 File Naming Convention

To make it easier to find the code, we used simple file naming conventions.

- `UniformGraphPicker*` to generate graph database using uniform distribution.
- `UniformGraphQueries*` to generate queries from graph database chosen from respective `UniformGraphPicker*` class.
- `ScalabilityGraphPicker` class generates five databases used for scalability experiment.
- `CoarseningIndex` for our graph-coarsening index.
- `CoarseningComparison*` for code that is used to run the comparison against competition.
- `CoarseningIndexDBUpdate*` for code that is used to run changing graph database comparison against competition.
- `StandaloneExp*` contains code to run experiment on our competition for standalone indexes.
- `ScalabilityExp` contains code to run scalability experiment on our competition.
- `IndexUpdateForDB*` contains code to run experiment on our competition for changing graph databases.
- `ADWINAutoUpdate*` contains code to run query workload change experiment with ADWIN change detection.

A.3.3 Chemical Dataset Conversion

Since the data from `eMolecules` was in SMILES format, and our index could primarily work with the other format, we converted each graph parsed accordingly. Therefore, when working with chemical dataset, appropriate files were automatically selected. The difference can be observed by looking at file extension. For SMILES, an extension of ".smiles" will be present and no extension for generic graph format.

A.3.4 Labelling Graphs for Social Networks

When working with social networks from SNAP [21], the edges were undirected, represented as *source-dest*. This format did not match the two formats in the previous section. Therefore, we used the SNAP library to parse the data and convert to the generic graph format. The problem was that the nodes and edges were not labeled. We used page rank to label the nodes. The edges were assumed to have the same label of 1. In case of signed networks, edge labels were 1 and 2 as negative edge labels did not work with previous implementation. The nodes were labeled by computing page rank, provided in the API, and uniformly distributing minimum page rank to maximum page rank into 10 labels from 0-9. Then, ego networks were generated for each node in the network.

To convert the dataset and label graphs, run `PR.py` first and then `reset_nodes.py` in `./dblp`, `./blogcatalog` and `./slashdot` folders. The raw files may not be present and may need to be downloaded from SNAP [21] or BlogCatalog [36] site. The code was written in Python 2.6. `reset_nodes.py` was needed because each ego network needed to start from node 0 regardless of its label.

A.3.5 Generating Databases And Queries

1. `UniformGraphPicker*` helps generate the graph database and write to appropriate directories.

2. `UniformGraphQueries*` helps generate the queries and write to appropriate directories. It first mines queries using gSpan and then writes them to the file system.

A.3.6 Experiments

- `CoarseningComparison*` is used to run the comparison against competition.
- `CoarseningIndexDBUpdate*` is used to run changing graph database comparison against competition.
- `StandaloneExp*` is used to run experiment on our competition for standalone indexes.
- `ScalabilityExp` is used to run scalability experiment on our competition.
- `IndexUpdateForDB*` is used to run experiment on our competition for changing graph databases.
- `ADWINAutoUpdate*` is used to run query workload change experiment with ADWIN change detection.

A.3.7 Compile And Run

The section shows how to compile and run the code using command line. Using Eclipse is straight forward. Some classes require command line arguments which can be setup

by going to Run→Run Configurations→Arguments and adding the arguments under Program Arguments textbox.

Command line:

```
cd ./src
javac -d ../bin -cp .:../lib/*:../lib/moa-release-2012.08.31/* <filename>.java
```

- For `UniformGraphPicker*`, `UniformGraphQueries*`, `CoarseningComparisonScalability*`, `CoarseningComparisonStandalone*`, `CoarseningIndexDBUpdate*`:

  ```
  java -cp bin:lib/*:lib/moa-release-2012.08.31/* <filename>
  ```

- For `CoarseningComparisonADWIN*`:

  ```
  java -cp bin:lib/*:lib/moa-release-2012.08.31/* <filename>
  <workload_selector>
  ```

- For `ScalabilityExp`, `StandaloneExp*`:

  ```
  java -cp bin:lib/*:lib/moa-release-2012.08.31/* <filename> <index_selector>
  ```

- For `ADWINAutoUpdate*`:

  ```
  java -cp bin:lib/*:lib/moa-release-2012.08.31/* <filename> <index_selector>
  <workload_selector>
  ```