GBLENDER: Towards Blending Visual Query Formulation and Query Processing in Graph Databases

Changjiu Jin[†] Sourav S Bhowmick[†] Xiaokui Xiao[†] James Cheng[†]
Byron Choi[§]

[†]School of Computer Engineering, Nanyang Technological University, Singapore
[§]Department of Computer Science, Hong Kong Baptist University, Hong Kong
cjjin|assourav|xkxiao|jamescheng@ntu.edu.sg, choi@hkbu.edu.hk

ABSTRACT

Given a graph database $\mathcal{D}$ and a query graph g, an exact subgraph matching query asks for the set S of graphs in $\mathcal{D}$ that contain g as a subgraph. This type of queries find important applications in several domains such as bioinformatics and chemoinformatics, where users are generally not familiar with complex graph query languages. Consequently, user-friendly visual interfaces which support query graph construction can reduce the burden of data retrieval for these users. Existing techniques for subgraph matching queries built on top of such visual framework are designed to optimize the time required in retrieving the result set S from $\mathcal{D}$, assuming that the whole query graph has been constructed. This leads to sub-optimal *system response time* as the query processing is initiated only *after* the user has finished drawing the query graph.

In this paper, we take the first step towards exploring a novel graph query processing paradigm, where instead of processing a query graph after its construction, it *interleaves* visual query construction and processing to improve system response time. To realize this, we present an algorithm called GBLENDER that prunes false results and prefetches partial query results by exploiting the latency offered by the visual query formulation. It employs a novel *action-aware* indexing scheme that exploits users' interaction characteristics with visual interfaces to support efficient retrieval. Extensive experiments on both real and synthetic datasets demonstrate the effectiveness and efficiency of our solution.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query processing

General Terms

Algorithms, Experimentation, Performance

Keywords

Graph Databases, Graph Indexing, Visual Query Formulation, Frequent Subgraphs, Infrequent Subgraphs, Prefetching

1. INTRODUCTION

Graphs provide a natural way of modeling data in a wide variety of domains, such as social networks, bioinformatics, chemoinformatics, and Semantic Web. For example, in chemoinformatics graphs are used to represent atoms and bonds in chemical compounds. In bioinformatics, protein interaction networks are graphs where nodes represent molecules and edges represent interaction between them. Therefore, it is evident that graph databases are growing rapidly in size in a wide spectrum of applications. As a result, there is a critical need for efficiently querying these growing graph databases.

In recent times, the database community has shown tremendous interest in proposing innovative solutions to query large graph databases [2, 5, 6, 9, 12–14, 17–21]. Querying graph data involves two key steps: *query formulation* and *efficient processing* of the formulated query. A number of query languages have been proposed for graphs which can be used to formulate a query in textual form. For instance, GraphLog [3] represents both data and queries as graphs. PQL [10] is a pathway query language designed for biological networks. More recently, GraphQL [6] was proposed for querying general graph structured data.

Most of the existing graph query processing techniques have focused on exact graph or subgraph matching queries. That is, given a graph database $\mathcal{D} = \{g_1, g_2, \ldots, g_n\}$ and a query graph q, find all the graphs in which q is a subgraph. Note that it is inefficient to sequentially scan $\mathcal{D}$ to process q because (a) accessing the whole graph database is costly and (b) subgraph isomorphism test is NP-complete [4]. Hence, state-of-the-art methods build graph indices to efficiently process graph queries. Such strategy is typically performed in two major steps: *filtering* and *candidate verification*. First, the filtering step uses the index to eliminate subset of the false results and produce a candidate answer set. Then, the candidate verification step verifies whether the query is indeed a subgraph of each candidate. Since the candidate answer set is typically much smaller than the database, index-based query processing is significantly more efficient than sequential scan of the graph database.

1.1 Motivation

While the database community has made considerable progress in devising efficient graph query processing strategies, formulating a graph query using a graph query language often demands considerable cognitive effort from the end user (consumer) and requires "programming" skill that is at least comparable to SQL. A user must be familiar with the syntax of the language, and must be able to express his/her needs accurately in a syntactically correct form. However, in many real life domains it is unrealistic to assume that users are proficient in expressing such textual queries. For example, in life sciences domain biologists may need to be familiar with

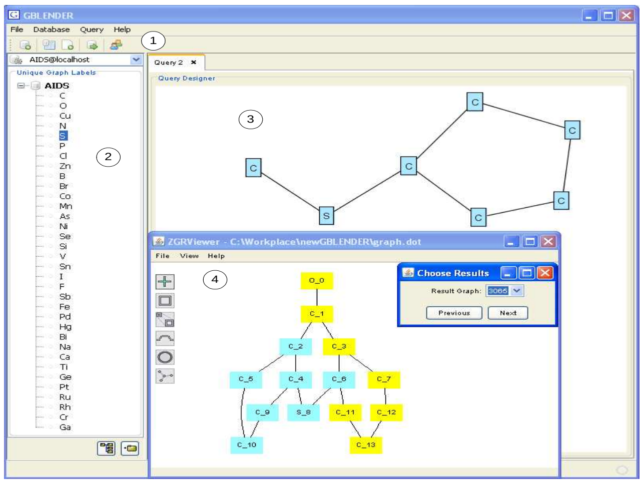


Figure 1: Visual interface of GBLENDER.

PQL to be able to formulate meaningful queries over biological networks. However, they cannot be expected to learn the complex syntax of PQL. In fact, the need for easy and intuitive techniques that can reduce the burden of query formulation is fundamental to the spreading of graph data management tools to wider community [8]. This highlights the need for a user-friendly visual querying scheme on top of existing graph processing methods to replace data retrieval aspects of graph query languages.

A visual interface for graph query construction typically enables a user to formulate a query by clicking-and-dragging items on the query canvas. Figure 1 depicts an example of such a visual interface for formulating graph queries. A user may sequentially drag items from Panel 2 to Panel 3 to create labeled nodes in the query graph and then create edges between them by clicking on relevant nodes. Once the user has finished construction of the query, he/she can click on the "Run" icon to execute the query. Typically, query evaluation in such a graphical framework can be performed in two key steps. First, the visual query is transformed into its textual or algebraic form. Second, the transformed query is evaluated using an existing state-of-the-art index-based graph processing method. Observe that the visual querying framework does not require a user to be familiar with the syntax of underlying graph query language. This is highly desirable in a wide variety of domains where a typical consumer is not proficient in database query languages.

A key feature of the traditional visual querying paradigm is that query evaluation is only initiated *after* the "Run" icon is clicked. Although the final query that a user intends to pose is revealed gradually in a step-by-step manner during query construction, it is not exploited by the query processor *prior* to clicking of the "Run" icon. This is primarily due to the fact that the data management community has traditionally considered visual interface-related issues more relevant to the human-computer interaction community and orthogonal to data processing. In this paper, we take the first step towards exploring a novel graph query processing paradigm by blending these two orthogonal areas. Specifically, we *interleave* query construction and query processing to prune false results and prefetch partial query results in a single-user environment by exploiting the latency offered by the GUI-based query formulation. A key objective of this new paradigm is to improve the *system response time* (SRT), which refers to the duration between the time a user presses the "Run" icon to the time when the user gets the query results. Note that in traditional graph processing paradigm SRT is identical to the time taken to evaluate the *entire* query. In contrast, in the new paradigm since we initiate query processing during query construction, SRT is the time taken to process a *part* of the query that is yet to be evaluated (if any). Often, as we shall see in Section 5, this results in significant improvement in SRT compared to traditional index-based graph processing methods.

Query processing on graphs in this new paradigm is challenging for a number of reasons. Firstly, the naïve strategy of matching every edge a user draws on the query canvas to the underlying database can be prohibitively expensive due to multiple subgraph isomorphism tests and repeated access to the disk. How can we blend query evaluation and query construction so that it can minimize disk access as well as subgraph isomorphism tests? Further, the number of candidate graphs for subgraph isomorphism should be manageable during the entire period. Secondly, what type of indexing schemes should we have to support such query processing paradigm? Indexing mechanism in this new paradigm should be effective even when the entire query is not known and must be able to exploit typical users' interaction behaviors with the visual interface for efficient pruning and retrieval. As we shall see in Section 3.2, existing state-of-the-art graph indexing schemes are not suitable for this purpose as they are oblivious to visual *actions* (or *steps*) taken by users during query construction. Further, they are primarily designed based on the assumption that the entire graph query is available. Thirdly, the *prefetching-based* graph query processing (we prefetch partial results) must be completely transparent from the user. A user's interaction behavior with the visual interface should not be affected by the query processing strategy. In this paper, we address all these issues.

1.2 Overview

In this paper we present an index-based method that blends visual query formulation and query processing, called GBLENDER (**G**raph **blender**). GBLENDER employs two novel *action-aware* indexing methods, called *action-aware frequent index* (A^2F) and *action-aware infrequent index* (A^2I), to support efficient matching of *frequent* and *infrequent* query fragments, respectively, while formulating a visual query graph. The A^2F index is a graph-structured index and has a *memory-resident* and a *disk-resident* components. In contrast to existing graph indexing schemes, its structure exploits some of the users' interaction characteristics with visual interfaces (e.g., visual queries grow incrementally in size during query formulation, smaller-sized query fragments appear more often in visual queries compared to larger-sized fragments). The A^2I-index indexes *discriminative infrequent subgraphs* (infrequent fragments whose subgraphs are all frequent) to prune the candidate space for infrequent queries. Note that except for FG-Index [2], none of the existing feature-based indexing schemes support infrequent fragments. FG-Index builds an index only for *infrequent edges* (infrequent fragment with only one edge) but not for infrequent subgraphs. We shall elaborate further on the differences in Section 3.2.

Based on the above indexing schemes, we propose an innovative matching paradigm for querying graphs. When a user draws a new edge on the query canvas during query formulation, GBLENDER searches the query fragment in the A^2F and A^2I indexes. If the query fragment is a frequent fragment, then it retrieves the identifiers of the matching graphs by probing the A^2F index. This identifier set is progressively refined as the size of the query grows gradually with the addition of new edges by the user. Note that in this work we assume that the user does not commit any mistake while formulating a query graph. If the final query remains frequent then the results are directly computed without subgraph isomorphism test. Otherwise, if the query fragment evolves to an infrequent fragment, then the algorithm probes the A^2I index to retrieve relevant identifiers of graphs containing discriminative infrequent fragments. Then, it retrieves the corresponding candidate graphs from the graph database which is progressively refined based on the subsequent actions by the user. Note that GBLENDER accesses the graph database only once (during infrequent fragment matching) throughout the query processing stage. When the "Run" icon is clicked, the system returns the exact results by filtering the false candidates using subgraph isomorphism test (if necessary).

We have applied GBLENDER to both real-world and synthetic datasets. Our experiments demonstrate that GBLENDER has excellent real-world performance and the system response time grows gracefully with increasing number of graphs in the database. We also show that GBLENDER significantly outperforms state-of-the-art indexing schemes based on traditional querying paradigm (highest observed factor being four orders of magnitude). In summary, the main contributions of this paper are as follows.

- We introduce an innovative graph matching paradigm that blends visual graph query construction and query processing to prefetch partial results during query formulation.
- In Section 3, we propose two novel action-aware indexing schemes, called *action-aware frequent index* (A^2F) and *action-aware infrequent index* (A^2I), that exploit typical visual interaction characteristics of users to facilitate efficient pruning and retrieval of partial results matching query fragments.
- We present a novel algorithm called GBLENDER that implements our proposed paradigm for exact subgraph matching queries by exploiting the indexing schemes (Section 4).
- By applying GBLENDER to real-world and synthetic datasets, in Section 5, we show its effectiveness, significant improvement of system response time over existing methods, and ability to gracefully handle increasing number of graphs in the database.

2. PRELIMINARIES

In this section, we first introduce concepts related to graph databases and queries which we shall be using subsequently. Then, we introduce the visual interface of GBLENDER for formulating visual queries.

2.1 Exact Subgraph Matching Problem

A graph G is denoted as (V, E), where V is the set of nodes and $E \subseteq |V| \times |V|$ is the set of (directed or undirected) edges in the graph. Nodes and edges can have labels as attributes specified by mappings $\phi : V \rightarrow \sum_{V_\ell}$ and $\psi : E \rightarrow \sum_{E_\ell}$ respectively, where $\sum_{V_\ell}$ is the set of node labels and $\sum_{E_\ell}$ is the set of edge labels. The *size* of G is denoted by $|G| = |E|$. For ease of presentation, we present our method using undirected graphs with labeled nodes. It is straightforward to extend our method to process other kinds of graphs.

A graph $G_1 = (V_1, E_1)$ is a *subgraph* of another graph $G_2 = (V_2, E_2)$ (or G_2 is a *supergraph* of G_1) if there exists a *subgraph isomorphism* from G_1 to G_2, denoted by $G_1 \subseteq G_2$ (or $G_2 \supseteq G_1$). The graph G_1 is called a *proper subgraph* of G_2, denoted as $G_1 \subset G_2$, if $G_1 \subseteq G_2$ and $G_1 \not\supseteq G_2$.

Definition 1. **(Subgraph Isomorphism)** *A subgraph isomorphism is an injective function $f : V_1 \rightarrow V_2$, such that (1) $\forall u \in V_1$, $\phi_1(u) = \phi_2(f(u))$, and (2) $\forall(u, v) \in E_1$, $(f(u), f(v)) \in E_2$ and $\psi_1(u, v) = \psi_2(f(u), f(v))$.*

2.2 Graph Fragments

We now introduce the notion of *frequent* and *infrequent graph fragments*.

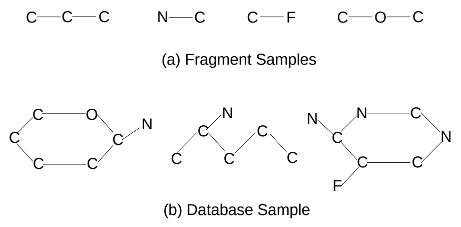


Figure 2: Fragment samples in a chemical compound database.

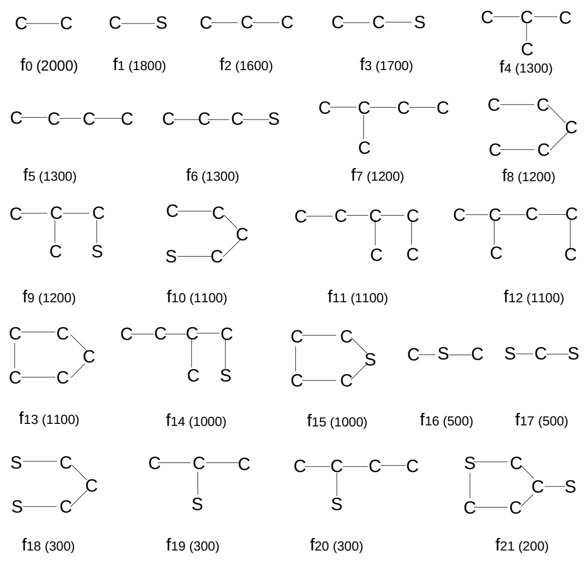


Figure 3: Frequent and infrequent fragments.

2.2.1 *Fragment Support Graph (FSG)*

Informally, we use the term *fragment* to refer to a small subgraph existing in graph databases or query graphs. We shall refer to a fragment in a query graph as *query fragment* in order to distinguish it from a fragment in the database. Let $\mathcal{D}$ be a graph database containing a set of graphs. In order to uniquely identify a graph in $\mathcal{D}$, we assign a unique id to each graph in the database. Let g be a subgraph of $G_i \in \mathcal{D}$ ($0 < i \leq |\mathcal{D}|$) and has at least one edge. Then, g is a *fragment* in $\mathcal{D}$. Note that g can be a path, a tree, or a graph. Also, some of the fragments in $\mathcal{D}$ may appear frequently in many graphs, while some other fragments may appear only in a few graphs. Figure 2 shows a sample of a graph database and some fragments in it. Observe that the fragments `C-C-C` and `C-N` can be found in every graph in Figure 2(b). However, the fragments `C-O-C` and `C-F` only appear once in the whole database.

Given a fragment $g \subseteq G$ and $G \in \mathcal{D}$, we refer to G as the *fragment support graph* (FSG) of g. We denote the set of FSGs of g as $\mathcal{D}_g$. $|\mathcal{D}_g|$ is called (*absolute*) *support*, denoted by $sup(g)$. Since each graph in $\mathcal{D}$ is denoted by a unique identifier, $fsgId(g)$ denotes the set of identifiers of the graphs in $\mathcal{D}_g$.

2.2.2 *Frequent Fragments*

A fragment g is *frequent* if its support is no less than $\alpha|\mathcal{D}|$ where α is the minimum support threshold. That is, if $g \in \mathcal{D}$ and $sup(g) \geq freq(\mathcal{D})$ where $freq(\mathcal{D}) = \alpha|\mathcal{D}|$ and $0 < \alpha < 1$ then g is a *frequent* fragment in $\mathcal{D}$. We denote the set of frequent fragments in $\mathcal{D}$ as $\mathcal{F}$. For example, let $|\mathcal{D}| = 10000$ and $\alpha = 0.1$. Then, $freq(\mathcal{D}) = 1000$. That is, all fragments with support larger than or equal to 1000 in $\mathcal{D}$ are frequent fragments. The fragments $f_0 - f_{15}$ in Figure 3 are frequent fragments as their supports are no less than 1000 (shown in parenthesis). Note that a frequent fragment's subgraph must be a frequent fragment [17].

2.2.3 *Infrequent Fragments*

Given a fragment $g \in \mathcal{D}$, if the support $sup(g) < freq(\mathcal{D})$ then g is an *infrequent* fragment. For example, in Figure 3 f_{16}-f_{21} are infrequent fragments as their support is less than 1000. We denote the set of infrequent fragments in $\mathcal{D}$ as $\mathcal{I}$. Obviously, a fragment is either frequent or infrequent in a given database. Note that the number of infrequent fragments in $\mathcal{D}$ depends on the minimum support threshold. As we increase the threshold towards 1, the number of infrequent fragments increases in $\mathcal{D}$. Consequently, it is computationally expensive to index all infrequent fragments in the database. To alleviate this problem, we focus our attention to those infrequent fragments whose subgraphs are all frequent. We refer to these fragments as *discriminative infrequent fragments*.

Definition 2. (**Discriminative Infrequent Fragment**) *Given $g \in \mathcal{I}$, let $sub(g)$ be the set of the subgraphs of g. If $sub(g) \subset \mathcal{F}$ or $|g| = 1$, then g is a discriminative infrequent fragment in $\mathcal{D}$.*

For example, consider Figure 4 that lists all the subgraphs of f_{19}, f_{20}, and parts of f_{21} (Figure 3). As all the subgraphs of f_{19} are frequent fragments, f_{19} is a discriminative infrequent fragment. Observe that f_{19} (as an infrequent fragment) is a subgraph of f_{20} and f_{21}. Hence, f_{20} and f_{21} are general infrequent fragments but not discriminative. Due to the same reasons, f_{16}, f_{17}, and f_{18} are discriminative infrequent fragments. In the sequel, we shall refer to discriminative infrequent fragment and discriminative fragment interchangeably. We denote a set of discriminative fragments in a database $\mathcal{D}$ as $\mathcal{I}_d$. It is easy to observe that an infrequent fragment may contain more than one discriminative fragments. We refer to the one with the smallest size as *minimal discriminative fragment*.

Definition 3. (**Minimal Discriminative Fragment (MDF)**) *Let $g \in \mathcal{I}$, $g' \subset g$, and $g' \in \mathcal{I}_d$. If $\nexists g'' \in \mathcal{I}_d$ such that $|g'| > |g''|$ and $g'' \subset g$, then g' is the minimal discriminative fragment of g, denoted as $mdf(g)$.*

For example, consider the infrequent fragment f_{21}. Observe that it contains three discriminative fragments (f_{16}, f_{18}, and f_{19}). Since f_{16} is the smallest among them, it is the MDF of f_{21}. Next, we discuss some of the key characteristics of infrequent fragments that we shall exploit subsequently.

If a graph g contains a discriminative fragment as its subgraph, then g must be an infrequent fragment. Formally,

LEMMA 1. *Let $g' \in \mathcal{I}_d$ and $g \in \mathcal{D}$. If $g' \subset g$ then $g \in \mathcal{I}$.*

PROOF. Since $g' \in \mathcal{I}_d$, $|\mathcal{D}_{g'}| < freq(\mathcal{D})$. Also as $g' \subset g$, $|\mathcal{D}_g| \leq |\mathcal{D}_{g'}|$. Therefore, $|\mathcal{D}_g| \leq |\mathcal{D}_{g'}| < freq(\mathcal{D})$. Hence, g is an infrequent fragment. $\Box$

On the other hand, if a graph is an infrequent fragment, then it must contain at least one discriminative fragment.

LEMMA 2. *Given $g \in \mathcal{I}$, $\exists g' \in \mathcal{I}_d$ such that $g' \subseteq g$.*

PROOF. If $\nexists$ ($g' \subset g$ and $g' \in \mathcal{I}_d$), then $sub(g) \subset \mathcal{F}$. Consequently, g itself is a discriminative fragment (By Definition 2). Hence, the above lemma holds. $\Box$

Next, we define the notion of *largest subgraph.* Let $g' \subset g$ and $|g'| = |g| - 1$. Then, g' is the *largest subgraph* of g. Note that g can have more than one largest subgraphs. We denote a set of largest subgraphs of g as $Lsub(g)$. If all the largest subgraphs of an infrequent fragment are frequent fragments, then the infrequent fragment must be a discriminative fragment.

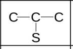
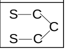
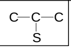
Id	f0	f1	f2	f3	f5	f6	f19	f16	f18
Sub(f21)	C—C	C—S	C—C—C	C—C—S	C—C—C—C	C—C—C—S		C—S—C	
Sub(f20)	C—C	C—S	C—C—C	C—C—S	C—C—C—C	C—C—C—S			
Sub(f19)	C—C	C—S	C—C—C	C—C—S					

Figure 4: Subgraphs of infrequent fragments.

THEOREM 1. *Given $g \in \mathcal{I}$, if $Lsub(g) \subset \mathcal{F}$, then $g \in \mathcal{I}_d$.*

PROOF. Assume that $g \in \mathcal{I}$ but $g \notin \mathcal{I}_d$. Then, $\exists g' \subset g, g' \in \mathcal{I}_d$ (based on Lemma 2). Hence, $\exists g'' \in Lsub(g)$, such that $g' \subseteq g''$. Therefore, $g'' \in \mathcal{I}$ (based on Lemma 1), which contradicts $Lsub(g) \subset \mathcal{F}$. Therefore, g is a discriminative fragment. $\Box$

As we shall see later, Theorem 1 can be used for fast identification of discriminative fragment. Based on the above discussion, it follows that if one of the subgraphs of g is a discriminative fragment, g is an infrequent fragment. Therefore, a discriminative fragment plays a central role in the formation of infrequent fragment and can be used in turn to identify an infrequent fragment. In practice, the number of discriminative fragments is significantly smaller than the total number of infrequent fragments, as will be demonstrated in Section 5.

2.3 Visual Interface of GBLENDER

Figure 1 depicts the screenshot of the visual interface of GBLENDER. It consists of four main panels. A user begins formulating a query by choosing a database as the query target and creating a new query canvas by clicking on the buttons in the Toolbar (Panel 1). The left panel (Panel 2) displays the unique labels of nodes that appear in the dataset in lexicographic order. In the query formulation process, the user chooses labels from this panel for creating the nodes in the query graph. The *Visual Query Designer* panel (Panel 3) depicts the area for formulating graph queries. A user drags a node that is part of the query from Panel 1 and drops it in Panel 3. Next, he/she adds another node in the same way. Then, she creates an edge between the added nodes by left and right clicking on them. Additional nodes and edges are added to the query graph by repeating these steps. Finally, the user can execute the query by clicking on the "Run" icon in the *Query Toolbar*. The *Results Window* (Panel 4) displays the query results.

3. ACTION-AWARE INDEXING

In this section, we present two indexing schemes, namely *action-aware frequent index* (A^2F) and *action-aware infrequent index* (A^2I), to support efficient matching of frequent and infrequent query fragments, respectively, while formulating a visual query graph. Our indexing schemes are *user action-aware*. That is, the structure of the index is designed to take advantage of typical actions a user undertakes in order to formulate a visual graph query. We begin by identifying the key features of such an *action-aware* index. In the sequel, we assume that frequent fragments are mined from the database using an existing technique e.g., *gSpan* [16].

3.1 Key Features of Action-Aware Index

A visual graph query can be formulated in different ways by following different sequences of GUI actions. Figure 5 shows two different sequences of visual actions (also referred to as *steps*), denoted by *Sequence 1* and *Sequence 2*, a user may undertake to formulate the query in Figure 1. We can make the following observations related to the query formulation process.

- Visual query formulation in GBLENDER follows a "node/edge-at-a-time" approach where a user incrementally adds new nodes or edges in the *visual query designer* panel (Panel 3 in Figure 1). Consequently, after every step the size of the query fragment grows by one. Recall that in this paper we focus on *error-oblivious* query graphs where a user correctly formulates the queries. Hence, deletion of edges due to mistakes committed by a user is beyond the scope of this work. Also, observe that the structure of the query fragment can evolve from a path to a tree or graph.

- At any step, the partial query graph formulated thus far, is either a frequent or infrequent fragment. Typically, as more edges are added, the chance of a query to remain frequent diminishes. Once it becomes infrequent, it remains as infrequent for rest of the formulation steps. For instance, in Figure 5 the partial query evolved from a frequent fragment to an infrequent one after Step 6 in *Sequence 1* whereas it becomes infrequent after the second step in *Sequence 2*.

In our proposed paradigm of blending visual query formulation and query processing, it is important to filter negative results after every visual action taken by a user. Consequently, we need an efficient indexing scheme which can exploit the above visual interaction characteristics effectively to prune false results. We envisage that such an *action-aware* indexing scheme should support the following key features: (a) It should be able to prune a part of irrelevant results even if only *partial* query graph is known during query formulation. (b) Since the size of a partial query graph g' grows by one, given a list of graphs that satisfy the fragment g' in *Step* i, it is important to support efficient strategy for identifying the graphs that match the fragment g'' (generated at *Step* $i+1$) where $g' \subset g''$ and $|g''| = |g'| + 1$. (c) A partial query graph may evolve from being a frequent fragment to an infrequent one in the database. Furthermore, it may also evolve from a simple path to a complex graph structure. Hence, the proposed strategy should be able to support pruning based on both graph-structured frequent and infrequent fragments. (d) Since smaller fragments always appear more often in different visual queries compared to larger-sized fragments, smaller-sized graph fragments should be efficiently indexed to support fast retrieval. (e) Lastly, since subgraph isomorphism testing is known to be NP-complete [4], the indexing scheme should minimize expensive candidate verification while retrieving partial results.

3.2 Why Existing Strategies Cannot be Used?

While state-of-the-art indexing strategies are certainly innovative and powerful, we found out that they cannot be directly adopted for efficiently blending visual query formulation and processing for the following reasons. Firstly, these schemes are based on the conventional paradigm that the *entire* query graph must be available *before* query processing. However, in our proposed paradigm query processing is initiated as soon as a fragment of the query graph is visually formulated. For instance, *gIndex* [17] uses *apriori*-like strategy to enumerate a set of fragments of the query by checking whether a fragment belongs to the underlying *frequent subgraph index*. In order to generate this fragment set, the entire query graph should be available. TreePi [19] indexes frequent and discriminative trees and adopts a new pruning technique based on the concept of Center Distance Constraints (CDC). The basic idea is that if the query graph appears in a candidate graph, distances between pairs of features in query graph must be preserved in the candidate graph as well. This distance computation requires the availability of the entire query graph. $(Tree + \Delta)$ [20] enumerates all frequent subtrees in a query graph q and computes the candidate answer set. If q is a non-tree cyclic graph, then it obtains a set of *discriminative graph features* to generate the candidate answer set. Observe

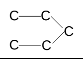
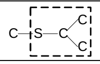
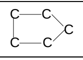
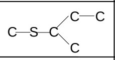
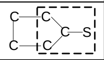
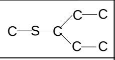
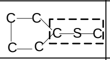
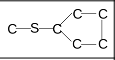
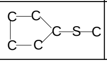
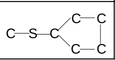
Steps	Sequence 1			Sequence 2		
	Action	Current Graph	Status	Action	Current Graph	Status
Step 1	C—C	C—C	frequent	C—S	C—S	frequent
Step 2	C—C	C—C—C	frequent	S—C	C—S—C	Infrequent & prefetch
Step 3	C—C	C—C—C—C	frequent	C—C	C—S—C—C	infrequent
Step 4	C—C		frequent	C—C		Infrequent & prefetch
Step 5	C—C		frequent	C—C		infrequent
Step 6	C—S		Infrequent & prefetch	C—C		infrequent
Step 7	S—C		Infrequent & prefetch	C—C		infrequent
Step 8	Click RUN		verification	Click RUN		verification

Figure 5: Two different sequences of query formulation steps.

that the entire query graph must be available to enumerate frequent subtrees as well as to determine if q is cyclic. CDIndex [14] is only suitable for graphs with limited sizes, as it exhaustively enumerates and indexes all the subgraphs in the database. It is not supportive to our paradigm not only due to the size limitation but also due to the requirement of expensive multiple subgraph isomorphism test for partial query graph fragments.

GString [9] is a semantic-based approach to index chemical compound databases. It converts a graph into its string representation and then uses suffix tree-based indexing scheme. A query graph is converted into a *GString* and *summary string*, and then matched against the suffix-tree. The generation of *GString* and summary string, requires availability of the entire query graph. C-tree [5] is a clustering-based index to support both subgraph queries and similarity queries. The graph closure is a "bounding box" containing structural information of the constituent graphs. A subgraph query is processed in two phases. First, candidate answer set is generated by traversing the C-tree and pruning nodes using an approximate subgraph isomorphism technique called *pseudo subgraph isomorphism*. Next, each candidate answer is verified for exact subgraph isomorphism. C-tree is not designed for repeated processing of partial query fragments as it will result in multiple pseudo subgraph isomorphism test.

Secondly, a key feature of action-aware indexing scheme is that it should be able to exploit both frequent and infrequent subgraph fragments to prune false results. However, very few existing techniques support both types of fragments. For example, *gIndex* [17] only indexes discriminative frequent graphs and assumes that they are most likely to appear in query graphs. This assumption may not hold for many applications as a user may submit various queries with arbitrary structures. TreePi [19] and $(Tree + \Delta)$ [20] index frequent and discriminative subtrees rather than subgraphs, as trees can be manipulated efficiently. FG-Index [2] uses frequent subgraphs as index features. Frequent graph queries are answered without verification and infrequent queries require only a small number of verifications. While it supports infrequent edges, it does not support infrequent graphs. C-tree and GString do not support frequent or infrequent fragments as indexing features.

Lastly, since the above indexes are designed for conventional subgraph matching paradigm, they do not require to support efficient traversal and retrieval of graph fragments g and g' where $|g| = |g'| + 1$. For example, FG-Index [2] indexes only individual subgraphs such that given a query q, at best it can directly retrieve

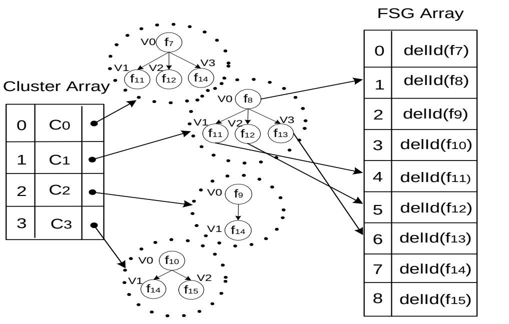


Figure 6: An example of DF-index.

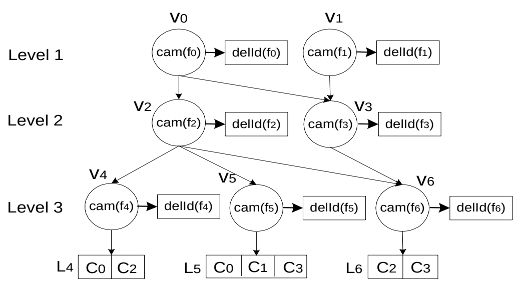

V0 V1
Level 1
cam(f0) delId(f0) cam(f1) delId(f1)
V2 V3
Level 2
cam(f2) delId(f2) cam(f3) delId(f3)
V4 V5 V6
Level 3
cam(f4) delId(f4) cam(f5) delId(f5) cam(f6) delId(f6)
L4 C0 C2
L5 C0 C1 C3
L6 C2 C3

Figure 7: An example of MF-index.

the result for q if q is indexed, but it must start the search from the beginning if $q \cup \{e\}$ needs to be matched. In other words, the search cannot move to $q \cup \{e\}$ directly from q. Further, relatively more efficient pruning of smaller-sized frequent fragments compared to larger-sized fragments is also not an important requirement for existing approaches.

3.3 Action-Aware Frequent (A^2F) Index

A challenge in creating an index for frequent fragments is that the frequent fragment set can be large for a small α and hence the index built on the frequent fragments can be too large to fit in the main memory. Then, the performance of repeated evaluation of partial query fragments may degrade as the processing needs frequent disk access. To address this issue, similar to FG-Index [2], we create a *memory-resident* and a *disk-resident* components of A^{2}F index. We refer to them as *memory-based frequent index* (MF-index) and *disk-based frequent index* (DF-index), respectively.

How do we determine which frequent fragment should reside where? To answer this question, we take a different strategy compared to FG-Index by exploiting a key feature of visual query formulation. Recall that the construction of visual queries always grows incrementally from small to larger-sized query fragments. Consequently, smaller frequent fragments are processed more frequently in various visual queries compared to their larger counterparts. We exploit this feature to determine where a frequent fragment should reside. Specifically, small-sized frequent fragments (frequently utilized) are stored in MF-index whereas larger frequent fragments (less frequently utilized) reside in DF-index. Formally, let $\beta \geq 1$ be the *fragment size threshold*. If $g \in \mathcal{F}$ and $|g| \leq \beta$, then index g into the MF-index. Otherwise, index g into the DF-index. Note that the sizes of MF-index and DF-index can be tuned by adjusting β based on the average size of typical queries and availability of memory. For instance, when β is the maximal size of frequent fragments, all the frequent fragments are indexed in MF-index. Even though it is faster to match frequent fragments in MF-index, it occupies larger memory space. In contrast, if β is too small, most of the frequent fragments are indexed in the DF-index and query processing needs to frequently access the disk. We shall empirically study the effect of β on query processing in Section 5. We now elaborate on the structure of these two types of frequent index.

3.3.1 Disk-Based A^2F Index (DF-Index)

Informally, DF-index is an array of *fragment clusters*. A *fragment cluster* is a directed graph $\mathcal{C} = (V_C, E_C)$ where each node $v \in V_C$ is a frequent fragment g where $|g| > \beta$. There is an edge $(v', v) \in E_C$ *iff* $g' \subset g$ and $|g| = |g'|+1$. We denote the root node (node with no incoming edge) of $\mathcal{C}$ as $root(\mathcal{C})$. Each fragment g of v is represented by its CAM code [7], denoted as $cam(g)$. We choose the maximal code among all possible codes of a graph by lexicographic order as this graph's canonical code.

Each node with fragment g in $\mathcal{C}$ points to a set of FSG identifiers of g (denoted as $delId(g)$ where $delId(g) \subseteq fsgId(g)$). Note that it is not space efficient to attach the complete list of FSG identifiers of g on each frequent fragment in the index as the size can be large when α is close to 1. Fortunately, the following property holds: given $g, g' \in \mathcal{F}$, if $g' \subset g$ then $fsgId(g) \cap fsgId(g') = fsgId(g)$ [2]. That is, node v' (representing g') and its child node v (representing g) shares a large number of FSGs. We exploit this property to make the index more space-efficient. We elaborate on this with a simple example. Figure 6 depicts an example of DF-index ($\beta = 3$) based on the frequent fragments in Figure 3. In the fragment cluster $\mathcal{C}_2$, we assign those FSG ids to f_9 that are not in f_{14}. Since $fsgId(f_{14}) \subset fsgId(f_9)$, $|delId(f_9)| = |fsgId(f_9)| - |fsgId(f_{14})| = 200$. For the leaf node f_{14}, $delId(f_{14}) = fsgId(f_{14})$. Also observe that we can retrieve the identifers of all FSGs of g by traversing all its children and adding them together. For instance, in the case of fragment cluster $\mathcal{C}_1$, $fsgId(f_8) = delId(f_8) \cup delId(f_{11}) \cup delId(f_{12}) \cup delId(f_{13})$.

Definition 4. (**DF-index**) *Given a set of frequent fragments $\mathcal{F}$ in a graph database $\mathcal{D}$ and fragment size threshold β, an* DF*-index constructed on $\mathcal{F}$ consists of the following components:*

- *An array, called Cluster Array (*CA*), stores a collection of fragment clusters. Let $CA[i]$ be the i-th entry in the* CA*. The fragment cluster stored in $CA[i]$ is assigned an identifier $\mathcal{C}_i$.*
- *A fragment cluster $\mathcal{C}_i$ is a graph $\mathcal{C}_i = (V_{C_i}, E_{C_i})$ where $v \in V_{C_i}$ represents a frequent fragment $g \in \mathcal{F}$ such that $|g| > \beta$. Each $(v', v) \in E_{C_i}$ represents the parent-child relationship between two vertexes. Fragment g is the child of g' iff $g' \subset g$ and $|g| = |g'| + 1$.*
- *An array, called* FSG *Array (*FA*), stores $delId$ list of distinct frequent fragments in* CA.
- *$\forall\, v \in V_{C_i}$ and $\forall i$, $v = (cam(g), j))$ where $cam(g)$ is the* CAM *code of g and $FA[j]$ contains $delId(g)$.*

3.3.2 Main Memory-Based A^2F Index (MF-Index)

The MF-index indexes all frequent fragments having size less than or equal to β. Similar to a fragment cluster, it is a directed graph $G_M = (V_M, E_M)$ where the nodes and edges have same semantics as $\mathcal{C}$. In addition, nodes representing frequent fragments of size β are leaf nodes in G_M and do not have any child fragments. Each leaf node $v \in V_M$ representing a fragment g, is additionally associated with a *fragment cluster list* $\mathcal{L}$ where each entry $\mathcal{L}_i$ points to a fragment cluster $\mathcal{C}_j$ in the DF-index such that $g \subset root(\mathcal{C}_j)$.

Definition 5. (**MF-index**) *Given a set of frequent fragments $\mathcal{F}$ in a graph database $\mathcal{D}$ and fragment size threshold β, an* MF*-index constructed on $\mathcal{F}$ is a graph $G_M = (V_M, E_M)$ where $v \in V_M$ represents a frequent fragment $g \in \mathcal{F}$ and satisfies the following conditions.*

Algorithm 1 ***BuildA2FIndex***

```
Input: A set of frequent fragments $\mathcal{F}$, fragment size threshold $\beta$
Output: MF-index and DF-index
 1: Sort $\mathcal{F}$ by size ascending order
 2: Index each $|g| = 1, g \in \mathcal{F}$ in MF-index
 3: for $g_i \in$ A$^2$F-index, $g_j \in \mathcal{F}$ do
 4:   if $g_i \subset g_j$ and $|g_j| = |g_i| + 1$ then
 5:     if $|g_i| = \beta$ then
 6:       if $g_j \notin$DF-index then
 7:         Index $g_j$ in $\mathcal{C}_k$, $k$++
 8:         Insert $\mathcal{C}_k$ in DF-index
 9:       end if
10:       Add $g_j$'s fragment cluster id in $g_i.\mathcal{L}$
11:     else
12:       if $|g_i| > \beta$ then
13:         Index $g_j$ in the same fragment cluster as $g_i$
14:       else
15:         Index $g_j$ in MF-index
16:       end if
17:       Connect $g_i$ and $g_j$ with an edge
18:       $delId(g_i) = delId(g_i) - fsgId(g_j)$
19:     end if
20:   end if
21: end for
```

- *for each* $v \in V_M$, $|g| \leq \beta$.
- *if v is not a leaf node then $v = (cam(g), delId(g))$ where $cam(g)$ is the* CAM *code of g and $delId(g)$ is a list of* FSG *identifiers of g s.t. $delId(g) \subset fsgId(g)$.*
- *if v is a leaf node then $v = (cam(g), delId(g), \mathcal{L})$ where $\mathcal{L}$ is a list of fragment cluster identifiers of g and $delId(g) = fsgId(g)$. Let $\mathcal{L}_i$ be the i-th entry of $\mathcal{L}$. Then, $\mathcal{L}_i$ contains an index j of* CA *such that $CA[j] = \mathcal{C}_j$ and $g \subset root(\mathcal{C}_j)$.*
- *Each $(v', v) \in E_M$ represents the parent-child relationship between two vertexes. Fragment g is the child of g' iff $g' \subset g$ and $|g| = |g'| + 1$.*

EXAMPLE 1. Figures 6 and 7 depict DF-index and MF-index, respectively, built based on the fragments listed in Figure 3 and $\beta = 3$. The fragments f_0 and f_1 are chosen as the root nodes in the MF-index as they have the least size ($|f_0| = |f_1| = 1$). Since f_2 and f_3 are supergraphs of f_0 and f_1 with one additional edge, they are connected to f_0, f_1 as their children, respectively. Similarly, f_4, f_5, and f_6 are inserted into the MF-index. Since the sizes of these fragments are 3, they are leaf nodes in the MF-index (Figure 7).

Next, we create a set of fragment clusters for each leaf node in the MF-index and insert them into the Cluster Array of DF-index (Figure 6). Since f_7 is the child of f_4 with size 4, we create a fragment cluster, denoted as $\mathcal{C}_0$, containing f_7 and it's children f_{11}, f_{12}, and f_{14}. Note that $root(\mathcal{C}_0) = f_7$. $\mathcal{C}_0$ is added to f_4's cluster list $\mathcal{L}_4$. We also add $delId(f_7)$, $delId(f_{11})$, $delId(f_{12})$, and $delId(f_{14})$ in the array FA of the DF-index. Similarly, we build the fragment clusters $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ and add them in CA.

3.3.3 *Algorithm for Building A^2F Index*

Algorithm 1 shows the top-down approach of the A^{2}F-index construction. Firstly, the frequent fragments are sorted in ascending order based on their size (Line 1). All frequent edges are indexed in the MF-index (Line 2). Given $g_i \in$ A^{2}F-index, $g_j \in \mathcal{F}$, if $g_i \subset g_j$ and g_j has one more edge than g_i, then g_j is a child of g_i (Line 4). Note that the cost of subgraph isomorphism test here is not significant as the frequent fragments are already sorted by their size. Consequently, for a given frequent fragment we just need to check only those fragments that have one additional edge. If $|g_i| = \beta$, g_i is a leaf node in the MF-index. Consequently, g_j should reside

Algorithm 2 ***BuildA2IIndex***

```
Input: $\mathcal{F}, \mathcal{D}$
Output: A$^2$I-index
 1: Get $\mathcal{I}_1$ from $\mathcal{D}$ to A$^2$I-index
 2: for $g_i \in \mathcal{F}, e_j \in \mathcal{F}_1$ do
 3:   $g_{new} = g_i + e_j$
 4:   if $g_{new} \in \mathcal{I}_d$ and $g_{new} \notin$ DFA then
 5:     $fsgId(g_{new}) \longleftarrow$ Retrieve FSG identifiers of $g_{new}$
 6:     if $|fsgId(g_{new})| > 0$ then
 7:       Add $g_{new}$ in A$^2$I-index with $fsgId(g_{new})$
 8:     end if
 9:   end if
10: end for
```

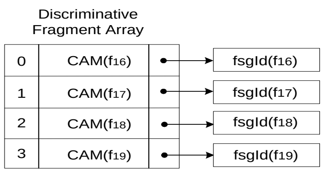


Figure 8: An example of A^{2}I index.

in the DF-index. If $g_j \notin$DF-index, the algorithm indexes g_j as the root of fragment cluster $\mathcal{C}_k$ and insert $\mathcal{C}_k$ in the cluster array of DF-index (Lines 6-9). It inserts g_j's fragment cluster id in g_i's fragment cluster list (Line 10). If $|g_i| > \beta$, then g_j is indexed in the same fragment cluster as g_i. Otherwise, it indexes g_j in the MF-index (Lines 12-16). Then it connects g_i and g_j with an edge and updates g_i's FSG id entries by deleting g_j's FSG ids (Lines 17-18). This process is repeated until all the frequent fragments are indexed.

3.4 Action-Aware Infrequent (A^2I) Index

The A^{2}I-index indexes infrequent fragments to prune the candidate space for infrequent queries. In order to ensure that the index is space-efficient, we index only the discriminative infrequent fragments $\mathcal{I}_d$ instead of infrequent fragments $\mathcal{I}$ as often in practice $|\mathcal{I}_d| \ll |\mathcal{I}|$. Except for FG-Index, none of the existing feature-based graph indexing schemes index infrequent fragments. FG-Index builds an index only for *infrequent edges* (infrequent fragments with only one edge). Consequently during query formulation, if an infrequent query fragment is complex-structured, then FG-Index is not very effective in reducing candidate space by pruning negative results. For example, consider Figure 3. The subgraphs of the infrequent fragment f_{21} with support 200 are listed in Figure 4. Observe that none of the subgraphs of f_{21} is an infrequent edge. As the candidate pruning strategy of FG-Index exploits only the frequent fragments and infrequent edges, the candidate space of f_{21} can only be reduced to 1000 by its largest frequent fragment f_{15}. However, it is possible to adapt the indexing scheme so that the candidate space can be reduced to 300 by its infrequent subgraph f_{18}. It is worth mentioning that reduction of candidate space reduces the number of subgraph isomorphism tests.

We now describe the structure of A^{2}I-index designed specifically to address the above issue by indexing infrequent fragments. Intuitively, it consists of an array of discriminative fragments (denoted as DFA) arranged in ascending order of their sizes. Each entry in DFA stores the CAM code of $g \in \mathcal{I}_d$ and a list of FSG identifiers of g ($fsgId(g)$). Figure 8 depicts an example of A^{2}I-index constructed using the discriminative fragments in Figure 3. For instance, $cam(f_{16})$ is stored in $DFA[0]$. Also, $DFA[0]$ has a pointer to the list of FSG identifiers of f_{16}. Note that as the support of each discriminative infrequent fragment is less than $\alpha|\mathcal{D}|$, it is possible to store A^{2}I-index in the memory (see Section 5).

Definition 6. (**A^{2}I-index**) *Given a set of discriminative infrequent fragments $\mathcal{I}_d$ in a graph database $\mathcal{D}$, an* A^{2}I*-index con-*

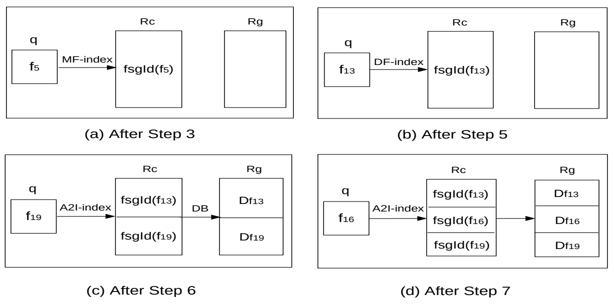


Figure 9: Candidates maintenance during query evaluation.

*structed on $\mathcal{I}_d$ consists of an array, called Discriminative Fragment Array (*DFA*), which stores $\mathcal{I}_d$. Let $DFA[i]$ be the i-th entry in the* DFA*. Then, $DFA[i] = (cam(g_k), fsgId(g_k))$ where $g_k \in \mathcal{I}_d$. Further, if $i < j$ are indexes of* DFA *then $|g_i| \le |g_j|$.*

3.4.1 Algorithm for Building A^2I Index

The algorithm for building A^{2}I-index is shown in Algorithm 2. We denote the sets of infrequent and frequent fragments with only one edge as $\mathcal{I}_1$ and $\mathcal{F}_1$, respectively. Firstly, we retrieve $\mathcal{I}_1$ from $\mathcal{D}$ and index them in the A^{2}I-index (Line 1). Next, we add a frequent edge e_j on the frequent fragment g_i to form a new graph g_{new} (Lines 2-3). Obviously, there are different ways to construct such a new graph by adding e_j to different nodes of g_i. We shall elaborate on this later. If g_{new} is not a frequent fragment and does not exist in DFA, then it checks if g_{new} is a discriminative fragment using Theorem 1 (Line 4). The algorithm retrieves the identifiers of FSGs of g_{new} from $fsdId(g_i)$ using subgraph isomorphism test (Line 5). If g_{new} exists, then it adds g_{new} and $fsgId(g_{new})$ in the A^{2}I-index (Lines 6-7). The algorithm repeats this process until no new fragment is generated.

Observe that there are two ways to add a frequent edge on a frequent fragment: (a) connect a new node on a node of the frequent fragment. (b) connect two existing nodes of a frequent fragment without introducing a new node. Let $g \in \mathcal{F}$ has n nodes and K be the number of frequent edges in $\mathcal{F}$. Let $|g_{new}|$ be the number of newly generated graphs of g. Then the largest possible value of $|g_{new}|$ is as follows: $|g_{new}| = Kn + (max(|g|) - min(|g|))$. The computational complexity to evaluate this equation is $O(n^2)$. We remove the frequent fragments, newly generated fragments that do not exist in the database, and existing discriminative fragments from the newly generated graphs at each step. Furthermore, as subgraph isomorphism test is used to retrieve $fsgId(g_{new})$ (Line 5), it is important to reduce the size of FSG space. We achieve this by considering only the FSGs in $\mathcal{D}_{g_i}$ instead of $\mathcal{D}$.

4. ACTION-AWARE QUERY PROCESSING

We now discuss how the action-aware indexes proposed in the preceding section can facilitate blending of query formulation and processing. In the sequel, we assume that a user *does not commit any errors* while formulating a visual query (no deletion of edges). Since a visual query is formulated step-by-step by adding edges incrementally, our proposed action-aware query processing algorithm, called GBLENDER, utilizes the latency offered by the GUI actions to retrieve partial results.

Algorithm GBLENDER is shown in Algorithm 3. The visual query q to be formulated, initially empty, is initialized as frequent in Line 1. Let R_c and R_g represent sets of candidate FSG identifiers and candidate graphs, respectively. When the GUI action adds a new edge on q, then if q's state is frequent, it matches q in MF-index or DF-index (Line 4-5). This step is encapsulated by the

Algorithm 3 GBLENDER

```
Input: A GUI action $Action$, A$^2$F-index, A$^2$I-index, and $\mathcal{D}$.
Output: $Results$ of the visual query fragment.
 1: $q.fr$=true
 2: if $Action$ is $e$ then
 3:     $q = q + e$
 4:     if $q.fr$ is $true$ then
 5:         $R_c \leftarrow$ FrequentFragment($q$, A$^2$F-index) /* Algorithm 4 */
 6:     end if
 7:     if $q.fr$ is $false$ then
 8:         $R_c \leftarrow$ InfrequentFragment($q$, A$^2$I-index) /* Algorithm 5 */
 9:         $R_g \leftarrow$PrefetchCandGraphs($R_c$, $\mathcal{D}$) /* Algorithm 6 */
10:     end if
11: else
12:     if $Action$ is $Run$ then
13:         if $q \in$ A$^2$F-index or A$^2$I-index then
14:             $Results = R_c$
15:         else
16:             $Results \leftarrow$ Verify($q$, $R_g$)
17:         end if
18:     end if
19: end if
```

Algorithm 4 *FrequentFragment*

```
Input: Query fragment $q$, set of candidate FSG ids $R_c$, A$^2$F-index
Output: Updated $R_c$
 1: if $|q| \le \beta$ then
 2:     $fsgId(q) \leftarrow$ Search($q$, MF-index)
 3:     if $fsgId(q) \ne \emptyset$ then
 4:         $R_c$=$fsgId(q) \cap R_c$
 5:     else
 6:         $q.fr = false$
 7:     end if
 8: else
 9:     $fsgId(q) \leftarrow$ Search($q$, DF-index)
10:     if $fsgId(q) \ne \emptyset$ then
11:         $R_c$=$fsgId(q) \cap R_c$
12:     else
13:         $q.fr = false$
14:     end if
15: end if
```

FrequentFragment procedure which we shall elaborate later. If q is an infrequent fragment, the algorithm invokes *InfrequentFragment* procedure to retrieve R_c (Line 8). The candidate graphs (R_g) satisfying R_c are fetched from $\mathcal{D}$ and q is set as an infrequent fragment (Line 9). If the GUI action is clicking the "Run" icon (Line 12), then if the current q is a frequent one or a discriminative fragment, the algorithm returns exact results from R_c (Lines 13-14). Otherwise the query is a "non-discriminative" infrequent fragment, and it returns results by verifying candidates in R_g (Line 16). Here we use commonly used Ullman's algorithm [15] for subgraph isomorphism test. Note that we can easily replace this step with a more efficient subgraph testing algorithm such as the one described in [12]. We now elaborate on the *FrequentFragment*, *InfrequentFragment*, and *PrefetchCandGraphs* procedures in detail.

***FrequentFragment* procedure.** Algorithm 4 shows the *FrequentFragment* procedure. If $|q| \le \beta$, then it searches q in the MF-index (Lines 1-2). Firstly, it transforms q into its CAM code and then it performs the graph isomorphism test by comparing the CAM code of q with those in the MF-index. Two graphs g and g' are isomorphic to each other, if and only if $cam(g) = cam(g')$ [7]. These steps are encapsulated by the *Search* procedure in Line 2. If q can be found in the MF-index, then the algorithm retrieves its FSG identifiers $fsgId(q)$ and updates R_c (Lines 3-4). Otherwise, q is identified as an infrequent fragment (Line 6). If $|q| > \beta$, then the algorithm executes the same steps in the DF-index (Lines 9-14). Note that the time complexity of this algorithm is $O(n)$ where n is the number of frequent fragments with size $|q|$.

EXAMPLE 2. Consider *Sequence 1* in Figure 5 to formulate the visual query in Figure 1. Figure 9 depicts the states of R_g and R_c during different steps. After Step 1, q matches f_0 in the MF-index. As a result, $fsgId(f_0)$ is assigned to R_c. After Step 2, the algorithm searches the new q among the children of f_0 (f_2 and f_3) in the MF-index. Since only f_2 matches q, $fsgId(f_2)$ replaces $fsg(f_0)$ in R_c. After Step 3, f_5 matches the new q and as a result $fsgId(f_5)$ replaces $fsgId(f_2)$ in R_c (Figure 9(a)).

After Step 4, since f_5 is a leaf node in the MF-index, no matched fragment can now be located in the MF-index. Hence, the algorithm searches the list of fragment cluster identifiers $\mathcal{L}$ of f_5 ($\mathcal{C}_0$, $\mathcal{C}_1$, and $\mathcal{C}_3$). Since f_8 in $\mathcal{C}_1$ matches q (Figure 6), $\mathcal{C}_1$ is retrieved from the DF-index to the memory. Then $fsgId(f_8)$ is computed from FA in the DF-index, which replaces $fsgId(f_5)$ in R_c. After Step 5, the new q is matched among the children of f_8 and f_{13} is selected. Consequently, $fsgId(f_{13})$ is assigned to R_c to replace $fsgId(f_8)$ (Figure 9(b)). So far the query graph has remained as a frequent fragment. Since we do not need to fetch candidates from $\mathcal{D}$ for frequent fragments, R_g has remained empty during this process. If the user clicks on the "Run" icon now, then the results of q is R_c.

***InfrequentFragment* procedure.** Algorithm 5 shows the procedure for evaluating infrequent fragments. Recall that there is at least one discriminative fragment in q (Lemma 2). Therefore, the algorithm finds the MDF of q, which can match an element in the A^{2}I-index. Firstly, it retrieves each subgraph (denoted as q_s) of q with n edges which is also a supergraph of the new edge e (Lines 1-2). Then, for each such q_s, it checks if there is a match in the A^{2}I-index (using the *Search* procedure). If there is, then q_s is $mdf(q)$. Consequently, it updates R_c by intersecting it with $fsgId(q_s)$ (Lines 2-5). Otherwise, the algorithm continues to search for $mdf(q)$ with $n+1$ edges by repeating the above operations.

Theoretically, searching for $mdf(q)$ has exponential complexity. Fortunately, in practice this does not adversely effect the performance of GBLENDER for two main reasons. Firstly, in practice the value of n is small as typically a user does not visually formulate very large queries. Secondly, once one $mdf(q)$ is found, we do not need to search for others any more as this MDF is sufficient to reduce the candidate space below $\alpha|\mathcal{D}|$.

EXAMPLE 3. Reconsider Example 2. The query is a frequent query in the first five steps. After Step 6, no matched frequent fragment can be found in the A^{2}F-index. Hence, q has evolved into an infrequent fragment. The algorithm now searches for a q_s which is an $mdf(q)$. In this case, it finds that $q_s = f_{19}$ which is matched in the A^{2}I-index. Consequently, R_c is updated as follows: R_c=$fsgId(f_{13}) \cap fsgId(f_{19})$ (Figure 9(c)). Next, the algorithm fetches the candidate graphs having ids in R_c from $\mathcal{D}$ and assigns them to R_g. After Step 7, as $q_s = f_{16}$ is $mdf(q)$, R_c is updated again (Figure 9(d)). The final candidate space can be calculated as following:

$$\begin{aligned}|R_c| &= |fsgId(f_{13}) \cap fsgId(f_{16}) \cap fsgId(f_{19})| \\ &\leq Min(|fsgId(f_{13})|, |fsgId(f_{16})|, |fsgId(f_{19})|) \\ &= |fsgId(f_{19})| = 300\end{aligned}$$

***PrefetchCandGraphs* procedure.** Algorithm 6 outlines the procedure for prefetching candidate graphs from the database for the subgraph isomorphism testing. Recall that prefetching is only necessary when the current query fragment is infrequent. Note that as the current query contains one infrequent fragment as subgraph,

Algorithm 5 ***InfrequentFragment***

```
Input: Infrequent query fragment $q$, $R_c$, and A$^2$I-index
Output: Updated $R_c$
1: for $n$=1 ... $|q|$ do
2:   for each $q_s$ s.t. $|q_s| = n$ and $q_s \subseteq q$ and $e \subseteq q_s$ do
3:     $fsgId(q_s) \leftarrow$ Search($q_s$, A$^2$I-index)
4:     if $fsgId(q_s) \neq \emptyset$ then
5:       $R_c$=$fsgId(q_s) \cap R_c$
6:       exit
7:     end if
8:   end for
9: end for
```

Algorithm 6 ***PrefetchCandGraphs***

```
Input: $R_c$, $R_g$, and $\mathcal{D}$
Output: $R_g$
1: if $R_g = \emptyset$ then
2:   $R_g \leftarrow$ Fetch($R_c, \mathcal{D}$) /* Retrieves from the database $\mathcal{D}$ */
3: else
4:   $R_g = R_g \cap R_c$
5: end if
```

the size of candidate results is definitely bounded by $\alpha|\mathcal{D}|$. In addition, the prefetching operation just needs to access the database only once during the entire infrequent query evaluation process. After that, as the query size grows, the new candidate set can be generated from the previous one (Line 4). The time complexity of this algorithm is $O(m)$ where m is the number of elements in R_c.

EXAMPLE 4. Reconsider Example 3. Since the current R_g is empty, candidate graphs are prefetched from the database based on the ids in R_c after Step 6 (Lines 1-2) as depicted in Figure 9(c). After Step 7, the new candidates are subset of the candidates in Step 6. Hence, R_g is updated by removing the graphs whose ids are not in R_c (Line 4) in Figure 9(d).

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of GBLENDER. The set of frequent fragments, which is used as an input to construct the action-aware indexes is mined using *gSpan* [16]. GBLENDER is implemented in Java JDK 1.6 and the results display component is implemented using *ZGRViewer* [11]. We run all experiments on an Intel Pentium4 3.4GHz machine with 3GB memory, running Linux 2.6.28 System.

To the best of our knowledge, there is no existing subgraph matching system that realizes our proposed paradigm. Hence, we compare GBLENDER (denoted by GBR for brevity) to the following state-of-the-art graph indexing methods: C-Tree [6] (denoted by CT), FG-Index [2] (denoted by FGI), and SSI [12][1]. CT is implemented in Java whereas both SSI and FGI are implemented in C++.

5.1 Experimental Setup

Datasets. We use the AIDS Antiviral dataset containing 43K graphs as real-world dataset. Note that this dataset has been used by several existing graph indexing schemes such as *gIndex* [17], FG-Index [2], C-Tree [5], and SSI [12]. The average size of a graph is 25 vertices and 27 edges. The maximum size of a graph is 222 vertices and 251 edges. There are 63 different types of atoms in the dataset. In accordance with [5, 12], we generate vertex-labeled graphs from the molecule structures (omitting Hydrogen atoms). Further, we remove the edge labels as it increases the chance of a fragment to be frequent.

[1]Despite our best efforts (including contacting the authors), we could not get the source code of the approaches discussed in [19, 20].

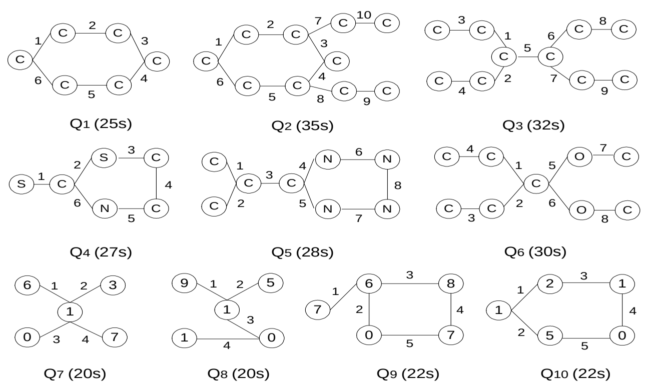


Figure 10: Queries on real and synthetic datasets.

Query	FGI		SSI		CT		GBR		$\|R_g\|$
	SRT	$\|R_c\|$	SRT	$\|R_c\|$	SRT	$\|R_c\|$	SRT	$\|R_c\|$	
Q_1	8	0	223	35241	11300	34736	0.002	0	32013
Q_2	27	0	324	14868	13300	20946	0.002	0	4280
Q_3	47	0	187	12836	11000	12904	0.005	0	8445
Q_4	76	NA	47	6927	6800	269	29	476	269
Q_5	260	NA	116	25601	6870	37	18	68	37
Q_6	200	NA	89	18252	7870	503	50	750	283

Table 1: Query performance on the AIDS dataset (in msec.)

To test the scalability of GBLENDER on database size, we use the synthetic graph dataset generator of FG-Index [2] (*Graphgen*) to generate five datasets with sizes from 20K to 100K. We set the number of distinct labels to 10. The average number of graph edges in each dataset is set to 30 and the average graph density is 0.1.

Querysets. We chose ten queries as shown in Figure 10. $Q_1 - Q_6$ are queries on the AIDS dataset whereas $Q_7 - Q_{10}$ are queries on the synthetic datasets. Since these queries are formulated by users using the visual interface, it is not realistic to expect a user to formulate large queries visually. Therefore, we chose query graphs whose sizes do not exceed 10. Note that GBLENDER can handle larger query graphs gracefully. Further, the labels on the edges of a query in Figure 10 represent the default sequence of steps for query formulation in GBLENDER. For example, in Q_4 the default sequence of steps for query formulation is: [(S,C), (C,S), (S,C), (C,C), (C,N), (N,C)]. Unless mentioned otherwise, we shall be using the default sequence for formulating a particular query.

Participants profile. In order to formulate visual queries, three unpaid male volunteers participated in the experiment. Participants ranged in ages from 21 to 27. All subjects have varying degree of familiarity with graph queries.

At the start, participants were trained to use the GUI of GBLENDER. For every query, the participants were given some time to determine the steps that are needed to formulate the query visually. This is to ensure that the effect of thinking time is minimized during query formulation. Note that faster a user formulates a query, the lesser time GBLENDER has for prefetching. The participants were given one query at a time. That is, only after the correct formulation of the current query, a participant was given the next query. If an error was committed by a participant then that particular formulation effort is ignored and he had to start afresh. Each query was formulated five times by each participant (using the default sequence unless specified otherwise) and reading of the first formulation of each query was ignored. The average query formulation time for a query by all participants is shown in parenthesis in Figure 10.

AIDS dataset size (K)	*1*	*5*	*10*	*20*	*40*
Index Size (MB)	0.025	0.128	0.306	0.724	1.55
Syn. dataset size (K)	*20*	*40*	*60*	*80*	*100*
Index Size (MB)	14.2	29.7	45	60.6	76

Table 2: Size of A^{2}I-index.

	GBR				FGI	CT	SSI
β	**3**	**5**	**8**	**11**			
Size	3.8+23.5	7.5+23	18.6+17.4	24.8+0	0.8+43.2	45	20.4

Table 3: Index size comparison (MB).

5.2 Performance on Real Graph Dataset

We discuss the performance of GBLENDER on the AIDS dataset from a variety of aspects. We set $\alpha = 0.1$ and $\beta = 8$ in all experiments unless specified otherwise. We follow the default settings of FGI, SSI, and CT as suggested in [2], [12], and [5], respectively. The largest size of the indexed frequent fragments is 10.

Candidate size and system response time (SRT). To evaluate the query performance of GBLENDER, we use the six queries (Q_1 - Q_6) on the AIDS dataset. Note that Q_1 - Q_3 are frequent query fragments and the remaining ones are infrequent queries. Table 1 shows the average *system response time* (SRT), the candidate size (denoted by $|R_c|$), and the size of result set (denoted by $|R_g|$). Recall that in GBLENDER, SRT refers to the duration between the time a user presses the "Run" icon to the time when the user gets the query results. The average SRT is computed by taking the average of the SRTs of all participants (last four formulations). In FGI, CT, and SSI, SRT refers to the execution time of a query. Each query was executed five times in each approach and the results from the first run were always discarded.

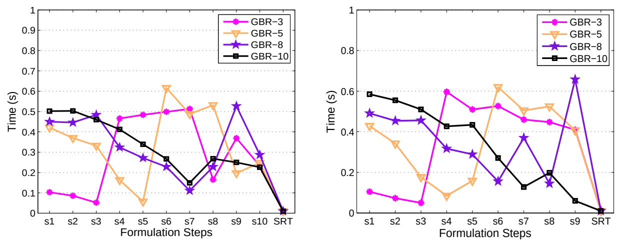
We can make the following observations. Firstly, GBR and FGI are orders of magnitude faster than SSI and CT for frequent queries (Q_1 - Q_3). This is because both these approaches follow verification-free frequent fragment matching strategies whereas SSI and CT need to verify a large candidate set. More importantly, GBR is up to four orders of magnitude faster than FGI. This is mainly due to the innovative paradigm of blending query formulation with query processing. Secondly, for the infrequent queries (Q_4 - Q_6), GBR has the best performance among all the approaches. Note that the performance gap between GBR and SSI is significantly reduced due to the fast subgraph isomorphism algorithm employed by SSI rather than its index pruning ability. Recall that in GBR we use Ullman's subgraph isomorphism algorithm which has been shown to be around two orders of magnitude slower than SSI (for query size equal to 10) [12]. Hence, we expect GBR to be orders of magnitude faster than SSI if the fast subgraph isomorphism algorithm is adopted. Thirdly, both FGI and GBR have $|R_c| = 0$ for frequent queries for reason mentioned above. Note that the candidate size of an infrequent query is not available for FGI (denoted as NA). Further, the candidate space of GBR is significantly smaller than SSI for infrequent queries. For instance, in Q_4, $|R_c|$ of SSI is 6927 whereas it is 476 for GBR. This again highlights the limitation of the pruning power of SSI's index.

Size of A^{2}I-index. Table 2 shows the size of A^{2}I-index for the AIDS and synthetic (for $\beta = 4$, $\alpha = 0.05$) datasets with different sizes. Note that the dataset size refers to the number of graphs in the dataset. It is evident that the index size is small enough to easily reside in the main memory.

Effect of β on index size. We now compare the index size of GBR to existing schemes. Note that the value of β in GBR affects the sizes of index in the memory and disk. Hence, we measure the size of index for different values of β. Table 3 reports the index

Query	*Sequence*	$Step_1$	$Step_2$	$Step_3$	$Step_4$	$Step_5$	$Step_6$	$Step_7$	$Step_8$	$Step_9$	Avg. SRT
Q_3	1,2,3,4,5,6,7,8,9	443	468	374	318	228	127	110	17	642	0.002
	5,2,1,3,4,6,7,9,8	520	450	321	194	374	210	146	17	650	0.002
Q_6	1,2,3,4,5,6,7,8	434	405	384	345	325	651	261	348		50
	5,6,1,2,3,4,7,8	150	335	34	660	361	28	25	336		70

Table 4: Effect of variation in query formulation sequence (in msec.)



(a) Q_2 (b) Q_3

Figure 11: Effect of β on frequent queries (in sec.).

size of different approaches. Since in GBR both MF-index and A^{2}I-index are memory-resident while DF-index is disk-resident, in the table the index size is represented as (*size in memory + size in disk*). Observe that the size of index in disk decreases with the increase in β as more frequent fragments are indexed in memory. Note that the value of β only affects the index construction of frequent fragments and has no effect on the infrequent fragments. Also, observe that GBR's index is smaller than that of FGI and CT.

Effect of β on SRT. We now study how the SRT of frequent query fragments are affected due to variation of β. Figure 11 depicts the results for the queries Q_2 and Q_3. The x-axis represents the sequence of formulation steps for a specific query (9 and 10 steps for Q_3 and Q_2, respectively) and the rightmost point represents the SRT. The y-axis represents the average time taken by all the participants (last four formulations). We denote the performances of GBR for different values of β as GBR-β. We can make the following observations. Firstly, β does not have significant impact on the SRTs of Q_2 and Q_3. Secondly, all the steps in both queries take less than a second to match query fragments. Note that the time taken to construct an edge visually in GBR is between 1 and 2 seconds. This justifies virtually "zero" SRT for frequent queries as prefetching of partial results can be completed while a user is constructing an edge on the visual interface. Note that this also justifies excellent performance for infrequent queries as the SRT essentially consists of verification time after the "Run" icon is clicked (Line 16 in Algorithm 3). Thirdly, since GBR-8 builds more frequent fragments in the MF-index compared to GBR-3 and GBR-5, the former takes relatively longer to search for matched frequent fragments in the MF-index. Lastly, GBR-3 at Step $s4$, GBR-5 at Step $s6$, and GBR-8 at Step $s9$ begin to access DF-index to search for larger frequent fragments. Consequently, the response time increases distinctly compared to previous steps. Note that such increase is not visible for GBR-10 as all the frequent fragments are in the MF-index.

Effect of α. In this experiment, we compare the performance of representative queries for different values of α. We vary the value of α from 0.05 to 0.2 for GBR, FGI, and SSI. The remaining parameters are set to their default settings. Note that CT does not have an α parameter. Variation of α may change the nature of a query. For instance, Q_2 is a frequent query when α=0.1, but it evolves to an infrequent query for α=0.15. On the other hand, Q_1 and Q_3 are always frequent fragments, while Q_4 and Q_6 are always infrequent queries. Figure 12 reports the SRT of representative queries for different values of α (in log scale). Firstly, for frequent queries (Figure 12(a), Figure 12(b) for $\alpha < 0.15$), GBR and FGI significantly outperform SSI and CT as the former are verification-free techniques. Also, none of them are significantly affected by variations in α as they remain frequent. Secondly, for infrequent queries (Figures 12(b) (for $\alpha \geq 0.15$), 12(c), and 12(d)) both GBR and SSI perform better than FGI and CT. Specifically, FGI mainly depends on frequent fragments to reduce the candidate space. The larger the value of α, the lesser frequent fragments are built in FGI, which degrades the pruning ability of FGI. In contrast, not only GBR benefits from the paradigm of blending of query formulation and processing, but also due to the fact that it uses both frequent and infrequent fragments to reduce the candidate space. Also SSI outperforms FGI and CT for the reasons discussed earlier.

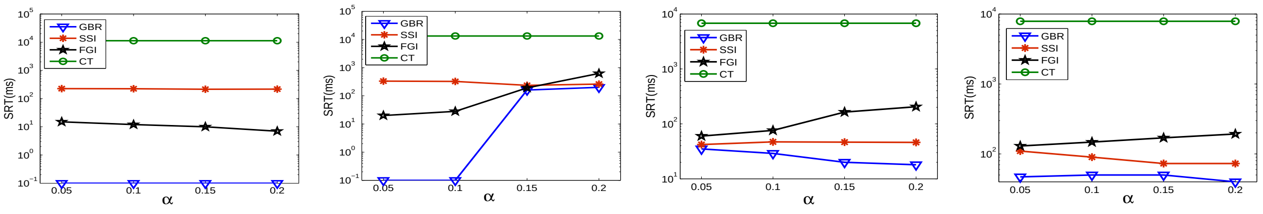
Effect of query formulation sequence. Observe that a visual query can be formulated by following different sequence of steps. In this experiment we assess the effect of these different sequences on the SRT. Table 4 lists two different formulation sequences for frequent query Q_3 and infrequent query Q_6, average times (all participants) to retrieve partial results, and the average SRT. For frequent queries, the formulation sequences only have minor effect on the prefetching time during the query formulation. Therefore, there is no change in the SRT. In contrast, for Q_6 different sequences may result in different SRT due to the change in candidate space. For example, the candidate sizes for Q_6 are 750 and 861 for the two sequences, respectively. However, the difference in SRT is not significant due to the following reasons. Firstly, if a query fragment evolves from a frequent fragment to an infrequent one, then the A^{2}F-index contributes to the reduction of the candidate space besides the A^{2}I-index. Secondly, we only retrieve the MDF (if available) instead of all discriminative fragments. Note that although the candidate space can change under different formulation sequences for the infrequent query, for any sequence it is always bounded in $\alpha|\mathcal{D}|$.

5.3 Performance on Synthetic Graph Dataset

We now assess the scalability of GBLENDER using the synthetic dataset and the queries Q_7 to Q_{10}. Here we compare GBR with only FGI as it has superior overall performance in terms of SRT and index pruning capability in comparison with SSI and CT. We set $\beta = 4$ and $\alpha \in \{0.01, 0.05\}$. We denote the performances of GBR for different values of α as GBR-α. Figure 13 depicts the performance results (in log scale) and confirms the strengths of GBLENDER. Observe that the SRT of GBR (less than $0.1s$) is two orders of magnitude faster than FGI across all datasets and α. Note that as α increases to 0.05, less frequent fragments are generated and as a result the performance of FGI-0.05 degrades compared to FGI-0.01, especially when the dataset is large. More importantly, our proposed paradigm enables GBLENDER to scale gracefully.

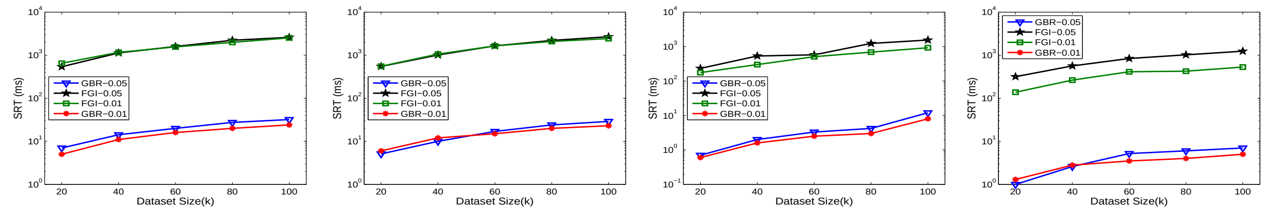
6. RELATED WORK

Recently, there have been a number of studies by the database community to develop techniques for subgraph isomorphism and graph isomorphism to speed up subgraph and similarity search over large graph databases [2, 5, 6, 9, 12–14, 17–21]. These efforts follow the conventional query processing paradigm where the formulation of a query graph is independent of its evaluation against the database. Typically, the *complete* query is first specified before it is processed. In contrast, GBLENDER realizes a novel query



(a) Q_1 (b) Q_2 (c) Q_4 (d) Q_6

Figure 12: Effect of α on queries (in msec.).



(a) Q_7 (b) Q_8 (c) Q_9 (d) Q_{10}

Figure 13: Scalability of GBLENDER (in msec.).

processing paradigm by blending two traditionally independent areas, namely human-computer interaction and database query processing. Specifically in the proposed approach, when a subgraph matching query is visually formulated, its evaluation is interleaved with the formulation activities. Hence, our method is orthogonal to existing studies related to subgraph query processing.

In order to speed up subgraph evaluation, most existing works focus on developing indexing techniques to support efficient searching. As discussed in Section 3.2, our proposed indexing scheme is different. In particular, unlike existing strategies, the design of the proposed indexing scheme is influenced by the characteristics of users' visual interaction behaviors during query formulation.

More germane to this work is the study described in [1] in the context of XML query processing. Similar to us, the authors proposed a technique that blends visual XML query formulation and query processing by exploiting the latency offered by the GUI-based query formulation to prefetch portions of the query results. Our proposed GBLENDER differs from this effort in the following ways. Firstly, we focus on graph queries instead of tree-structured XML queries. Evaluation of graph queries is typically more challenging than tree queries due to its inherent computational complexity. Secondly, as [1] is built on top of a relational backend, it leverages on existing well-known indexing schemes and SQL queries to efficiently prefetch partial results. In contrast, we propose a novel users' action-aware indexing scheme to support efficient computation of partial results that match different fragments of a visual subgraph matching query.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented GBLENDER - an exact subgraph matching tool for matching graph queries formulated using a visual interface. It is the first work that makes a strong connection between graph query processing and visual query formulation to improve system response time. GBLENDER employs a novel indexing scheme, which exploits some of the users' interaction characteristics with visual interfaces to support efficient pruning and retrieval. The innovative subgraph matching algorithm used by GBLENDER exploits the latency offered by the GUI-based query formulation to prune false results and prefetch partial query results in a single-user environment. Experimental studies on real and synthetic graphs validated the merit of our proposed graph processing paradigm.

We have barely scratched the surface of this novel query processing paradigm. We are currently exploring how other types of graph queries (e.g., similarity search) can be realized in this framework. Also, we wish to explore how query formulation errors can be gracefully integrated in this paradigm. In summary, the results of this paper are an important first step in this regard.

Acknowledgments. The authors thank H. He for providing C-Tree; X. Yan for providing gSpan; and H. Shang for providing SSI. This work is partly funded by the AcRF Tier-1 Grant (M52020073) from the Ministry of Education, Singapore.

8. REFERENCES

[1] S. S. BHOWMICK, S. PRAKASH. Every Click You Make, I Will be Fetching It: Efficient XML Query Processing in RDBMS Using GUI-driven Prefetching. In *ICDE*, 2006.
[2] J. CHENG, Y. KE, W. NG, A. LU. FG-Index: Towards Verification-Free Query Processing On Graph Databases.*In SIGMOD*, 2007.
[3] M. P. CONSENS, A. O. MENDELZON. GraphLog: A Visual Formalism for Real Life Recursion. *In PODS*, 1990.
[4] S. A. COOK. The Complexity of Theorem-Proving Procedures. *In STOC*, 1971.
[5] H. HE, A. K. SINGH. Closure-Tree: An Index Structure for Graph Queries. *In ICDE*, 2006.
[6] H. HE, A. K. SINGH. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. *In SIGMOD*, 2008.
[7] J. P. HUAN, W. WANG. Efficient Mining of Frequent Subgraph in the Presence of Isomorphism. *In ICDM*, 2003.
[8] H. V. JAGADISH, A. CHAPMAN, A. ELKISS ET AL. Making Database Systems Usable. *In ACM SIGMOD*, 2007.
[9] H. JIANG, H. WANG, P. S. YU, S. ZHOU. GString: A Novel Approach for Efficient Search in Graph Databases. *In ICDE*, 2007.
[10] U. LESER. A Query Language for Biological Networks. *In Bioinformatics*, 21:ii33–ii39, 2005.
[11] E. PIETRIGA. A Toolkit for Addressing HCI Issues in Visual Language Environments.*In IEEE Symp. on Vis. Lang. and Human-Centric Comp.*, 2005.
[12] H. SHANG, Y. ZHANG, X. LIN, J. X. YU. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *In VLDB*, 2008.
[13] S. TRIL, U. LESER. Fast and Practical Indexing and Querying of Very Large Graphs. *In SIGMOD*, 2007.
[14] D. W. WILLIAMS, J. HUAN, W. WANG. Graph Database Indexing Using Structured Graph Decomposition. *In ICDE*, 2007.
[15] J. R. ULLMAN. An Algorithm for Subgraph Isomorphism. *J. ACM*, 23(1):31–42, 1976.
[16] X. YAN, J. HAN. gSpan: Graph-based Substructure Pattern Mining. *In ICDM*, 2002.
[17] X. YAN, P. S. YU, J. HAN. Graph Indexing: A Frequent Structure-Based Approach. *In SIGMOD*, 2004.
[18] X. YAN, P. S. YU, J. HAN. Substructure Similarity Search in Graph Databases. *In SIGMOD*, 2005.
[19] S. ZHANG, M. HU, J. YANG. TreePi: A Novel Graph Indexing Method. *In ICDE*, 2007.
[20] P. ZHAO, J. X. YU, P. S. YU. Graph Indexing: Tree + delta $\geq$ Graph. *In VLDB*, 2007.
[21] J. L. ZOU, L. CHEN, Y. LU. A Novel Spectral Coding in a Large Graph Database. *In EDBT*, 2008.