

Graphs-at-a-time: Query Language and Access Methods for Graph Databases

Huahai He^{*}

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
huahai@cs.ucsb.edu

Ambuj K. Singh

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106, USA
ambuj@cs.ucsb.edu

ABSTRACT

With the prevalence of graph data in a variety of domains, there is an increasing need for a language to query and manipulate graphs with heterogeneous attributes and structures. We propose a query language for graph databases that supports arbitrary attributes on nodes, edges, and graphs. In this language, graphs are the basic unit of information and each query manipulates one or more collections of graphs. To allow for flexible compositions of graph structures, we extend the notion of formal languages from strings to the graph domain. We present a graph algebra extended from the relational algebra in which the selection operator is generalized to graph pattern matching and a composition operator is introduced for rewriting matched graphs. Then, we investigate access methods of the selection operator. Pattern matching over large graphs is challenging due to the NP-completeness of subgraph isomorphism. We address this by a combination of techniques: use of neighborhood subgraphs and profiles, joint reduction of the search space, and optimization of the search order. Experimental results on real and synthetic large graphs demonstrate that our graph specific optimizations outperform an SQL-based implementation by orders of magnitude.

Categories and Subject Descriptors

H.2.3 [Database Management]: Languages—*Query Languages*; H.2.4 [Database Management]: Systems—*Query processing*

General Terms

Algorithms, Languages, Performance

Keywords

Graph query language, Graph algebra, Query optimization

^{*}Author's current address is Google Inc., Mountain View, CA 94043, huahai@google.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...\$5.00.

1. INTRODUCTION

Data in multiple domains can be naturally modeled as graphs. Examples include the Semantic Web [28], GIS, images [2], videos [20], social networks, Bioinformatics and Cheminformatics. Semantic Web standardizes information on the web as a graph with a set of entities and explicit relationships. In Bioinformatics, graphs represent several kinds of information: a protein structure can be modeled as a set of residues (nodes) and their spatial proximity (edges); a protein interaction network can be similarly modeled by a set of genes/proteins (nodes) and physical interactions (edges). In Cheminformatics, graphs are used to represent atoms and bonds in chemical compounds.

The growing heterogeneity and size of the above data has spurred interest in diverse applications that are centered on graph data. Existing data models, query languages, and database systems do not offer adequate support for the modeling, management, and querying of this data. There are a number of reasons for developing native graph-based data management systems. Considering expressiveness of queries: we need query languages that manipulate graphs in their full generality. This means the ability to define constraints (graph-structural and value) on nodes and edges *not* in an iterative one-node-at-a-time manner but simultaneously on the entire object of interest. This also means the ability to return a graph (or a set of graphs) as the result and not just a set of nodes. Another need for native graph databases is prompted by efficiency considerations. There are heuristics and indexing techniques that can be applied only if we operate in the domain of graphs.

1.1 Graphs-at-a-time Queries

Abstractly, a graph query takes a graph pattern as input, retrieves graphs from the database which contain (or are similar to) the query pattern, and returns the retrieved graphs or new graphs composed from the retrieved graphs. Examples of graph queries can be found in various domains:

- Find all heterocyclic chemical compounds that contain a given aromatic ring and a side chain. Both the ring and the side chain are specified as graphs with atoms as nodes and bonds as edges.
- Find all protein structures that contain the α - β -barrel motif [4]. This motif is specified as a cycle of β strands embraced by another cycle of α helices.
- Given a query protein complex from one species, is it functionally conserved in another species? The pro-

tein complex may be specified as a graph with nodes (proteins) labeled by Gene Ontology [12] terms.

- Find all instances from an RDF (Resource Description Framework [22]) graph where two departments of a company share the same shipping company. The query graph (of three nodes and two edges) has the constraints that nodes share the same company attribute and the edges are labeled by a “shipping” attribute. Report the result as a single graph with departments as nodes and edges between nodes that share a shipper.
- Find all co-authors from the DBLP dataset (a collection of papers represented as small graphs) in a specified set of conference proceedings. Report the results as a co-authorship graph.

As illustrated above, there is an increasing need for a language to query and manipulate graphs with heterogeneous attributes and structures. The language should be native to graphs, general enough to meet the heterogeneous nature of real world data, declarative, and yet implementable. Most importantly, a graph query language needs to support the following feature.

- Graphs should be the basic unit of information. The language should explicitly address graphs and queries should be graphs-at-a-time, taking one or more collections of graphs as input and producing a collection of graphs as output.

1.2 Graph Specific Optimizations

A graph query language is useful only if it can be efficiently implemented. This is especially important since one encounters the usual bottlenecks of subgraph isomorphism. As graphs are special cases of relations, graph queries can still be reduced to the relational model. However, the general-purpose relational model allows little opportunity for graph specific optimizations since it breaks down the graph structures into individual relations. Let us consider a simple example as follows. Figure 1 shows a graph query and a graph where each node has a single label as its attribute (nodes with the same label are distinguished by subscripts).

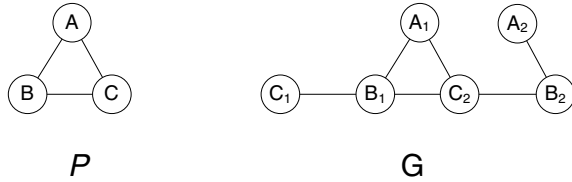


Figure 1: A sample graph query and database graph

An SQL-based implementation would store the graph in two tables. Table V(vid, label) stores the set of nodes¹ where vid is the node identifier. Table E(vid1, vid2) stores the set of edges where vid1 and vid2 are end points of each edge. The graph query can then be implemented by multiple joins:

```
SELECT V1.vid, V2.vid, V3.vid
FROM   V AS V1, V AS V2, V AS V3,
       E as E1, E as E2, E as E3
```

¹For convenience, the terms “vertex” and “node” are used interchangeably in this paper.

```
WHERE V1.label = 'A' AND V2.label = 'B' AND V3.label = 'C'
AND V1.vid = E1.vid1 AND V1.vid = E3.vid1
AND V2.vid = E1.vid2 AND V2.vid = E2.vid1
AND V3.vid = E2.vid2 AND V3.vid = E3.vid2
AND V1.vid <> V2.vid AND V1.vid <> V3.vid
AND V2.vid <> V3.vid;
```

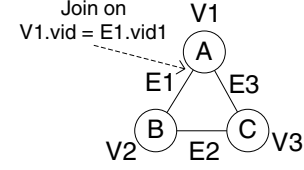


Figure 2: SQL-based implementation

As can be seen in the above example, the global view of graph structures is lost in the SQL query. This prevents pruning of the search space that utilizes local or global graph structural information. For instance, nodes A_2 and C_1 in G can be safely pruned since they have only one neighbor. Node B_2 can also be pruned after A_2 is pruned. Furthermore, the SQL query involves many join operations. Traditional query optimization techniques such as dynamic programming do not scale well with the number of joins. This makes SQL-based implementations inefficient.

1.3 Our Approach

In this paper, we propose *GraphQL*, a graph query language that uses a graph pattern as the basic operational unit. A graph pattern consists of a graph structure and a predicate on attributes of the graph. To allow flexible manipulation on graph structures (useful for definition of graph queries as well as database graphs), we introduce the notion of *formal languages for graphs*. The core of GraphQL is a *graph algebra* in which the selection operator is generalized to graph pattern matching and a composition operator is introduced for rewriting matched graphs. In terms of expressive power, GraphQL is relationally complete and is contained in Datalog [24]. The nonrecursive version of GraphQL is equivalent to the relational algebra.

In the second part of the paper, we consider the evaluation of graph queries. Access methods for the selection operator turn out to be the main challenge, especially when graphs are large. We accelerate the basic graph pattern matching algorithm by three techniques that exploit graph structural information. First, we generate the search space with local pruning using neighborhood subgraphs or their profiles. Second, we reduce the overall search space simultaneously using global structural information. Third, we optimize the search order based on a cost model designed for graphs. As we demonstrate in experimental results, the combination of these three techniques allows us to scale to both large queries and large graphs.

Our work has the following contributions:

1. We introduce the notion of formal languages for graphs. It is useful for manipulating graphs and is the basis of our query language (Section 2).
2. We propose the GraphQL query language. This language supports graphs as the basic unit of information, arbitrary attributes, and set-oriented operations (Section 3).

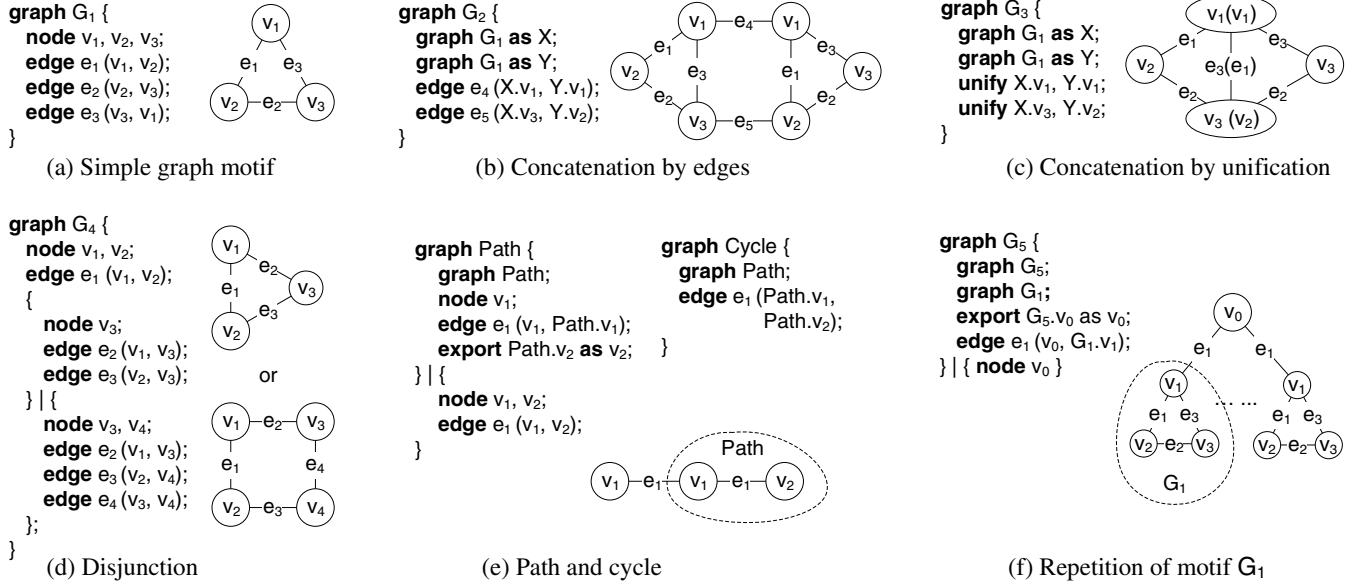


Figure 3: Examples of graph motifs

- We define a graph algebra along the line of the relational algebra. The graph algebra generalizes the selection operator to graph pattern matching and introduces a composition operator for rewriting matched graphs. In terms of expressive power, we prove that the graph algebra is relationally complete and is contained in Datalog (Section 3.3).
- We propose efficient access methods for the selection operator over large graphs. Experimental results on large real and synthetic graphs show that our graph specific optimizations outperform an SQL-based implementation by orders of magnitude (Sections 4 and 5).

2. FORMAL LANGUAGE FOR GRAPHS

In this section, we introduce the notion of formal languages for graphs. This notion is useful for composing and manipulating graph structures. It serves as a basis of our graph query language.

In classical formal languages [17], a formal grammar consists of a finite set of terminals and nonterminals, and a finite set of production rules that generate strings of characters. We extend this notion of formal grammars from strings to the graph domain, where the basic operational unit is a graph structure, namely a *graph motif*. A graph motif can be either a simple graph or composed of other graph motifs by means of concatenation, disjunction, and repetition. A *graph grammar* is a finite set of graph motifs. The *language* of a graph grammar is the set of all graphs derivable from graph motifs of that grammar.

2.1 Simple Graph Motifs

A simple graph motif is a normal graph. It consists of a set of nodes and a set of edges. Each node, edge, or graph is identified by a *variable*. The variable name can be omitted if not referenced elsewhere. Nodes and edges correspond to terminals, whereas graphs correspond to nonterminals.

Every node, edge, or graph may have multiple attributes. Figure 3(a) shows a simple graph motif and its graphical representation.

2.2 Complex Graph Motifs

A graph motif can be composed of other graph motifs. In existing grammars, a string is obtained by concatenation, disjunction, or repetition of other strings. A string connects to other strings through its head and tail, which is specified implicitly. In the graph domain, however, a graph may connect to other graphs through arbitrary nodes. Therefore, one needs to explicitly specify these interconnections.

2.2.1 Concatenation

A graph motif can be composed of two or more graph motifs. The constituent motifs can be either left unconnected or concatenated in one of two ways. One way is to connect nodes in each motif by new edges. Figure 3(b) shows an example of concatenation by edges. Graph motif G_2 is composed of two motifs G_1 of Figure 3(a). The two motifs are connected by two edges. To avoid name conflicts, aliases are introduced using the keyword “as.”

The other way of concatenation is to *unify* nodes in each motif. Two edges are unified automatically if their respective end nodes are unified. Figure 3(c) shows an example of concatenation by unification.

A *member* of a graph motif refers to a node, an edge, or a graph motif declared in the body of that motif. A graph motif is *recursive* if its body or derived body contains itself as a member. Member variables are visible from the scope of declaration and any nested scope. Variables declared in other scopes can be referenced through their enclosing graph variables.

2.2.2 Disjunction

A graph motif can be defined as a disjunction of two or more graph motifs. Figure 3(d) shows an example of disjunction. In graph motif G_4 , two anonymous graph motifs

are declared (comprising of node v_3 or nodes v_3 and v_4). Only one of them is selected and connected to the rest of G_4 . All the constituent graph motifs should have the same “interface” to the outside.

2.2.3 Repetition (Kleene Star)

In existing formal grammars, a repetition takes the form $S \rightarrow S_1^*$. It can also be written as $S \rightarrow S_1 S \mid \varepsilon$. We specify repetition of graph motifs through this kind of self recursion. Figure 3(e) shows the construction of a path and a cycle. In the base case, the path has two nodes and one edge. In the recurrence step, the path contains itself as a member, adds a new node v_1 which connects to v_1 of the nested path, and exports the nested v_2 so that the new path has the same “interface.” The keyword “**export**” is equivalent to declaring a new node and unifying it with the nested node. Graph motif *Cycle* is composed of motif *Path* with an additional edge that connects the end nodes of the *Path*.

Recursion in GraphQL is not limited to paths and cycles. Figure 3(f) illustrates an example where the repetition unit is a graph motif. Motif G_5 contains an arbitrary number of motif G_1 and a root node v_0 . The declaration recursively contains G_5 itself and a new G_1 , with $G_1.v_1$ connected to v_0 , where v_0 is exported from the nested G_5 . The first resulting graph consists of node v_0 alone, the second consists of node v_0 connected to G_1 through edge e_1 , the third consists of node v_0 connected to two instances of G_1 through edge e_1 , and so on.

3. GRAPH QUERY LANGUAGE

In this section, we describe the graph query language. We first describe the data model. Next, we define a graph pattern which is the main building block of a graph query. We then define a graph algebra for the query language and investigate its expressive power. Finally, we illustrate the graph query syntax through an example.

3.1 Data Model

In the GraphQL data model, each node, edge, or graph can have arbitrary attributes. We use a *tuple*, a list of name and value pairs, to denote these attributes. The tuple can have an optional *tag* that denotes the type of the tuple. Figure 4 shows a sample graph that represents a paper (the nodes are unconnected). Node v_1 has two attributes “title” and “year”. Nodes v_2 and v_3 have a tag “author” and an attribute “name”.

```
graph G <inproceedings> {
  node v1 <title="Title1", year=2006>;
  node v2 <author name="A">;
  node v3 <author name="B">;
};
```

Figure 4: A sample graph

In the relational model, tuples are the basic unit of information. Each relational algebraic operator manipulates collections of tuples. In GraphQL, graphs are the basic unit of information. Each operator takes one or more collections of graphs as input and generates a collection of graphs as output. This is similar to the TAX model [18] where trees are the basic unit and the operators work on collections of trees. A graph database consists of one or more collections

of graphs. Graphs in a collection do not necessarily have identical structures and attributes. However, they can still be processed in a uniform way by adhering to a graph pattern.

3.2 Graph Patterns

Essentially, a graph pattern is a graph motif plus a predicate on attributes of the motif. A graph pattern is used to select graphs of interest. It is the main building block of a graph query.

Definition 1. (Graph Pattern) A *graph pattern* is a pair $\mathcal{P} = (\mathcal{M}, \mathcal{F})$, where \mathcal{M} is a graph motif and \mathcal{F} is a predicate on the attributes of the motif.

The predicate \mathcal{F} can be a combination of boolean or arithmetic comparison expressions. Figure 5 shows a sample graph pattern. The predicate can be broken down to predicates on individual nodes or edges, as shown on the right side of the figure.

<pre>graph P { node v1; node v2; } where v1.name="A" and v2.year>2000;</pre>	or	<pre>graph P { node v1 where name="A"; node v2 where year>2000; };</pre>
-------------------------------------------------------------------------------------	----	---------------------------------------------------------------------------------

Figure 5: A sample graph pattern

Next, we define the notion of **graph pattern matching** which generalizes subgraph isomorphism with evaluation of the predicate.

Definition 2. (Graph Pattern Matching) A graph pattern $\mathcal{P}(\mathcal{M}, \mathcal{F})$ is matched with a graph G if there exists an **injective mapping** $\phi: V(\mathcal{M}) \rightarrow V(G)$ such that i) For $\forall e(u, v) \in E(\mathcal{M})$, $(\phi(u), \phi(v))$ is an edge in G , and ii) predicate $\mathcal{F}_\phi(G)$ holds.

A graph pattern is recursive if its motif is recursive (see Section 2.2.1). A recursive graph pattern is matched with a graph if one of its derived motifs is matched with the graph.

Mapping Φ :
 $\Phi(P.v_1) \rightarrow G.v_2$
 $\Phi(P.v_2) \rightarrow G.v_1$

Figure 6: A mapping between the graph pattern in Figure 5 and the graph in Figure 4

Figure 6 shows an example of graph pattern matching between the pattern in Figure 5 and the graph in Figure 4.

Once a graph pattern is matched to a graph, the binding between them can be used to access the graph. This allows one to access a collection of graphs uniformly even though they may have heterogeneous structures and attributes. We use a *matched graph* to denote the binding between a graph pattern and a graph.

Definition 3. (Matched Graph) Given an injective mapping ϕ between a pattern \mathcal{P} and a graph G , a matched graph is a triple $\langle \phi, \mathcal{P}, G \rangle$ and is denoted by $\phi\mathcal{P}(G)$.

Although formally a triple, a matched graph has all characteristics of a graph. Thus, all concepts that apply to a

graph also apply to a matched graph, e.g., a collection of matched graphs is also a collection of graphs.

A graph pattern can match a graph in multiple places, resulting in multiple bindings (matched graphs). This is considered further when we discuss the selection operator in Section 3.3.1.

3.3 Graph Algebra

We define a graph algebra along the lines of the relational algebra. However, there are two important distinctions. First, the selection operator is now generalized to graph pattern matching. Second, a composition operator is introduced to generate new graphs from matched graphs.

3.3.1 Selection

A selection operator σ is defined using a graph pattern \mathcal{P} . It takes a collection of graphs \mathcal{C} as input and produces a collection of graphs that match the graph pattern, denoted by $\sigma_{\mathcal{P}}(\mathcal{C})$.

A graph pattern can match a graph many times. Thus, a selection could return many instances for each graph. We use an option “exhaustive” to specify whether it should return one or all possible mappings from the graph pattern to the graph. Whether one or all mappings are required depends on the application. The output is a collection of matched graphs:

$$\sigma_{\mathcal{P}}(\mathcal{C}) = \{\phi_{\mathcal{P}}(G) \mid G \in \mathcal{C}\}$$

3.3.2 Cartesian Product and Join

A Cartesian product operator takes two collections of graphs \mathcal{C} and \mathcal{D} and produces a collection of graphs as output. Each output graph is composed of a graph from \mathcal{C} and another from \mathcal{D} . The constituent graphs are unconnected:

$$\mathcal{C} \times \mathcal{D} = \{ \text{graph } \{ \text{graph } G_1, G_2; \} \mid G_1 \in \mathcal{C}, G_2 \in \mathcal{D} \}$$

The join operator can be defined by a Cartesian product followed by a selection: $\mathcal{C} \bowtie_{\mathcal{P}} \mathcal{D} = \sigma_{\mathcal{P}}(\mathcal{C} \times \mathcal{D})$.

In a *valued join*, the join condition is a predicate on attributes of the constituent graphs, e.g., “ $G_1.\text{name} = G_2.\text{name}$.” In a *structural join*, the constituent graphs can be concatenated by edges or unification. This is specified using a composition operator which is described next.

3.3.3 Composition

The composition operator generates new graphs by combining information from matched graphs. We first define graph templates that specify the output structure of the graphs, and then define the composition operator.

Definition 4. (Graph Template) A *graph template* \mathcal{T} consists of a list of formal parameters which are graph patterns, and a template body that defines a graph by using variables from the graph patterns.

Once actual parameters (matched graphs) are given, a graph template is *instantiated* to a real graph. This is similar to the invocation of a function: the template body is the function body; the graph patterns are the formal parameters; the matched graphs are the actual parameters. The resulting graph can be denoted by $\mathcal{T}_{\mathcal{P}_1.. \mathcal{P}_k}(G_1, \dots, G_k)$.

Figure 7 shows a sample graph template $\mathcal{T}_{\mathcal{P}}$ and an instantiated graph $\mathcal{T}_{\mathcal{P}}(G)$. \mathcal{P} is the formal parameter of the

$\mathcal{T}_{\mathcal{P}} = \text{graph } \{$ <div style="margin-left: 20px;"> $\text{node } v_1 \langle \text{label} = \mathcal{P}.v_1.\text{name} \rangle;$ $\text{node } v_2 \langle \text{label} = \mathcal{P}.v_2.\text{title} \rangle;$ $\text{edge } e_1(v_1, v_2);$ </div> $\}$	$\mathcal{T}_{\mathcal{P}}(G) = \text{graph } \{$ <div style="margin-left: 20px;"> $\text{node } v_1 \langle \text{label} = \text{"A"} \rangle;$ $\text{node } v_2 \langle \text{label} = \text{"Title1"} \rangle;$ $\text{edge } e_1(v_1, v_2);$ </div> $\}$
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 7: A graph template and its instantiation. \mathcal{P} and G are shown in Figure 5 and Figure 4.

template. The template body consists of two nodes constructed from \mathcal{P} and an edge between them. Given the actual parameter G , the template is instantiated to a graph.

A primitive composition operator ω is defined using a graph template $\mathcal{T}_{\mathcal{P}}$ which has a single parameter. It takes a collection of matched graphs \mathcal{C} as input and produces a collection of graphs as output:

$$\omega_{\mathcal{T}_{\mathcal{P}}}(\mathcal{C}) = \{\mathcal{T}_{\mathcal{P}}(G) \mid G \in \mathcal{C}\}$$

In general, a composition operator may allow two or more collections of graphs as input. This can be expressed by a primitive composition operator and the Cartesian product operator:

$$\omega_{\mathcal{T}_{\mathcal{P}_1, \mathcal{P}_2}}(\mathcal{C}_1, \mathcal{C}_2) = \omega_{\mathcal{T}_{\mathcal{P}}}(\mathcal{C}_1 \times \mathcal{C}_2),$$

where $\mathcal{P} = \text{graph } \{ \text{graph } \mathcal{P}_1, \mathcal{P}_2; \}$.

Projection and Renaming, two other operators of the relational algebra, can be expressed using the composition operator. The set operators (union, difference, intersection) can also be defined easily. In terms of expressive power, the five operators (selection, Cartesian product, primitive composition, union, and difference) are complete.

Algebraic laws are important for query optimization. Since our graph algebra is defined along the lines of the relational algebra, laws of relational algebra carry over.

3.4 FLWR Expressions

We adopt the FLWR (For, Let, Where, and Return) expressions in XQuery [3] as the syntax of our graph query language. Since a graph algebra has been defined, we skip the query syntax for brevity.

```

graph P {
  node v1 <author>;
  node v2 <author>;
};
C := graph {};
for P exhaustive in doc("DBLP")
let C := graph {
  graph C;
  node P.v1, P.v2;
  edge e1(P.v1, P.v2);
  unify P.v1, C.v1 where P.v1.name=C.v1.name;
  unify P.v2, C.v2 where P.v2.name=C.v2.name;
}

```

Figure 8: A graph query that generates a co-authorship graph from the DBLP dataset

Figure 8 shows an example that generates a co-authorship graph \mathcal{C} from a collection of papers. The query states that any pair of authors in a paper should appear in the co-authorship graph with an edge between them. The graph

```

DBLP:graph G1 {
  node v1 <author name="A">;
  node v2 <author name="B">;
};
graph G2 {
  node v1 <author name="C">;
  node v2 <author name="D">;
  node v3 <author name="A">;
};

```


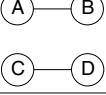
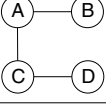
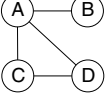
Iteration	Mapping	Co-authorship graph C
1	$\Phi(P.v_1) \rightarrow G_1.v_1$ $\Phi(P.v_2) \rightarrow G_1.v_2$	
2	$\Phi(P.v_1) \rightarrow G_2.v_1$ $\Phi(P.v_2) \rightarrow G_2.v_2$	
3	$\Phi(P.v_1) \rightarrow G_2.v_1$ $\Phi(P.v_2) \rightarrow G_2.v_3$	
4	$\Phi(P.v_1) \rightarrow G_2.v_2$ $\Phi(P.v_2) \rightarrow G_2.v_3$	

Figure 9: A possible execution of the Figure 8 query

pattern P matches a pair of authors in a paper. The **for** clause selects all such pairs from the data source. The **let** clause places each pair in the co-authorship graph and adds an edge between them. The unifications ensure that each author appears only once. Again, two edges are unified automatically if their end nodes are unified.

Figure 9 shows a running example of the query. The DBLP collection consists of two graphs G_1 and G_2 . The pair of author nodes (A, B) is first chosen and an edge is inserted between them. The pair (C, D) is chosen next and the (C, D) subgraph is inserted. When the third pair (A, C) is chosen, unification ensures that the old nodes are reused and an edge is added between existing A and C. The processing of the fourth pair adds one more edge and completes the execution.

The query can be translated into a recursive algebraic expression:

$$C = \sigma_J(\omega_{\tau_{P,C}}(\sigma_P(\text{"DBLP"}), \{C\}))$$

where $\sigma_P(\text{"DBLP"})$ corresponds to the **for** clause, $\tau_{P,C}$ is the graph template in the **let** clause, and J is a graph pattern for the join condition: $P.v_1.name = C.v_1.name \ \& \ P.v_2.name = C.v_2.name$. The algebraic expression turns out to be a structural join that consists of three primitive operators: Cartesian product, primitive composition, and selection.

3.5 Expressive Power

We first show that the relational algebra (RA) is contained in GraphQL.

THEOREM 1. ($RA \subseteq GraphQL$) *For any RA expression, there exists an equivalent GraphQL algebra expression.*

PROOF. We can represent a relation (tuple) in GraphQL using a graph that has a single node with attributes as the

tuple. The primitive operations of RA (selection, projection, Cartesian product, union, difference) can then be expressed in GraphQL. The selection operator can be simulated using a graph pattern with the given predicate as the selection condition. For projection, one rewrites the projected attributes to a new node using the composition operator. Other operations (product, union, difference) are straightforward as well. \square

Next, we show that GraphQL is contained in Datalog. This is proved by translating graphs, graph patterns, and graph templates into facts and rules of Datalog.

THEOREM 2. ($GraphQL \subseteq Datalog$) *For any GraphQL algebra expression, there exists an equivalent Datalog program.*

PROOF. We first translate all graphs of the database into facts of Datalog. Figure 10 shows an example of the translation. Essentially, we rewrite each variable of the graph as a unique constant string, and then establish a connection between the graph and each node and edge. Note that for undirected graphs, we need to write an edge twice to permute its end nodes.

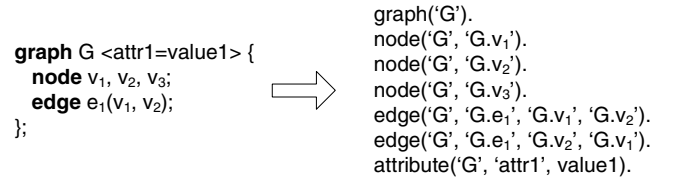


Figure 10: The translation of a graph into facts of Datalog

For each graph pattern, we translate it into a rule of Datalog. Figure 11 gives an example of such translation. The body of the rule is a conjunction of the constituent elements of the graph pattern. The predicate of the graph pattern is written naturally. It can then be shown that a graph pattern matches a graph if and only if the corresponding rule matches the facts that represent the graph.

Subsequently, one can translate the graph algebraic operations into Datalog in a way similar to translating RA into Datalog. Thus, we can translate any GraphQL algebra expression into an equivalent Datalog program. \square

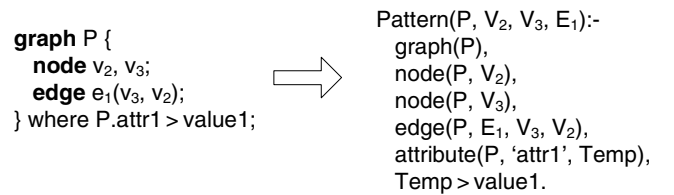


Figure 11: The translation of a graph pattern into a rule of Datalog

It is well known that nonrecursive Datalog (nr-Datalog) is equivalent to RA. Consequently, the nonrecursive version of GraphQL (nr-GraphQL) is also equivalent to RA.

COROLLARY 1. $nr-GraphQL \equiv RA$.

4. ACCESS METHODS

In this section, we discuss access methods for the selection operator, i.e., given a graph pattern and a collection of graphs, how to produce a collection of matched graphs.

Generally, a graph database can be classified into two categories. One category is a large collection of small graphs, e.g., chemical compounds. The main challenge for this category is to reduce the number of pairwise graph pattern matchings. A number of graph indexing techniques have been proposed to address this challenge [15, 29, 35].

In the second category, the graph database consists of a few very large graphs, e.g., protein interaction networks, Web information, social networks. The challenge here is to accelerate the graph pattern matching itself. In this paper, we focus on access methods for large graphs. However, the proposed methods are also applicable to small graphs.

We first describe the basic graph pattern matching algorithm in Section 4.1, and then discuss accelerations to the basic algorithm in Sections 4.2, 4.3, and 4.4. We restrict our attention to nonrecursive graph patterns and in-memory processing.

4.1 Graph Pattern Matching

A graph pattern matching takes a graph pattern \mathcal{P} and a graph G as input, and produces one or all feasible mappings as output. It is used as a subroutine for the selection operator. Algorithm 1 outlines the basic algorithm.

The predicate of graph pattern \mathcal{P} is rewritten as predicates on individual nodes \mathcal{F}_u 's and edges \mathcal{F}_e 's. Predicates that cannot be pushed down, e.g., $u_1.\text{label}=u_2.\text{label}$, remain as the graph-wide predicate \mathcal{F} . We define the *feasible mates* of node u as follows.

Definition 5. (Feasible Mates) The feasible mates $\Phi(u)$ of node u is the set of nodes in graph G that satisfies predicate \mathcal{F}_u :

$$\Phi(u) = \{v | v \in V(G), \mathcal{F}_u(v) = \text{true}\}.$$

The algorithm consists of two phases. The first phase (lines 1–4) retrieves the feasible mates for each node u in the pattern. The resulting product $\Phi(u_1) \times \dots \times \Phi(u_k)$ forms the *search space* over which the second phase of the algorithm (Lines 7–26) searches for subgraph isomorphism.

Definition 6. (Search Space) The search space of a graph pattern matching is defined as the product of feasible mates for each node of the graph pattern: $\Phi(u_1) \times \dots \times \Phi(u_k)$, where k is the size of the graph pattern.

The second phase (lines 7–27) searches in a depth-first manner for matchings between the graph pattern and the graph. Procedure $\text{Search}(i)$ iterates on the i^{th} node to find feasible mappings for that node. Procedure $\text{Check}(u_i, v)$ examines if u_i can be mapped to v by considering their edges. Line 12 maps u_i to v . Lines 13–16 continue to search for the next node or if it is the last node, evaluate the graph-wide predicate. If it is true, then a feasible mapping $\phi : V(\mathcal{P}) \rightarrow V(G)$ has been found and is reported (line 15). Line 16 stops searching immediately if only one mapping is required.

The graph pattern and the graph are represented as a vertex set and an edge set, respectively. In addition, adjacency lists of the graph pattern are used to support line 21. For line 22, edges of graph G can be represented in a hashtable where keys are pairs of the end points. To avoid repeated

Algorithm 1: Graph Pattern Matching

Input: Graph Pattern \mathcal{P} , Graph G

Output: One or all feasible mappings $\phi_{\mathcal{P}}(G)$

```

1 foreach node  $u \in V(\mathcal{P})$  do
2    $\Phi(u) \leftarrow \{v | v \in V(G), \mathcal{F}_u(v) = \text{true}\}$ 
3   // Local pruning and retrieval of  $\Phi(u)$  (Section 4.2)
4 end
5 // Reduce  $\Phi(u_1) \times \dots \times \Phi(u_k)$  globally (Section 4.3)
6 // Optimize search order of  $u_1, \dots, u_k$  (Section 4.4)
7  $\text{Search}(1)$ ;
8 void  $\text{Search}(i)$ 
9 begin
10  foreach  $v \in \Phi(u_i)$ ,  $v$  is free do
11    if not  $\text{Check}(u_i, v)$  then continue;
12     $\phi(u_i) \leftarrow v$ ;
13    if  $i < |V(\mathcal{P})|$  then  $\text{Search}(i + 1)$ ;
14    else if  $\mathcal{F}_{\phi}(G)$  then
15      Report  $\phi$ ;
16      if not exhaustive then stop;
17  end
18 end
19 boolean  $\text{Check}(u_i, v)$ 
20 begin
21  foreach edge  $e(u_i, u_j) \in E(\mathcal{P}), j < i$  do
22    if edge  $e'(v, \phi(u_j)) \notin E(G)$  or not  $\mathcal{F}_e(e')$  then
23      return false;
24  end
25  return true;
26 end
```

evaluation of edge predicates (line 22), another hashtable can be used to store evaluated pairs of edges.

The worst-case time complexity of Algorithm 1 is $O(n^k)$ where n and k are the sizes of graph G and graph pattern \mathcal{P} , respectively. This complexity is a consequence of subgraph isomorphism that is known to be NP-hard. In practice, the running time depends on the size of the search space. Next, we discuss possible ways to accelerate Algorithm 1 by reducing this search space and exploring it in the best order.

1. How to reduce the size of $\Phi(u_i)$ for each node u_i ? How to efficiently retrieve $\Phi(u_i)$?
2. How to reduce the overall search space $\Phi(u_1) \times \dots \times \Phi(u_k)$?
3. How to optimize the search order?

We present three techniques that respectively address the above questions. The first technique prunes each $\Phi(u_i)$ individually and retrieves it efficiently through indexing. The second technique prunes the overall search space by considering all nodes in the pattern simultaneously. The third technique applies ideas from traditional query optimization to find the right search order.

4.2 Local Pruning and Retrieval of Feasible Mates

We can index attributes of the graph nodes for fast retrieval of feasible mates. This avoids a sequential scan of all nodes in a large graph. To reduce the size of $\Phi(u_i)$ even

further, we can go beyond nodes and **consider neighborhood subgraphs of the nodes**. The neighborhood information can be exploited to prune infeasible mates at an early stage.

Definition 7. (Neighborhood Subgraph) Given graph G , node v and radius r , the neighborhood subgraph of node v consists of all nodes within distance r (number of hops) from v and all edges between the nodes.

Node v is a feasible mate of node u_i only if the neighborhood subgraph of u_i is sub-isomorphic to that of v (with u_i mapped to v). Note that if the radius is 0, then the neighborhood subgraphs degenerate to nodes.

Although neighborhood subgraphs have high pruning power, they incur a large computation overhead. This overhead can be reduced by **representing neighborhood subgraphs by their light-weight profiles**. For instance, one can define the profile as a sequence of the node labels in lexicographic order. **The pruning condition then becomes whether a profile is a subsequence of the other.**

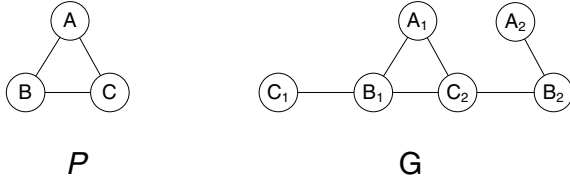


Figure 12: A sample graph pattern and graph

Nodes of G	Neighborhood sub-graphs of radius 1	Profiles
A ₁		ABC
A ₂		AB
B ₁		ABCC
B ₂		ABC
C ₁		BC
C ₂		ABBC

Figure 13: Feasible mates using neighborhood subgraphs and profiles. The resulting search spaces are also shown for different pruning techniques.

Figure 12 shows the sample graph pattern \mathcal{P} and the database graph G again for convenience. Figure 13 shows the neighborhood subgraphs of radius 1 and their profiles for nodes of G . If the feasible mates are retrieved using node attributes, then the search space is $\{A_1, A_2\} \times \{B_1, B_2\} \times \{C_1, C_2\}$. If the feasible mates are retrieved using neighborhood subgraphs, then the search space is $\{A_1\} \times \{B_1\} \times \{C_2\}$. Finally, if the feasible mates are retrieved using profiles, then the search space is $\{A_1\} \times \{B_1, B_2\} \times \{C_2\}$. These are shown in the right side of Figure 13.

Algorithm 2: Refine Search Space

Input: Graph Pattern \mathcal{P} , Graph G , Search space $\Phi(u_1) \times \dots \times \Phi(u_k)$, level l
Output: Reduced search space $\Phi'(u_1) \times \dots \times \Phi'(u_k)$

```

1 begin
2   foreach  $u \in \mathcal{P}, v \in \Phi(u)$  do Mark  $\langle u, v \rangle$ ;
3   for  $i \leftarrow 1$  to  $l$  do
4     foreach  $u \in \mathcal{P}, v \in \Phi(u), \langle u, v \rangle$  is marked do
5       //Construct bipartite graph  $\mathcal{B}_{u,v}$ 
6        $N_{\mathcal{P}}(u), N_G(v)$ : neighbors of  $u, v$ ;
7       foreach  $u' \in N_{\mathcal{P}}(u), v' \in N_G(v)$  do
8          $\mathcal{B}_{u,v}(u', v') \leftarrow \begin{cases} 1 & \text{if } v' \in \Phi(u'); \\ 0 & \text{otherwise.} \end{cases}$ 
9       end
10      if  $\mathcal{B}_{u,v}$  has a semi-perfect matching then
11        Unmark  $\langle u, v \rangle$ ;
12      else
13        Remove  $v$  from  $\Phi(u)$ ;
14        foreach  $u' \in N_{\mathcal{P}}(u), v' \in N_G(v), v' \in \Phi(u')$ 
15          do Mark  $\langle u', v' \rangle$ ;
16      end
17      if there is no marked  $\langle u, v \rangle$  then break;
18    end
19  end

```

If the node attributes are selective, e.g., many unique attribute values, then one can index the node attributes using a **B-tree or hashtable, and store the neighborhood subgraphs or profiles as well**. Retrieval is done by indexed access to the node attributes, followed by pruning using neighborhood subgraphs or profiles. Otherwise, if the node attributes are not selective, one may have to index the neighborhood subgraphs or profiles. **Recent graph indexing techniques** [7, 15, 19, 29, 31, 34, 35, 36, 37] or **multi-dimensional indexing methods such as R-trees** can be used for this purpose.

4.3 Joint Reduction of Search Space

We reduce the overall search space iteratively by the refinement procedure of pseudo subgraph isomorphism, an approximation algorithm previously developed in [15]. Essentially, this technique checks for each node u and its feasible mate v whether the adjacent subtree of u in \mathcal{P} is sub-isomorphic to that of v in G . This check can be done recursively on the depth of the adjacent subtrees: the level l subtree of u is sub-isomorphic to that of v only if a bipartite graph $\mathcal{B}_{u,v}$ between neighbors of u and v has a **semi-perfect matching**, i.e., **all neighbors of u are matched**. In the bipartite graph at level l , an edge is present between two nodes u' and v' only if the level $l-1$ subtree of u' is sub-isomorphic to that of v' .

Algorithm 2 outlines the refinement procedure. At each iteration (lines 3–18), a bipartite graph $\mathcal{B}_{u,v}$ is constructed for each u and its feasible mate v (lines 5–9). If $\mathcal{B}_{u,v}$ has no semi-perfect matching, then v is removed from $\Phi(u)$, thus reducing the search space (line 13).

The algorithm has two implementation improvements on the refinement procedure discussed in [15]. First, it avoids unnecessary bipartite matchings. A pair $\langle u, v \rangle$ is marked if it needs to be checked for semi-perfect matching (lines 2,

4). If the semi-perfect matching exists, then the pair is unmarked (lines 10–11). Otherwise, the removal of v from $\Phi(u)$ (line 13) may affect the existence of semi-perfect matchings of the neighboring $\langle u', v' \rangle$ pairs. As a result, these pairs are marked and checked again (line 14). Second, the $\langle u, v \rangle$ pairs are stored and manipulated using a hashtable instead of a matrix. This reduces the space and time complexity from $O(k \cdot n)$ to $O(\sum_{i=1}^k |\Phi(u_i)|)$. The overall time complexity is $O(l \cdot \sum_{i=1}^k |\Phi(u_i)| \cdot (d_1 d_2 + M(d_1, d_2)))$ where l is the refinement level, d_1 and d_2 are maximum degrees of \mathcal{P} and G respectively, and $M()$ is the time complexity of maximum bipartite matching ($O(n^{2.5})$ for Hopcroft and Karp's algorithm [16]).

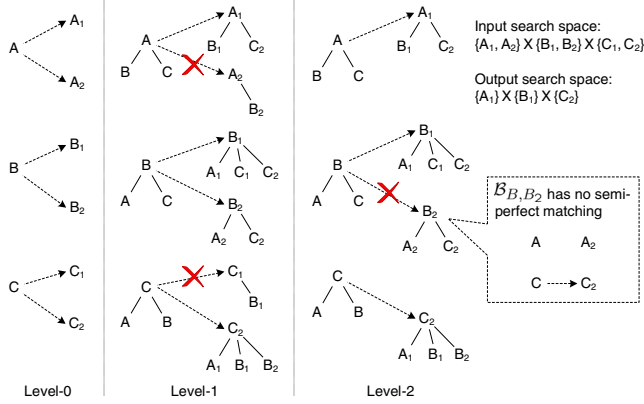


Figure 14: Refinement of the search space

Figure 14 shows an execution of Algorithm 2 on the example in Figure 12. At level 1, A_2 and C_1 are removed from $\Phi(A)$ and $\Phi(C)$, respectively. At level 2, B_2 is removed from $\Phi(B)$ since the bipartite graph \mathcal{B}_{B, B_2} has no semi-perfect matching (note that A_2 was already removed from $\Phi(A)$).

Whereas the neighborhood subgraphs discussed in Section 4.2 prune infeasible mates by using local information, the refinement procedure in Algorithm 2 prunes the search space globally. The global pruning has a larger overhead and is dependent on the output of the local pruning. Therefore, both pruning methods are indispensable and should be used together.

4.4 Optimization of Search Order

Next, we consider the search order of Algorithm 1. The goal here is to find a good search order for the nodes. Since the search procedure is equivalent to multiple joins, it is similar to a typical query optimization problem [5]. Two principal issues need to be considered. One is the cost model for a given search order. The other is the algorithm for finding a good search order. The cost model is used as the objective function of the search algorithm. Since the search algorithm is relatively standard (e.g., dynamic programming, greedy algorithm), we focus on the cost model and illustrate that it can be customized in the domain of graphs.

4.4.1 Cost Model

A search order (aka a query plan) can be represented as a rooted binary tree whose leaves are nodes of the graph pattern and each internal node is a join operation. Figure 15 shows two examples of search orders.

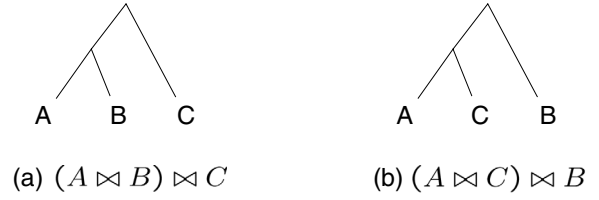


Figure 15: Two examples of search orders

We estimate the cost of a join (a node in the query plan tree) as the product of cardinalities of the collections to be joined. The cardinality of a leaf node is the number of feasible mates. The cardinality of an internal node can be estimated as the product of cardinalities of collections reduced by a factor γ .

Definition 8. (Result size of a join) The result size of join i is estimated by

$$Size(i) = Size(i.left) \times Size(i.right) \times \gamma(i)$$

where $i.left$ and $i.right$ are the left and right child nodes of i respectively, and $\gamma(i)$ is the reduction factor.

A simple way to estimate the reduction factor $\gamma(i)$ is to approximate it by a constant. A more elaborate way is to consider the probabilities of edges in the join: Let $\mathcal{E}(i)$ be the set of edges involved in join i , then

$$\gamma(i) = \prod_{e(u,v) \in \mathcal{E}(i)} P(e(u,v))$$

where $P(e(u,v))$ is the probability of edge $e(u,v)$ conditioned on u and v . This probability can be estimated as

$$P(e(u,v)) = \frac{freq(e(u,v))}{freq(u) \cdot freq(v)}$$

where $freq()$ denotes the frequency of the edge or node in the large graph.

Definition 9. (Cost of a join) The cost of join i is estimated by

$$Cost(i) = Size(i.left) \times Size(i.right)$$

Definition 10. (Cost of a search order) The total cost of a search order Γ is estimated by

$$Cost(\Gamma) = \sum_{i \in \Gamma} Cost(i)$$

For example, let the input search space be $\{A_1\} \times \{B_1, B_2\} \times \{C_2\}$. If we use a constant reduction factor γ , then $Cost(A \bowtie B) = 1 \times 2 = 2$, $Size(A \bowtie B) = 2\gamma$, $Cost((A \bowtie B) \bowtie C) = 2\gamma \times 1 = 2\gamma$. The total cost is $2 + 2\gamma$. Similarly, the total cost of $(A \bowtie C) \bowtie B$ is $1 + 2\gamma$. Thus, the search order $(A \bowtie C) \bowtie B$ is better than $(A \bowtie B) \bowtie C$.

4.4.2 Search Order

The number of all possible search orders is exponential in the number of nodes. It is expensive to enumerate all of them. As in many query optimization techniques, we consider only left-deep query plans, i.e., the outer node of each join is always a leaf node. The traditional dynamic programming would take an $O(2^k)$ time complexity for a

graph pattern of size k . This is not scalable to large graph patterns. Therefore, we adopt a simple greedy approach in our implementation: at join i , choose a leaf node that minimizes the estimated cost of the join.

5. EXPERIMENTAL STUDY

In this section, we evaluate the performance of the proposed access methods on large real and synthetic graphs. The access methods were written in Java with Sun JDK 1.6.

For comparison, we implement the SQL-based approach to graph pattern matching as described in Figure 2, except that only a count of the results is returned so as to minimize the communication cost. We use MySQL 5.0.45 as the database server. MySQL is configured as: storage engine=MyISAM (non-transactional), key_buffer_size=256M. Other parameters are set as default. For each large graph, two tables $V(\text{vid}, \text{label})$ and $E(\text{vid1}, \text{vid2})$ are created as in Figure 2. B-tree indices are built for each field of the tables.

All the experiments were run on an AMD Athlon 64 X2 4200+ 2.2GHz machine with 2GB memory running MS Win XP Pro.

5.1 Biological Network

We experimented with a yeast protein interaction network [1]. This graph consists of 3112 nodes and 12519 edges. Each node represents a unique protein and each edge represents an interaction between proteins.

To allow for meaningful queries, we add Gene Ontology (GO) [12] information to the proteins. This ontology is a hierarchy of categories that describes cellular components, biological processes, and molecular functions of genes and their products (proteins). Each GO term is a node in the hierarchy and has one or more parent GO Terms. Each protein has one or more GO terms. The original GO terms in the yeast network consist of 2205 distinct labels. We relax these GO terms by using their high level ancestors, which consist of 183 distinct labels. This relaxation allows us to find more meaningful and general patterns. At the same time, it makes the problem harder since the labels are less selective. We index the node labels using a hashtable, and store the neighborhood subgraphs and profiles with radius 1 as well.

We evaluate the access methods using two extreme kinds of graphs as queries: cliques and paths. For biological datasets, the former may correspond to protein complexes, while the latter may correspond to transcriptional or signaling pathways.

5.1.1 Clique Queries

We generated clique queries by varying the sizes of cliques from 2 to 7 (sizes greater than 7 have no answers). For each size, we generate a complete graph and assign each node a random label. The random label is selected from the top 40 most frequent labels. We generate 1000 clique queries and average the results. The queries are divided into two groups according to the number of answers returned: low hits (less than 100 answers) and high hits (more than 100 answers). Queries having no answers are not counted in the statistics. If a query has too many hits (more than 1000), then the graph pattern matching is terminated immediately and the query is counted in the group of high hits.

We evaluate the pruning power of the retrieval methods

and the refinement step by comparing their resulting search space to the baseline search space. The baseline search space consists of feasible mates by checking node attributes only. The *reduction ratio* of search space is defined by

$$\gamma(\Phi, \Phi_0) = \frac{|\Phi(u_1)| \times \dots \times |\Phi(u_k)|}{|\Phi_0(u_0)| \times \dots \times |\Phi_0(u_k)|}$$

where Φ_0 refers to the baseline search space.

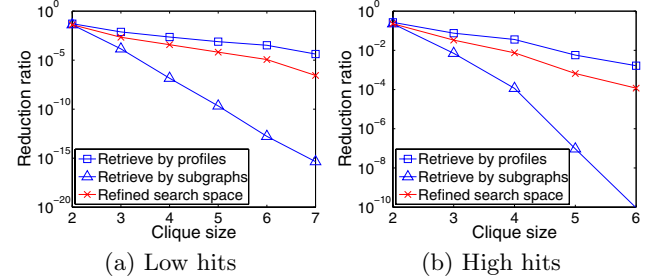


Figure 16: Search space for clique queries

Figure 16 shows the reduction ratios of search space by different methods. “Retrieve by profiles” finds feasible mates by checking profiles and “Retrieve by subgraphs” finds feasible mates by checking neighborhood subgraphs (Section 4.2). “Refined search space” refers to search space reduction by the refinement step discussed in Section 4.3 (the input search space is generated by “Retrieve by profiles”). In the refinement step, the maximum refinement level ℓ is set as the size of the query. As can be seen from the figure, the refinement procedure always reduces the search space retrieved by profiles. Retrieval by subgraphs results in the smallest search space. This is due to the fact that neighborhood subgraphs for a clique query is actually the entire clique.

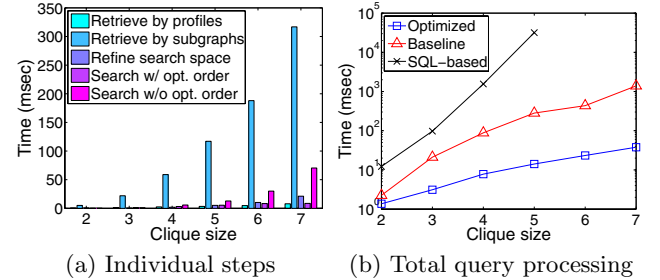


Figure 17: Running time for clique queries (low hits)

Figure 17(a) shows the average processing time for individual steps under varying clique sizes. The individual steps include retrieval by profiles, retrieval by subgraphs, refinement, search with the optimized order (Section 4.4), and search without the optimized order. The time for finding the optimized order is negligible since we take a greedy approach in our implementation. As shown in the figure, retrieval by subgraphs has a large overhead although it produces a smaller search space than retrieval by profiles. Another observation is that the optimized order improves upon the search time.

Figure 17(b) shows the average total query processing time in comparison to the SQL-based approach on low hits queries. The “Optimized” processing consists of retrieval by

profiles, refinement, optimization of search order, and search with the optimized order. The “Baseline” processing consists of retrieval by node attributes and search without the optimized order on the baseline space. The query processing time in the “Optimized” case is improved greatly due to the reduced search space.

The SQL-based approach takes much longer time and does not scale to large clique queries. This is due to the unpruned search space and the large number of joins involved. Whereas our graph pattern matching algorithm (Section 4.1) is exponential in the number of nodes, the SQL-based approach is exponential in the number of edges. For instance, a clique of size 5 has 10 edges. This requires 20 joins between nodes and edges (as illustrated in Figure 2).

5.1.2 Path Queries

The second class of queries that we considered is at the other extreme of connectivity: path queries. The size of this query was varied between 2 and 10. The input search space for the refinement step is “Retrieve by profiles.” Other settings were similar to that for clique queries.

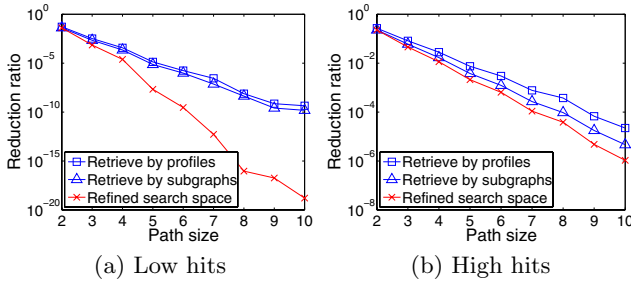


Figure 18: Search space for path queries

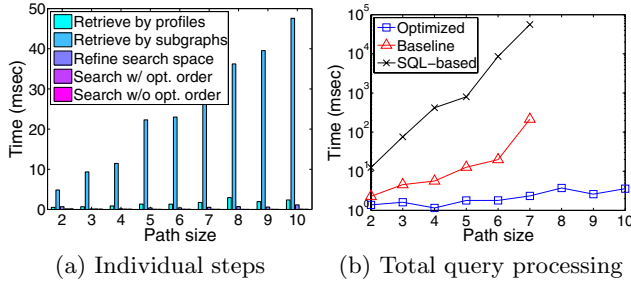


Figure 19: Running time for path queries (low hits)

Figure 18 shows the reduction ratios for path queries. Unlike clique queries, the refinement step (Section 4.3) here produces the smallest search space. This is because the refinement step reduces the overall search space, whereas local pruning (either by profiles or by subgraphs) cannot capture global information.

Figure 19(a) shows the individual time for path queries (some of the plots are not visible because of small values). The individual steps take less time than those for clique queries. This is because the path queries are simpler than clique queries. The search steps (with and without the optimized order) are negligible. This is a result of the small search space produced by the refinement step. Figure 19(b) shows the total time for path queries. The “Optimized” pro-

cessing here consists of retrieval by profiles, refinement, optimization of search order, and search with the optimized order. Again, the “Optimized” processing improves greatly upon the “Baseline” processing. The SQL-based processing scales better than in the case of clique queries since path queries requires less number of joins. But it still takes much more time than “Optimized” processing.

5.2 Synthetic Graphs

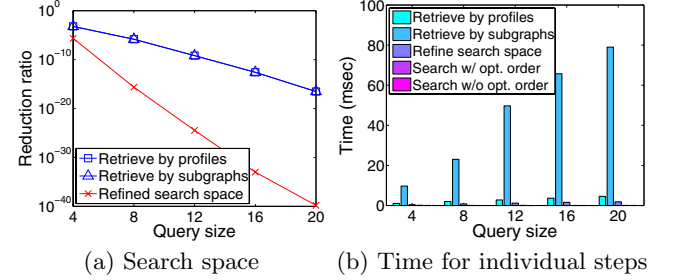


Figure 20: Search space and running time for individual steps (synthetic graphs, low hits)

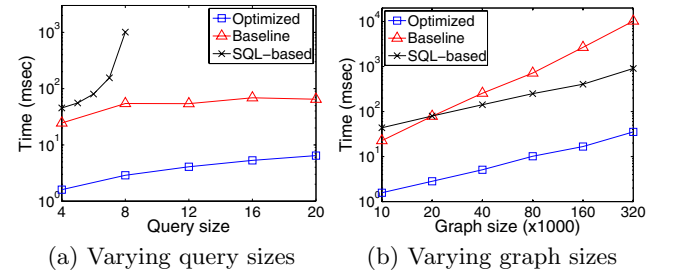


Figure 21: Running time (synthetic graphs, low hits)

We also evaluate the access methods on synthetic graphs. The synthetic graphs are generated using a simple Erdős-Rényi [11] random graph model: generate n nodes, and then generate m edges by randomly choosing two end nodes. Each node is assigned a label (100 distinct labels in total). The distribution of the labels follows Zipf’s law, i.e., probability of the x^{th} label $p(x)$ is proportional to x^{-1} . The queries are generated by randomly extracting a connected subgraph from the synthetic graph.

We first fix the size of synthetic graphs n as $10K$, $m = 5n$, and vary the query size between 4 and 20. Figure 20 and Figure 21(a) shows the search space and time. We found that the best combination is retrieval by profiles, followed by the refinement step, and search with the optimized order. Again, the search steps take little time due to the small search space produced by the refinement step. The SQL-based processing takes much longer time and is not appropriate for large queries.

Then, we set the query size to 4, and vary the size of synthetic graphs n between $10K$ and $320K$ and $m = 5n$. Figure 21(b) shows the total time. As can be seen, the “Optimized” processing scales better than the “Baseline” processing for increasing graph sizes. The SQL-based approach also scales to large graphs for small queries, but it still takes much longer time than the “Optimized” processing.

To summarize our experiments, retrieval by profiles produces small search space with little overhead. The refinement step (Section 4.3) greatly reduces the search space. The overhead of the search step is well compensated by the extensive reduction of search space. A practical combination would be retrieval by profiles, followed by refinement, and then search with an optimized order. This combination scales well with various query sizes and graph sizes. SQL-based processing is not scalable to large queries. Overall, the optimized processing performs orders of magnitude better than the SQL-based approach. While small improvements in SQL-based implementations can be achieved by careful tuning and other optimizations, the results show that query processing in the graph domain has clear advantages.

6. RELATED WORK

A number of query languages have been proposed for graphs. GraphLog [10] represents both data and queries as graphs. In terms of expressive power, GraphLog was showed equivalent to stratified linear Datalog. GOOD [14] is a graph-oriented object data model that transforms graphs by node and edge addition and deletion. GraphDB [13] uses an object-oriented data model for database graphs and queries. GOQL [30] also uses an object-oriented graph data model and is an extension to OQL. PQL [21] is a pathway query language for biological networks. The language is derived from SQL and is implemented on top of an RDBMS.

Some of the recent interest in graph query languages has been spurred by Semantic Web and the accompanying SPARQL query language [23]. This model describes a graph by a set of triples, each of which describes an (attribute, value) pair or an interconnection between two nodes. The SPARQL query language works primarily through a pattern which is a constraint on a single node. All possible matchings of the pattern are returned from the graph database. A general graph query language could be more powerful by providing primitives for expressing constraints on the entire result graph simultaneously.

In XML databases, TAX [18] is a tree algebra for XML. TAX uses a pattern tree to match interesting nodes. The pattern tree consists of a tree structure and a predicate on nodes of the tree. GraphQL generalizes this idea to describe a graph pattern.

GraphQL is different from the above query languages in that graphs are chosen as the basic unit. In comparison to SQL, GraphQL has a similar algebraic system, but the algebraic operators are defined directly on graphs. In comparison to OODB, GraphQL queries are set-oriented, whereas OODB accesses objects in a navigational manner. On data representation, GraphQL is semistructured and does not cast strict and pre-defined data types or schemas on graphs. In contrast, SQL presumes a strict schema in order to store data. Furthermore, GraphQL can be efficiently implemented using graph specific optimizations. Table 1 outlines the main differences between GraphQL and other query languages.

Graph grammars have been used previously for modeling visual languages and graph transformations in various domains [26, 25]. Our work is different in that our emphasis has been on a query language and database implementations.

In graph indexing, GraphGrep [29] uses enumerated paths as index features to filter unmatched graphs. GIndex [35] uses discriminative frequent fragments as index features to

Table 1: Comparison of different query languages

Language	Basic unit	Query style	Semi-structured
GraphQL	graphs	set-oriented	yes
SQL	tuples	set-oriented	no
TAX	trees	set-oriented	yes
GraphLog	nodes/edges	logic pro.	-
OODB (GOOD, GraphDB, GOQL)	nodes/edges	navigational	no

improve filtering rates and reduce index sizes. Closure-tree [15] organizes graphs into a tree-based index structure using graph closures as the bounding boxes. GString [19] converts graph querying to subsequence matching. TreePi [36] uses frequent subtrees as index features. Williams et al. [34] decompose graphs and hash the canonical forms of the resulting subgraphs. SAGA [31] enumerates fragments of graphs and answers are generated by assembling hits of the query fragments. FG-index [7] uses frequent subgraphs as index features. Frequent graph queries are answered without verification and infrequent queries require only a small number of verifications. Zhao et al. [37] show that frequent tree-features plus a small number of discriminative graphs are better than frequent graph-features. While the above techniques can be used as access methods for the case of a large collection of small graphs, this paper addresses access methods for the case of a single large graph.

Another line of graph indexing addresses reachability queries in large directed graphs [6, 8, 9, 27, 32, 33]. In a reachability query, two nodes are given and the answer is whether there exists a path between the two nodes. Reachability queries correspond to recursive graph patterns which are paths (Figure 3(e)). Indexing and processing of reachability queries are generally based on spanning trees with pre/post-order labeling [6, 32, 33] or 2-hop-cover [8, 9, 27]. These techniques can be incorporated into access methods for recursive graph pattern queries.

7. CONCLUSION

We have presented GraphQL, a novel query language for graphs with arbitrary attributes and sizes. GraphQL has a number of appealing features. Graphs are the basic unit and graph structures are composable using the notion of formal languages for graphs. We developed efficient access methods for the selection operator using the idea of neighborhood subgraphs and profiles, refinement of the overall search space, and optimization of the search order. Experimental studies on real and synthetic graphs validated the access methods.

In summary, graphs are prevalent in multiple domains. This paper has demonstrated the benefits of working with native graphs for queries and database implementations. Translations of graphs into relations are unnatural and cannot take advantage of graph-specific heuristics. The coupling of graph-based querying and native graph-based databases produces interesting possibilities from the point of view of expressiveness and implementation techniques. We have barely scratched the surface and much more needs to be done in matching characteristics of queries and databases to appropriate heuristics. The results of this paper are an important first step in this regard.

Acknowledgments

This work was supported in part by NSF grants IIS-0612327 and DBI-0213903. We would like to thank members of the DBL at UCSB for reading drafts of the paper.

Repeatability assessment result

All the results in this paper were verified by the SIGMOD repeatability committee. The code and data used in the paper are available at <http://www.sigmod.org/codearchive/sigmod2008/>.

8. REFERENCES

- [1] S. Asthana et al. Predicting protein complex membership using probabilistic network reliability. *Genome Research*, May 2004.
- [2] S. Berretti, A. D. Bimbo, and E. Vicario. Efficient matching and indexing of graph models in content-based retrieval. In *IEEE Trans. on Pattern Analysis and Machine Intelligence*, volume 23, 2001.
- [3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C, <http://www.w3.org/TR/xquery/>, 2007.
- [4] C. Branden and J. Tooze. *Introduction to protein structure*. Garland, 2 edition, 1998.
- [5] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.
- [6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB '05*, pages 493–504, 2005.
- [7] J. Cheng, Y. Ke, W. Ng, and A. Lu. FG-Index: towards verification-free query processing on graph databases. In *Proc. of SIGMOD '07*, 2007.
- [8] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.
- [9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.
- [10] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, 1990.
- [11] P. Erdős and A. Rényi. On random graphs I. *Publ. Math. Debrecen*, (6):290–297.
- [12] Gene Ontology. <http://www.geneontology.org/>.
- [13] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *Proc. of VLDB'94*, pages 297–308, 1994.
- [14] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. of PODS '90*, pages 417–424, 1990.
- [15] H. He and A. K. Singh. Closure-Tree: An Index Structure for Graph Queries. In *Proc. of ICDE'06*, Atlanta, 2006.
- [16] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 1973.
- [17] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.
- [18] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. of DBPL'01*, 2001.
- [19] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. GString: A novel approach for efficient search in graph databases. In *ICDE*, 2007.
- [20] J. Lee, J. Oh, and S. Hwang. STRG-Index: Spatio-temporal region graph indexing for large video databases. In *Proc. of SIGMOD*, 2005.
- [21] U. Leser. A query language for biological networks. *Bioinformatics*, 21:ii33–ii39, 2005.
- [22] F. Manola and E. Miller. RDF Primer. W3C, <http://www.w3.org/TR/rdf-primer/>, 2004.
- [23] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C, <http://www.w3.org/TR/rdf-sparql-query/>, 2007.
- [24] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, chapter 24 Deductive Databases. McGraw-Hill, third edition, 2003.
- [25] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing. In *11th International IEEE Symposium on Visual Languages*, 1995.
- [26] G. Rozenberg (Ed.). *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.
- [27] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proc. of ICDE '05*, pages 360–371, 2005.
- [28] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.
- [29] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. of PODS*, 2002.
- [30] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.
- [31] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2), 2007.
- [32] S. Tril and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD '07*, pages 845–856, 2007.
- [33] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. of ICDE '06*, page 75, 2006.
- [34] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.
- [35] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A frequent structure-based approach. In *Proc. of SIGMOD*, 2004.
- [36] S. Zhang, M. Hu, and J. Yang. TreePi: A novel graph indexing method. In *ICDE*, 2007.
- [37] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta \geq graph. In *Proc. of VLDB*, pages 938–949, 2007.