Chapter 4

QUERY LANGUAGE AND ACCESS METHODS FOR GRAPH DATABASES*

Huahai He*
Google Inc.
Mountain View, CA 94043, USA
huahai@google.com

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 present a graph query language (GraphQL) that supports bulk operations on graphs with arbitrary structures and annotated attributes. In this language, graphs are the basic unit of information and each query manipulates one or more collections of graphs at a time. The core of GraphQL is 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 graph specific optimizations outperform an SQL-based implementation by orders of magnitude.

*This is a revised and extended version of the article "Graphs-at-a-time: Query Language and Access Methods for Graph Databases", Huahai He and Ambuj K. Singh, In Proceedings of the 2008 ACM SIGMOD Conference, http://doi.acm.org/10.1145/1376616.1376660. Reprinted with permission of ACM.
*Work done while at the University of California, Santa Barbara.

Keywords: Graph query language, Graph algebra, Graph pattern matching

1. Introduction

Data in multiple domains can be naturally modeled as graphs. Examples include the Semantic Web [32], GIS, images [3], videos [24], 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

Generally, 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 [5]. 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 protein complex may be specified as a graph with nodes (proteins) labeled by Gene Ontology [14] terms.
- Find all instances from an RDF (Resource Description Framework [26]) 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 4.1 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).

Consider an SQL-based approach to the sample graph query. The graph in the database can be modeled in two tables. Table V(vid, label) stores the set of nodes[1] 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 expressed as an SQL query with multiple joins:

[1]For convenience, the terms "vertex" and "node" are used interchangeably in this chapter.

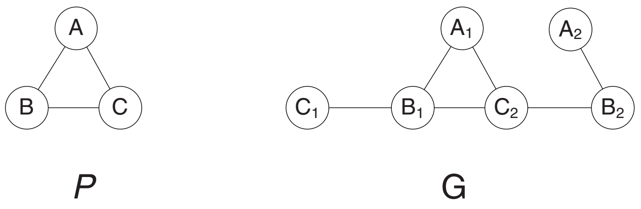


Figure 4.1. A sample graph query and a graph in the database

```
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
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;
```

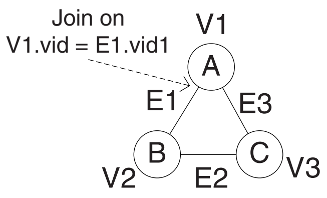


Figure 4.2. SQL-based implementation

As can be seen in the above example, although the graph query can be expressed by an SQL query, the global view of graph structures is lost. 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 GraphQL

This chapter presents *GraphQL*, a graph query language in which graphs are the basic unit of information from the ground up. GraphQL uses a graph pattern as the main building block of a query. A graph pattern consists of a graph structure and a predicate on attributes of the graph. Graph pattern matching is defined by combining subgraph isomorphism and predicate evaluation. The core of GraphQL is a bulk *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. In terms of expressive power, GraphQL is relationally complete and is contained in Datalog [28]. The nonrecursive version of GraphQL is equivalent to the relational algebra.

The chapter then describes efficient processing of the selection operator over large graph databases (either a single large graph or a large collection of graphs). We first present a basic graph pattern matching algorithm, and then apply three graph specific optimization techniques to the basic algorithm. The first technique prunes the search space locally using neighborhood subgraphs or their profiles. The second technique performs global pruning using an approximation algorithm called pseudo subgraph isomorphism [17]. The third technique optimizes the search order based on a cost model for graphs. Experimental study shows that the combination of these three techniques allows us to scale to both large queries and large graphs.

GraphQL has a number of distinct features:

1 Graph structures and structural operations are described by the notion of formal languages for graphs. This notion is useful for manipulating graphs and is the basis of the query language (Section 2).

2 A graph algebra is defined along the line of the relational algebra. Each graph algebraic operator manipulates graphs or sets of graphs. 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, the graph algebra is relationally complete and is contained in Datalog (Section 3.3).

3 An efficient implementation of the selection operator over large graphs is presented. Experimental results on large real and synthetic graphs show that graph specific optimizations outperform an SQL-based implementation by orders of magnitude (Sections 4 and 5).

2. Operations on Graph Structures

In order to define graph patterns and operations on graph structures, we need a formal way to describe graph structures and how they can be combined into new graph structures. As such we extend the notion of formal languages [20] from the string domain to the graph domain. The notion deals with graph structures only. Description of attributes on graphs will be discussed in the next section.

In existing formal languages (e.g., regular expressions, context-free languages), a formal grammar consists of a finite set of terminals and nonterminals, and a finite set of production rules. A production rule consists of a nonterminal on the left hand side and a sequence of terminals and nonterminals on the right hand side. The production rules are used to derive strings of characters. Strings are the basic units of information.

In a *formal language for graphs*, the basic units are graph structures instead of strings. The nonterminals, called *graph motifs*, are either simple graphs or composed of other graph motifs by means of *concatenation*, *disjunction*, or *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.

A simple graph motif represents a graph with constant structure. It consists of a set of nodes and a set of edges. Each node, edge, or graph is identified by a *variable* if it needs to be referenced elsewhere. Figure 4.3 shows a simple graph motif and its graphical representation.

```
graph G1 {
   node v1, v2, v3;
   edge e1 (v1, v2);
   edge e2 (v2, v3);
   edge e3 (v3, v1);
}
```

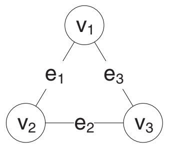


Figure 4.3. A simple graph motif

A complex graph motif consists of one or more graph motifs by concatenation, disjunction, or repetition. In the string domain, a string connects to other strings implicitly through its head and tail. In the graph domain, a graph may connect to other graphs in a structural way. These interconnections need to be explicitly specified.

2.1 Concatenation

A graph motif can be composed of two or more graph motifs. The constituent motifs are either left unconnected or concatenated in one of two ways. One way is to connect nodes in each motif by new edges. Figure 4.4(a) shows an example of concatenation by edges. Graph motif G_2 is composed of two motifs G_1 of Figure 4.3. The two motifs are connected by two edges. To avoid name conflicts, alias names of G_1 are used.

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 4.4(b) shows an example of concatenation by unification.

Concatenation is useful for defining Cartesian product and join operations on graphs.

2.2 Disjunction

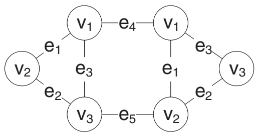
A graph motif can be defined as a disjunction of two or more graph motifs. Figure 4.5 shows an example of disjunction. In graph motif G_4, two anonymous graph motifs are declared (comprising of node v_3 or nodes v_3 and v_4). Only one of them is selected and connected to the rest of G_4. In disjunction, all the constituent graph motifs should have the same "interface" to the outside.

2.3 Repetition

A graph motif may be defined by itself to derive recursive graph structures. Figure 4.6(a) 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 $Path$.

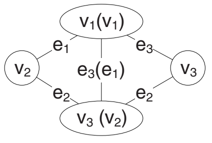
Recursions in the graph domain are not limited to paths and cycles. Figure 4.6(b) illustrates an example where the repetition unit is a graph motif. Motif G_5 contains an arbitrary number of motif G_1 and a root node v_0. The

```
graph G2 {
  graph G1 as X;
  graph G1 as Y;
  edge e4 (X.v1, Y.v1);
  edge e5 (X.v3, Y.v2);
}
```



(a)

```
graph G3 {
  graph G1 as X;
  graph G1 as Y;
  unify X.v1, Y.v1;
  unify X.v3, Y.v2;
}
```



(b)

Figure 4.4. (a) Concatenation by edges, (b) Concatenation by unification

```
graph G4 {
  node v1, v2;
  edge e1 (v1, v2);
  {
    node v3;
    edge e2 (v1, v3);
    edge e3 (v2, v3);
  } | {
    node v3, v4;
    edge e2 (v1, v3);
    edge e3 (v2, v4);
    edge e4 (v3, v4);
  };
}
```

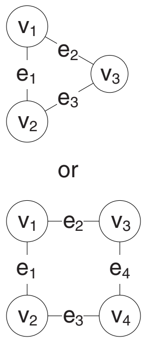
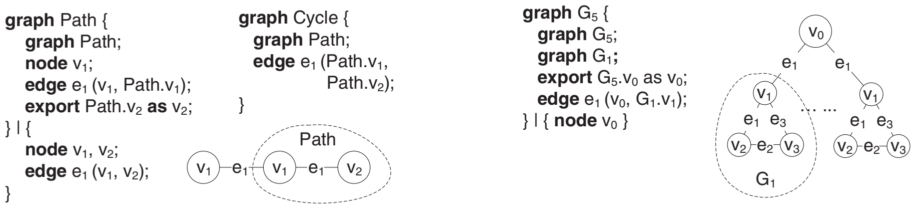


Figure 4.5. Disjunction

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.



```
graph Path {
  graph Path;
  node v1;
  edge e1 (v1, Path.v1);
  export Path.v2 as v2;
} | {
  node v1, v2;
  edge e1 (v1, v2);
}
```

```
graph Cycle {
  graph Path;
  edge e1 (Path.v1,
           Path.v2);
}
```

```
graph G5 {
  graph G5;
  graph G1;
  export G5.v0 as v0;
  edge e1 (v0, G1.v1);
} | { node v0 }
```

(a) (b)

Figure 4.6. (a) Path and cycle, (b) Repetition of motif G_1

3. Graph Query Language

This section presents the GraphQL query language. We first describe the data model. Next, we define graph patterns and graph pattern matching. We then present a graph algebra and its bulk operators which is the core of the graph query language. Finally, we illustrate the syntax of the graph query language through an example.

3.1 Data Model

Graphs in the real world contain not only graph structural information, but also attributes on nodes and edges. In GraphQL, we use a *tuple*, a list of name and value pairs, to represent the attributes of each node, edge, or graph. A tuple may have an optional *tag* that denotes the tuple type. Tuples are *annotated* to the graph structures so that the representations of attributes and structures are clearly separate. Figure 4.7 shows a sample graph that represents a paper (the graph has no edges). 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 4.7. A sample graph with attributes

In the relational model, tuples are the basic unit of information. Each algebraic operator manipulates collections of tuples. A relational query is always equivalent to an algebraic expression which is a combination of the operators. A relational database consists of one or more tables (relations) 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. A graph database consists of one or more collections of graphs. Unlike the relational model, graphs in a collection do not necessarily have identical structures and attributes. However, they can still be processed in a uniform way by binding to a graph pattern.

The GraphQL data model is similar to the TAX model [22] as for XML. In TAX, trees are the basic unit and the operators work on collections of trees. Trees in a collection have similar but not identical structures and attributes. This is captured by a pattern tree.

3.2 Graph Patterns

A graph pattern is the main building block of a graph query. Essentially, it consists of a graph motif and a predicate on attributes of the motif. The graph motif specifies constraints on graph structures and the predicate specifies constraints on attributes. A graph pattern is used to select graphs of interest.

Definition 4.1. *(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 $\mathcal{F}$ is a combination of boolean or arithmetic comparison expressions. Figure 4.8 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.

```
graph P {                              graph P {
  node v1;                               node v1 where name="A";
  node v2;                   or          node v2 where year>2000;
} where v1.name="A"                    };
  and v2.year>2000;
```

Figure 4.8. A sample graph pattern

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

Definition 4.2. *(Graph Pattern Matching) A graph pattern $\mathcal{P}(\mathcal{M}, \mathcal{F})$ is matched with a graph G if there exists an injective mapping ϕ: $V(\mathcal{M}) \to V(G)$ 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 2.3). A recursive graph pattern is matched with a graph if one of its derived motifs is matched with the graph.

Mapping Φ:
$\Phi(P.\mathrm{v}_1) \rightarrow \mathrm{G.v}_2$
$\Phi(P.\mathrm{v}_2) \rightarrow \mathrm{G.v}_1$

Figure 4.9. A mapping between the graph pattern in Figure 4.8 and the graph in Figure 4.7

Figure 4.9 shows an example of graph pattern matching between the pattern in Figure 4.8 and the graph in Figure 4.7.

If a graph pattern is matched to a graph, the binding between them can be used to access the graph (either graph structural information or attributes on the graph). As a graph pattern can match many graphs, this allows us to access a collection of graphs uniformly even though the graphs may have heterogenous structures and attributes. We use a *matched graph* to denote the binding between a graph pattern and a graph.

Definition 4.3. *(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 a matched graph is formally defined by a triple, it has all characteristics of a graph. Thus, all terms and conditions that apply to a graph also apply to a matched graph. For example, a collection of matched graphs is also a collection of graphs. As such it can match another graph pattern, resulting in another collection of matched graphs (two levels of bindings).

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 3.3.0.

3.3 Graph Algebra

We define a graph algebra along the lines of the relational algebra. This allows us to inherit the solid foundation and experience of the relational model. All relational operators have their counterparts or alternatives in the graph algebra. These operators are defined directly on graphs since graphs are now the basic units of information. In particular, the selection operator is generalized to graph pattern matching; a composition operator is introduced to generate new graphs from matched graphs.

Selection (σ). A selection operator σ takes a graph pattern $\mathcal{P}$ and a collection of graphs $\mathcal{C}$ as arguments, and produces a collection of matched graphs as output. The result is denoted by $\sigma_{\mathcal{P}}(\mathcal{C})$:

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

A graph database may consist of a single large graph, e.g., a social network. A single large graph and a collection of graphs are treated in the same way. A collection of graphs is a special case of a single large graph, whereas a single large graph is considered as many inter-connected or overlapping small graphs. These small graphs are captured by the graph pattern of the selection operator.

A graph pattern can match a graph many times. Thus, a selection could return many instances for each graph in the input collection. We use an option "exhaustive" to specify whether it should return one or all possible mappings between the graph pattern and the graph. Whether one or all mappings are required depends on the application.

Cartesian Product ($\times$) and Join ($\bowtie$). A Cartesian product operator takes two collections of graphs $\mathcal{C}$ and $\mathcal{D}$ as input, and produces a collection of graphs as output. Each graph in the output collection is composed of a graph from $\mathcal{C}$ and another from $\mathcal{D}$. The constituent graphs are unconnected:

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

As in the relational algebra, the join operator in the graph algebra 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. The constituent graphs are unconnected in the resultant graph. No new graph structures are generated. Figure 4.10 shows an example of valued join.

```
graph {
  graph G1, G2;
} where G1.id = G2.id;
```

Figure 4.10. An example of valued join

In a *structural join*, the constituent graphs can be concatenated by edges or unification. New graph structures are generated in the resultant graph. This is specified through a composition operator which is described next.

Composition (ω). Composition operators are used to generate new graphs from existing (matched) graphs. In order to specify the composition operators, we introduce the concept of graph templates.

Definition 4.4. *(Graph Template) A* graph template $\mathcal{T}$ *consists of a list of formal parameters which are graph patterns, and a template body which is defined by referring to the graph patterns.*

Once actual parameters (matched graphs) are given, a graph template is *instantiated* to a real graph. This is similar to invoking 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 $\mathcal{T}_{\mathcal{P}_1..\mathcal{P}_k}(G_1, ..., G_k)$.

```
TP = graph {
    node v1 <label=P.v1.name>;
    node v2 <label=P.v2.title>;
    edge e1 (v1, v2);
  }

          (a)
```

```
TP (G) = graph {
      node v1 <label="A">;
      node v2 <label="Title1">;
      edge e1 (v1, v2);
   }

          (b)
```

Figure 4.11. (a) A graph template with a single parameter $\mathcal{P}$, (b) A graph instantiated from the graph template. $\mathcal{P}$ and G are shown in Figure 4.8 and Figure 4.7.

Figure 4.11 shows a sample graph template and a graph instantiated from the graph template. $\mathcal{P}$ is the formal parameter of the 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.

Now we can define the composition operator. A *primitive composition* operator ω takes a graph template $\mathcal{T}_{\mathcal{P}}$ with a single parameter, and a collection of matched graphs $\mathcal{C}$ as input. It produces a collection of instantiated graphs as output:

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

Generally, a composition operator allows two or more collections of graphs as input. This can be expressed by a primitive composition operator and a Cartesian product operator, the latter of which combines multiple collections of graphs into one:

$$\omega_{\mathcal{T}_{\mathcal{P}_1,\mathcal{P}_2}}(\mathcal{C}_1, \mathcal{C}_2) = \omega_{\mathcal{T}_{\mathcal{P}}}(\mathcal{C}_1 \times \mathcal{C}_2),$$
$$\text{where } \mathcal{P} = \textbf{graph} \ \{ \ \textbf{graph} \ \ \mathcal{P}_1, \ \mathcal{P}_2; \}.$$

Other operators. 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 basic operators (selection, Cartesian product, primitive composition, union, and difference) are complete. Other operators and any algebraic expressions can be expressed as combinations of these five operators.

Algebraic laws are important for query optimization as they provide equivalent transformations of query plans. Since the 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 [4] as the syntax of our graph query language. The query syntax is shown in Appendix 4.A. We illustrate the syntax through an example.

```
graph P {
   node v1 <author>;
   node v2 <author>;
} where P.booktitle=”SIGMOD”;
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 4.12. A graph query that generates a co-authorship graph from the DBLP dataset

Figure 4.12 shows an example that generates a co-authorship graph 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 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 4.13 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.

```
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">;
      };
```

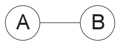
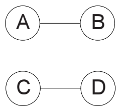
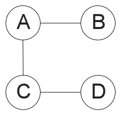
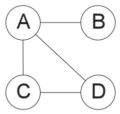
Iteration	Mapping	Co-authorship graph C
1	$\Phi(P.v_1) \rightarrow G_1.v_1$ $\Phi(P.v_2) \rightarrow G_1.v_2$	
2	$\Phi(P.v_1) \rightarrow G_2.v_1$ $\Phi(P.v_2) \rightarrow G_2.v_2$	
3	$\Phi(P.v_1) \rightarrow G_2.v_1$ $\Phi(P.v_2) \rightarrow G_2.v_3$	
4	$\Phi(P.v_1) \rightarrow G_2.v_2$ $\Phi(P.v_2)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

Figure 4.13. A possible execution of the Figure 4.12 query

3.5 Expressive Power

We now discuss the expressive power of GraphQL. We first show that the relational algebra (RA) is contained in GraphQL.

Theorem 4.5. *(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 4.6. *(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 4.14 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.

```
graph G <attr1=value1> {
  node v1, v2, v3;
  edge e1(v1, v2);
};
```

$\Longrightarrow$

```
graph('G').
node('G', 'G.v1').
node('G', 'G.v2').
node('G', 'G.v3').
edge('G', 'G.e1', 'G.v1', 'G.v2').
edge('G', 'G.e1', 'G.v2', 'G.v1').
attribute('G', 'attr1', value1).
```

Figure 4.14. The translation of a graph into facts of Datalog

For each graph pattern, we translate it into a rule of Datalog. Figure 4.15 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. □

```
graph P {
  node v2, v3;
  edge e1(v3, v2);
} where P.attr1 > value1;
```

$\Longrightarrow$

```
Pattern(P, V2, V3, E1):-
  graph(P),
  node(P, V2),
  node(P, V3),
  edge(P, E1, V3, V2),
  attribute(P, 'attr1', Temp),
  Temp > value1.
```

Figure 4.15. 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 4.7. *nr-GraphQL $\equiv$ RA.*

4. Implementation of the Selection Operator

We now discuss efficient implementation of the selection operator. Other graph algebraic operators can find their counterpart implementations in relational databases, and future research opportunities are open for graph specific optimizations.

Generally, graph databases can be classified into two categories. One category is a large collection of small graphs, e.g., chemical compounds. The selection operator returns a subset of the collection as answers. The main challenge in 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 [17, 34, 40]. Graph indexing plays a similar role for graph databases as B-trees for relational databases: only a small number of graphs need to be accessed. Scanning of the whole collection of graphs is not necessary.

In the second category, the graph database consists of one or a few very large graphs, e.g., protein interaction networks, Web information, social networks. Graphs in the answer set are not readily present in the database and need to be constructed from the single large graph. The challenge here is to accelerate the graph pattern matching itself. In this chapter, we focus on the second category.

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. Recursive graph pattern matching and disk-based access methods remain as future research directions.

4.1 Graph Pattern Matching

Graph pattern matching is essentially an extension of subgraph isomorphism with predication evaluation (Definition 4.2). Algorithm 4.1 outlines the basic graph pattern matching 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., "$u_1.label = u_2.label$", remain in the graph-wide predicate $\mathcal{F}$. For each node u in pattern $\mathcal{P}$, there is a set of candidate matched nodes in G with respect to $\mathcal{F}_u$. These nodes are called *feasible mates* of node u and is denoted by $\Phi(u)$:

Definition 4.8. *(Feasible Mates) The feasible mates $\Phi(u)$ of node u is the set of nodes in graph G that satisfies predicate F_u:*

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

The feasible mates of all nodes in the pattern define the search space of graph pattern matching:

Definition 4.9. *(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:*

$$\Phi(u_1) \times .. \times \Phi(u_k),$$

where k is the number of nodes in the graph pattern.

Algorithm 4.1: 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     $\Phi(u) \leftarrow \{v | v \in V(G), \mathcal{F}_u(v) = \textbf{true}\}$
 3     // Local pruning and retrieval of $\Phi(u)$ (Section 4.2)
 4 end
 5 // Reduce $\Phi(u_1) \times .. \times \Phi(u_k)$ globally (Section 4.3)
 6 // Optimize search order of $u_1, .., u_k$ (Section 4.4)
 7 Search(1);
 8 void Search($i$)
 9 begin
10     foreach $v \in \Phi(u_i)$, $v$ is free do
11         if not Check($u_i$, $v$) then continue;
12         $\phi(u_i) \leftarrow v$;
13         if $i < |V(\mathcal{P})|$ then 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 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
```

Algorithm 4.1 consists of two phases. The first phase (lines 1–4) retrieves the feasible mates for each node u in the pattern. The second phase (Lines 7–26) searches over the product $\Phi(u_1) \times .. \times \Phi(u_k)$ in a depth-first manner for subgraph isomorphism. Procedure Search(i) iterates on the i^{th} node to find feasible mappings for that node. Procedure 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 evaluation of edge predicates (line 22), another hashtable can be used to store evaluated pairs of edges.

The worst-case time complexity of Algorithm 4.1 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.

We now consider possible ways to accelerate Algorithm 4.1:

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 $\Phi(u_1) \times .. \times \Phi(u_k)$?

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

Node attributes can be indexed directly using traditional index structures such as B-trees. This allows for fast retrieval of feasible mates and avoids a full scan of all nodes. To reduce the size of feasible mates $\Phi(u_i)$'s 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 4.10. *(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.

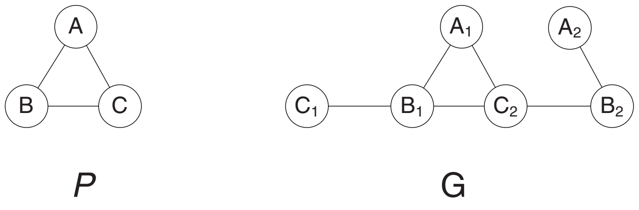
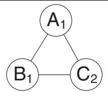
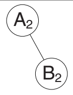
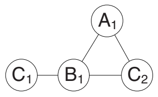
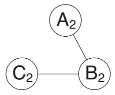
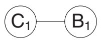
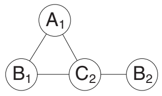


Figure 4.16. A sample graph pattern and graph

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

Search space
Retrieve by nodes: $\{A_1, A_2\} \times \{B_1, B_2\} \times \{C_1, C_2\}$
Retrieve by neighborhood subgraphs: $\{A_1\} \times \{B_1\} \times \{C_2\}$
Retrieve by profiles of neighborhood subgraphs: $\{A_1\} \times \{B_1, B_2\} \times \{C_2\}$

Figure 4.17. Feasible mates using neighborhood subgraphs and profiles. The resulting search spaces are also shown for different pruning techniques.

Figure 4.16 shows the sample graph pattern $\mathcal{P}$ and the database graph G again for convenience. Figure 4.17 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 4.17.

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 [9, 17, 23, 34, 36, 39–42] 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 an approximation algorithm called Pseudo Subgraph Isomorphism [17]. This prunes the search space by considering the whole pattern and the space $\Phi(u_1) \times .. \times \Phi(u_k)$ simultaneously. Essentially, this technique checks for each node u in pattern $\mathcal{P}$ and its feasible mate v in graph G whether the adjacent subtree of u is sub-isomorphic to that of v. The check can be defined recursively on the depth of the adjacent subtrees: the level l subtree of u is sub-isomorphic to that of v only if the level $l-1$ subtrees of u's neighbors can all be matched to those of v's neighbors. To avoid subtree isomorphism tests, a bipartite graph $\mathcal{B}_{u,v}$ is defined between neighbors of u and v. If the bipartite graph has a semi-perfect matching, i.e., all neighbors of u are matched, then u is level l sub-isomorphic to v. In the bipartite graph, 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', or equivalently the bipartite graph $\mathcal{B}_{u',v'}$ at level $l-1$ has a semi-perfect matching. A more detailed description can be found in [17].

Algorithm 4.2 outlines the refinement procedure. At each iteration (lines 3–20), 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 [17]. 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,

Algorithm 4.2: Refine Search Space

```
Input: Graph Pattern $\mathcal{P}$, Graph $G$, Search space $\Phi(u_1) \times .. \times \Phi(u_k)$,
       level $l$
Output: Reduced search space $\Phi'(u_1) \times .. \times \Phi'(u_k)$
begin
 foreach $u \in \mathcal{P}, v \in \Phi(u)$ do Mark $\langle u, v \rangle$;
 for $i \leftarrow 1$ to $l$ do
     foreach $u \in \mathcal{P}, v \in \Phi(u)$, $\langle u, v \rangle$ is marked do
         //Construct bipartite graph $\mathcal{B}_{u,v}$
         $N_{\mathcal{P}}(u), N_G(v)$: neighbors of $u, v$;
         foreach $u' \in N_{\mathcal{P}}(u), v' \in N_G(v)$ do
             $\mathcal{B}_{u,v}(u', v') \leftarrow \begin{cases} 1 & \text{if } v' \in \Phi(u'); \\ 0 & \text{otherwise.} \end{cases}$
         end
         if $\mathcal{B}_{u,v}$ has a semi-perfect matching then
             Unmark $\langle u, v \rangle$;
         else
             Remove $v$ from $\Phi(u)$;
             foreach $u' \in N_{\mathcal{P}}(u), v' \in N_G(v), v' \in \Phi(u')$ do
                 Mark $\langle u', v' \rangle$;
             end
         end
     end
     if there is no marked $\langle u, v \rangle$ then break;
 end
end
```

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 [19]).

Figure 4.18 shows an execution of Algorithm 4.2 on the example in Figure 4.16. 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 Algo-

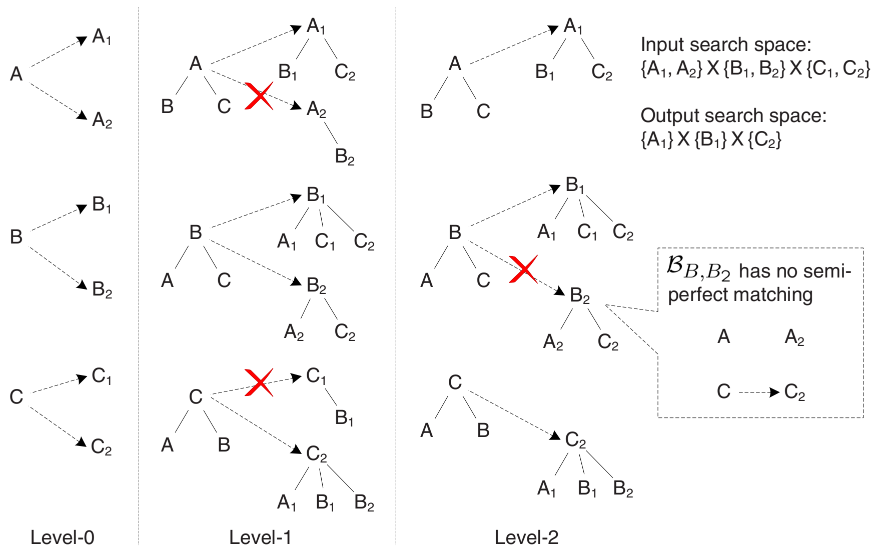

Input search space:
{A1, A2} X {B1, B2} X {C1, C2}
Output search space:
{A1} X {B1} X {C2}
$\mathcal{B}_{B,B_2}$ has no semi-perfect matching
Level-0
Level-1
Level-2

Figure 4.18. Refinement of the search space

rithm 4.2 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 4.1. 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 [7]. 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.

Cost Model. A search order (a.k.a. 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 4.19 shows 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.

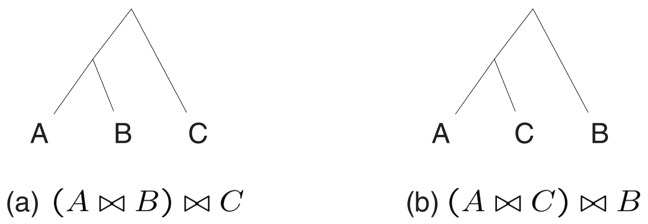


Figure 4.19. Two examples of search orders

Definition 4.11. *(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 4.12. *(Cost of a join) The cost of join i is estimated by*

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

Definition 4.13. *(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 $(A \bowtie B) \bowtie C$.

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 presented graph pattern matching algorithms on large real and synthetic graphs. The graph specific optimizations are compared with an SQL-based implementation as described in Figure 4.2. MySQL server 5.0.45 is used and configured as: storage engine=MyISAM (non-transactional), key_buffer_size = 256M. Other parameters are set as default. For each large graph, two tables V(vid, label) and E(vid1, vid2) are created as in Figure 4.2. B-tree indices are built for each field of the tables.

The presented graph pattern matching algorithms were written in Java and compiled with Sun JDK 1.6. 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

the real dataset is a yeast protein interaction network [2]. 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) [14] terms to the proteins. The Gene 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. We use high level GO terms as labels of the proteins (183 distinct labels in total). We index the node labels using a hashtable, and store the neighborhood subgraphs and profiles with radius 1 as well.

Clique Queries. The clique queries are generated with sizes (number of nodes) between 2 and 7 (sizes greater than 7 have no answers). For each size, a complete graph is generated with each node assigned a random label. The random label is selected from the top 40 most frequent labels. A total of 1000 clique queries are generated and the results are averaged. 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. Queries having too many hits (more than 1000) are terminated immediately and counted in the group of high hits.

To evaluate the pruning power of the local pruning (Section 4.2) and the global pruning (Section 4.3), we define the *reduction ratio* of search space as

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

where Φ_0 refers to the baseline search space.

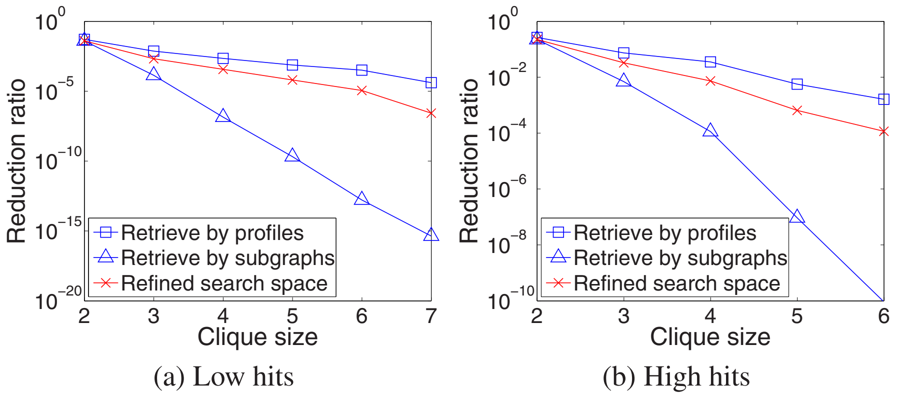


Figure 4.20. Search space for clique queries

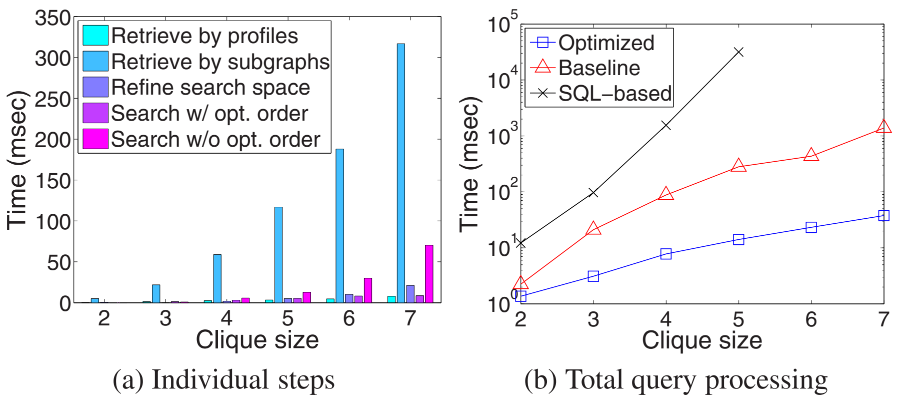


Figure 4.21. Running time for clique queries (low hits)

Figure 4.20 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 the global pruning discussed in Section 4.3 where the input search space is generated by “Retrieve by profiles”. 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 4.21(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.

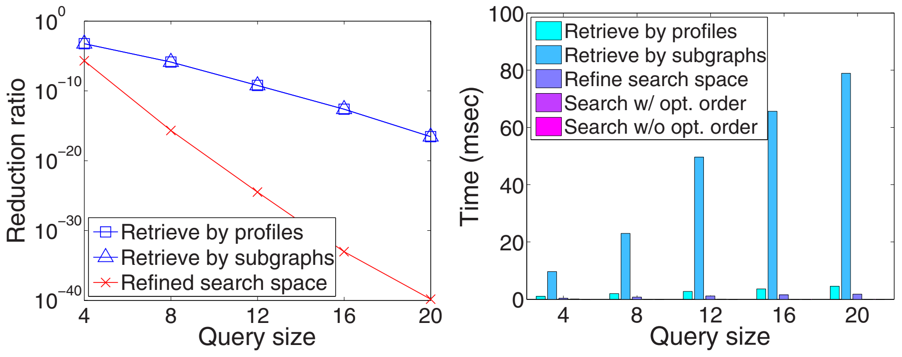
Figure 4.21(b) shows the average total query processing time in comparison to the SQL-based approach on low hits queries. The “Optimized” processing consists of retrieval by profiles, refinement, optimization of search order, and search with the optimized order. The “Baseline” processing consists of 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 4.2).

5.2 Synthetic Graphs

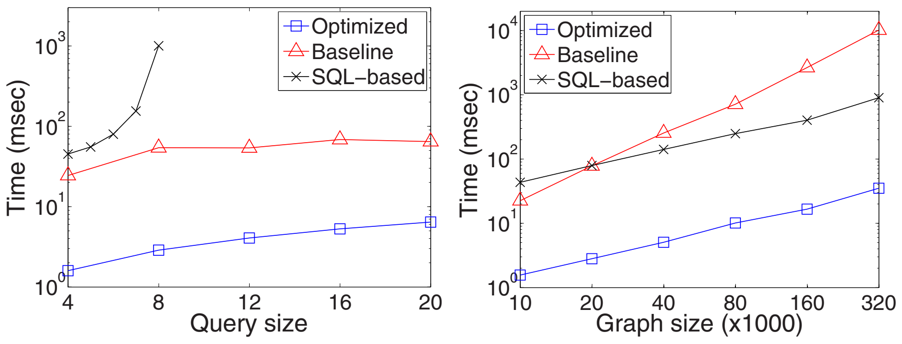
The synthetic graphs are generated using a simple Erdős-Rényi [13] random graph model: generate n nodes, and then generate m edges by randomly choosing two end nodes. Each node is assigned a label (100 distinct labels in total). The distribution of the labels follows Zipf’s law, i.e., probability of 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 4.22 shows the search space and processing time for individual steps. Unlike clique queries, the global pruning produces the smallest search space, which outperforms the local pruning by full neighborhood subgraphs.



(a) Search space (b) Time for individual steps

Figure 4.22. Search space and running time for individual steps (synthetic graphs, low hits)



(a) Varying query sizes (graph size: 10K) (b) Varying graph sizes (query size: 4)

Figure 4.23. Running time (synthetic graphs, low hits)

Figure 4.23 shows the total time with varying query sizes and graph sizes. As can be seen, The SQL-based approach is not scalable to large queries, though it scales to large graphs with small queries. In either case, the "Optimized" processing produces the smallest running time.

To summarize the experimental results, retrieval by profiles has much less overhead than that of retrieval by subgraphs. 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

6.1 Graph Query Languages

A number of graph query languages have been historically available for representing and manipulating graphs. GraphLog [12] represents both data and queries graphically. Nodes and edges are labeled with one or more attributes. Edges in the queries are matched to either edges or paths in the data graphs. The paths can be regular expressions with possibly negation. A query graph is a graph with a distinguished edge. The distinguished edge introduces a new relation for nodes. The query graph can be naturally translated into a Datalog program where the distinguished edge corresponds to a new predicate (relation). A graphical query consists of one or more query graphs, each of which can use predicates defined in other query graphs. The predicates among them thus form a dependence graph of the graphical query. GraphLog queries are graphical queries in which the dependence graph must be acyclic. In terms of expressive power, GraphLog was shown to be equivalent to stratified linear Datalog [28]. GraphLog does not provide any algebraic operations on graphs, which is important for practical evaluation of queries.

In the category of object-oriented databases, GOOD [16] is a graph-oriented object data model. GOOD models an object database instance by a directed labeled graph, where objects in the database and attributes on the objects are both represented as nodes of the graph. GOOD does not distinguish between atomic, composed and set objects. There are only printable nodes and non-printable nodes. The printable nodes are used for graphical interfaces. As for edges, there are only functional edges and non-functional edges. The functional edges point to unique nodes in the graph. Both nodes and edges can have labels, which are defined by an object database scheme. GOOD defines a transformation language that contains five basic operations on graphs: node addition and deletion, edge addition and deletion, and abstraction that groups common nodes. These operations are defined using the notion of a pattern that describes subgraphs embedded in the object database instance. The transformation language is used for both querying and updates. In terms of expressive power, the transformation language can express operations on sets and recursive functions.

GraphDB [15] is another object-oriented data model and query language for graphs. In the GraphDB data model, the whole database is viewed as a single graph. Objects in the database are strong-typed and the object types support inheritance. Each object is associated with an object type and an object identity. The object can have data attributes or reference attributes to other objects. There are three kinds of object classes: simple classes, linked classes, and path classes. Objects of simple classes are nodes of the graph. Objects of link classes are edges and have two additional references to source and target simple objects. Objects of path classes have a list of references to node and edge objects in the graph. A query consists of several steps, each of which creates or manipulates a uniform sequence of objects, a heterogeneous sequence of objects, a single object, or a value of a data type. The uniform sequence of objects have a common tuple type, whereas the heterogenous sequence may belong to different object classes and tuple types. Queries are constructed in four fundamental ways: derive, rewrite, union, and custom graph operations. The derive statement is similar to the usual select...from...where statement, and can be used to specify a subgraph pattern, which is formulated as a list of node objects, edge objects, or either of them occurring in a path object. The rewrite operation transforms a heterogenous sequence of objects into a new sequence. The union operation transforms a heterogenous sequence into a uniform one by taking the least common tuple type. The graph operations are user-defined, e.g., shortest path search.

GOQL [35] also uses an object-oriented graph data model and is extended from OQL. Similar to GraphDB, GOQL defines object types for nodes, edges, paths, and graphs. As in OQL, GOQL uses the usual select...from...where statement to specify queries. In addition, it uses temporal operators next, until and connected to define path formulas. The path formulas can be used as predicates on sequences and paths in the queries. For query processing, GOQL translates queries into an object algebra (O-Algebra) with the extended temporal operators. PQL [25] is a pathway query language for biological networks. The language extends SQL with path expressions and is implemented on top of an RDBMS. In all these languages, the basic objects are nodes and edges as in the object-oriented data model, and paths as extended by the respective languages. Querying on graph structures are explicitly constructed from the basic objects.

More recently, XML databases have been studied intensively for tree-based data models and semistructured data. XML databases can be generally implemented in two approaches: mapping to relational database systems [33] or native XML implementations [21]. In the second approach, TAX [22] is a tree algebra for XML that operates natively on trees. 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. Tree pattern matching thus plays an important role in XML query processing [1, 6]. GraphQL generalizes the idea of tree patterns to graph patterns. Graph patterns is the main building block of a graph query and graph pattern matching is an important part of graph query processing. Both GraphQL and TAX generalize the relational algebraic operators, including selection, product, set operations. TAX has additional operators such as copy-and-paste, value updates, node deletion and insertion. GraphQL can express these operations by the composition operator.

Some of the recent interest in Semantic Web has spurred Resource Description Framework (RDF) [26] and the accompanying SPARQL query language [27]. 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.

Table 4.1. 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

Table 4.1 outlines the comparison between GraphQL and other query languages. GraphQL is different from other query languages in that graphs are chosen as the basic unit of information. This means graphs or sets of graphs are used as the operands and return types in all graph operations. Graph structures are thus preserved and carried over atomically. This is useful not only from a user's perspective but also for query optimizations that rely on graph structural information. 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 declarative and set-oriented, whereas OODB accesses single objects in a navigational manner (i.e., using references to access objects one after another in the object graph). With regard to data model and representation, GraphQL is semistructured and does not cast strict and predefined data types or schemas on nodes, edges, and graphs. In contrast, SQL presumes a strict schema in order to store data. OODB requires objects (nodes and edges) to be strong-typed. In comparison to XML databases, the main difference lies in the underlying data model. GraphQL deals with the graph (networked) data model, whereas XML databases deal with the hierarchical data model.

Graph grammars have been used previously for modeling visual languages and graph transformations in various domains [30, 29]. Our work is different in that our emphasis has been on a query language and database implementations.

6.2 Graph Indexing

Graph indexing is useful for graph pattern matching over a large collection of small graphs. GraphGrep [34] uses enumerated paths as index features to filter unmatched graphs. GIndex [40] uses discriminative frequent fragments as index features to improve filtering rates and reduce index sizes. Closure-tree [17] organizes graphs into a tree-based index structure using graph closures as the bounding boxes. GString [23] converts graph querying to subsequence matching. TreePi [41] uses frequent subtrees as index features. Williams et al. [39] decompose graphs and hash the canonical forms of the resulting subgraphs. SAGA [36] enumerates fragments of graphs and answers are generated by assembling hits of the query fragments. FG-index [9] 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. [42] 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 chapter addresses graph pattern matching for the case of a single large graph.

Another line of graph indexing addresses reachability queries in large directed graphs [8, 10, 11, 31, 37, 38]. 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 4.6(a)). Indexing and processing of reachability queries are generally based on spanning trees with pre/post-order labeling [8, 37, 38] or 2-hop-cover [10, 11, 31]. These techniques can be incorporated into access methods for recursive graph pattern queries.

7. Future Research Directions

Physical Storage of Graph Data. Graphs in the real world are heterogeneous in both the structures and the underlying attributes. It is challenging to store graphs on disks for efficient storage and fast retrieval. What is the appropriate storage unit, nodes, edges, or graphs? In the category of a large collection of small graphs, how to store graphs with various sizes to fixed-length pages on disks? In the category of a single large graph, how to decompose the large graph into small chunks and preserve locality? Traditional storage techniques need to be re-considered, and new graph-specific heuristics might be devised to address these questions.

Implementation of Other Graph Operators. This chapter only addresses implementation of the selection operator. Other operators, such as joins on two collections of graphs, might be a challenge if the inter-graph join conditions are not trivial. In addition, operators such as ordering (ranking), aggregation (OLAP processing), are interesting research directions on their own.

Scalability to Very Large Graph Databases. The presented techniques consider graphs with millions of nodes and edges, or millions of small graphs. Graphs in some domains, such as Internet, social networks, are in the scale of tera-bytes or even larger. Graphs at this scale cannot be processed by single machines. Large-scale parallel and distributed schemes are needed for graph storage and query processing.

8. Conclusion

We have presented GraphQL, a 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 chapter 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 chapter are an important first step in this regard.

Acknowledgments

This work was supported in part by NSF grants IIS-0612327.

Appendix: Query Syntax of GraphQL

```
Start ::= ( GraphPattern ";" | FLWRExpr ";" )* <EOF>

GraphPattern ::=  "graph" [<ID>] [Tuple] "{"
                        MemberDecl *
                  "}" ["where" Expr]

MemberDecl ::= "node" NodeDecl ("," NodeDecl)* ";"
```

```
             | "edge" EdgeDecl ("," EdgeDecl)* ";"
             | "graph" <ID>  ( "," <ID> )* ";"
             | "unify" Names "," Names ("," Names)* ";"

NodeDecl ::= [<ID>][Tuple] ["where" Expr]

EdgeDecl ::= [<ID>]"(" Names "," Names")" [Tuple] ["where" Expr]

Tuple ::= "<"[<ID>] (<ID>"="Literal)* ">"

FLWRExpr ::= "for" ( <ID> | GraphPattern )
             ["exhaustive"] "in" "doc" "(" string ")"
             ["where" Expr]
             ( "return" GraphTemplate |
               "let" <ID> "=" GraphTemplate )

GraphTemplate ::= "graph" [<ID>] [TupleTemplate] "{"
                      TMemberDecl *
                  "}" | <ID>

TMemberDecl ::= "node" TNodeDecl ("," TNodeDecl)* ";"
              | "edge" TEdgeDecl ("," TEdgeDecl)* ";"
              | "graph" <ID>  ( "," <ID> )* ";"
              | "unify" Names "," Names ("," Names)* ["where" Expr] ";"

TNodeDecl ::= [<ID>][TupleTemplate]

TEdgeDecl ::= [<ID>]"("Names "," Names")"[TupleTemplate]

TupleTemplate ::= "<"[<ID>] (<ID>"="Expr)* ">"

Expr ::= Term ( Op Expr )*

Op ::=   "|" | "&"  | "+" | "-"  | "*" | "/" |
         "==" | "!=" | ">" | ">=" | "<" |"<="

Term ::=  "(" Expr ")" | Literal | Names

Names ::= <ID> ("." <ID>)*

Literal ::= int | float | string
```

References

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–, 2002.

[2] S. Asthana et al. Predicting protein complex membership using probabilistic network reliability. *Genome Research*, May 2004.

[3] 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.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C, `http://www.w3.org/TR/xquery/`, 2007.

[5] C. Branden and J. Tooze. *Introduction to protein structure*. Garland, 2 edition, 1998.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[7] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *PODS*, 1990.

[13] P. Erdős and A. Renyi. On random graphs I. *Publ. Math. Debrecen*, (6):290–297, 1959.

[14] Gene Ontology. `http://www.geneontology.org/`.

[15] R. H. Guting. GraphDB: Modeling and querying graphs in databases. In *Proc. of VLDB'94*, pages 297–308, 1994.

[16] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *Proc. of PODS '90*, pages 417–424, 1990.

[17] H. He and A. K. Singh. Closure-Tree: An Index Structure for Graph Queries. In *Proc. of ICDE '06*, Atlanta, USA, 2006.

[18] H. He and A. K. Singh. Graphs-at-a-time: Query Language and Access Methods for Graph Databases. In *Proc. of SIGMOD '08*, pages 405–418, Vancouver, Canada, 2008.

[19] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Computing*, 1973.

[20] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[21] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *VLDB J.*, 11(4):274–291, 2002.

[22] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. of DBPL'01*, 2001.

[23] H. Jiang, H. Wang, P. S. Yu, and S. Zhou. GString: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[24] J. Lee, J. Oh, and S. Hwang. STRG-Index: Spatio-temporal region graph indexing for large video databases. In *Proc. of SIGMOD*, 2005.

[25] U. Leser. A query language for biological networks. *Bioinformatics*, 21:ii33–ii39, 2005.

[26] F. Manola and E. Miller. RDF Primer. W3C, `http://www.w3.org/TR/rdf-primer/`, 2004.

[27] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C, `http://www.w3.org/TR/rdf-sparql-query/`, 2007.

[28] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, chapter 24 Deductive Databases. McGraw-Hill, third edition, 2003.

[29] J. Rekers and A. Schurr. A graph grammar approach to graphical parsing. In *11th International IEEE Symposium on Visual Languages*, 1995.

[30] G. Rozenberg (Ed.). *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, volume 1. World Scientific, 1997.

[31] 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.

[32] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, 2006.

[33] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[34] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proc. of PODS*, 2002.

[35] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, 1999.

[36] 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.

[37] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD '07*, pages 845–856, 2007.

[38] 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.

[39] D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.

[40] X. Yan, P. S. Yu, and J. Han. Graph Indexing: A frequent structure-based approach. In *Proc. of SIGMOD*, 2004.

[41] S. Zhang, M. Hu, and J. Yang. TreePi: A novel graph indexing method. In *ICDE*, 2007.

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