LWI and Safari: A New Index Structure and Query Model for Graph Databases

Srinath Srinivasa*
Indian Institute of Information Technology,
26/C, Electronics City,
Bangalore, India 560100.
sri@iiitb.ac.in

Martin Maier†
C L Infotech Pvt Ltd.,
214, 7th Block,
Koramangala,
Bangalore, India 560095.
martin@clinf.com

Mandar R. Mutalikdesai
Indian Institute of Information Technology,
26/C, Electronics City,
Bangalore, India 560100.
mandar@iiitb.ac.in

ABSTRACT

Graph databases are gaining importance in several emerging applications, especially molecular biology. In many existing approaches, such databases are regarded as a "schemaless" collection of labeled graphs. However, there are often user-defined schemes that help in limiting the search space while answering a query and to deliver meaningful results. Techniques based only on index structures do not exploit such situations. This paper presents our work on a graph database system called GRACE, where a Data Manipulation Language (DML) called Safari is proposed for graph databases and is closely integrated with structural indexes in the DBMS. Users may define schematic structures over a subset of graphs in the database and add them into the database as any other member graphs. The query model in turn can use such member graphs to define its search space in order to deliver more meaningful results. Queries can be composed, so that schemas defining search spaces can be generated dynamically. An augmenting index structure called labeled walk index (LWI) is also proposed that is extensively used for answering structural queries in Safari.

1. INTRODUCTION

Databases of labeled graphs are finding applications in many emerging areas – most significantly in molecular biology. Molecular structures stored as labelled graphs often need to be retrieved based on structural similarity or subgraph isomorphism.

Determining structural similarity or subgraph isomorphism are both NP-complete in complexity. As a result, various techniques ranging from pre-processing to heuristics based approximate searches have been explored [1, 3, 6, 13, 18].

In a graph database, not only comparing individual graphs is difficult, it is also required to make this comparison against all graphs in the database.

In *walk-based* indexing approaches like Graphgrep [3], all walks in the member graphs are enumerated until a maximum walk length l_w. The query is also similarly decomposed and the problem reduces to a set of string matching tasks. In *frequent substructure* based approaches like gIndex [18], member graphs are indexed based on frequently occurring substructures contained in them. A query graph is also searched for occurrences of these substructures and the search space is thus narrowed down. Indexing and searching over structures is facilitated by storing the graphs in a "canonical form" that is in the form of a string.

However, a shortcoming with existing approaches is that they treat the database as a "schemaless" collection of graphs. The default search space for all queries is the set of all member graphs in the database.

In reality however, several classifications and schematic structures are employed by users to relate graphs in the database. In the realm of protein structures, there exist several schemes like SCOP [12], CATH [2] and FSSP [9] which relate protein structures based on several parameters.

One can envisage a user query of the form: *Which is the biggest common cluster of proteins that are classified similarly under both SCOP and CATH?* Such queries cannot be answered in a straightforward fashion in existing approaches.

User defined schemes reflect domain knowledge; and as a result, queries that are posed and answered within such a context are likely to be more useful than those that blindly search the entire database.

This paper presents an approach that enables the user to define one or more schemes and formulate complex queries over these schemes. A DML called Safari is proposed using which, users may relate graphs in the database to form schematic structures. The most general form for a schema is a graph itself. Hence, a schema can be added to the database as any other member graph. A Safari query enables a user to choose a member graph as schema and specify that the query be answered within its context. The result of such queries are again graph structures. Hence queries can be composed

*Contact author
†On study leave from Fachhochschule Regensburg, Prüfeninger Strasse 58 93049, Regensburg, Germany

ADVANCES IN DATA MANAGEMENT 2005
Jayant Haritsa, T.M. Vijayaraman (Editors)
© CSI 2005

and schema graphs can be generated dynamically.

In addition, data mining techniques based on filtration is also employed for fast structural searches. The technique is built around an index structure called "labeled walk index (LWI)". Both Safari and LWI are presented in this paper.

2. THE OVERALL GRACE MODEL

Overall structure of a GRACE database: A GRACE database $\mathcal{G}$ is a collection of *member graphs*, $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$. Each member graph G_i, is "labeled graph" of the form: $G_i = (V, E, V_L, \lambda, V_N, \gamma)$. Here, V is a set of nodes or vertices, $E \subseteq V \times V$ is a set of edges, V_L is a set of vertex labels, $\lambda : V \to V_L$ is the vertex labeling function, V_N is the set of node "names" and $\gamma : V \to V_N$ is the node naming function. The vertex set of G_i is refered by $V(G_i)$ and the edge set is refered by $E(G_i)$.

In a labeled graph, it is generally the case that the number of node labels is far less than the number of nodes. This characteristic is utilized in building reasonably small structural indexes that can return graphs based on approximate matching. The label of a node represents the node's "type" and the node name refers to a specific node instance of a given label. Distinct nodes having the same label should have different names.

We do not consider edge labels in this paper for the sake of simplicity; although edge labels can be incorporated seamlessly.

For each node in a member graph its label/name combination is unique. For example, in a graph representing an organic molecule, the node C_1 denotes node named "1" belonging to type (having a label) C. Similarly, a node representing N_1 represents node named "1" belonging to type "N". The name distinguishes a node from other nodes having the same label.

Attributes in a member graph are in the form of

$$(name, type, value)$$

triples, where *name* is the name of the attribute, *type* is its type and *value* is the value of the attribute. In the current implementation, only one attribute type (string) is supported. Attributes can be associated with nodes, edges and the graph itself.

Figure 2 shows a graph depicting the molecule benzene written in a syntax called the *modified DOT form* that is supported by GRACE. The graph has a label "aromatic" and name "benzene". It has an attribute named "discoverer" whose value is "Kekule" associated with it. Node attributes are defined next. This graph does not have individual node attributes. Instead all nodes having a given label share the specified attributes for the labels "C" and "H" respectively. Edge definitions follow next and edge attributes are defined after each edge definition.

Schema graphs: A member graph $G_S \in \mathcal{G}$ is called a "schema graph" if there exists another member graph $G_i \in \mathcal{G}$ such that $G_i \in V(G_S)$. In other words, a member graph which contains at least one node that refers to another member graph in the database is called a schema graph.

Referencing a member graph is done using one of the following ways:

1. Using the *gid* of the member graph. Each member graph is given a unique graph id *gid* upon insertion. This can be used by schema graphs to reference member graphs.

2. Using a name/label combination. All member graphs appear as nodes in at least one schema graph called the *default* graph as explained further below. The graph label and name combination appear as node name and labels in the default graph. As a result, the combination of graph labels and names should be unique throughout the database.

3. Using a *Safari* query. As explained in Section 3, queries in Safari return graph structures. Nodes in schema graphs can use such queries to reference dynamically generated views of the database.

```
graph aromatic_benzene
[(discoverer, string, ``Kekule'')]
{
  C [(element, string, ``Carbon'')];
  H [(element, string, ``Hydrogen'')];
  ;
  C_0--H_0 [(bond, char, 's')];
  C_0--C_1 [(bond, char, 's')];
  C_1--H_1 [(bond, char, 's')];
  C_1--C_2 [(bond, char, 'd')];
  C_2--H_2 [(bond, char, 's')];
  C_2--C_3 [(bond, char, 's')];
  C_3--H_3 [(bond, char, 's')];
  C_3--C_4 [(bond, char, 'd')];
  C_4--H_4 [(bond, char, 's')];
  C_4--C_5 [(bond, char, 's')];
  C_5--H_5 [(bond, char, 's')];
  C_5--C_0 [(bond, char, 'd')];
}
```

Figure 1: Representation of a benzene molecule graph

The first two kinds of referencing techniques are called *static* referencing and the last technique is called *dynamic* referencing. Insertion of a schema graph fails if any of its nodes try to refer to a graph that does not exist in the database. This is in order to maintain referential integrity. Note that, it can happen only when static referencing techniques are used. A query returns a null graph when no results match the query. A null graph is a legitimate graph and hence referential integrity for the schema graph is not violated.

The graph referenced by a node in a schema graph may in turn be a schema graph itself. This can continue to any levels and in fact, circular and self references are also allowed. A schema can refer to itself as one of its nodes[1].

When a member graph G_i becomes a node in a schema graph G_S, the graph attributes of G_i become the node attributes of whichever node represents G_i in G_S.

The "default" graph: By default, all member graphs belong to a schema called the *default* graph. In its original form, the default graph is a disconnected graph, comprising only of nodes representing every member graph (including

[1]This is as long as there are no contradictory semantics associated with the self reference like the Russell's paradox. Self references per se, need not lead to paradoxes.

itself) in the database. Edges can be added to the default graph to establish relationships among member graphs.

For example, if member graphs are protein structures, edges among nodes in the *default* graph may denote root-mean-square distances (RMSD) that are known among many pairs of protein molecules.

In a general sense however, it is not advisable to add edges directly to the default graph. This is because addition (and deletion) of nodes to (and from) the default graph is automatic. Also addition of other schema graphs into the database will be reflected in the default graphs as well. This may interfere with the semantics of relationships among nodes in the graph.

Graph addition and special attributes: When a member graph is inserted into the database, it is given a unique "graph id" represented by *gid*. The *gid* of a graph is added as a graph attribute into the graph, and as a node attribute for the node which represents this graph in the default graph. In addition to the *gid*, the name and label of each node and of the graph itself are added as corresponding attributes. The *gid* can be searched using attribute name `__gid`, and names and labels can be searched using attribute names `__name` and `__label` respectively.

Graph deletion: When a member graph G_i is *deleted* from the database, all schema graphs which have *static* references to G_i will be modified. Nodes corresponding to G_i in these schema graphs, as well as all their incident edges will be deleted.

It is forbidden to delete the *default* graph. Even an empty database contains the default graph, which contains just one node: a reference to the default graph itself.

Graph deletions are not yet supported in the current implementation of GRACE, however the corresponding referential index structures to handle deletions have been designed.

3. THE Safari DATA AND QUERY MODEL

3.1 Overview

The syntax for specifying graphs in GRACE is a modified form of the DOT format developed under the Graphviz project at AT&T[2]. A language called Safari is proposed that can manipulate graph objects specified in the modified DOT form and manage their storage and retrieval in databases. Safari constructs are divided into four types:

Graph modification constructs: These include constructs where nodes and edges can be added or deleted from graphs. In addition, attributes for nodes, edges and for the graph itself can be managed.

Graph retrieval constructs: These include constructs which take a graph as input and return a subgraph based on the specified condition.

Database modification constructs: These include constructs that add and delete graphs to and from a database.

Database retrieval constructs: These constructs search the database for graphs matching the specified condition and returns either a member graph or a dynamically created meta-graph as the query result. A meta-graph is a "pure" schema graph where all nodes refer to other member graphs.

[2]http://www.research.att.com/ erg/graphviz/info/lang.html

In this paper, we shall be mainly considering retrieval constructs; specific examples of other constructs would be provided as and when necessary. Also structural query constructs are not considered in this section. They are taken up separately in the next section.

3.2 Graph retrieval constructs

Graph retrieval constructs take a graph and return a subgraph. These operations are completely in-memory operations. No database access is performed.

The main construct for graph retrieval is called the `selecton` operator (pronounced as "select on"), denoted by σ_o. The syntax of `selecton` is as follows:

$$\sigma_o \ \ \langle condition \rangle \ \ [\langle graphref \rangle]$$

The σ_o construct takes two arguments, of which the second one is optional. The first argument is a graph retrieval condition. Some example functions that specify these conditions are outlined in Table 1. The second argument is the graph over which retrieval is to be performed. If this is omitted, the *default* graph for the current database is chosen.

Note that since σ_o does not perform any database operations, the graph provided in the argument should already be in memory. When a database is chosen (with the "`use`" command, explained later), the *default* graph and a few index structures are loaded into memory.

The $graphref$ or the graph reference that forms the third parameter of the command-line (or the second argument of the command)[3] can be provided in one of the following ways:

1. Omit the third parameter to automatically take the *default* graph reference
2. Provide the name of a session variable holding a graph
3. Provide an inline DOT formatted graph
4. Place another Safari query output.

Suppose we have a session variable called PDB1a8i which stores the molecular structure of the protein 1a8i. The following query:

```
C1a8i = selecton Edgeattr(''BondType'',''Hydrogen'')
           selecton
              Edge(Nodeattr(''__label'',''C''),
                Nodeattr(''__label'',''C''))
              PDB1a8i;
```

returns a subgraph containing hydrogen bonds among carbon molecules. The inner `selecton` selects all edges between nodes having label "C" and the outer `selecton` prunes this to select only those edges which have a "BondType" attribute as "Hydrogen". Note the assignment operator which assigns the output graph to another session variable called C1a8i.

[3]We shall use the terms "third parameter" of the command-line and the "second argument" of the command, interchangeably.

Condition	Comments
Nodeattr(String, String)	Returns all nodes which have an attribute matching the first parameter, whose value matches second parameter. When two or more nodes are returned, any edges among them are also returned.
Edgeattr(String, String)	Returns all edges which have an attribute matching the first parameter, whose value matches second parameter. When an edge is returned the corresponding nodes are also returned.
Edge(Nodeattr(), Nodeattr())	Returns all edges that exist between nodes matching the first argument and nodes matching the second argument.
Walk(Nodeattr(), Nodeattr(), n)	Returns all walks of length n or below between nodes matching the first argument and nodes matching the second argument.

Table 1: Example graph retrieval conditions

3.3 Graph modification constructs

These constructs take a graph as input and return a modified graph. All these are done in memory and no database access is performed. Some of the graph modification constructs include the following:

addnode The syntax of addnode is of the form:

```
addnode label_name [name1=value1,...] graphref;
```

The node with the specified name and label is added to the graph specified in the graphref, which could be either a variable, query result or an inline reference. The modified graph is then returned. If a node with the specified name and value already exists, then the list of attributes is merged with the existing node.

addedge The syntax of addedge is:

```
addedge label1_name1 -- label2_name2
          [name1=value1,...] graphref;
```

An edge is added between nodes label1_name1 and label2_name2. If no such nodes exist, then they are created. If an edge already exists, then the specified attribute list is merged with that of the existing edge.

merge This is of the form:

```
merge graphref1 graphref2;
```

If $graphref1 = (V_1, E_1)$ and $graphref2 = (V_2, E_2)$ then the functioning of merge would be $graphref2 = (V_1 \cup V_2, E_1 \cup E_2)$.

Consider the following query:

```
CCN1a8i = merge C1a8i
        selecton Edgeattr(``BondType'',``Hydrogen'')
          selecton
            Edge(Nodeattr(``__label'',``C''),
              Nodeattr(``__label'',``N''))
            PDB1a8i;
```

This query first generates a subgraph showing all hydrogen bonds between "C" and "N" atoms by the two `selecton` statements. The earlier graph having only "C" labeled nodes is then merged into this graph to to show bonds between "C" labeled nodes and between "C" and "N" labeled nodes.

Instead of the session variable C1a8i in the above example, the complete `selecton` query can be written instead.

3.4 Database retrieval constructs

Database retrieval constructs take logical conditions and match graphs in the database. The search space may be limited by specifying a schema, without which the entire graph (the *default* schema) will be used as the search space.

There are two main graph retrieval constructs: `selectin` represented by σ_i and `selectgraph` represented by σ_g.

The `selectin` operator is specified as:

$$\sigma_i \ \langle condition \rangle \ \ [\langle graphref \rangle]$$

Here, *condition* is a logical condition on either attributes in a graph or its structure, and *graphref* is the usual graph reference which corresponds to the current *default* graph if omitted.

The `selectin` operator works as follows:

1. The graph specified in the third parameter is used as a schema and all the graphs referred by it are searched for the specified *condition*
2. The schema graph (third parameter) is then pruned to remove all nodes that either do not refer to other graphs, or refer to graphs that do not match the condition. When a node is removed, its incident edges are also removed.
3. The pruned schema graph is then returned.

The `selectin` operator creates a "derived view" of the database from an existing schema by matching conditions from graphs in the database. Contrast this with `selecton` in which conditions are matched within the graph itself.

Consider the following query:

```
selectin EdgeAttr(``BondType'', ``Triple'')
         Amino_acids;
```

The query takes the schema graph referred by the variable `Amino_acids` and returns only those nodes (and edges among them if any) that represent graphs containing at least one triple bond.

The `selectgraph` operator (σ_g) is used to select a member graph from a schema. The syntax of σ_g is as follows:

$$\sigma_g \ \langle condition \rangle \ \ [\langle graphref \rangle]$$

If *condition* matches a single graph in the provided *graphref*, this member graph is fetched from the database and returned. If *condition* matches more than one member graph,

Command	Comments
`createdatabase <name>`	Creates a new graph database with the given name and populates it with the default graph.
`use <name>`	Uses the specified database as the current database. This entails loading the new default graph and index structures into memory and discarding any previously loaded ones.
`insertgraph <graphref>`	Inserts the specified graph into the database. Insertion fails if the name/label combination of the graph already exists in the database. When a graph is inserted, it is given a new gid.
`deletegraph <attribute list>`	Delete the member graph matching the list of attributes specified in the attribute list. Deletion fails if no graphs match or more than one graph matches attribute list. Specifying either `__gid` or `__name` and `__label` can help in uniquely identifying member graphs.

Table 2: Some database manipulation commands

then σ_g acts as σ_i and returns that part of the graph provided in *graphref* that match the given condition. A new graph attribute `__ismeta` is created and set to 1 in the returned graph to denote that the returned graph is a metagraph and not a member graph.

The following conditions may be used to uniquely identify graphs: *gid* and *name/label* combination.

As usual, the third *graphref* parameter is optional and refers to the *default* graph if omitted.

Consider the following query:

```
selectgraph (Nodeattr(‘‘__name’’, ‘‘Phenylalanin’’))
    selectin EdgeAttr(‘‘BondType’’, ‘‘Double’’)
             Amino_acids;
```

Assuming a bio-molecular database, this query searches for a molecule named "Phenylalanin" among the set of molecules described by the `Amino_acids` schema that have a double bond.

3.5 Database modification constructs

These constructs enable the user to create a database, add graphs to the database, delete graphs from the database and drop databases. In the present implementation, there is no facility to modify graphs in place. The user should delete a graph and add the modified member graph separately.

Database manipulation constructs are not directly relevant to the topic of the paper. Hence, some of the Safari commands that manipulate the database, are summarized in Table 2.

4. INDEX STRUCTURES AND QUERY ANSWERING

4.1 Index structures in GRACE

This section describes index structures that are used in GRACE to handle queries. Three kinds of index structures are used by default in a GRACE database. These are:

1. The attribute-value index (AVI)
2. The graph location index (GLI) and
3. The label-walk index (LWI)

AVI is used for attribute searches over the database. AVI is an inverted index that stores gid values of graphs matching specific attribute and value pairs. Since the current implementation of GRACE supports only string attributes, AVI suffices for all attribute related queries.

The GLI is used to quickly retrieve a graph based either on its gid or its name/label combination. Graphs are stored in one or more data files and the number of graphs in a data file is determined by a configurable parameter called "graphs per file" (gpf). The GLI forms a secondary index into these data files, indexing every graph entered into the database.

AVI and GLI are not covered in depth in this paper.

The third index, LWI, is used for structure based searches. *Label Walks* are used to codify structural properties of member graphs. A label walk is a walk[4] comprising of only node labels (without the node names). A label walk is hence a data type representing the set of all walks in the graph having this sequence of labels. Since undirected graphs are considered, a label walk is the same as the `reverse` of itself. Hence the label walk `C-C-N` is the same as `N-C-C`. GRACE stores only the lexicographically smaller label walk between a given label walk and its reverse. If directed graphs are assumed, this check need not be performed.

Operations of LWI can be divided into two phases, the *index generation* and the *query processing*. Index generation is a one time effort over each graph in the database, and is performed whenever graphs are added to the database. The query processing applies an incremental strategy in generating label walks for progressively refining query details.

4.2 Index Generation

Whenever a new graph G_i is added to the database, label walks in G_i are generated up to a maximum length l_w. Walk generation is performed by a depth-first search (DFS) procedure starting from all nodes in the graph.

The set of all label walks in G_i having label l is depicted by $w_l(G_i)$. For each label l, the number of label walks or $|w_l(G_i)|$ is stored in the LWI index. The index structure is in the form of a prefix tree (called the *LWI tree*) that stores label walks, their number of appearances and the ids of the graphs in which they can be found.

The label walk index LWI is organized as a database-wide prefix tree as shown in Figure 2. Each node in the LWI tree holds information about the label walk starting from the root to itself. Each node points to a sorted table of pairs of the form (num, gid), that pairs member graph gids with

[4]A walk in a graph is a sequence of nodes $v_1 v_2 \ldots v_n$, such that there exists an edge between any two consecutive nodes and no edge repeats itself in the walk. Note that nodes may repeat in a walk.

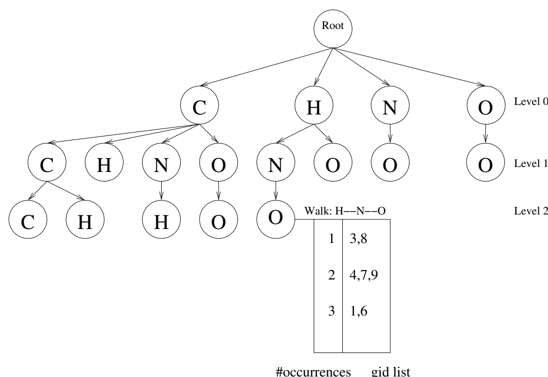

Figure 2: The LWI tree

the number of occurrences of the present label walk in the graph. There are no intermediate nodes in the LWI tree, except the root node. Each node holds information about label walks until itself starting from the root.

In the figure an LWI tree is shown which is created after addition of one or more graphs into the database. The figure also shows the table associated with a node. The position of the node from the root indicates that it represents the labeled walk `H--N--O`. The table associated with this node has two columns: the number of occurrences of this walk and the set of gids that have such an occurrence. In the example database of Figure 2, walk `H--N--O` appears once in graphs with ids 3 and 8, while it appears twice in graphs with ids 4, 7 and 9.

When a query graph Q is presented for structural similarity search, walks in the graph are enumerated and a vector is created comprising of pairs of the form $(walk, proj)$. Here $walk$ is a walk present in Q, while $proj$ is the number of occurrences of $walk$ in Q. The LWI is then searched to look for entries of all $walk$ values present in Q such that the number of occurrences is $proj \pm w$, where w is a nearness threshold that can be configured as a database parameter.

Query answering is done in a phased manner where each *refinement* constitutes exploring one level further in the LWI tree. This is explained in more detail in the next section.

4.3 Processing structural queries

Two main kinds of structural queries are supported by GRACE:

Similarity: Similarity searches take a graph as parameter and returns a set of "structurally similar" graphs. Structural distance between graphs is the *edit distance*, or the number of node/edge edit operations that are required to transform one graph to another. Structural similarity searches return graphs at small edit distances to the query graph.

Substructure: Substructure queries take a query graph as input and returns all graphs that are likely to have the query graph as one of its subgraphs.

Structural queries are processed by generating label walks from the query graph and comparing them with number of appearances in the LWI tree. Generation of label walks of the query graph is done in a phased manner so that query results can be returned quickly and can be progressively refined.

All label walks for a given gid at any given level are treated as a *vector* depicting the position of the graph at that level. When a query graph is similarly vectorized, query results would be based on distances between the query graph vector and the member graph vectors. This approach can be contrasted with that of Graphgrep [3], where all walks from the graph are stored to produce results based on *exact* matching. The LWI is many orders of magnitude smaller than a data structure storing all walks in the member graphs. However, the vectorization based approach used in GRACE provides results based on approximate matching.

The algorithm for pruning the search space for structure based queries is based on the mining technique called FBT [14] developed by the GRACE team for graph mining applications. FBT is based on the concept of filtration, rather than incremental construction that is employed by other graph mining algorithms. In order to mine for common subgraphs in a database of graphs, FBT starts with an assertion that all graphs in the database are isomorphic, and progressively starts pruning edges that refute such an assertion. The process then converges to the maximal common subgraphs among the graphs.

A similar technique is used for structure based searches in LWI. Given a query graph Q and a schema graph S, a similarity search proceeds as follows.

1. Start from level $l \leftarrow 0$ (first level after the root) of the LWI tree
2. Let W_S be the set of all gids present in the schema graph S
3. Let Q_l be the set of all label walks and their corresponding number of occurrences in the query graph (i.e. the vector of query graph) at level l
4. Let L_l be the set of all label walk vectors in the LWI tree at level l
5. Compare Q_l with L_l and obtain the set of all gids W_L that are "close enough" to the query vector, based on a window parameter δ.
6. Compute $R_l = W_S \cap W_L$ as the set of query results at level l
7. Since query results at level $l + 1$ has to be contained in the query results at level l, remove all gids from W_S that are not in R_l. That is, $W_S = W_S \cap R_l$.
8. If any walk in L_l contains no gids from the new W_l, mark the corresponding node in the LWI tree as "useless". The entire sub-tree under this node can be ignored when comparing at the next level
9. Set $l \leftarrow l + 1$ and return to step 2 until the desired number of refinements are done

The "close enough" function is computed by considering a sphere of radius δ around the query graph vector at each level and adding all vectors that lie within this radius. For substructure queries, the "close enough" function is replaced

by the “contains” function which returns the set of all vectors that lie *above* the query vector.

The LWI tree is augmented with another data structure called the layered star schema proposed in [15] for fast spatial searches in a discrete space having different levels.

4.4 Structural query constructs in Safari

In Safari, structural queries can be performed by providing a *graphref* as input parameter to functions like `similar()`, `contains()` and `mcg()`. Here `mcg` stands for “maximal common subgraph” between the query graph and the *graphref* in the third parameter.

All the above structural query constructs come in two forms:

1. Single-step structural query, where structural searches are performed for one level above the previous search if any, and
2. Multi-step structural query, where structural searches are performed for a pre-specified number of levels.

For example, the `similar()` function has the following forms: `similar(graphref)` and `similar(graphref, depth)`. In the first form, the given *graphref* is searched for similar graphs in a given domain for one level. If the domain is a usual schema graph, then only level 0 is searched. If the domain is the result of a previous structural search, then similarity search is done for one level above the number of levels searched in the domain.

Consider the following query:

```
selectin similar(fructose)
       selectin similar(fructose);
```

This query searches the entire database (using the *default* graph as the search domain) and returns molecular structures similar to fructose till a depth of 2 (levels 0 and 1). The inner `selectin` returns a pruned *default* graph containing graphs whose vectors lie within distance δ from the vector of fructose at level 0. The returned graph is given a special attribute called `__slevel` whose value is set to 0 (the level until which search was performed). When the `similar()` search in the outer `selectin` sees this attribute in the domain, it knows the next level which has to be searched.

Progressive structural searches not only provide interactive response times and enable the user to choose between speed and accuracy; it is also useful in queries that cluster structural isomers from member graphs. The following set of queries shows such an example.

```
level0 = selectin similar(fructose);
flevel1 = selectin similar(fructose) level0;
glevel1 = selectin similar(glucose) level0;
```

The first query above retrieves a set of graphs that are structurally similar to fructose at level 0 (i.e. in terms of the number of different kinds of atoms). The second query refines this to the next level; but the third query takes results from the first query and compares it against *another* graph: glucose. This query in effect says that: *among all graphs that are similar to fructose in terms of atoms, which are similar to glucose in terms of bonds between atoms?* While the first query gets all structural isomers of fructose, the third query separates those molecules that are similar to glucose.

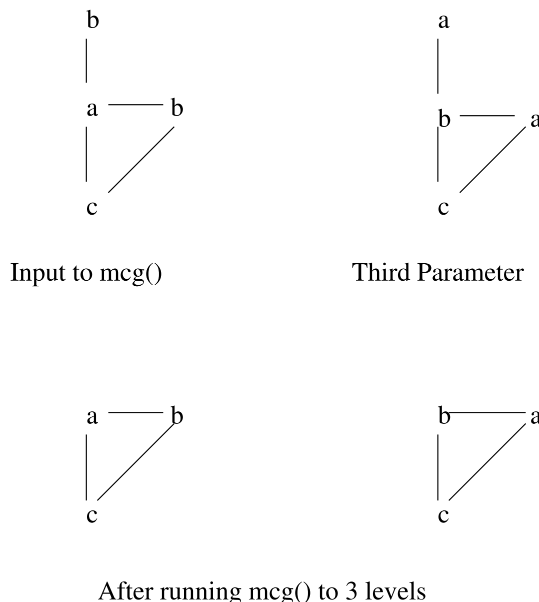

Figure 3: Illustration of `mcg()`

Similarity searches can be “instrumented” at several levels in order to separate structurally similar graphs into different classes.

The `similar()` search can also accept a *depth* parameter which specifies the depth to which similarity search should proceed. Hence the query:

```
selectin similar(fructose, 3)
      selectin similar(fructose);
```

searches the database for graph structures similar to fructose until a depth of 3, over the previous depth of 1 (or until level 0). The return value of this query is a meta-graph where the `__slevel` is set to 3.

The `contains()` function behaves in an analogous fashion to `similar()`. Both `similar()` and `contains()` are used in `selectin` queries.

The `mcg()` function is used in `selecton` queries in order to compare individual graphs. The `mcg()` function takes a *graphref* as input and compares it with the *graphref* provided as the third parameter. Without a second parameter, `mcg()` compares till a depth of 1 and removes all label walks in both graphs that are not common.

Figure 3 illustrates how `mcg()` works. The first graph in the figure is given as input to `mcg` and the second graph is the third parameter of `selecton`. After `mcg()` runs for 1 level, it finds that the labels `a`, `b` and `c` are common to both graphs. Hence the graphs are unchanged. Even after the next refinement, the graphs are unchanged as all the level 1 label walks (`a-b`, `a-c`, `b-c`) are common. However, in the next refinement, label walk `b-a-b` found in the input graph is not present in the other graph. Similarly, the label walk `a-b-a` found in the other graph is not present in the input graph. The graphs are then pruned to result in the graphs

shown in the lower half of the figure.

The `mcg()` function has to maintain two graphs across different invocations. In order to support this, the output of `mcg()` is a "composite graph" that contains both the pruned graphs. In a composite graph, nodes are tagged with an extra attribute that identify to which graph they belong. The graph also has an attribute `__mlevel` that specifies the level to which `mcg()` has already been run on these graphs. When an `mcg()` command detects a composite graph as the third parameter, it ignores any *graphref* provided as its input parameter. This parameter may be omitted in such cases.

The pruning concept used in `mcg()` is derived from the FBT [14] algorithm for mining graph databases.

5. THE ANMOL SUITE FOR MOLECULAR ANALYSIS

Initially GRACE was designed with molecular biology applications in mind. However, GRACE has now evolved into an application independent graph database that can manage labeled graphs. Molecular biology applications are now addressed in a suite called AnMol (expanding to "Analysis of Molecules").

AnMol is an OLAP application for managing bio-molecular data. It has been tested primarily on protein records available from the Protein Data Bank (PDB) [11]. The design goals of AnMol are meant towards *analytical* queries – that return aggregate properties of the data set. Hence, querying collective properties like finding a cluster of similar molecules is given more importance in AnMol, than finding precise structural differences between pairs of molecules.

Structures of bio-molecules are usually represented by a set of (x, y, z) coordinates for each atom in the molecule. A commonly used mechanism for comparing two molecular structures is to use the root-mean-square (RMSD) distance between *corresponding* atoms of two molecules. The main challenge in RMSD calculation is to obtain this correspondence among the atoms.

In the DALI algorithm [10] all-pairs distances between atoms are computed to create a distance matrix for each molecule. Molecules are then aligned by aligning the matrices. This is done by swapping rows of one of the matrices until the largest common sub-matrix of distances is found.

A similar technique is also used in *geometric hashing* [7], that was introduced in the area of computer vision. Geometric hashing has also been applied in comparing molecular structures [8, 16]. Geometric hashing hashes objects in a 3D space by computing all-pairs distances among them.

However, establishing correspondences between distance matrices poses a problem in the above approaches. When objects in a 3D space are all of different types, it is simpler to establish this correspondence. However, molecules are characterized by large numbers of similar atoms (graph nodes with the same labels), which requires distance matrices to be first aligned before comparison.

In addition to 3D structures, several other features of biomolecules also form basis for comparison. For instance, similarity in the structure of hydrophobic regions in two or more proteins may be interesting, even if their 3D shapes differ. Geometric hashing cannot be used to compare molecules based on such non-spatial features.

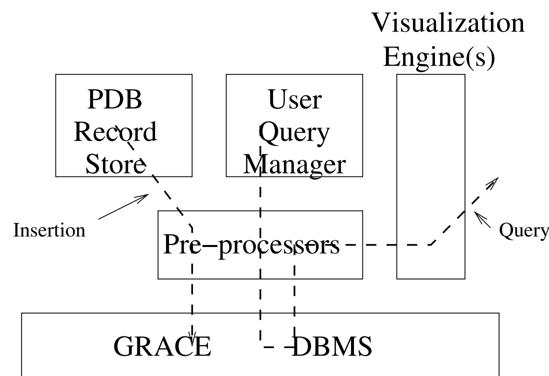

Figure 4: Overall AnMol architecture

In contrast, AnMol considers labeled graphs as the fundamental representation of molecules. Nodes and edges in these graphs may represent several things like: atoms and covalent bonds, atoms and hydrogen bonds, structural motifs and distance, etc. AnMol provides a set of *pre-processors* that take a 3D representation of bio-molecules (usually a PDB record in case of proteins) and convert them into one of these various labeled graphs.

Labeled graphs thus created are managed by the GRACE DBMS. AnMol supports structural queries based on similarity and substructure. Given a molecular structure, a similarity search returns other similar molecules. The measure of similarity can be biased by assigning relative weights to the different kinds of labeled graphs. For instance, assigning high weightage to hydrogen bond graphs and lesser weightage to covalent bond graphs would favour molecules that are similar in the hydrogen bonding structure than covalent bonding structure.

Once relevant graphs are retrieved from the database, AnMol ranks them based on the relative weightages assigned for each kind of graph. The ranked list is then linked with the original PDB records containing the 3D structure. The user can then use freely available visualization tools for rendering the returned molecules.

Figure 4 schematically depicts the overall architecture of AnMol.

6. PERFORMANCE ANALYSIS

GRACE implementation was tested on a set of PDB records pre-processed for proximity structure among atoms (all atoms that are less than 6 Angstroms from one another being connected), and a set of synthetic graphs.

Evaluation was performed on a Pentium II with 256MB of RAM. Unfortunately, comparison with related software like GraphGrep and gIndex could not be performed. GraphGrep had compilation problems due to some library requirement which could not be met till date; and gIndex could not be obtained.

PDB graphs were used mainly to measure graph loading times. With a maximum walk length of 4, insertion of a graph (which includes generation of its LWI entry) is shown in Figure 5.

While insertion is fast enough for a maximum walk length

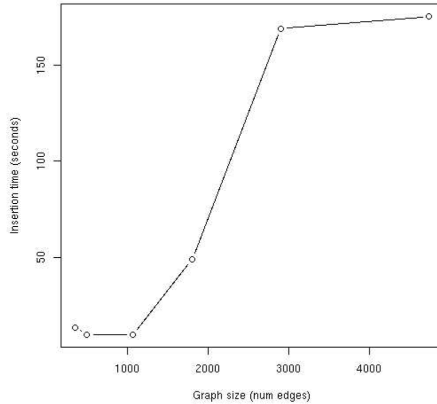

Figure 5: Insertion time against graph size for PDB graphs

of 4, a walk length of 5 made insertion take several hours when size of the graph increased beyond 4000 edges.

Insertion is hence performed in a batch mode; the control returns to the user as soon as a graph is given for insertion. The user is notified once insertion is complete.

In order to determine query result accuracy, a set of synthetic graphs were used. A graph generator that took a seed graph and generated graphs of various lengths was employed to populate the database.

Given a member graph as input, and a similarity region of radius $\delta = 0$, similarity search yielded accurate results in a maximum of 2 refinements. The same was true of substructure queries. When one of the seed graphs were given as input, accurate results were obtained after a maximum of 2 refinements.

The graph database contained 100 graphs ranging between 250 to 1000 edges for this experiment, and a maximum walk length of 4 was used. The queries took an average of 0.02 seconds (including the refinements) for substructure queries and 0.03 seconds for similarity queries.

7. RELATED LITERATURE

The schema in a traditional RDBMS can be represented by an Entity-Relationship labeled graph. Hence an RDBMS is a graph database; only, it stores different instances of the *same* graph – the schema graph.

Graph databases developed in the early 90s in OODBMS frameworks have sought to query and manipulate such schemas and their instances directly by treating them as graphs, rather than converting them to tables. These objectives and design principles are fundamentally different from the objectives of GRACE. Some representative examples are taken up here for comparison.

The GOOD [4] database system allows for storage of several graphs in a database. Graph nodes are objects that can be invoked to perform query and update operations. A notion of schema graphs is also proposed. A schema graph in GOOD is a graph comprising of only node labels without names. All instances of this schema are graphs that *have the same structure* but with different node names. By contrast, in GRACE, a schema graph contains meta-data that establishes relationships among member graphs, each of whom may have different structures unrelated to the schema structure itself. GOOD also does not seem to support structure based retrieval.

GOAL [5] is another example. GOAL is an extension of GOOD which supports different kinds of nodes representing simple data types, classes of objects and relationship types. GOAL also allows schema graphs (and hence all its instances) to be modified by addition and deletion of graph elements.

Several other database systems like XML data bases, spatial databases, hypertext webbases, etc. have used graph theoretic representation and query mechanisms. However, they are not directly relevant to GRACE which is meant for retrieval of labeled graphs based on structural similarity.

Several other database systems have been proposed that have similar objectives as that of GRACE. These systems are contrasted based on the approaches taken for structure-based retrievals.

In the SUBDUE database [6], a concept called hierarchical conceptual clustering is used to identify and index substructures in a graph database. Whenever an interesting substructure is found, all occurrences of the substructure are abstracted away by a single node. This is however a time-consuming process since it requires several structural comparisons. In addition, when two or more interesting substructures overlap, abstracting one will lose information about the other. A compression technique similar to that of SUBDUE was employed in earlier versions of GRACE [13, 15], but have been replaced by walk based approaches because of the above problems.

Graphgrep [3] is most similar to GRACE. However, Graphgrep stores all walks from graphs for exact matching and all searches are executed over the entire database. In contrast, GRACE allows the user to specify schematic constructs that restrict the search space. GRACE also uses vectorization concepts that help in returning "similar" graphs in an efficient fashion.

The gIndex approach [18] indexes frequently occurring substructures in order to query graph databases. Frequent substructures are mined using their earlier algorithm called gSpan [17]. Mining and graph matching is based on rewriting graphs into a canonical form that represents them as strings. Rewriting large graphs is a time-consuming process, especially when a large query graph has to obtain results in an interactive time frame. Also, if the graph database contains very few recurring substructures, the index is likely to become sparse. In contrast, GRACE indexes all structural features and provides for progressive refinements of query

results. The user can also restrict search spaces for more meaningful results by using appropriate schema graphs.

Several index structures based on label paths have been proposed in XML literature. Since they are primarily oriented towards XPath queries and not structural similarity, they are not referred here.

8. CONCLUSIONS

GRACE presents a mechanism for managing data and meta-data in graph databases in a uniform fashion. The Safari constructs enable a user to dynamically create new search spaces and schematic structures and search within them.

For future directions, a number of new functions are envisaged for Safari, that can expand the scope of GRACE from supporting structural queries over undirected labeled graphs to more general graph theoretic queries. Some potential application areas that are planned to be addressed using GRACE include knowledge management, XML databases and managing web usage patterns.

9. ADDITIONAL AUTHORS

K. A. Gowrishankar, Gopinath P.S., Indian Institute of Information Technology, Bangalore

10. REFERENCES

[1] S. Beretti, A. D. Bimbo, and E. Vicario. Efficient macthing and indexing of graph models in content-based retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23, 2002.

[2] CATH. Protein structure classification. *http://www.biochem.ucl.ac.uk/ bsm/cath/*.

[3] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *Proceedings of the International Conference in Pattern recognition (ICPR), Quebec, Canada*, 2002.

[4] M. Gyssens, J. Paredaens, J. V. den Bussche, and D. van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994.

[5] J. Hidders and J. Paredaens. Goal, a graph-based object and association language. *CISM - Advances in Database Systems*, pages 247–265, 1993.

[6] I. Jonyer, L. B. Holder, and D. J. Cook. Hierarchical conceptual structural clustering. *International Journal on Artificial Intelligence Tools*, 10:107–136, 2001.

[7] Y. Lamdan and H. Wolfson. Geometric hashing: A general and efficient model-based recognition scheme. In *Proc. of the IEEE Int'l Conf. on Computer Vision*, 1988.

[8] N. Leibowitz, Z. Y. Fligelman, R. Nussinov, and H. J. Wolfson. Multiple structural alignment and core detection by geometric hashing. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 169–177. AAAI Press, 1999.

[9] L.Holm and C. Sander. Fold classification based on structure-structure alignment of proteins. *http://www2.ebi.ac.uk/dali/fssp/*.

[10] L.Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *Journal of Molecular Biology*, 23, 1993.

[11] PDB. The protein data bank. *http://www.pdb.org/*.

[12] SCOP. Structural classification of proteins. *http://scop.mrc-lmb.cam.ac.uk/scop/*.

[13] S. Srinivasa, S. Acharya, H. Agrawal, and R. Khare. Vectorization of structure to index graph databases. In *Proceedings of IASTED Int'l Conf. on Information Systems and Databases (ISDB'02), Tokyo, Japan*. Acta Press, 2002.

[14] S. Srinivasa and L. BalaSundaraRaman. A filtration based technique for mining maximal common subgraphs. *Under revision towards publication in IEEE TKDE*, 2003.

[15] S. Srinivasa and S. Kumar. A platform based on the multi-dimensional data model for analysis of bio-molecular structures. *Proceedings of VLDB 2003, Berlin, Germany*, 2003.

[16] X. Wang and J. T. Wang. Protein classification: A geometric hashing approach. In *Computational Biology and Genome Informatics*. World Scientific Publishing Company, 2003.

[17] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of ICDM 2002*, 2002.

[18] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure based approach. In *Proceedings of SIGMOD 2004, Paris, France*, June 2004.