Data, Storage and Index Models for Graph Databases

Chapter · August 2011

CITATIONS
13

READS
2,668

1 author:

Srinath Srinivasa
International Institute of Information Technology Bangalore
94 PUBLICATIONS **284** CITATIONS

SEE PROFILE

Data, Storage and Index Models for Graph Databases

Srinath Srinivasa
International Institute of Information Technology
Bangalore, India
sri@iiitb.ac.in

ABSTRACT
Management of graph structured data has important applications in several areas. Queries on such data sets are based on structural properties of the graphs, in addition to values of attributes. Answering such queries pose significant challenges, as reasoning about structural properties across graphs are typically intractable problems. This chapter provides an overview of the challenges in designing databases over graph datasets. Different application areas that use graph databases, pose their own unique set of challenges, making the task of designing a generic graph-oriented DBMS still an elusive goal. The purpose of this chapter is to survey some of the piecemeal solutions that have been proposed to address specific challenges in graph data management and suggest an overall structure in which these different solutions can be meaningfully placed.

Keywords: Graph data management, Graph database, Structural index, Graph isomorphism, Graph data models, Graph storage models

INTRODUCTION
Increasingly, management of data representing *relationship structures* across entities is an important issue in several application domains. Often, semantics are hidden in the way different entities are related, which may not be obvious by studying each entity in isolation. Querying on properties of such relationship structures are sometimes as important, if not more so, as querying for values of specific attributes.

Some application examples follow:

- In proteomics – or the study of proteins, it is well known that the conformation, or the 3D structure, of a protein is strongly correlated to the functioning of the protein (Ex: Graves et. al, 1995; Colinge and Bennett, 2007). The conformation map of a protein is formed by interactions between the different atoms that make up the protein molecule. Studying these interaction patterns hence, is of immense importance
- In the field of computer-aided design of electrical circuits, some interconnection patterns across electrical devices keep recurring in different problems. Managing data about electrical circuits and comparing similarities between two or more circuits is of significant importance
- In the field of information security, a commonly occurring challenge is to look for anomalous behavior by one or more entities, like people, software, etc. in the system. Anomalies constitute deviations from the norm that are distinct from "noise" or "novel" deviations – both of which also constitute deviations from the norm (Chandola et. al

2009). A central aspect of anomaly detection constitutes studying and reasoning about the *relationships* that an entity has with other entities in the system

- Designers of public infrastructure like airports, hospitals, etc. face a number of conflicting requirements concerning safety, efficiency, reachability, cost, etc. Conflicting requirements like these are often modeled as *network design* and *facility location* problems (cf. Brandes and Erlebach, 2005) over graphs. Since designing a facility from scratch is often a hard problem, architects use design patterns depicting best practices and architectural features that have been recorded over time. Management of such datasets requires the database to support structural queries over graphs.

Management of graph structured data is not a new problem in the field of databases. Most data structures at the application level usually need to deal with some form of graph structure. Early approaches to data management did not have clear separations between logical and physical views of data. As a result, graph structured data from the application were stored directly as a networked set of records at the physical level. An example of this is the Network data model (Taylor and Frank, 1976).

Later approaches towards data management had clearer separations between the physical layout of records on disk and the logical structuring of the database schema. The most popular among these is the relational data model (Codd 1983), which has become the standard for most commercial DBMS today. The relational data model gave mathematical underpinnings to the logical structuring of data, based on concepts of relations, functional dependencies and normal forms. Relations in a relational schema are captured in two different forms: an n-ary relation across several attributes of a given semantic entity; and relations across entities. The former kind of relation is represented in the form of a table – every column in a table is related to every other, by virtue of being attributed to a single entity. The second form of relation is represented by means of foreign keys across tables.

So, in a sense, every relational database represents a graph structured logical model for the data. However, conventional relational databases are inadequate for the challenges of managing graph data as illustrated in the application examples. They can be summarized as follows:

- A relational database represents several instances of a single graph structure that is captured by the schema. Graph databases on the other hand, may need to contend with several graphs with different structures as part of the same database
- While a graph can be represented as a set of relations (for example, modeling each edge as a relation) and stored as a set of rows in a table, doing so would violate an implicit assumption in relational databases that each row of a table represents an entity instance that is independent of all other instances of that type
- Index structures in relational databases are focused on answering queries over values of attributes. Answering queries concerning relationship structures usually requires a large amount of joins, making it very expensive.

The motivation for developing native database models for graph structures have been varied. Graph based logical data models generated a lot of interest in object-oriented databases in the late 1980s and early 1990s (Kim, 1990). The primary motivation for using graphs in object-oriented databases was to enable representation of complex objects, type systems, associations and inheritances natively in the database.

Some examples are: GOAL (Hidders & Paredaens, 1993), GOOD (Gemis et al., 1993), and GraphDB (Güting, 1994). As mentioned earlier, the main motivations behind using graphs in object databases, centered upon *representation of complex objects* and relationships. However, these are inadequate for the kinds of challenges that require graph data management today. For instance, supporting queries over structural properties across several graphs, has not been one of the main elements of interest in object databases. In addition, in most object databases, graph structured data needed to conform to a predefined schema (Angles & Gutierrez, 2008), thus making their structural properties known a priori.

Graph data management has been addressed in several problems involving knowledge management. Some of the early applications included knowledge representation problems in artificial intelligence, where management of *semantic networks* have been a primary concern. Examples include: Telos (Mylopoulos et al., 1990), Controlled Decomposition Model (CDM) (Topaloglou, 1993), etc. Semantic networks represent relationships across disparate entities, thus forming a large graph data structure. Such databases also need to provide support for inference, which typically requires indexing *paths* over the semantic networks.

Later, in the fields of digital libraries and the Semantic Web, representing knowledge constructs have become a primary focus. In addition, research efforts on management of semi-structured data address a variety of graph related problems as part of data management, like path constraints, inclusions, extents, inverses etc. Some examples include (Abiteboul, 1997; Abiteboul & Vianu, 1997), (Bertino et al., 2003), (Buneman et al., 1998).

Application areas like bioinformatics, have posed significant problems of graph data management. Many concepts concerning bioinformatics – be they molecular structures, metabolic pathways or protein interaction structures – are naturally represented in the form of graphs. The late 1990s and the early 2000s have seen several research models for representation of biological data in the form of graphs. Some examples include: GRACE (Srinivasa et al., 2002; Kumar & Srinivasa 2003; Srinivasa et al., 2005), (Borgelt & Berthold, 2002), GraphGrep (Guigno & Shasha, 2002), gSpan (Yan & Han, 2002), etc. Management of graph data in bioinformatics applications continues to be an area of active research interest.

In more recent times, the popularity of online social networking and collaborative bookmarking sites have created huge amounts of graph structured data, which cannot be reliably stored and managed using relational databases. As a result, there have been several approaches to create custom storage and query models for managing such large graph data. Several of these disparate efforts have also come together under a banner called NoSQL (which, rather than representing a rejection of SQL, is claimed to stand for "Not Only SQL") databases. To be sure, NoSQL is not just about storing graph structured data – but web scale data. There are several underlying models that are used by databases that call themselves NoSQL. For instance, CouchDB[1] from the Apache Foundation, is a document store that supports ACID (atomicity, consistency, isolation and durability) semantics on a large document repository. It employs the MapReduce[2] model for creating views and interacting with applications. Cassandra[3], also by the Apache Foundation, implements a column store based on Google's BigTable model. Cassandra is used by several sites like FaceBook, Twitter and Digg to manage large graph datasets. The Java based neo4j[4] implements a transactional graph data store that provides native support for describing several graph related properties like nodes, edges, paths, etc.

[1] CouchDB Technical Overview. http://couchdb.apache.org/docs/overview.html
[2] Wikipedia page for MapReduce. http://en.wikipedia.org/wiki/MapReduce
[3] Cassandra home page. http://cassandra.apache.org/
[4] Neo4j website. http://neo4j.org/

It is quite clear that graph data management is a pertinent issue in several application areas. However, research efforts towards graph data management have been piece-meal and isolated, with no overarching theory of a graph DBMS emerging. Even more fundamentally, there is no clear consensus on what constitutes a graph database – other than the fact that the data elements of interest lie in the patterns of relationships, in addition to values of attributes.

The objective of this chapter is to introduce some issues of concern pertaining to graph data management, and where relevant, contrast it with similar problems in relational databases. It is not meant to be a survey of existing graph databases; however relevant literature would be cited as part of the process of characterizing this space.

Specifically three kinds of issues are considered: data models, storage models and index structures for graph databases. Data models address the logical model of the graph database. This in turn depends on the type of graph that is stored as well as the way in which the graph data is envisaged to be used. Storage models look into physical storage structures for graph data. Specifically, we shall look into the contrasting approaches of mapping graph databases onto relational data stores versus native storage structures for graph databases. The last issue of concern – that of index structures – looks into structural indexes. A structural index is a augmented data structure for a graph database that extracts and pre-computes certain structural features of the graph data and stores it in a global index structure. This index is then used for quick answering of structure-based queries.

A topic conspicuous by its absence, is query language and query processing for graph databases. This is a vast area of research interest in itself and is omitted from this chapter for reasons of brevity and space constraints. The interested reader is recommended to read (Angles & Gutierrez, 2008; Sakr & Al-Naymat, 2010) for surveys addressing query models in graph databases.

DATA MODELS FOR GRAPH DATABASES

Data models are concerned with the logical structure of the database. Based on constraints posed by the application areas, there are different paradigms of logical models for graph databases. To understand this space, we need to first consider the different kinds of graph data that are typically handled by applications requiring graph databases.

Types of Graph Data

A graph represents a set of relationships among a set of objects or entities of interest. In the most generic form, a graph is represented as a tuple: $G = (V,E)$, where V is a set of *nodes* or *vertices* and $E \subseteq V \times V$ represents pair-wise relationships across the nodes.

Nodes and edges of a graph may contain attributes, whose values may be of interest. However, they can usually be searched by storing them in a relational data store. We will not be concerned with querying for values of attributes of nodes and edges – rather, we will be interested in queries over the graph structure itself.

A graph is said to be *directed* if for any pair of nodes $u,v \in V$, $(u,v) \neq (v,u)$. The graph is said to be undirected if the direction of the relationship does not matter. In such cases, an edge is also represented as a 2-element subset of vertices, rather than an ordered pair.

A dataset depicting hyperlinks across web pages can be seen as a directed graph, where the nodes are web pages and edges are hyperlinks. On the other hand, a dataset representing friendships on a social networking site like Facebook, can be seen as an undirected graph, where a friendship relation from A to B implies a similar relationship from B to A.

In terms of storage, in undirected graphs, a pair of nodes that are connected by an edge, need to be stored only once in order to represent their relationship, while for a directed graph, a pair of nodes may need to be stored twice depending on the existence of bi-directional relationships. The directed nature of graphs also has a bearing on some indexing structures. Some kinds of structure indexes depend upon reducing a graph structured data into a string by performing a depth-first traversal (DFS) of the graph and then encoding the resultant DFS tree into a string. A DFS traversal on undirected connected graphs always results in a tree, while for directed graphs, a DFS traversal may result in a forest, depending on where the traversal started (Cormen, et al., 2001).

A graph is said to be *labeled* if there exist sets V_L and E_L, representing sets of node and edge labels, and label assignment functions functions δ: $V \rightarrow V_L$ and γ: $E \rightarrow E_L$ respectively. Usually the following holds: $|V_L| \ll |V|$ and $|E_L| \ll |E|$. That is, the number of node and edge labels are much less than the number of nodes and edges of the graph.

A label can be seen as a *type* assignment for nodes and edges. All nodes and edges having the same label can be seen as belonging to the same type. Labels are *categorical* values – that is, they cannot be ordered or compared using a relation like less-than-or-equal-to (≤).

As an example, an organic molecule comprises of a large number of relatively small types of atoms like Carbon, Hydrogen, Oxygen, etc. Each Carbon atom is a different node in the graph, but has the same label as all other Carbon atoms. Similarly, edge types representing covalent bonds in an organic molecule can have one of three labels: single bond, double bond or triple bond.

For graph databases, labels provide a means by which nodes and edges can be bucketed together. Such a bucketing can be used to address problems in both storage and indexing of graphs in the database.

A special form of labeled graph data are *bi-partite* or more generally, k-*partite* graphs. A k-partite graph is a labeled graph where $|V_L| = k$ and $|E_L| = 1$ with the following constraint on the edges: an edge can connect two nodes if and only if they do not have the same label.

Bi-partite and k-partite graphs depict data sets showing interaction between two or more classes of entities. For instance, an online bookstore like Amazon.com contains data about customers buying books. This entire dataset can be represented as a bi-partite graph where nodes belong to one of two types – customer and book. Edges connecting customers and books depict which customers have bought which book. Similarly, a social bookmarking engine like del.icio.us (now called delicious.com) has a tri-partite dataset made of the following kinds of nodes: users, tags and URLs. Users use tags to annotate URLs. This can be captured in the tri-partite graph as edges across: users and tags, users and URLs and tags and URLs.

Edges in a graph may also be associated with numerical weights. Such graphs are called *weighted graphs*. Here, the edge set E is defined as: $E \subseteq V \times V \times \Re$, where $\Re$ is the set of real numbers.

Depending on the application context, these weights may take on different meanings. For instance, a road network can be represented in the form of a graph, where intersections are denoted by nodes and roads by weighted edges. The weight on each edge in this case, would depict the distance across intersections. Unlike labels in a labeled graph, weights represent quantities, rather than types, and can be compared.

Queries over a weighted graph typically require some form of a min/max computation over paths or substructures. Examples include shortest path queries in road network datasets, finding near-cliques with maximum pair-wise energy in molecular structures, etc.

Nodes in a graph can be sometimes associated with *spatial coordinates* in the form of (x,y,z) or latitude, longitude and altitude values. Such forms of graphs are called spatial graphs – graph structures embedded inside S^3 – the 3-dimensional sphere (Kobayashi, 1994). Examples include road networks, where each node – representing an intersection, can be annotated with a latitude, longitude and altitude coordinates. Similarly, the tertiary structure of a protein molecule is represented as a spatial graph. Here, nodes represent atoms and links represent covalent bonds across atoms. In addition to these, a protein is also characterized by its 3D conformance, where the 3D coordinates of each atom is important to determine the structural (and hence functional) characteristics of the protein molecule.

In spatial graphs, the neighborhood of a node can be defined in two ways – the *spatial neighborhood*, comprising of other nodes that are close in terms of their coordinates; and the *logical neighborhood* defined by edge connectivity. The existence of two kinds of neighborhoods can be exploited in problems like facilitating routing and shortest-path queries in spatial graphs.

Some kinds of applications require management of *hypergraph* data. A hypergraph is a generalization over a graph structure, where an edge may connect any number of nodes, rather than just a pair of nodes. Formally, a hypergraph is of the form H = (V,E), where V is the set of nodes and $E \subseteq 2^V-\{\phi\}$, where 2^V is the powerset (set of all subsets) of V and, ϕ is the empty set. Hypergraph data management have been explored in fields like knowledge representation and object databases. Examples include GROOVY (Levene & Poulovassilis, 1991) and Hy+ (Consens & Mendelzon, 1993). Higher-order relationships – relationships that relate more than a pair of entities – often cannot be reduced to a set of pair-wise relationships without losing semantics.

Another concept related to that of hypergraphs is the notion of a *hypernode*. A hypernode in a graph is a vertex that represents another graph structure (which in turn may contain more hypernodes). Hypernodes represent a way by which graph structures can be aggregated into a conceptual hierarchy, and enable graphs to be used both as a data element and a schema element. Examples of graph databases supporting hypernodes include (Levene & Poulovassilis, 1990), (Jonyer, et al., 2000) and (Srinivasa, et al., 2005).

As is apparent, although management of graph data is required in several application areas, there are different kinds of graph data that needs to be managed – each suitable for a different application area, and having its own specific forms of queries that need to be supported. It is not surprising then that, there is no single data model for a graph database. The next subsection explores some of the different data models of graph databases and their suitability for specific forms of graph data.

Data Models

Different kinds of graph database models have been proposed, each of them being designed in the context of some application area. Broadly however, data models for graph databases can be divided into the following kinds:

1. Database as a collection of graphs
2. Database as a nested graph
3. Database as a graph

Each of these data models are explained in more detail below.

Database as a collection of graphs. In the first model of graph databases, the database can be seen modeled as D = (M,I), where M represents a set of *member* graphs and I represents a set of *auxiliary* graphs that make up the database. Each member graph $G \in M$ is of the form G = (V,E), and represents a distinct data structure from all other member graphs. The set I also comprises of one or more graphs that provides navigational support for the database management system to index into the database.

The graphs in I are strictly meta-data and cannot be part of the result set of a query. The actual data of interest are confined to the graphs in M. Each member graph in M may have a different structure, and there is no overarching schema that describes this collection. While meta-data is managed separately as index structures, this is different from having a schema that is defined a priori, and that describes how the database is organized.

Member graphs are also typically small in comparison with the size of the database. Queries on such databases focus on matching structural properties *across* member graphs in the database and return zero or more member graphs from M as the query result.

Datasets of molecular structures and protein conformance datasets can be seen as examples of such a database. Many such databases exist in the field of bioinformatics, that provide various query services, including queries that search the dataset based on structural properties of the member graphs. The NAR database summary[5] lists several such databases providing online services pertinent to nucleic acid research, many of which manage datasets in the form of a collection of graphs. Other examples of databases as a collection of graphs, include: GraphGrep (Guigno & Shasha, 2002; Shasha et al., 2002) and GRACE (Srinivasa, et al., 2002).

Database as a nested graph. In the second form of graph databases, the database is still a collection of member graphs. But, member graphs are connected together by other graphs describing schematic structures, which in turn can be described by higher level schema graphs; converting the entire database into a single *nested graph* model. One of the main design objectives behind such graph databases is the rich expressiveness that is obtained by treating data and meta-data seamlessly using a common data structure.

Database as a nested graph model have been primarily employed in object database models and database models for knowledge management. In such databases, meta-data do not play an

[5] NAR database summary. http://www.oxfordjournals.org/nar/database/a/

auxiliary role. Data and meta-data are both part of the data stored in the database and can be returned in response to a query. Some examples follow.

The hypernode database model (Levene & Poulovassilis, 1990; Levene & Loizou, 1995) models the database as a single data structure called the *hypernode*. A hypernode is a graph structure comprising of nodes and edges, where each node in turn can represent other hypernodes.

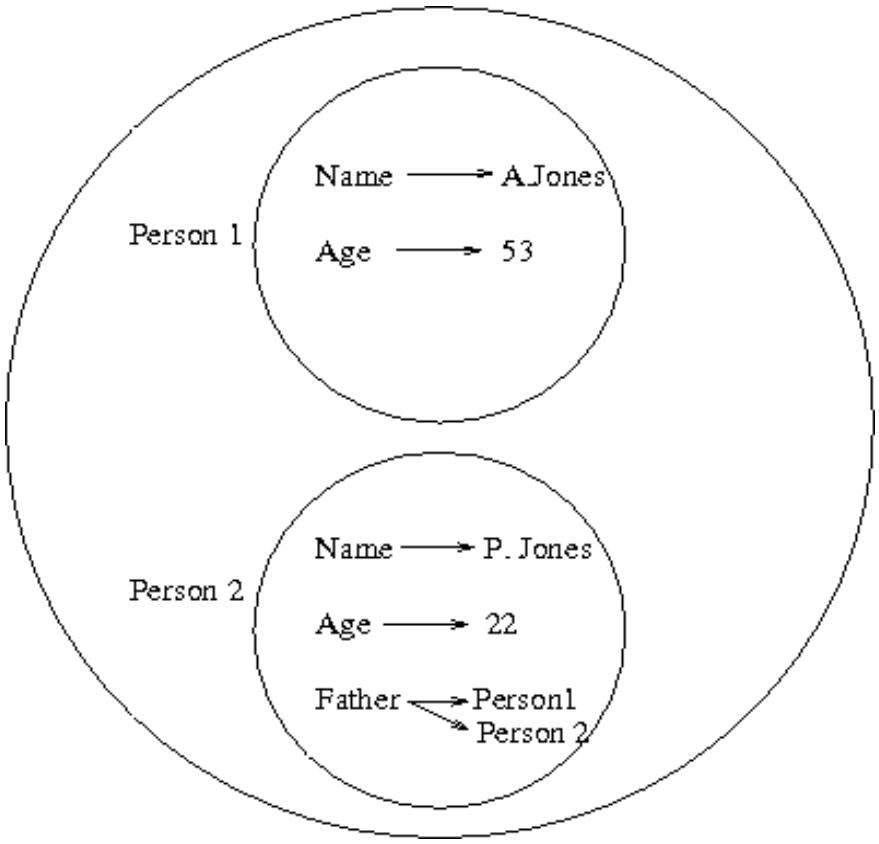


Figure 1: *An example hypernode database comprising of a single hypernode with two nodes, each of which represents a graph*

Figure 1 shows an example hypernode database showing two "Person" nodes, where each node in turn represents a graph showing relationships between attribute names and values. Formally, a hypernode is defined as a triple (G,V,E), where G is the label of the hypernode, V is a set of nodes and E is a set of directed edges connecting the nodes. Hypernodes can be used to represent a variety of concepts including aggregation, composition, encapsulation and cyclic relationships. The hypernode model also comes with a programming language over hypernodes, supporting a specific operator called INFER, that can infer new hypernodes based on existing hypernodes in the database based on a specified set of rules.

The *Safari* data model described as part of the GRACE graph database (Srinivasa et al., 2005) has a somewhat similar approach. The graph database comprises of several graphs, where a node in a graph can be classified as either a "data" node or a "meta" node. A data node comprises of atomic data objects, while meta nodes refer to other graphs in the database. When a new database is

created, it creates a single graph called **_default**, which comprises of a single meta-node pointing to itself. On insertion of a member graph into the database, a corresponding meta node is automatically created in the **_default** graph, which points to the new member graph. By default, the **_default** graph contains no edges, but edges can always be added to member graphs at any time. Member graphs can in turn, contain a combination of data and meta nodes, and relationships between them.

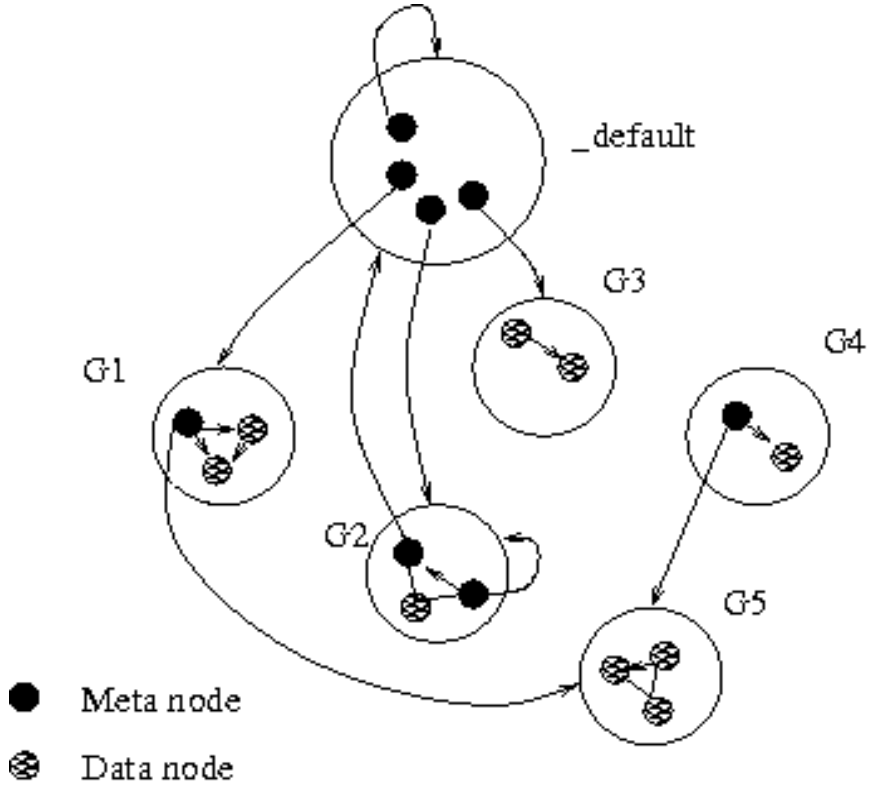


Figure 2: *A schematic diagram of a Safari database. Not all member graphs are shown as meta-nodes in the* ***_default*** *graph, however, in the implementation, each member graph contains a representative meta node in* ***_default****.*

Figure 2 schematically depicts a graph database in the Safari model. A meta node in any graph can refer to another graph in one of two ways – *statically* using the graph id of the target graph, or *dynamically*, using a standing query. The Safari data model comes with a query language that supports closed operations on graph selections. That is, the input and output to these operators is a single graph. There are three such closed operators: select-in, select-on and select-graph. The first two induces new graph structures by searching "inside" (by unfolding the meta nodes and expanding the search) a graph or "on" a graph (without unfolding the meta nodes). The select-graph operator takes a graph that has one or more meta-nodes and returns one of the target graphs pointed by the meta nodes that match the query.

A meta-node with a standing query hence can point to graphs that don't physically exist in the database, and are materialized at query time.

A third example for nested graph database models are RDF[6] databases. RDF, which is a W3C (World Wide Web Consortium) standard is used to provide precise meta-data for web resources. RDF is made of triples of the form (subject, predicate, object). Here, subject and object are resources, where a resource is anything that has a Universal Resource Identifier (URI) associated with it. A collection of RDF triples that describes a web resource can be seen as a directed graph with edge labels. Since such an RDF description can be made available as a web resource, an RDF description can describe resources that are in turn RDF descriptions themselves, making an RDF store a nested graph.

Database as a graph. The third model for graph databases represents the entire database as one big graph. Unlike the previous approach, the graph comprises of only data elements. There may be other augmented graphs for indexing into the data graph, however, these data structures are not integrated into the data graph. This is a common approach used for handling web-scale graph data like hyperlink networks, social network data, etc.

In principle, the database as a graph model can be seen as a special case of the "Database as a collection of graphs" model, where the collection size is 1. However, in practice, the design principles of these two kinds of databases have very different motivations. In web-scale graph data, structural queries *on* the graph is of primary concern, while in a collection of graphs, structural queries *across* the graphs are of interest. Also, web-scale graph data usually need to support several kinds of structural queries like shortest-path queries, community or near-clique detection queries, centrality queries, etc. that are of little interest in a collection of graphs where each member graph is relatively small in size.

Some examples of this model of graph databases follow.

The neo4j graph database represents the entire database as a graph[7]. It supports two kinds of first-class objects – nodes and relations. Both of them can store one or more properties in the form of attributes and their values. The entire database is then modeled as a set of nodes and relations across nodes. For graph related queries, a framework called "Traverser" is provided that can support a variety of traversal and neighborhood queries. A Traverser query comprises of four components: (a). A starting node, (b). the kinds of relationships to traverse, (c). a stop criteria and (d). a selection criteria that selects a set of nodes as the result from the set of nodes obtained by the traversal.

Dex (Martínez-Bazan, et al., 2007) is a Java based library for managing large graph datasets. Dex supports labeled multi-graphs (graphs having more than one edge between a pair of nodes), where edges can be either directed or undirected. Nodes and edges can be associated with a set of attributes and values, which can be queried upon. Dex also introduces a concept called "virtual edge" that connects nodes having the same value for a given attribute. Virtual edges are non-materialized edges and are used for managing referential integrity, in much the same way as foreign keys are used in relational databases.

A Dex database is in the form of a graph that is an instance of the DbGraph class. Dex supports a set of basic graph traversal queries over which other sophisticated algorithms are implemented. The basic support includes operations like "Explode" that gets the set of all edges of a given type from a given node, and "Neighbors" that returns the set of nodes that are adjacent to a given node. Dex also implements a number of graph algorithms in packages. These include shortest-path

---

[6] Resource Description Framework (RDF). http://www.w3.org/RDF/

[7] Neo4j Technical Introduction. http://dist.neo4j.org/neo-technology-introduction.pdf

algorithms, graph traversal algorithms (breadth-first and depth-first traversals), algorithms to find strongly connected components of directed graphs, etc.

STORAGE MODELS FOR GRAPH DATABASES

As with any non-standard database, storage models for graph databases can be seen from two perspectives – graph databases implemented on a relational store, and graph databases implemented over native storage models.

Graph Databases over Relational Storage

A graph structure can be stored as a set of relations in a conventional relational database table. Attributed Relational Graph (ARG) has been a popular model of graph structures in the late 80s and 90s, and is continued to be used today in applications like computer vision and pattern recognition (Tsai & Fu, 1979; Chang & Fu, 1980; Sanfeliu & Fu, 1983; Kim et. al., 2010). ARGs are labeled undirected graphs where both nodes and edges are labeled. Much of the work on ARGs use conventional databases and store an ARG in one or more tables.

An unlabeled graph can be stored in a relational table with two columns, by treating each edge as a row and the two columns representing nodes on either side of the edge. Labels and attributes can be maintained separately in other tables and referred by foreign keys.

Another example for a graph database implemented over relational storage is Periscope/GQ, by the University of Michigan (Tian, 2008). Periscope/GQ implements a graph database as an application over the PostgreSQL relational database engine. The graph database supports labeled graphs that are either directed or undirected. The database uses three relational tables – a "Graphs" table, a "Nodes" table and an "Edges" table to implement the entire database.

While a relational representation of graphs does capture every property of the graph data structure, storing graphs as relational tables is inefficient for large graphs. Even fairly simple traversal algorithms like finding the k-hop neighbors of a given node would require costly self joins on the table making it impractical for managing large graph datasets.

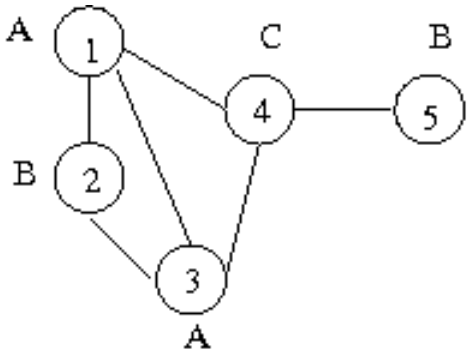
Native Models for Graph Storage

For managing large graphs and/or large collections of graphs, several innovative models of native storage have been proposed. We review some example methods of native graph storage below.

Graph Fingerprinting. GraphGrep (Guigno & Shasha, 2002) is an application independent graph database that manages a collection of undirected, labeled graphs. Here, only nodes are assumed to have labels, while edges are considered unlabeled.

Graph storage in GraphGrep comprises of two elements – label paths and a fingerprint table. A member graph of a GraphGrep database is of the form $G = (V, E, V_L, \delta)$ where V is a set of nodes, E is a set of edges V_L is the set of node labels and $\delta: V \rightarrow V_L$ represents label assignment for the nodes. Each node in such a member graph is said to have a unique identifier that distinguishes it from all other nodes.

For such a member graph, two kinds of paths are defined: an *id-path* and a *label-path*. An id-path of length k comprises of a sequence of k node ids such that there exists an edge between any two consecutive node ids. A label-path of length k comprises of a sequence of k node labels, such that there exist at least one id path of length k, where the node ids have the corresponding labels of the label path. A member graph is then represented as a set of label-paths, up to some maximum length (denoted as l_p), where each label-path comprises of a set of id-paths. Label-paths are computed starting from all nodes in the graph up to the maximum length.



(a)

A = {(1), (3)}
B = {(2), (5)}
C = {(4)}
A–A = {(1,3), (3,1)}
A–B = {(1,2), (3,2)}
A–C = {(1,4), (3,4)}
B–A = {(2,1), (2,3)}
B–C = {(5,4)}
C–A = {(4,1), (4,3)}
C–B = {(4,5)}
A–B–A = {(1,2,3)}

Figure 3: *Label-paths and the corresponding set of id-paths for an example graph*

Figure 3 shows an example member graph and a computation of label-paths with the corresponding set of id-paths.

Once the label-paths of a member graph are computed, the entire database itself is stored in the form of a table called the fingerprint table. The table lists all label-paths occurring in the database and for each member graph, denotes the number of id-paths that the graph has for the given label-path.

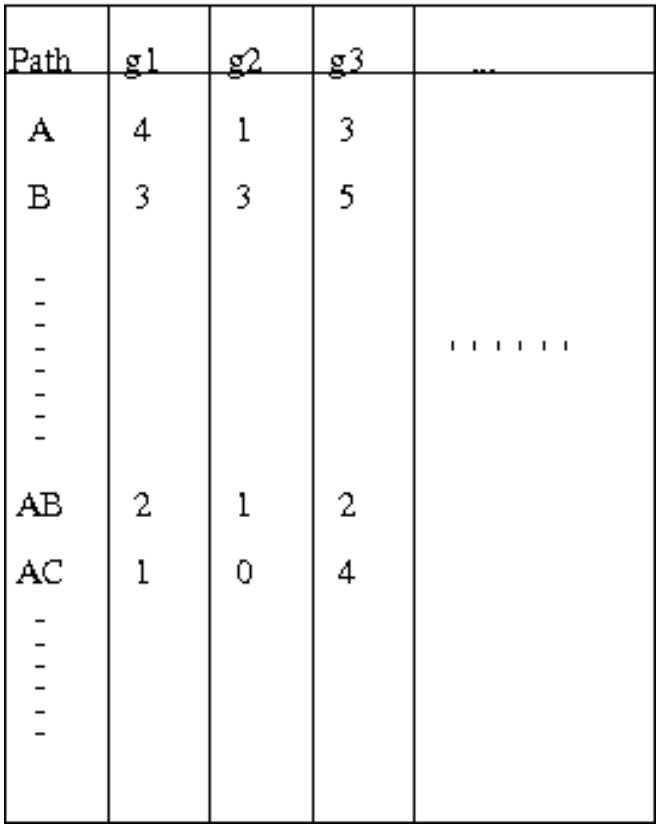


Figure 4: *An example fingerprint table for a GraphGrep database*

Figure 4 shows an example fingerprint table for a GraphGrep database. This table also serves as an index structure that is used to filter the candidate set of graphs in response to a structural query. A later version of GraphGrep called RelaxGrep (Bonnici et al., 2010) also supports *relaxed graph matching* – that is matching a query graph Q with some of its nodes and edges removed.

Minimum DFS sequence. Another innovative method of storing graphs is to encode them in the form of unique sequences in a way that comparisons related to graph structure can be mostly reduced to string comparisons.

String comparisons are much faster than structure comparisons. As a result, such an encoding can answer structural queries efficiently. The complexity is relegated to the task of reducing a graph to a unique string so that string comparisons are possible. Since this operation is a one-time operation and performed off line, it has no bearing on the query response time. However, such an approach would be suitable primarily for graph databases that are read-only, since any modification to the member graphs will require re-computation of its unique string.

An example of such an approach is gSpan (Yan & Han, 2002), where a member graph is reduced to a unique string that represents a depth-first search (DFS) tree of the graph. The approach is called minimum DFS encoding and is explained below.

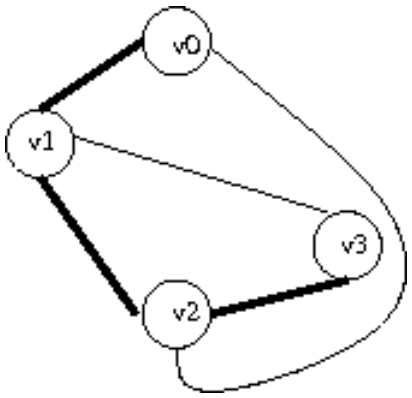


Figure 5: *A DFS traversal on an undirected graph with tree and back edges*

Consider a connected, undirected graph as shown in Figure 5. A depth-first search on such a graph starting from a node (v_0 in the example) creates a DFS tree, by traversing the graph in a depth-first manner. The edges that form the DFS tree are called *tree edges* (or *forward edges*) and the rest of the edges of the graph are termed *back edges*.

The classification of edges into tree and back edges are based on the discovery times of the nodes on either side of an edge (Cormen, et al., 2001). For a given node v, let d(v) be the discovery time of node v. Discovery time is a logical time stamp that denotes the time at which the node was first encountered. An edge of the form (u,v) is a tree edge if $d(u) < d(v)$ and is a back edge if $d(u) > d(v)$.

A given graph generates several DFS trees when the process begins from each node separately. Each DFS tree generated is written in the form of a sequence of edges. If $e_1 = (u_1,v_1)$ and $e_2 = (u_2, v_2)$ are both forward edges in the sequence, then e_1 precedes e_2 in the DFS sequence iff $d(v_1) < d(v_2)$. Similarly, if e_1 and e_2 are both back edges, then e_1 precedes e_2 iff either $d(u_1) < d(u_2)$ or $d(u_1) = d(u_2)$ and $d(v_1) < d(v_2)$. If e_1 and e_2 are not the same kind of edge, then e_1 precedes e_2 iff the following holds:

1. e_1 is a back edge, e_2 is a forward edge and $d(u_1) < d(v_2)$
2. e_1 is a forward edge, e_2 is a back edge and $d(v_1) \leq d(u_2)$

The above rules are collectively shown to create a linear ordering over all edges of a DFS tree. Once all DFS trees from a graph are mapped to sequences, the lexicographically smallest DFS sequence is chosen to represent the *canonical labeling* of the graph. It is also shown that isomorphic graphs have the same canonical labeling.

Basis Graph. Storing graphs as path indexes or DFS sequences have several advantages. They can support fast searches based on structural properties of member graphs. However, they are mainly suitable for read-only databases. This is because conversion of graph structures into fingerprints or canonical DFS sequences is a costly operation. This is carried out at graph insertion time, and is a one-time operation. However, if member graphs are modified, then every modification requires re-calculation of the fingerprint or canonical DFS sequence, making such an approach unsuitable for graph databases where member graphs are subject to frequent modifications.

Singh and Srinivasa (Singh & Srinivasa, 2006) propose an alternative approach to graph storage that favors a graph database that is read-write in nature, but which also needs to support structural similarity searches. This model, called "Basis Graph" represents a combined storage and index structure for member graphs in a database supporting labeled, undirected graphs.

In this model, the entire database is represented as a single graph called the Basis Graph. The nodes of the Basis Graph represent all unique node labels that are present in the database. Edges in the Basis Graph represent one or more labeled edges from one or more graphs in the database. The undirected Basis Graph also supports edges from a node to itself. This indicates the presence of edges between two nodes of the same label, somewhere in the database.

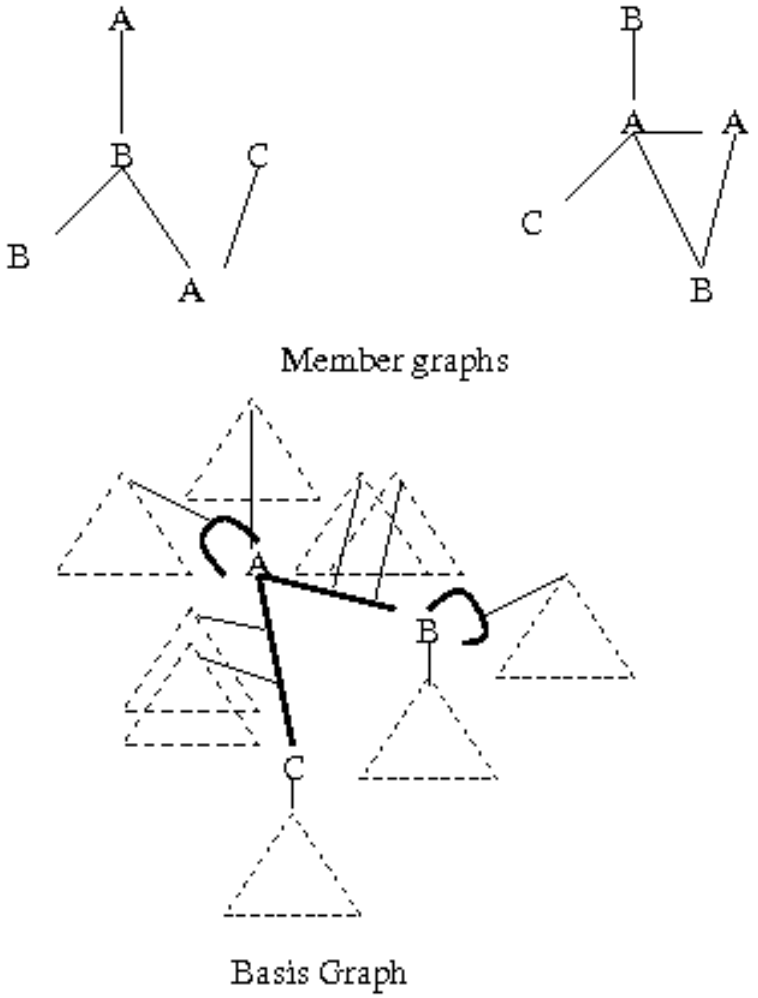


Figure 6: *Basis graph for two labeled member graphs*

Figure 6 schematically shows the basis graph created from two member graphs. Edges of the basis graph are shown in bold and the dashed triangles in the graph depict B+ tree structures that are associated with each node and edge of the basis graph. Every node of the basis graph has an associated B+ tree[8], while every edge connecting distinct labels have two B+ trees associated with them. For edges connecting a node to itself, there is one B+ tree associated with it.

[8] Wikipedia page for B+ tree. http://en.wikipedia.org/wiki/B%2B_tree

B+ trees are efficient data structures to store, access and delete sorted records. The primary key for the B+ trees associated with node C of the basis graph maintains records of the form (gid, nid, rid). Here, gid is the identifier of a member graph, nid is the identifier of a node in the member graph having label C and rid is the identifier of the record (in a conventional record store) that contains attributes and values associated with the node. The pair gid and nid jointly form the primary key for the records of the B+ tree. Similarly, for a B+ tree associated with an edge of the form C-D of the basis graph, maintains records of the form (gid, nid1, nid2, rid), where gid is the graph identifier, and the record represents a C-D edge in the graph between nodes nid1 and nid2. The other B+-tree with this edge stores records of the form (gid, nid2, nid1, rid) representing the same edge in the opposite direction. Although this appears redundant, maintaining two B+ trees helps in fast retrieval and structure matching.

Algorithms are proposed over the basis graph for insertion, modification, deletion and query based on gids as well as structural similarity on the member graphs. Even though a B+ tree is an index structure, in this model, it forms part of the database storage system as well, since the member graph structures are not stored elsewhere other than in the basis graph.

STRUCTURE INDEXES FOR GRAPH DATABASES

One of the main objectives of graph databases is to answer queries on structural properties of the graph structured data. Structural queries come in various forms. In transport network databases, a structural query may be in the form of route-finder query for finding the shortest or best path between destinations. In a molecular database, a structural query may be in the form of (sub)structure similarity between the query graph and one of the member graphs. In a hyperlink or social network database, a structural query may involve finding communities in the form of near cliques, bipartite cores, etc.

Many algorithms involving graph structures are intractable. For instance, subgraph isomorphism is known to be NP-complete, while graph isomorphism is one of the few problems, which although known to be on NP, it is not known whether it belongs to P or NP-complete[9]. Given that graph databases have to contend with several member graphs, graphs of extremely large sizes and the requirement for an interactive query response time, it is unrealistic to compute exact structural match algorithms at query time.

To address this, a variety of structure indexes have been proposed. The design principle behind a structural index is to extract and index structural properties of member graphs, typically at insertion time, and use this to filter the search space rapidly in response to a query. Optionally, the filtration phase is followed by an exact matching phase in which, precise results are returned. Alternatively, some approaches view graph database queries as information retrieval queries, and resort to *ranking* the query results with some confidence measure, rather than performing exact matches. In any case, the structural index is one of the most important components of a graph database.

Structure indexes can be seen as belonging to one of different indexing paradigms. These are explained in detail below.

---

[9] Wikipedia page on Graph Isomorphism Problem. http://en.wikipedia.org/wiki/Graph_isomorphism_problem

Path-based Index Structures

A path-based structural index extracts paths from member graphs and indexes them in a global data structure. Depending on the nature of the graphs, the extracted paths may be reassembled into a single data structure like a tree, poset or another graph. Path information so extracted, is used to answer queries like shortest-paths and (sub)structure similarity.

The fingerprint table used in GraphGrep (Guigno & Shasha, 2002) can be seen as a path-based structure index. Here, label paths up to a maximum length are enumerated starting from every node in member graphs and the number of such label paths are recorded in the fingerprint table. This table is used for fast filtering of candidate graphs for structural similarity queries.

The labeled walk index (LWI) of the GRACE graph database (Srinivasa, et. al., 2005) takes a similar approach. All label walks of member graphs are enumerated and organized in the form of a tree. Since GRACE manages only undirected graphs, only the lexicographically smaller label walk is maintained. At each level l in the tree, each member graph is represented as a vector showing the number of label walks of each type of length l.

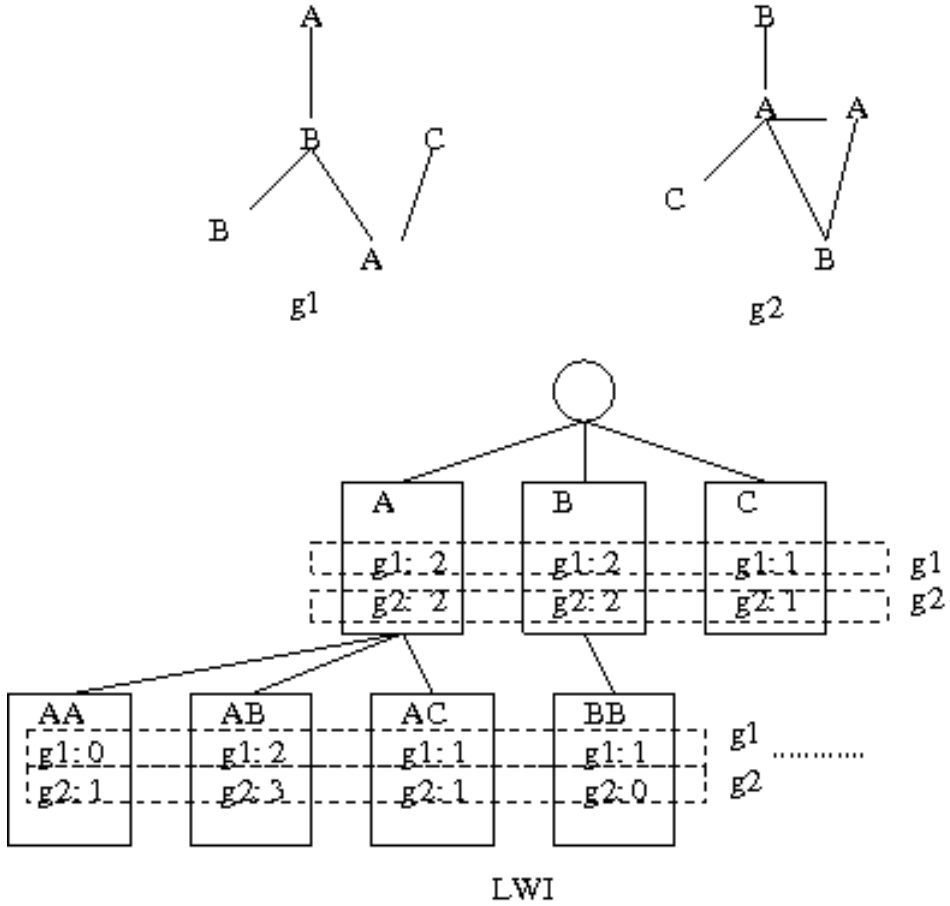


Figure 7: *The LWI tree for two labeled member graphs*

Figure 7 shows a set of two member graphs and the resultant LWI tree for the graphs. Similar to the fingerprint table of GraphGrep, LWI is used for answering structural similarity queries. For a

given query graph, walks are enumerated in steps of 1 starting with walks of length 1. A set of query results are returned based on the vector distance between the query graph and the member graphs at this level. The GRACE application then allows the user to refine results, which means that label walks are enumerated for one more level. The process of refinement of query results can continue till the maximum depth of the LWI tree.

While the above two examples were about path-based indexes for structure matching in a collection of graphs, path-based indexes have been used for finding efficiently evaluating path expressions and shortest paths in graphs where the entire database is a single graph.

An example is of edge-labeled graphs, which are directed-graph structured data where edges have labels. Several kinds of recursive queries on relational databases, like finding transitive closures of relations, can be represented as solving path-expressions over labeled graph databases. Mendelzon and Wood (Mendelzon & Wood, 1995) show that evaluating such path expressions is NP-complete for queries represented as regular expressions over path labels.

Jin, et al (Jin, et. al., 2010) address the issue of answering labeled constraint reachability (LCR) queries in a database represented as a directed, edge-labeled graph. An LCR query takes a set of edge labels A and a pair of nodes u and v, and returns true if there exists a path between u and v whose edge labels are wholly contained in A. In order to answer such queries, a poset is constructed that maintains path-label sets between all pairs of nodes. For any two elements M, N of the path-label poset $M \leq N$, iff $M \subseteq N$. The maximal elements of the poset would then contain what are termed "minimal sufficient path-label sets" between the pairs of nodes.

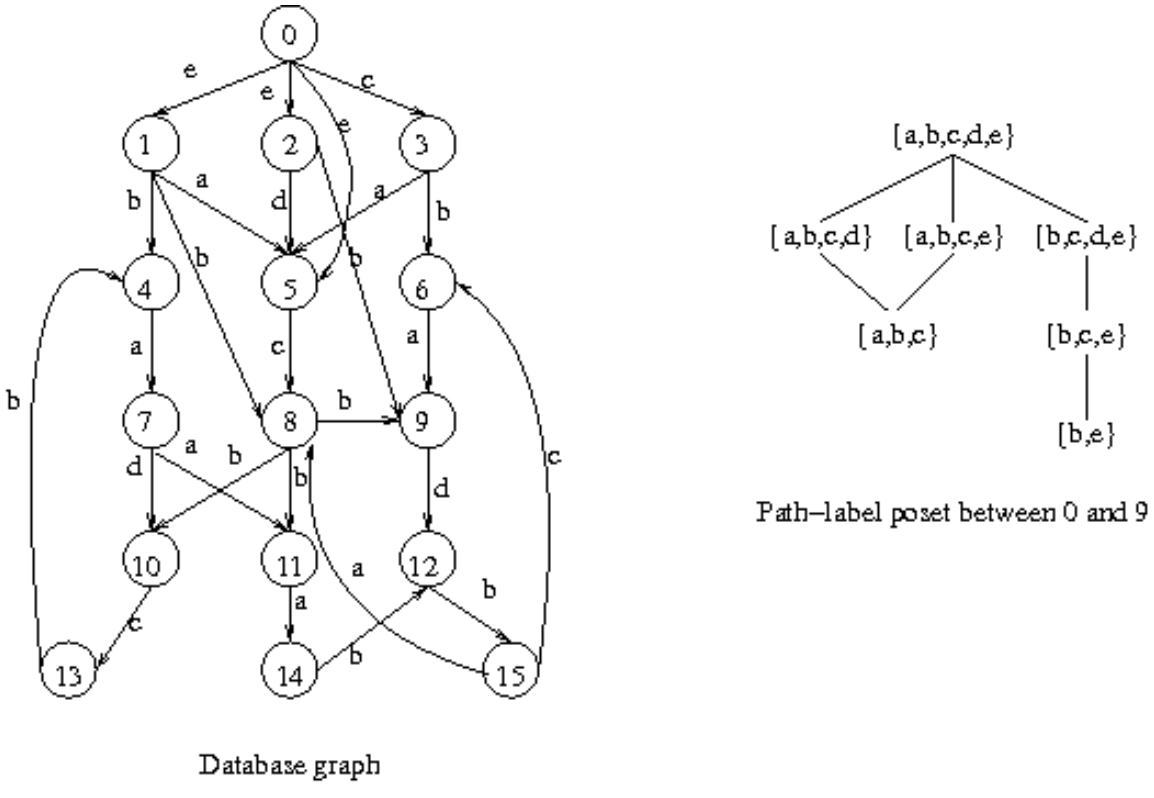


Figure 8: *A sample graph database and path-label poset between nodes 0 and 9*

Figure 8 shows a sample graph database and the path-label poset between nodes 0 and 9. The poset is constructed using a dynamic programming approach that is a generalization of the Floyd-Warshall algorithm[10] for shortest paths and transitive closures. The computational complexity of the index construction across all pairs of nodes is shown to be $O(|V|^3 2^{|\Sigma|})$, where $|V|$ is the number of nodes in the database, and $|\Sigma|$ is the number of edge labels. LCR queries are answered by traversing the path-label poset.

Path-based index structures have also been proposed in the context of XPath queries for XML databases. XML databases represent data sets that are in the form of a hierarchy and an XPath query is in the form of a path expression that returns a sequence of XML elements matching that expression. XPath index structures index the XML databases and pre-compute several relationships like ancestor sets, descendant sets, predecessor sets and successor sets of individual elements. An example of such an index is MTree (Pettovello & Fotouhi, 2006), which aims to create a native index structure for XML databases, rather than answering XPath queries by using conventional indexes like B+ trees. Other examples of path-based index structures over XML datasets include (Cooper, et. al., 2001; Min, et. al., 2005).

Path-based indexes play a central role in answering shortest-path and route-finder queries in transportation network datasets. Sanders and Schultes (Sanders & Schultes, 2007) provide a good overview of several indexing approaches to aid in route planning algorithms.

Subgraph-based Index Structures

The second model of structural index is to extract and index *subgraph structures* from the database. A common strategy here is to use *data mining* techniques to look for frequently occurring subgraphs (sometimes, frequently occurring substructures are also called motifs, in applications like computational proteomics). These motifs are then combined together in a global index structure, which is used to filter candidate graphs in response to a query. Some examples follow.

Yan et. al., (Yan, et. al., 2004) present a data structure called gIndex for maintaining frequently occurring substructures in a database of labeled, undirected graphs. Member graphs are searched in a breadth-first fashion and all *discriminative fragments* having a minimum support in the database are indexed. A frequently occurring structural fragment x is called "discriminative" if its presence cannot be predicted by the set of all frequently occurring subgraphs of x. In order to facilitate sub-graph isomorphism checks required for extracting frequent subgraphs, the graphs are first encoded into sequences using the canonical DFS encoding developed in their earlier work gSpan (Yan & Han, 2002), and that was introduced in the subsection on storage models for graph databases. Canonical encoding reduces sub-graph isomorphism checks to sequence comparisons. It also facilitates representation of the frequent subgraph index (called gIndex) as a prefix tree of all canonical sequences of all frequently occurring subgraphs.

He and Singh (He & Singh, 2006) present an index structure called "Closure Tree" for labeled, undirected collection of graphs, that is analogous to R-trees in spatial databases. A closure tree is a structure based index based on the concept of a *graph closure*. A graph closure is defined using a union function over graphs, where the union of two nodes results in a node with attributes and labels as unions of the corresponding sets of attributes and labels. The closure of two graphs is obtained by first establishing a correspondence between the nodes and edges of the two graphs.

[10] Wikipedia page on the Floyd-Warshall algorithm http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

Where nodes and edges are missing, dummy nodes and edges are created. The best correspondence between graphs is hence a mapping that minimizes the number of dummy nodes and edges. Computing the closure of two graphs gives properties similar to that of the Minimum Bounding Rectangle (MBR) computation in spatial databases. Closures are then aggregated into a closure tree, forming the structural index. A closure tree is a tree structure whose leaves represent member graphs and internal nodes represent closures. Each internal node is a closure of all its children.

Zhang et. al (Zhang, et. al, 2007), propose an approach based on indexing frequently occurring *tree* structures in a database containing a collection of labeled, undirected graphs. Tree structures are mined from a database with a level-wise edge-increasing graph mining method. The minimum support for frequent trees is a non-decreasing function that increases with the size of the trees being mined. For trees of size 1 (single edges), a minimum support of 1 is used, hence indexing all edges in the graphs. Once tree structures are mined, they are re-written in a canonical form starting from their centers. A node in a graph is said to be its "center" if it has the smallest radius (the longest of the shortest paths from the node to all other nodes). Trees are shown to have either a single center or two centers that are connected by an edge. Using this property, a canonicalization scheme is proposed where each frequent subtree is reduced into a string and stored in a conventional index structure like a trie or a B+ tree. When a query graph Q is provided, it is first partitioned into a set of feature trees that are indexed. A partitioning of a graph Q is a set of subgraphs such that they don't overlap and they collectively cover all nodes of Q. The partitions are then compared against the index structure to retrieve structurally similar member graphs.

Williams et. al (Williams, et al, 2007) present a technique for graph decomposition, where structural features can be extracted from small dense graphs with labeled nodes and edges. First, graphs are represented in a canonical form based on their adjacency matrix. Given a labeled graph G with n nodes, containing node and edge labels, its adjacency matrix M is an n x n matrix where each diagonal element captures the corresponding node label and each off-diagonal element captures the edge label. The absence of an edge is indicated by a special symbol 0. The lower triangle of M is then written in the form of a sequence by reading it row-wise. The canonical label is then the greatest lexicographic sequence thus obtained.

Next, a given graph G is decomposed into subgraphs and organized in the form of a poset. An element P of the subset represents a subgraph of G and for any pair of subgraphs P and Q, P is connected to Q with a directed edge only if P is a subgraph of Q and there is no subgraph P' such that P' is a subgraph of Q and P is a subgraph of P'.

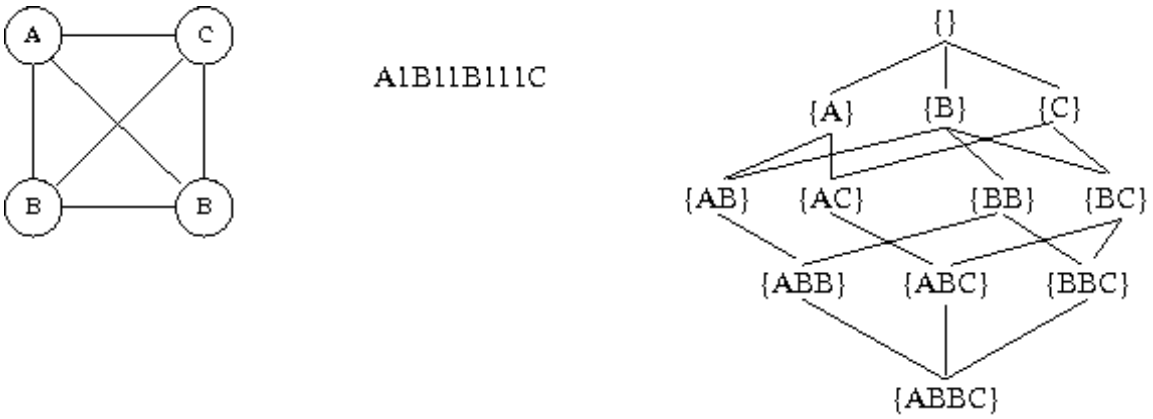


Figure 9: *An example graph, its canonical label and decomposition poset*

Figure 9 shows an example graph, its canonical label based on its adjacency matrix and a poset showing its decomposed set of subgraphs. The decomposition posets from all the member graphs are merged to create a decomposition poset for the entire database. Since all nodes in the decomposition graph are represented in the form of strings, they can be hashed using any technique for string hashing. When a query graph is given, its own decomposition matrix is created and the hash of each of the nodes of its decomposition poset are computed. If the top element of this poset matches a maximal element in the decomposition poset of the database, it means that the query graph is isomorphic to a member graph. If it matches an internal element of the database poset, then it means that the query graph is a subgraph of a member graph. Maximal common subgraphs between a given query graph and one or more of the member graphs can also be computed using the decomposition poset.

Spectral Methods for Indexing

Spectral methods for indexing graph databases employ concepts from spectral graph theory, where features of member graph(s) are represented as vectors in a hypothetical space. Optionally, multi-dimensional index structures may be used to efficiently search this space for answering structural queries.

Shokoufandeh et. al (Shokoufandeh et al, 2005) present a spectral indexing method for indexing a database of hierarchical structures represented in the form of directed acyclic graphs (DAGs). The proposed technique is aimed at managing databases of image representations, but could easily be applied to databases of DAGs in other domains as well.

Member DAGs in the database are represented in the form of adjacency matrices whose elements take on values from {-1, 0 1} representing a back edge, no edge and a forward edge respectively. Hence a DAG over *n* nodes becomes an anti-symmetric $n \times n$ matrix. Given a member graph G, its *spectrum* $\Gamma(G)$ is defined as a vector comprising of the magnitudes of all its eigenvalues placed in non-ascending order.

The spectrum hence becomes the feature vector of a member graph. The spectrum of a DAG can also be used to match graphs with distortions, where distortions represent a set of edits (addition, deletion and modification of nodes and edges) from the original graph. If $A^{m \times m}$ represents a DAG, and if it is distorted to another DAG of the form $H^{n \times n}$ by the addition of n-m new nodes,

the transformation $\psi : A \rightarrow$ H is called the *lifting operator*. A lifting operator is said to be spectrum preserving, if A and $\psi(A)$ have the same set of non-zero eigenvalues. The authors also show that any arbitrary lifting operator can be represented as the sum of a spectrum preserving lifting operator and the addition of a noise matrix.

The spectrum of member graphs in the database are thus computed and stored in an index structure representing a multi-dimensional space. Similarity match is computed by performing a k nearest neighbor (kNN) search in the index space.

Zou et. al (Zou, et al, 2008) propose a spectral method for graph indexing based on computing the eigen values of member graphs. Given a simple, undirected graph G, a theorem called the Interlacing Theorem in Ggraph Theory, relates the eigen values of G and any *induced subgraph* H of G. An induced subgraph of a graph G is made of a subset of all the vertices of G and any edges among them. Given a graph G with n eigen values $\lambda_1 \ldots \lambda_n$ sorted in descending order and an induced subgraph H of G with m eigen values $\beta_1 \ldots \beta_n$ sorted in descending order, the interlacing theorem states that $\lambda_{n-m+i} \leq \beta_i \leq \lambda_i$ for $i = 1 \ldots m$.

However, any arbitrary subgraph of a given graph G, may not correspond to the induced subgraph having the same set of nodes from G. This is because an induced subgraph is expected to have all edges between its nodes that are present in the original graph G. Hence the interlacing theorem cannot be directly used for subgraph isomorphism checks. In order to overcome this, the authors propose a concept of spectral *vertex topology signature* for every vertex in a member graph.

The vertex topology signature of any node *v*, is the set of top *t* eigen values for the level-*n* path tree starting from *v*. Here *n* and *t* are configurable parameters that trade off between accuracy and the speed of computation. The vertex signature trees have the property that if graph Q is a subgraph of another graph G, then the corresponding vertex path trees of Q would be induced subgraphs of the path trees of G.

CONCLUSION

The objective of this chapter was to provide an overview of the main challenges in designing graph databases. To this end, three specific challenges were chosen – data models, storage models and index structures. Some representative approaches from the literature were also introduced for each of the above challenges.

As is apparent, graph data management has attracted immense research attention, but has still eluded strong foundations or an overarching best practice or design principles. Research and commercial interest in graph databases is still strong as of this writing. One of the first international workshops dedicated to graph databases was held on July 15th 2010[11] in China. Similarly, neo4j which claims to be the first industry-grade, commercial graph database was released in 2009.

As the amount of data grows and the need for identifying patterns and extracting semantics becomes stronger, graph data management becomes an imminent challenge. Much of the research efforts introduced in this chapter go a long way in addressing such challenges.

[11] First International Workshop on Graph Databases (WGD 2010) home page http://www.icst.pku.edu.cn/IWGD2010/index.html

REFERENCES

Abiteboul, S. (1997). Querying Semi-Structured Data. *Proceedings of the 6th International Conference on Database Theory*, pages 1-18.

Abiteboul, S. & Vianu, V. (1997). Regular path queries with constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Tucson, Arizona, United States, May 11 - 15, 1997). PODS '97. ACM, New York, NY, 122-133.

Angles, R., Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*. 40(1), pages 1—39.

Bertino, E., Elmagarmid, A.K., Hacid, M-S. (2003). Ordering and Path Constraints over Semistructured Data. *Journal of Intelligent Information Systems*, 20(2), pages 181-206.

Bonnici, V., Giugno, R., Pulvirenti, A., Ferro, A., Shasha, D. (2010). RelaxGrep: Approximate Graph Searching by Query Relaxation. Supplementary Proceedings of Pattern Recognition in Bioinformatics.5th IAPR International Conference, PRIB 2010, Nijmegen, The Netherlands.

Borgelt, C. & Berthold, M. R. (2002). Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In *Proceedings of the 2002 IEEE international Conference on Data Mining* (December 09 - 12, 2002). ICDM. IEEE Computer Society, Washington, DC, 51.

Brandes, U., Erlebach, T. (Eds.) (2005). *Network Analysis: Methodological Foundations* [outcome of a Dagstuhl seminar, 13-16 April 2004]. Lecture Notes in Computer Science, 3418 Springer 2005, ISBN 3-540-24979-6.

Buneman, P., Fan, W. & Weinstein, S. (1998). Path constraints on semi-structured and structured data. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, United States, June 01 - 04, 1998). PODS '98. ACM, New York, NY, 129-138.

Chandola, V., Banerjee, A., & Vipin Kumar. (2009). Anomaly Detection : A Survey. *ACM Computing Surveys*. 41(3).

Chang, N.S. & Fu, K.S. (1980). A relational database system for images. Pictorial Information Systems. Lecture Notes in Computer Science 80, Springer.

Codd, E., F. (1983). A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM,* 26(1) 64–69.

Colinge J., Bennett K., L. (2007). Introduction to Computational Proteomics. *PLoS Computational Biolology,* 3(7), e114. doi:10.1371/journal.pcbi.0030114.

Consens, M. & Mendelzon, A. (1993). Hy+: a Hygraph-based query and visualization system. *SIGMOD Record*, 22(2), pages 511–516.

Cooper, B., Sample, N., Franklin, M. J., Hjaltason, G. R. & Shadmon, M. (2001). A Fast Index for Semistructured Data. In *Proceedings of the 27th international Conference on Very Large Data Bases* (September 11 - 14, 2001). P. M. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 341-350.

Cormen, T.H., Leiserson, C.E, Rivest, R.L. & Stein, C. (2001). *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.

Gemis, M., Paredaens, J., Thyssens, I., van den Bussche, J. (1993). GOOD: A Graph-Oriented Object database System. *ACM SIGMOD Record*, 22, pages 505—510.

Graves, M., Bergeman, E. R., & Lawrence, C. B. (1995). A Graph-Theoretic Data Model for Genome Mapping Databases. In *28th Annual Hawaii International Conference on System Sciences (HICCS'95)*.

Giugno, R. & Shasha, D. (2002). GraphGrep: A Fast and Universal Method for Querying Graphs. *Proceedings of the International Conference in Pattern recognition (ICPR)*, Quebec, Canada.

Güting, R., H. (1994). GraphDB: Modeling and Querying Graphs in Databases. Proceedings of the 20th International Conference on Very Large Databases (VLDB '94), pages 297—308.

He, H. & Singh, A. (2006). Closure-Tree : An Index Structure for Graph Queries. Proceedings of the 22nd International Conference on Data Engineering, (ICDE '06), IEEE Computer Society Press.

Hidders, J. & Paredaens, J. (1993). GOAL: A Graph-Based Object and Association Language. *Advances in Database Systems: Implementations and Applications, CISM*, pages 247–265.

Jonyer, I., Holder, L.B. & Cook, D.J. (2000) Graph-Based Hierarchical Conceptual Clustering in Structural Databases'. In the *Proceedings of the Seventeenth National Conference on Artificial Intelligence*.

Jin, R., Hong, H., Wang, H., Ruan, N. & Xiang, Y. (2010). Computing label-constraint reachability in graph databases. In *Proceedings of the 2010 International Conference on Management of Data* (Indianapolis, Indiana, USA, June 06 - 10, 2010). SIGMOD '10. ACM, New York, NY, 123-134.

Kim, D.H., Yun, I.D. & Lee, S.U. (2010). Attributed relational graph matching based on the nested assignment structure. *Pattern Recognition.* 43(3), pages 914-928

Kim, W. (1990). *Introduction to Object-Oriented Databases.* The MIT Press, 1990. ISBN 0-262-11124-1.

Kobayashi, K. (1994). On the Spatial Graph. *Kodai Math. Journal*. 17(3), pages 511-517.

Kumar, S., & Srinivasa, S. (2003). A Database for Storage and Fast Retrieval of Structure Data: A Demonstration. In Proceedings of the 19th International Conference on Data Engineering (ICDE'03).

Kuper, G. M. & Vardi, M. Y. (1984). A new approach to database logic. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Waterloo, Ontario, Canada, April 02 - 04, 1984). PODS '84. ACM, New York, NY, pages 86-96.

Levene, M. & Loizou, G. (1995). A Graph-Based Data Model and its Ramifications. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(5) pages 809–823.

Levene, M. & Poulovassilis, A. (1990). The Hypernode Model and its Associated Query Language. *In Proc. of the 5th Jerusalem Conference on Information Technology*, pages 520–530. IEEE Computer Society Press.

Levene, M. & Poulovassilis, A. (1991). An Object-Oriented Data Model Formalised Through Hypergraphs. *Data & Knowledge Engineering (DKE)*, 6(3), pages 205–224.

Mylopoulos, J., Borgida, A., Jarke, M. & Koubarakis, M. (1990). Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems (TOIS)*, 8(4), pages 325 —362.

Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M. & Larriba-Pey, J. (2007). Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the Sixteenth ACM Conference on Conference on information and Knowledge Management* (Lisbon, Portugal, November 06 - 10, 2007). CIKM '07. ACM, New York, NY, 573-582.

Mendelzon, A.O. & Wood, P.T. (1995). Finding Regular Simple Paths in Graph Databases**.** *SIAM Journal of Computing,* 24(6).

Min, J., Chung, C. & Shim, K. (2005). An adaptive path index for XML data using the query workload. *Inf. Syst.* 30, 6 (Sep. 2005), 467-487.

Pettovello, P. M. & Fotouhi, F. (2006). MTree: an XML XPath graph index. In *Proceedings of the 2006 ACM Symposium on Applied Computing* (Dijon, France, April 23 - 27, 2006). SAC '06. ACM, New York, NY, 474-481.

Sakr, S., Al-Naymat, G. (2010). Graph Indexing and Querying: A Review. *International Journal of Web Information Systems,* 6(2), pp. 101 – 120

Sanders, P. & Schultes, D. (2007). Engineering Fast Route Planning Algorithms. In *6th Workshop on Experimental Algorithms (WEA)*, Springer LNCS 4525, pp. 23-36.

Sanfeliu, A. & Fu, K.S. (1983). A Distance Measure Between Attributed Relational Graphs for Pattern Recognition. IEEE Trans. Systems, Man, and Cybernetics. 13, pages 353-363.

Shasha, D., Wang, J.T-L. & Giugno, R. (2002). Algorithmics and Applications of Tree and Graph Searching. *In Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, Madison, Wisconsin.

Singh, M.H. & Srinivasa, S. (2006). BasisGraph: Combining Storage and Structure Index for Similarity Search in Graph Databases. Proceedings of the 13th International Conference on Management of Data (COMAD '06), New Delhi, India.

Shokoufandeh, A., Macrini, D., Dickinson, S., Siddiqi, K., Zucker, S.W. (2005). Indexing Hierarchical Structures using Graph Spectra. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 27(7), pp.1125-1140.

Srinivasa, S., Acharya, S., Agrawal, H. & Khare, R. (2002). Vectorization of Structure to Index Graph Databases. *Proceedings of IASTED Int'l Conf. on Information Systems and Databases (ISDB'02)*, Acta Press, Tokyo, Japan.

Srinivasa, S., Maier, M. & Mutalikdesai, M. (2005). LWI and Safari: A New Index Structure and Query Model for Graph Databases, *COMAD 2005*, Goa, India.

Srinivasa, S. & Singh, M.H. (2005). GRACE: A Graph Database System. *COMAD 2005b*, Hyderabad, India.

Taylor, R., W. & Frank, R., L. (1976). CODASYL Data-base Management Systems. *ACM Computing Surveys,* 8(1) 67–103.

Topaloglou, T. (1993). Storage management for knowledge bases. In *Proceedings of the Second international Conference on information and Knowledge Management* (Washington, D.C., United States, November 01 - 05, 1993). B. Bhargava, T. Finin, and Y. Yesha, Eds. CIKM '93. ACM, New York, NY, 95-104.

Tian, Y. (2008). *Querying Graph databases*. PhD Thesis, The University of Michigan.

Tsai, W.H. & Fu, K.S. (1979). Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE Transactions on Systems, Man and Cybernetics*, 9, pages 757-768.

Williams, D., Huan, J. & Wang, W. (2007). Graph Database Indexing Using Structured Graph Decomposition. *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE)*.

Yan, X. & Han, J. (2002). gSpan: Graph-Based Substructure Pattern Mining. *Proceedings of ICDM 2002*, pages 721-724.

Yan, X., Yu, P. S. & Han, J. (2004). Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international Conference on Management of Data* (Paris, France, June 13 - 18, 2004). SIGMOD '04. ACM, New York, NY, 335-346.

Zhang, S., Hu, M. & Yang, J. (2007). TreePi: A Novel Graph Indexing Method. *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 966-975.

Zou, L., Chen, L., Yu, J.X. and Lu, Y. (2008) A novel spectral coding in a large graph database. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology (EDBT '08),* ACM, New York, NY, USA, 181-192.

Keywords: Graph database, Data Model, Storage Model, Structural Index, Substructure Matching, Similarity Matching, Graph Encoding, Hypernodes, Hypergraphs