A Query Language for Analyzing Networks

Anton Dries Siegfried Nijssen Luc De Raedt

K.U.Leuven, Celestijnenlaan 200A, Leuven, Belgium
{anton.dries,siegfried.nijssen,luc.deraedt}@cs.kuleuven.be

ABSTRACT

With more and more large networks becoming available, mining and querying such networks are increasingly important tasks which are not being supported by database models and querying languages. This paper wants to alleviate this situation by proposing a data model and a query language for facilitating the analysis of networks. Key features include support for executing external tools on the networks, flexible contexts on the network each resulting in a different graph, primitives for querying subgraphs (including paths) and transforming graphs. The data model provides for a closure property, in which the output of every query can be stored in the database and used for further querying.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data Models*

General Terms

Design, Theory

Keywords

Graph databases, Inductive databases, Data mining

1. INTRODUCTION

In many applications it is increasingly common to represent data as a large graph or network: examples include social networks, bibliographic networks and biological networks. Modeling the data as graphs is convenient in these applications as traditional graph-based concepts, such as paths, cliques, node degrees, edge degrees, and so on, are useful in their analysis. An increasing number of tools have been developed that operate on graphs, such as algorithms for finding cliques [21], important nodes [4, 18], important connections [18], or methods for classifying [18] or clustering nodes [12]. The main challenge in these applications is often to analyze the network in order to discover new knowledge and use that knowledge to improve the network.

Discovering new knowledge in databases (also known as KDD) typically involves a process in which multiple operations are repeatedly performed on the data, and as new insights are gained, the data is being transformed and extended. As one example consider a bibliographical network with authors, papers, and citations such as that gathered by Citeseer or Google Scholar. Important problems in such bibliographical networks include: entity resolution [3], which is concerned with detecting which nodes (authors or papers) refer to the same entity, and collective classification [18], where the task is to categorize papers according to their subject. This type of analysis typically requires one to call multiple tools and to perform a lot of transformations on the data. The key contribution of this paper is a new data model and accompanying query language that is meant to support such processes. This is in line with the work on *inductive databases*, which provide an integration of databases and data mining. Support for data mining can be realized by extending database languages such as SQL with primitives that integrate data mining deeply in the query language, cf. M-SQL [10], MineRule [16], DMQL1 [7] and SiQL [20], or by support for executing algorithms more as black boxes, cf. SINDBAD [20] or Oracle's Data Mining Extensions. All these inductive database approaches are extensions of the relational model (and often also of SQL). To the best of our knowledge, there are no inductive database languages yet that specifically target network data. It is also clear that for networked data directly applying the relational model is not an option. The reason is that the relational model does not support any data structures or operations related to graphs (such as subgraph matching or dealing with paths). In addition, the existing inductive database extensions typically only operate on a single table in a relational database and it is unclear how to apply this to networked data.

To deal with graphs, a number of graph databases, with corresponding query languages, have already been proposed [2]. These graph databases, however, lack some of the functionality that is needed in an inductive database. Before being able to apply a data mining algorithm on some data, the data must be in the correct format and must contain the relevant descriptors. Therefore, it is essential that an inductive database supports pre-processing of the data it contains and also accommodates multiple views on the same database, allowing to treat the database different in one context than in another. Our database model and language naturally support this requirement by employing a *uniform*

CIKM'09, November 2–6, 2009, Hong Kong, China.

representation of nodes and edges, which allows one to easily define different contexts on the network. Each context corresponds to a particular subnetwork. Using our uniform representation it becomes easy to, for instance, swap the roles of the nodes and the edges or realize other kinds of transformations. Our model also supports the *closure property* as it is essential that the user be able to execute a tool on the output of another tool. Therefore, each query starts from a network and produces an output that can be considered part of a network as well. The database is also not focused on one particular type of graph, but is able to work with directed, undirected, weighted or unweighted graphs, or hypergraphs and to easily switch between these models.

To deal with these challenges, we contribute a novel data model in which the database can be considered one network, that is, a large graph. We propose a query language, called BiQL, based on the well-known SQL query language that can be used to perform basic operations on the network to support the manipulation and transformation of the network. The language that we introduce provides the ability to call external data mining tools in a way similar to commercial systems (for instance, the Data Mining Extensions of Oracle) or research systems (for instance, SINDBAD [20]). It does not incorporate data mining primitives more deeply as it is still unclear what these primitives should look like in general, even in the case of a dataset consisting of a single table in a relational database: for instance, MineRule [16] and DMQL1 [7] focus on a small number of data mining operations.

The outline of this paper is as follows. In Section 2 we provide a set of queries that illustrate common queries in a data analysis setting. In Section 3 we summarize the type of functionality that is required to support these queries and the functionality required for supporting data mining. We provide a summary of a literature study which shows that existing graph databases are lacking. In Section 4 we propose our data model, which includes a structural part, an integrity part and a basic manipulation part. We study the operational semantics and relationships of our model to the relational model in Section 5. Extensions of the query language are discussed in Section 6. In Section 7 we show that we can indeed express the queries we wish to support. In Section 8 we conclude.

2. EXAMPLE QUERIES

Typical applications in network analysis include bibliographical and biological networks. In bibliographical networks one typically has entities such as `Author`s, `Paper`s and possibly `Venue`s. Furthermore, there are relationships such as an author having *authored* a paper, a paper *citing* another paper, and a paper being *published* in a venue. Standard examples of databases in this domain include Citeseer, DBLP and Google Scholar. Similarly, in biological networks, there are entities such as `Protein`s, `Tissue`s and `Gene`s; relationships of interest include proteins being *expressed* in tissues, genes *coding* for proteins, and proteins *interacting* with other proteins. The mining of such networks (and many others, such as social networks and the Internet) is a challenging and important task. It also imposes new demands on the underlying database model, which are well illustrated by the following queries and operations, set in the context of bibliographic datasets.

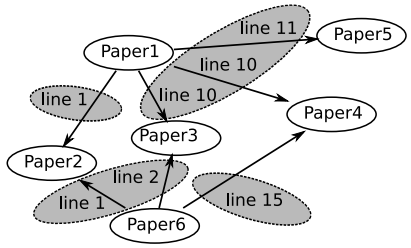


Figure 1: An incomplete citation graph, where the locations in papers of citations are indicated; gray circles indicate clusters of citations we wish to find.

Example 1. *Can we define a co-authorship relation and a citation relation, which states which authors cite each other?*

The main feature of this query is that it adds additional edges in the graph.

Example 2. *Can we find occurrences of the following pattern: author X cites author Y, which cites author X, possibly indirectly?*

In this query we are provided with a graph pattern, which we wish to match with the data.

Example 3. *Assume given a tool for computing quasi-cliques in a graph (for instance [21]). Can we find quasi cliques in the co-authorship graph?*

This query applies an external tool on a graph extracted from the database.

Example 4. *Can we find the influence graph of a paper, that is, the set of papers that have been influenced by a paper, that is, the recursive closure of papers referencing the paper?*

In this query we are provided with a complicated graph pattern, which we wish to use to group related nodes together.

Example 5. *Consider the following problem. In each paper, multiple other papers are cited; each citation occurs at a certain position (for instance, one paper is cited in the introduction, while another is cited in the conclusions). It is reasonable to assume that papers which are cited close to each other, are related to each other. Can we cluster the citations in every paper?*

An example is given in Figure 1. One possible way to answer this query is by using a graph clustering algorithm [12], which clusters nodes in a graph. In this case we would like to cluster the citations of every paper, and hence, we need to treat the citations as nodes. To deal with this query we need a uniform representation of nodes and edges, as this provides the flexibility to specify what are considered to be edges, and what are considered to be nodes.

3. REQUIREMENTS & RELATED WORK

3.1 Requirements and Design Choices

The main motivation and target application for our data model and query language is supporting exploratory data analysis on networked data. This results in the following requirements and design choices.

Small is beautiful.

In line with the relational database model, we believe that the data model should have the smallest possible number of concepts and primitives necessary for the representation and manipulation of the data. This property has to a large extent facilitated the development of the theory and the implementations of relational databases. This is a primary design principle that we used throughout the development of our data model.

As a consequence, we do not wish to introduce special language constructs to deal with complicated types of networks (directed, undirected, labeled, hypergraphs, etc.) or sets of graphs; we do not wish to treat attributes of edges differently than attributes of nodes. When introducing a novel type of graph, it should not be necessary to extend the basic data types.

Uniform representation of nodes and edges.

The most immediate consequence of the former choice is that we wish edges and nodes to be represented in a uniform way. We will do this by representing both edges and nodes as objects that are linked together by links that have no specific semantics. A uniform representation however does not only avoid having to introduce special syntax for different types of graphs or attributes, it also allows one to easily generate different views on a network. For instance, in a bibliographic database, we may have objects such as papers, authors and citations. In one context one could analyze the co-author relationship, in which case the authors are viewed as nodes and the papers as edges, while in another context, one could be more interested in citation-analysis, in which case the papers are the nodes and the citations the edges. Another example in which this interchangeability is important is given in Example 5 of the previous paragraph. The power of the uniform representation is hence also that it enables us to keep the underlying low-level network the same; only the interpretation of this network differs.

Flexible Contexts.

In order to support the knowledge discovery process, the user must be given the ability to extract different parts of the network and to transform them into the particular data format or graph model that the external data mining algorithms needs. This explains why we introduce the notion of contexts and classes. Contexts define the subgraphs of interest that can be used by, for example, graph mining algorithms or visualization tools. Contexts allow the user to extract information and present it into, for example, a traditional edge-labeled graph by selecting which objects should act as edges and which ones as nodes. Class definitions can also be used to create hypergraphs, sets of graphs, directed or undirected as well as other representations of the network. Using contexts is beneficial because it allows us to combine a general, application-independent underlying data structure with the most natural, possibly specific graph representations required for each application individually.

External calls.

While in, for example, relational databases there exists a set of core primitives in which all database operations of interest can be expressed, such a set of primitives is not yet available for data mining. This is why we have chosen to incorporate data mining by providing support for calling external procedures. The inputs to these external tools can be any type of network that, in principle, can be passed on as a context to the external tool.

Closure property.

Not only the inputs to data analysis algorithms matter but also the outputs. Therefore, it is essential that the result of any operation, query or external call can be used as the starting point for further queries and operations. This can be realized by enforcing a closure property which states that the result of any operation or query generates information that can be added to the database network (either permanently or temporarily). The information created by the query combined with the original database can therefore be queried again.

This closure property can be combined with contexts to provide integration with data mining tools. We can for example transform part of our network into a set of graphs in which we want to look for frequent patterns [11]. The results of the frequent pattern miner can then be added to the database network and used in subsequent queries. By integrating existing tools for graph mining and data mining the system becomes a powerful inductive database for information networks.

SQL-based.

To represent queries, a language is needed. There are many possible languages that could be taken as starting point, such as SQL, relational algebra or Datalog. Our approach is similar to that of the relational model: we aimed for a data model on which multiple equivalent ways to represent queries can be envisioned. Therefore, we employ an SQL-like language, but we will also show how to represent queries in a small extension of Datalog. In the end, this similarity to the relational model, both in the choice for a data model with a small number of types, and in the range of query languages feasible on top of it, should make the model more convenient for the many users familiar with the relational model.

Semi-structured data.

Data and information in many networks of interest comes from a wide variety of sources and is often heterogeneous in nature. Hence, it is impractical to require that the user formally defines the database schema describing the structure of the database. Our model therefore does not impose this requirement but is based instead on a semi-structured data model that supports working with heterogeneous data in a flexible way.

3.2 Related work

A number of query languages for graph databases have been proposed, many of which have been described in a recent survey [2]. However, none of these languages was designed for supporting knowledge discovery processes and each language satisfies at most a few of the above mentioned properties. For instance, GraphDB [5] and GOQL [19] are based on an object-oriented approach, with provisions for specific types of objects for use in networks such as nodes, edges and paths. This corresponds to a more structured data model that does not uniformly represent nodes and edges. In addition, these languages target other applications: GraphDB has a strong focus on representing spatially embedded networks such as highway systems or power lines, while GOQL [19], which extends the Object Query Language

(OQL), is meant for querying and traversing paths in small multimedia presentation graphs. Both languages devote a lot of attention to querying and manipulating paths: for example, GraphDB supports regular expressions and path rewriting operations.

GraphQL [8] provides a query language that is based on formal languages for strings. It provides an easy, yet powerful way of specifying graph patterns based on graph structure and node and edge attributes. In this model graphs are the basic unit and graph specific optimizations for graph structure queries are proposed. The main objective of this language is to be general and to work well on both large sets of small graphs as well as small sets of large graphs. However, extending existing graphs is not possible in this language; flexible contexts are not supported.

PQL [13] is an SQL-based query language focussed on dealing with querying biological pathway data. It is mainly focussed on finding paths in these graphs and it provides a special path expression syntax to this end. The expressivity of this language is, however, limited and it has no support for complex graph operations.

GOOD [6] was one of the first systems that used graphs as its underlying representation. Its main focus was on the development of a database system that could be used in a graphical interface. To this end it defines a graphical transformation language, which provides limited support for graph pattern queries. This system forms the basis of a large group of other graph-oriented object data models such as Gram [1] and GDM [9].

Hypernode [14] uses a representation based on hypernodes, which make it possible to embed graphs as nodes in other graphs. This recursive nature makes them very well suited for representing arbitrarily complex objects, for example as underlying structure of an object database. However, the data model is significantly different from a traditional network structure, which makes it less suitable for modeling information networks as encountered in data mining.

A similar, but slightly less powerful representation based on hypergraphs is used in GROOVY [15]. This system is primarily intended as an object-oriented data model using hypergraphs as its formal model. It has no support for graph specific queries and operations.

More recently, approaches based on XML and RDF such as SPARQL [17], are being developed. They use a semi-structured data model to query graph networks in heterogenous web environments; support for creating new nodes and flexible contexts is not provided.

While most of the systems discussed here use a graph-based data model and are capable of representing complex forms of information, none of them uses a uniform representation of edges and nodes (and its resulting flexible contexts), nor supports integration of KDD tools.

4. DATA MODEL

Our data model consists of several parts: (1) the *structural part* of the data model; (2) the *manipulation part* of the data model; (3) the *integrity part* of the data model.

4.1 Data Structures

The main choice we have to make in our data model is based on reconciling two requirements:

- representing a large number of graph and edge types;
- supporting graph theoretic concepts, such as paths or subgraphs.

Addressing these requirements, the data structure that we propose consists of the following components.

The object store, which contains all objects in a database. Objects are uniquely identified by an object identifier. Each object can contain an arbitrary list of attribute-value pairs describing its features.

The link store, which contains directed links between objects. They can be represented as (ordered) pairs of object identifiers, and do not have any attributes.

The domain store, which contains named sets of objects. A domain name allows users to identify a set of objects.

The context store, which contains named sets of domain names. A context name allows users to identify a set of domains in a query. Each domain in a context is given a name within that context. Hence, each context consists of $\lambda_1 \mapsto \lambda_2$ pairs, where λ_2 is a name occurring in the domain store, and λ_1 is the name given to this domain within the context. Optionally a context is assigned to a *class* (see Section 4.3).

The main design choice in this data structure is not to allow attributes on links. This ensures that within our data model links are very light and implicit. Between every pair of objects a link may or may not exist, but we do not specify how links are stored. In our query language, a basic operation is to check if a link exists between two objects.

Domains are used to group nodes of a certain type together, such as `authors` or `papers` in our example. Whereas nodes do not have names that can be used in the query system, domains have names that can be used. Domains are grouped in contexts; domains can be in an arbitrary number of contexts.

One may think of the objects as nodes in a graph, and of the links as unlabeled binary edges between these nodes. However, this raises the question how we represent edge labeled graphs or hypergraphs. This is clarified in the following example.

EXAMPLE 6 (EDGE LABELED GRAPH). *Assume given the objects and links in Table 1, belonging to domains X and Y, together constituting the context G. Then we can visualize context G as given in Figure 2. In this example, one may think of nodes* A, ..., E *as authors, and as the edges expressing strengths of co-authorships.*

Hence, the main choice that we have made is, in a sense, that also edges are represented as objects. An edge object is linked to the nodes it connects. Even though this may not seem intuitive, or could seem a bloated representation, the advantages of this choice outweigh the disadvantages because:

- by treating both edges and nodes as objects, we obtain simplicity and uniformity in dealing with attributes;
- it is straightforward to treat (hyper)edges as nodes (or nodes as (hyper)edges);
- it is straightforward to link two edges, for instance, when one wishes to express a similarity relationship between two edges.

obj-id	features
X1	{label=A, color=red}
X2	{label=B, color=yellow}
X3	{label=C, color=blue}
X4	{label=D, color=yellow}
X5	{label=E, color=red}
Y1	{weight=0.2}
Y2	{weight=0.5}
Y3	{weight=0.8}
Y4	{weight=0.2}
Y5	{weight=0.9}

(a) Object store

Y1	↔	X1
Y1	↔	X2
Y2	↔	X1
Y2	↔	X3
Y3	↔	X2
Y3	↔	X3
Y4	↔	X3
Y4	↔	X4
Y5	↔	X3
Y5	↔	X5

(b) Link store

name	objects
X	{ X1, X2, X3, X4, X5 }
Y	{ Y1, Y2, Y3, Y4, Y5 }

(c) Domain store

name	domains	type
G	{ Nodes ↦ X, Edges ↦ Y}	labeled_graph

(d) Context store

Table 1: Example database

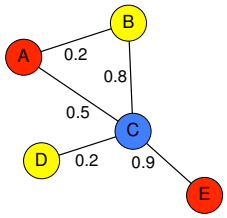


Figure 2: A visualization of context G in the example database in Table 1, where we use domain X as nodes and domain Y as edges.

This flexibility is further illustrated in the following example.

Example 7 (Edge Labeled Graph Set). *Assume given in addition to the objects and links in Table 1 the objects and links in Table 2, which are part of domain Z and are used to define context S. Then we can visualize this context S as given in Figure 3. In this example, both Z1 and Z2 can be thought of as identifying* subgraphs *of graph G in Figure 2.*

4.2 Data Manipulation

Now that we have a basic understanding of how the data is organized in the database, we can focus on querying this information. In this paragraph we introduce the main components of the BiQL query language, which allows for querying using an SQL-like notation; we will show the relationship of BiQL to the traditional relational model, including a more detailed discussion of its operational semantics, in Section 5.

To store the result of a query as a set of domains, we use the `CREATE` statement preceding the query. In general a query looks like this.

```
CREATE <names of new domains> AS
SELECT <definitions of domains>
FROM <selection from domains>
WHERE <predicate on attributes of objects>
```

obj-id	features
Z1	{name=part1}
Z2	{name=part2}

(a) Object store

name	objects
Z	{ Z1, Z2 }

(b) Domain store

Z1	→	X1	Z2	→	X2
Z1	→	X2	Z2	→	X3
Z1	→	X3	Z2	→	X4
Z1	→	Y1	Z2	→	X5
Z1	→	Y2	Z2	→	Y3
Z1	→	Y3	Z2	→	Y4
			Z2	→	Y5

(c) Link store

name	domains	type
S	{ Nodes ↦ X, Edges ↦ Y, Parts ↦ Z}	labeled_graphset

(d) Context store

Table 2: Example database; only elements additional to Table 1 are listed.

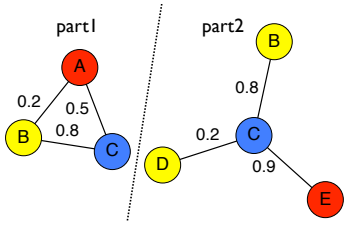


Figure 3: The example database in Table 2 conceived as graph set.

A query can be used to define multiple new domains at once by listing multiple names and definitions in the `CREATE` and `SELECT` statements, respectively. However in this paper we will only focus on the basic case in which a query only creates a single domain.

A simple example of such a query is this:

```
CREATE Y' AS
SELECT E
FROM Y E
WHERE E.weight > 0.4
```

This statement creates a new domain `Y'`; the objects that are inserted in this domain are obtained by letting a variable `E` range over the objects in domain `Y`; those objects which have a `weight` attribute with a value higher than 0.4 are inserted.

For Y defined as in Table 1, the resulting domain contains the following set of identifiers.

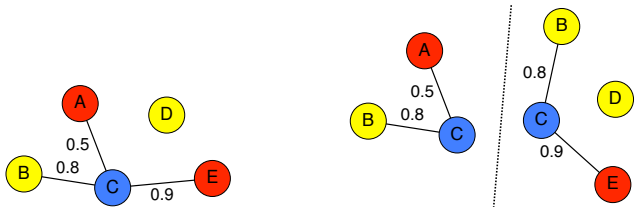
$$Y' = \{Y2, Y3, Y5\}$$

We can define a new context using this domain, using the following statement.

```
CREATE G' AS INSTANCE labeled_graph
WITH X as Nodes, Y' as Edges
```

Figure 4 illustrates this for the graphs from Figures 2 and 3.

Below we provide more extensive details for the `FROM`, `WHERE` and `SELECT` parts of a query.



(a) `labeled_graph WITH X as Nodes, Y' as Edges`

(b) `labeled_graphset WITH Z as Parts, X as Nodes, Y' as Edges`

Figure 4: View on the database using the new table Y'.

4.2.1 `WHERE` *statement*

In the `WHERE` statement constraints are expressed based on the features of the objects that the variables refer to. Our language supports all basic constraints that SQL supports. These are

- comparisons of attributes of an object with a constant;
- comparisons of attributes of two objects;
- compositions of basic atoms into formulas using ANDs (",") and ORs.

There is, however, an important difference. As the data is semi-structured, we do not have a fixed data schema. It is hence possible that the value of an attribute is used that an object does not possess. By default, we assume that such an atom always evaluates to false. Furthermore, if the object possesses multiple values for the same attribute, only one must satisfy the constraint.

4.2.2 `FROM` *statement*

The most important part of the query is the `FROM` statement. The main difference in our query language, compared to the the `FROM` statement in SQL, is that we use *path expressions* in the `FROM` statement to specify structural constraints on the variables.

The general form of the simple `FROM` statement is as follows.

```
<from_statement> ::= FROM <path_defs>
<path_defs>      ::= <path_def> "," <path_defs> |
                     <path_def>
<path_def>       ::= DISTINCT(<path_defs>) |
                     <path_statement>
<path_statement> ::= <obj_statement>
                     { ( "--" | "->" | "<-" |
                          "-/" | "-/>" | "</-" )
                       <obj_statement> }
<obj_statement>  ::= <domain> [ <variable> ]
```

A simple example of a query, which can be applied on the data in Table 1, is this:

```
CREATE Z'
SELECT P
FROM Z P -- Y E
WHERE E.weight >= 0.9
```

This statement creates a new domain Z'; those objects from domain Z are inserted that are linked to objects in domain Y with a weight higher than 0.9. In the example of Figure 3 we insert only Z2 in domain Z'; effectively, we insert only those subgraphs which have an edge with weight at least 0.9.

In general the `FROM` statement consists of a list of path expressions. A path expression consists of a series of domain identifiers separated by the

`--` or `->` or `<-` or `-/-` or `-/>` or `</-`

operators, which represent the presence of links in any direction, the presence of links in a given direction, the absence of links in any direction, or the absence of links in a given direction, respectively.

We can think of a path expression as a path in which the nodes are labeled with domains identifiers. When also treating the data as a labeled graph, evaluation takes place by searching for subgraphs homomorphic with the path.

In addition, path expressions can be enclosed in the `DISTINCT` statement. In this case, subgraph isomorphism is employed instead of subgraph homomorphism.

If path expressions share variables, these variables are required to match to the same object in the data.

Figure 5: A simple graph pattern

EXAMPLE 8. *Consider the following `FROM` and `WHERE` statement:*

```
FROM DISTINCT(X N1 -- Y -- X N2 -- Y -- X N3,
                                X N2 -- Y -- X N4)
WHERE N1.color = red    AND N2.color = blue
  AND N3.color = yellow AND N4.color = black
```

One can visualize this path expression, together with the `WHERE` statement, as the graph in Figure 5. Essentially, the pattern consists of two path expressions, linked together through a common node (N2). The pattern consists of 4 nodes and 3 edges, in which we have given variable names to the nodes. We pose additional constraints on these nodes. By the `DISTINCT` statement we enforce that no two nodes in the pattern will be matched to the same node in the data.

As a final feature, given that domains are sets of objects, we assume that the standard set operations `UNION`, `INTERSECT`, and `MINUS` can also be applied in the `FROM` part. Hence, we can express the following query:

```
FROM ( X UNION Y ) N
```

which puts nodes and edges in one domain.

4.2.3 *The* `SELECT` *statement*

The `SELECT` statement expresses which variables in the query are used to define a new domain. In our previous examples, queries always returned subsets of objects of existing domains. However, it can often be useful to create new objects. We illustrate this using the following query.

```
CREATE PatZ AS
SELECT <N1,E1,N2,E2,N3>
FROM X N1 -- Y E1 -- X N2 -- Y E2 -- X N3
WHERE N1.color = red
  AND N2.color = blue
  AND N3.color = yellow
```

The angle brackets (`<...>`) express a grouping operation; for each combination of objects `N1`, `E1`, `N2`, `E2`, `N3` for which the path expression matches the data, a new object is created. This object is by default only linked (in both directions) to the objects referred to by `N1`, `E1`, `N2`, `E2` and `N3`. Essentially, each object in the result represents an occurrence of a given subgraph in the data. The nodes and edges involved in this occurrence can be recovered by considering the links between the occurrence object and the nodes and edges in the data.

This new set of objects is useful, as we can think of each object as representing a graph in a set. This is illustrated in Figure 6.

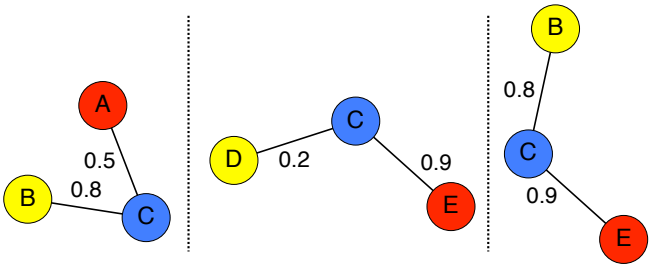


Figure 6: `graphset WITH PatZ AS Parts, X AS Nodes, Y AS Edges`

By default, newly created objects have no attributes assigned to them. However, attributes can be assigned by listing them in curly brackets (`{...}`), for example

```
SELECT <E>{E.*, weight: E.weight/2}
FROM Y E
WHERE E.weight > 0.4
```

which copies all the attributes from the original object but replaces the weight attribute by its old value divided by two.

The same notation can be used to determine the links with new objects. For instance, in the above example, we may want to connect the new edges with the same nodes as the old ones, and make the connection between the old and new edge directed. This can be done by adding the statement `E -- *` to the query, which tells the system to copy all of `E`'s links to the new object (similar to `E.*` for attributes), and the statement `->E`, which tells the system to insert a directed edge from the new object to the object `E`.

```
SELECT <E>{E.*, weight: E.weight/2, E -- *, -> E}
FROM Y E
WHERE E.weight > 0.4
```

4.3 Tool Integration

A main feature of our system is that it is capable of interacting with external tools. These tools differ in the type of input and output they require; some algorithms require directed graphs, others undirected node-labeled hypergraphs, or sets of these. We need a mechanism to exchange information with algorithms, and ensure that data of the correct type is provided to these algorithms. As pointed out, we represent these graph types in a uniform way, and use contexts to group appropriate domains together. Our main mechanism for ensuring that contexts satisfy the requirements of particular graph types are *classes* and *integrity constraints*. A basic example of a class is the following:

```
CLASS graph {
  Nodes,
  Edges
}
```

This defines a class with two domains, which are called `Nodes` and `Edges`. A context is an instance of class `graph` if it contains two domains under these aliases.

For each class, the database may support a number of export and import file formats. This allows the database to export a context representing a labeled graph in a format that tools operating on labeled graphs support, or to export a context representing a set of labeled graphs in a file format that tools operating on sets of graphs support.

In many cases, a context should fulfill more complex constraints than just having domains with certain names. An important aspect of classes is that they may include integrity constraints, which we discuss now.

4.4 Data Integrity

Structural integrity constraints are often desired in our data model. For instance, in traditional graphs an edge object must have degree 2. In this section we propose a mechanism for expressing constraints that domains should fulfill. The strategy that we have chosen is to allow for integrity constraints of the following kinds:

```
CONSTRAINT Domain1 SUBSET Domain2
CONSTRAINT Domain1 DISJOINT Domain2
CONSTRAINT Domain1 EQUAL Domain2
```

Here, the domains may be domains occurring in the data, but may also be computed by queries expressed over these domains. The constraint `Domain1 DISJOINT Domain2`, for instance, requires that two domains do not have common objects. Constraints can be part of classes, in which case the constraints are expressed over domains in the class. An example is provided below.

Example 9 (Labeled Graphset). *A context representing a graph set must consist of three domains: Parts, Edges and Nodes. Furthermore, if an edge is included in a part, the nodes to which it connects must be in the same part. We can express this by:*

```
CLASS labeled_graphset {
  Parts, Edges, Nodes,
  CONSTRAINT Edges DISJOINT
     SELECT E
     FROM Nodes N--Edges E--Parts P,
          Nodes N-/-Parts P
}
```

The query in this constraint selects all edges of which a node is not linked to the same part as the edge is. This query should not return edges.

When a context is created and declared to be an instance of a class, all constraints specified for the class should be satisfied, otherwise we assume that the instantiation fails.

Tools operate on and return instances of classes. Assume that we have an algorithm that operates on edge labeled graphs, and returns quasi-cliques as a set of subgraphs. We can call this algorithm as follows to create a context C:

```
CREATE C AS INSTANCE QuasiCliques(G)
```

A context is passed as a parameter to the algorithm. The ability to input and output contexts to external tools ensures that these tools are integrated within the database system.

5. OPERATIONAL SEMANTICS & THE RELATIONAL MODEL

Till now we have introduced our query language in an informal manner. We lack the space to formalize the operational semantics of our language extensively here. However, in this section we will provide the basic operational semantics by showing how queries can be translated into a logical representation. This will also clarify the relationship of our model to the relational model, and will show that a small extension to Datalog is sufficient to deal with the queries posed in this language.

First, we can formalize the database as a set of Datalog facts. This is straightforward:

- if an object O has attribute value V for attribute A, we represent this using a fact $\texttt{attribute}(\texttt{O}, \texttt{V}, \texttt{A})$
- if an object O_1 has a link to object O_2 we represent this using a fact $\texttt{link}(\texttt{O}_1, \texttt{O}_2)$
- if an object O is included in domain d, we represent this using a fact $\texttt{d}(\texttt{O})$

We can distinguish two types of basic queries. The first has a variable name in the `SELECT`:

CREATE *domain_name* AS
SELECT *Variable*
FROM $domain_1$ $Variable_1$-- ... --
$domain_n$ $Variable_n$
WHERE $predicate_1(...)$,...,$predicate_m(...)$

It is straightforward to express this query in Datalog:

$$\texttt{domain\_name}(\texttt{Variable}) :- \texttt{domain}_1(\texttt{Variable}_1), \texttt{link}(\texttt{Variable}_1, \texttt{Variable}_2)..., \texttt{domain}_n(\texttt{Variable}_n), \texttt{predicate}_1(...), ..., \texttt{predicate}_m(...)$$

Here the link constraint $Variable_i$--$Variable_j$ is mapped into a predicate $\texttt{link}(\texttt{Variable}_\texttt{i}, \texttt{Variable}_\texttt{j})$. Multiple path expressions are transformed into multiple series of `link` predicates. To deal with $DISTINCT(Variable_1, ..., Variable_n)$ requirements we can add atoms of the kind $\texttt{Variable}_\texttt{i} \neq \texttt{Variable}_\texttt{j}$ with $1 \leq i < j \leq n$. More involved is a query in which new objects are created:

CREATE *table_name* AS
SELECT <$Variable'_1$,..., $Variable'_k$> {attr:value}
FROM $domain_1$ $Variable_1$-- ... --
$domain_n$ $Variable_n$
WHERE $predicate_1(...)$,...,$predicate_m(...)$

Here it is required that

$$\{Variable'_1, ..., Variable'_k\} \subseteq \{Variable_1, ..., Variable_n\}$$

The problem with this query is that it introduces a new object, and hence, we need to introduce new object identifiers. Datalog does not provide a mechanism for introducing new object identifiers. We address this as follows. We assume that we can use one function

$$\texttt{id}(\texttt{domain\_name}, \texttt{Variable}'_1, ..., \texttt{Variable}'_\texttt{k}),$$

to construct a new identifier for a domain and a combination of object identifiers. Then the above query can be translated into the following queries in Prolog:

- to obtain new objects:

$$\texttt{domain\_name}(\texttt{id}(\texttt{domain\_name}, \texttt{Variable}'_1, ..., \texttt{Variable}'_\texttt{k})) :- \texttt{domain}_1(\texttt{Variable}_1), ..., \texttt{domain}_n(\texttt{Variable}_n), \texttt{predicate}_1(...), ..., \texttt{predicate}_m(...)$$

- to obtain the links to these objects:

$$\texttt{link}(\texttt{id}(\texttt{domain\_name}, \texttt{X}_1, ..., \texttt{X}_\texttt{k}), \texttt{X}_\texttt{i}) :- \texttt{domain\_name}(\texttt{id}(\texttt{domain\_name}, \texttt{X}_1, ..., \texttt{X}_\texttt{k}))$$

  for all $1 \leq i \leq k$;

- to obtain the attributes:

$$\texttt{attribute}(\texttt{id}(\texttt{domain\_name}, \texttt{X}_1, ..., \texttt{X}_\texttt{k}), \texttt{attr}, \texttt{value}) :- \texttt{domain\_name}(\texttt{id}(\texttt{domain\_name}, \texttt{X}_1, ..., \texttt{X}_\texttt{k})).$$

To deal with function calls, we assume that the output of the called algorithm is converted in appropriate Datalog facts.

6. AN EXTENDED QUERY LANGUAGE

In Section 4.2 we gave a description of the basic syntax of our query language. In this section we describe some possible extensions to this query language that make it more powerful: aggregates, aliases and regular expressions.

Aggregates.

Aggregate functions like count, sum, min, ... can be used to define new attribute values in the `SELECT` clause or act as constraints in the `WHERE` clause. They are computed over sets of objects. For example,

```
SELECT <N>{N.*, N -- *,
           in-degree: count(E),
           in-weight: avg(E.weight)}
FROM X N <- Y E
```

calculates for each node the in-degree (number of incoming edges) and the average weight of these edges. The expression `<N>` groups all occurrences of the graph pattern, such that we have a group of occurrences for each value for `N`. For each group, the aggregation operators are executed once; in this case, the aggregation goes over all possible assignments to `E` in the group.

Aggregates may also be used in the `WHERE` clause to restrict the results of a query. For example, adding the statement

```
WHERE count(E) > 1
```

to the query above will restrict the results to nodes with more than one incoming edge. The grouping used for executing this condition is defined by the object creation statement in the `SELECT` clause.

Aliases.

In the pattern that expresses the graph of Figure 6 (see Section 4.2) we used a path expression containing five variables; we had to provide a name for each of the individual variables in the pattern. Using an alias we can rewrite this query, eliminating the need to name every variable in the pattern in the `SELECT` statement.

```
DEFINE PatZ AS
SELECT <Pattern>
FROM (X N1 -- Y -- X N2 -- Y -- X N3)
```

```
                                        AS Pattern
WHERE N1.color = red AND N2.color = blue
                     AND N3.color = yellow
```

Aliases are especially useful in combination with regular expressions, because the number of variables defined in these expressions is unknown; we will use an alias to refer to each possible occurrence of the regular expression.

Regular expressions.

The path expressions we used in all the previous examples were limited in their expressivity because the length of the connections needed to be known. By using regular expressions we can overcome this limitation.

For example one could express a path of arbitrary length as

```
FROM X (-- Y -- X)* -- Y -- X
```

This expression can be expanded into

```
FROM X -- Y -- X
FROM X -- Y -- X -- Y -- X
FROM X -- Y -- X -- Y -- X -- Y -- X
...
```

We do not allow variable names inside a repetition block, as there may be no or multiple assignments to these variables. We can, however, assign an alias to such a block, which allows us to access the objects matched by the regular expression.

```
FROM X A ((-- Y -- X)* -- Y) AS Path -- X B
```

This alias can be used in combination with, for example, aggregate functions to calculate the length of the path,

```
SELECT <Path>{ length: count(Path->Y) }
```

or to select paths according to length,

```
WHERE count(Path->Y) < 5
```

A special construct can also be used to specify the shortest path:

```
WHERE count(Path->Y) is minimal
```

In this query we use a construction in which `Path->Y` represents all nodes in domain Y linked to the object Path.

7. ILLUSTRATIVE EXAMPLES

In Section 2 we pointed out some queries that should be expressible. Using the principles outlined above, we can now verify that our language indeed supports these examples.

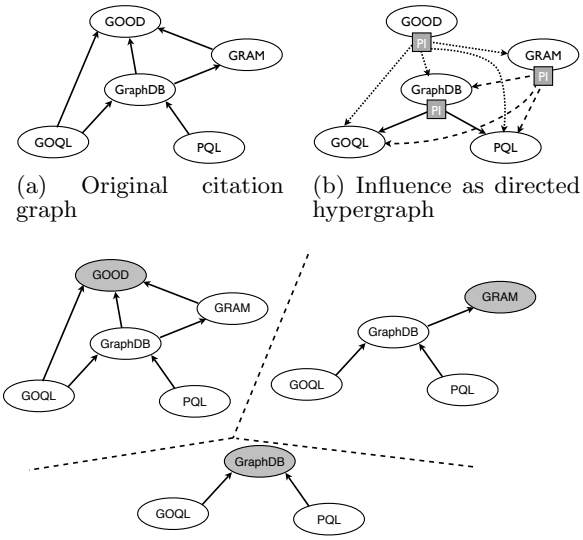
First we can define the co-authorship relation as follows (Example 1).

```
CREATE CoAuthors AS
SELECT <A,B>{strength: count(P)}
FROM Author A -- Paper P -- Author B
```

This creates a new `CoAuthor` object for every pair of authors, and adds the number of papers they share as attribute. Each co-author attribute is also linked to both authors in the relation.

The definition of the author citation relation is similar.

```
CREATE AuthorCites AS
SELECT <A,B>{ A ->, -> B}
FROM Author A -- Paper -> Cites ->
                           Paper -- Author B
```



(a) Original citation graph

(b) Influence as directed hypergraph

(c) Influence as set of graphs

Figure 7: Result of influence query on the citation graph of graph database papers

However, because the citation relation is directed, we need to override the default behavior of creating undirected links to objects `A` and `B`. We do this by explicitly specifying the direction of the links between the new object `<A,B>` and its children `A` and `B`.

To find authors mutually citing each other (Example 2), we can use the `AuthorCites` relation we just defined.

```
CREATE MutualCitation AS
SELECT <A,B>
FROM Author A -> AuthorCites -> Author B,
     Author B -> AuthorCites ->
         ( Author -> AuthorCites )* -> Author A
```

To find quasi cliques in this graph (Example 3) we can call an external algorithm. This algorithm takes a graph with `Authors` as nodes and `MutualCitations` as edges.

```
CREATE G AS INSTANCE labeled_graph WITH
     Author AS NODES,
     MutualCitation AS EDGES


CREATE C AS INSTANCE QuasiCliques(G)
```

We can find for each paper the set of papers that cite it directly or indirectly (Example 4). This can be done using a regular expression.

```
CREATE PaperInfluence AS
SELECT <X>{ X ->, -> Y }
FROM Paper X (-> Cites -> Paper)* ->
                          Cites -> Paper Y
```

This query creates for each paper `X` a new object and links it to (1) paper `X` itself, (2) every paper `Y` for which a citation path exists. Figure 7 shows the result of this query on an example citation graph. The result of the query is a directed hypergraph, which can also be treated as a set of citation graphs by defining a suitable context.

Finally, we will study how to cluster citations (Example 5). First, we need to add similarities between citations, assuming we have a `location` attribute in every citation.

```
CREATE CitationSimilarity AS
SELECT <X,Y>
FROM Paper P -> Cites X, Paper P -> Cites Y
WHERE abs ( X.line - Y.line ) < 3
```

Using the resulting relations between citations, we can create a graph and execute a graph clustering algorithm.

```
CREATE G AS INSTANCE labeled_graph WITH
     Cites AS NODES,
     CitationSimilarity AS EDGES

CREATE C AS INSTANCE GraphCluster(G)
```

The graph clustering algorithm is assumed to have a similar output as the quasi-clique algorithm, and returns a set of subgraphs, each corresponding to a cluster.

8. CONCLUSIONS

Motivated by the need to have database support for the analysis and mining of large networks we contributed a novel data model and query language (BiQL) that can act as the basis for an inductive database system. The data model is based on a number of design choices that make – as far as the authors are aware – it unique as compared to other graph database formalisms. These features include the uniform treatment of nodes and edges, providing for contexts, enforcing the closure property and supporting external calls to other procedures. This last choice should allow the easy coupling of the database to external data mining algorithms. This is similar to what happens in existing database systems that support data mining in this way, such as SINDBAD [20] and Oracle's Data Mining extensions. Other important features of the language include its support for graph-based querying (using, for instance, path or subgraph oriented queries) and the easiness with which networks can be transformed into one another.

Nevertheless, there remain several important issues for further research. The first of these concerns the development of an efficient and scalable implementation, which should enable us to experiment with several application databases and to realize a true inductive database. Today, a prototype of this system exists, implemented in Prolog and based on the Datalog representation described in Section 5. In future work we aim to extend this prototype to a scalable and fully functional system containing the different extensions described in Section 6 of this paper.

Other interesting issues concern the incorporation of mechanisms for supporting probabilistic reasoning in such networks as well as providing a user interface and a visualization interface that should enable the naive end-user to operate the system. These topics will be studied further in the European BISON project.

Acknowledgements

This work was supported by the European Commission under the 7th Framework Programme FP7-ICT-2007-C FET-Open, contract no. BISON-211898 and by a Postdoc grant from the Research Foundation — Flanders.

9. REFERENCES

[1] B. Amann and M. Scholl. Gram: a graph data model and query language. In *HT*, pages 201–211. ACM, 1992.

[2] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008.

[3] I. Bhattacharya and L. Getoor. Collective entity resolution in relational data. *Data Engineering Bulletin*, 29(2):4–12, 2006.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[5] R. H. Güting. GraphDB: Modeling and querying graphs in databases. In *VLDB*, pages 297–308. Morgan Kaufmann Publishers Inc., 1994.

[6] M. Gyssens, J. Paredaens, and D. van Gucht. A graph-oriented object database model. In *PODS*, pages 417–424. ACM, 1990.

[7] J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-based multidimensional data mining. *IEEE Computer*, 32(8):46–50, 1999.

[8] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *SIGMOD*, pages 405–418. ACM, 2008.

[9] J. Hidders. Typing graph manipulation operations. In *ICDT*, pages 394–409. Springer-Verlag, 2003.

[10] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[11] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[12] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal on Parallel and Distributed Computing*, 48(1):96–129, 1998.

[13] U. Leser. A query language for biological networks. *Bioinformatics*, 21(2):33–39, 2005.

[14] M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *JCIT*, pages 520–530. IEEE Computer Society Press, 1990.

[15] M. Levene and A. Poulovassilis. An object-oriented data model formalised through hypergraphs. *Data and Knowledge Engineering*, 6(3):205–224, 1991.

[16] R. Meo, G. Psaila, and S. Ceri. An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, 2(2):195–224, 1998.

[17] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2008.

[18] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3):93–107, 2008.

[19] L. Sheng, Z. Ozsoyoglu, and G. Ozsoyogly. A graph query language and its query processing. In *ICDE*, pages 572–581. IEEE Computer Society Press, 1999.

[20] J. Wicker, L. Richter, K. Kessler, and S. Kramer. SINDBAD and SiQL: An inductive database and query language in the relational model. In *ECML/PKDD (2)*, volume 5212 of *Lecture Notes in Computer Science*, pages 690–694. Springer, 2008.

[21] Z. Zeng, J. Wang, L. Zhou, and G. Karypis. Coherent closed quasi-clique discovery from large dense graph databases. In *KDD*, pages 797–802. ACM, 2006.