GraphDB: Modeling and Querying Graphs in Databases

Ralf Hartmut Güting
Praktische Informatik IV, FernUniversität Hagen
D-58084 Hagen, Germany
gueting@fernuni-hagen.de

Abstract

We propose a data model and query language that integrates an explicit modeling and querying of graphs smoothly into a standard database environment. For standard applications, some key features of object-oriented modeling are offered such as object classes organized into a hierarchy, object identity, and attributes referencing objects. Querying can be done in a familiar style with a *derive statement* that can be used like a *select … from … where*. On the other hand, the model allows for an explicit representation of graphs by partitioning object classes into simple classes, link classes, and path classes whose objects can be viewed as nodes, edges, and explicitly stored paths of a graph (which is the whole database instance). For querying graphs, the derive statement has an extended meaning in that it allows one to refer to subgraphs of the database graph. A powerful *rewrite* operation is offered for the manipulation of *heterogeneous sequences of objects* which often occur as a result of accessing the database graph. Additionally there are special graph operations like determining a shortest path or a subgraph and the model is extensible by such operations. Besides being attractive for standard applications, the model permits a natural representation and sophisticated querying of *networks*, in particular of *spatially embedded networks* like highways, public transport, etc.

This work was supported by the ESPRIT Basic Research Project 6881 AMUSING

1 Introduction

The work described in this paper arose from the observation that existing data models and query languages do not offer adequate support for the modeling and querying of *networks*. In particular, we are interested in *spatially embedded networks* which are an important part of geographic information, for example, highways, rivers, public transport systems, power and phone lines etc. Current spatial database models and systems (e.g. [SvH91, RoFS88, OrM88, Gü89]) can well enough represent the geometry of such networks but have no concept of their connectivity.

We feel that the most natural representation of a highway network (taking it as a prototype for spatial networks) is to view it as a graph whose nodes are highway junctions, whose edges are highway sections, and where highways are just certain paths over this graph. Therefore we would like to offer a data model capable to express this directly so that one can define a graph structure with corresponding node, edge, and path objects. For querying, special graph operations should be provided such as finding a shortest path, determining a subgraph within a given radius from a start node, etc.

On the other hand, modeling and querying networks is certainly not the only thing a user wants to do; hence, all of the more traditional applications should be supported as well, and preferably in a style that is not too different from what one knew before. The challenge is therefore to achieve a smooth integration of the desired graph modeling into a more classical environment. Ideally, if one is not interested in networks, this model should be usable like any of the well-known models, e.g. a relational, functional, or object-oriented one.

The purpose of this paper is to present a data model and query language that achieves such a smooth integration. On the one hand, we show that traditional applications can be modeled and queried in a familiar style, and indeed, better than before, because this model offers very attractive features to represent relationships between objects and to use them in queries. So we claim that even without considering networks, this model is suitable and quite interesting as a general purpose data model. On the other hand, sophisticated modeling and querying of networks is possible, as we demonstrate by a number of examples. Our approach can be summarized as follows:

- The data model contains a few salient features of object-oriented models: A database is a collection of object classes. Objects have identity and a tuple structure; attributes may be data or object-valued. Classes are organized in an inheritance hierarchy. Central tool for querying is a *derive statement* which so far offers similar capabilities as the traditional *select … from … where*.
- The data model offers graphs: There are three different *kinds* of object classes called *simple classes*, *link classes*, and *path classes*. Simple objects play the role of nodes in the database graph. Link objects are objects with additional distinguished references to source and target simple objects. Path objects are objects with an additional list of references to simple and link objects that form a path over the database graph.
- For querying the graph structure, (1) the derive statement has an extended meaning: In the *on-clause* (the counterpart to the *from …*) one can refer to connected subgraphs of the database graph and so specify relationships between simple objects, link objects, and path objects. (2) There is a special tool for sophisticated manipulation of heterogeneous sequences of objects (paths, in particular). (3) There is a collection of graph operations; they can specify argument subgraphs of the database by regular expressions over link class names (edge types). (4) The database graph can be extended or restricted dynamically within a query.

In this paper we give a formal definition of the data model and describe some key elements of the query language. This is a short version of [Gü94], where many aspects are treated in more depth, in particular, definition of subclasses and rewriting of sequences (paths). In [Gü94] additionally a system architecture and implementation strategy are described which are used in the GraphDB prototype we are currently implementing.

In the literature, the manipulation of graphs in databases has received quite a bit of attention, a survey can be found in [MaS90]. However, to our knowledge nowhere has the focus been on an *explicit representation of graphs* together with a *smooth integration* into standard modeling and querying. Most authors assume that graphs can be modeled implicitly in terms of the usual features of a given data model, e.g. the relational model [Kung86, StR86, Ag87, BiRS90], or a functional model [Rose86]. In most proposals the authors do not really care how graphs are represented but just focus on the abstract graph structure [CrMW87a, CrMW87b, CrN89, Rose86]. For querying, two main strategies are to offer general purpose facilities that allow to express graph traversal problems (like recursion, iteration) [Rose86, StR86], or to offer special operators [Ag87, Rose86, CrN89]. [BiRS90] propose an SQL extension based on the idea to generate a set of paths in the *from-clause* from which interesting paths are selected. In [CrMW87a, CrMW87b, CoM90, CoM93] the idea is to formulate a query as a set of graphs which are viewed as *patterns*; all subgraphs of the database instance are returned matching these patterns. In all of these approaches there is no explicit modeling of graphs within a general database environment and therefore no problem of integration with the data model. In some cases, graph querying has a very different style from the rest of data manipulation. – In contrast to the work above, in [GyPV90a, GyPV90b, Andr92, GePTV93] the approach is to model the database directly and entirely as a graph and to express *all* queries in terms of a few powerful graph manipulation primitives. Graphical user interfaces are offered for direct input of queries and visualization of results in terms of graph structures. This is mainly intended as an end user interface to a database system that may itself use another data model.

We feel that an explicit modeling of graphs is very desirable for several reasons: (i) It leads to a more natural modeling; graph structures are visible for the user, (ii) queries can refer directly to this graph structure, (iii) path objects can be defined (not present in any of the other models) and they are the interesting entities in most networks, (iv) the system can offer special data structures for graphs, and (v) the system can use efficient graph algorithms designed to utilize the special graph data structures.

The general approach of this paper has been pursued in our own previous work [Gü91, ErG91] and that of collaborators [AmS92]. In [Gü91] relations and graphs coexist. A problem with that approach is that a graph consisting only of nodes is practically the same as a relation and it is hard to separate graphs from each other and from relations in a database design. In [ErG91] graphs occur in an environment with object classes but are still separate entities. There is the same problem of partitioning a database into graphs. Also, in both approaches it becomes a nuisance to mention the graph arguments in many places in queries. The querying facilities offered in this paper go far beyond those of [Gü91, ErG91]. Amann and Scholl [AmS92] offer a few selected features of our model (node and edge objects, but no paths) in the context of hypertext applications.

The paper consists of two major sections, describing the data model and querying, respectively.

2 The Data Model

This section introduces the data model of GraphDB. We start with an overview and show the modeling of some example applications. In the following subsections, the model is developed more formally and systematically, defining bottom up the notions of data types, object types, and tuple types, three kinds of object classes, a database and the database graph. The main purpose in the design of the data model is to achieve a "seamless" integration of graph structures and graph operations into the usual facilities for data modeling and querying.

2.1 Overview

A database is a collection of object classes which are partitioned into three kinds of classes, called *simple classes*, *link classes*, and *path classes*. Objects of a *simple class* are on the one hand just like objects in other models: They have an object type and an object identity and can have attributes whose values are either of a *data type* (e.g. integer, string) or of an *object type* (that is, an attribute may contain a reference to another object). So the structure of an object is basically that of a tuple or record. On the other hand, objects of a simple class are *nodes* of the *database graph* – the whole database can also be viewed as a single graph. Objects of a *link class* are like objects of a simple class but additionally contain two distinguished references to source and target objects (belonging to simple classes), which makes them *edges* of the database graph. Finally, an object of a *path class* is like an object of a simple class, but contains additionally a list of references to node and edge objects which form a path over the database graph.

Besides the graph structure, object classes are organized into a class hierarchy and there are related notions of subtyping among tuple types, object types, and data types. Let us now consider some examples of data modeling with these facilities.

Standard Applications. As a simple standard application, consider the representation of books and their authors. We describe the database schema by showing corresponding data definition commands.

```
create class book = title: STRING,
   publisher: STRING, year: INTEGER;
create class person = name: STRING,
   address: STRING;
create link class wrote from person to
   book;
```

Here we have two simple classes *book* and *person* and a link class *wrote*. Observe how a link class can directly represent a many-many relationship. Attributes may be defined for link classes in the same way as for simple classes. Attributes may also contain object references. For example, we might define persons to contain a reference to their home country:

```
create class state = name: STRING,
   region: REGIONS;
create class person = name: STRING,
   address: STRING, country: state;
```

Here REGIONS is a geometric data type describing the area covered by the state.

Highway Network. The highway network is a relatively simple example of a spatially embedded network. It is a graph whose nodes are highway junctions and exits; each of those has an associated point in the geometric (or geographic) plane. We assume junctions are characterized by a name and exits by a number. Edges of this graph are highway sections: pieces of road between junctions and/or exits with an associated geometry which is a polyline in the plane. The most interesting objects of this network are highways: they correspond to paths over the graph given by junctions, exits, and highway sections.

```
create class vertex = pos: POINT;
create vertex class junction = name:
   STRING;
create vertex class exit = nr: INTEGER;
create link class section = route: LINE,
   no_lanes: INTEGER, top_speed: INTEGER
   from vertex to vertex;
create path class highway = name: STRING
   as section+;
```

Here junctions and exits are introduced as subclasses of a simple class *vertex*, which means they inherit the *pos* attribute. It also means that objects of both classes can be used as sources and targets of *section* edges of the graph. Indeed, there can also be "pure" *vertex* objects as nodes in the graph; they are useful to separate highway sections with different values of attributes such as *no_lanes*. The *highways* themselves are defined to be paths over *section* edges. Essentially the expression behind the keyword *as* is a regular expression defining a *path type* which in turn describes a set of paths of the database graph. Path types are more interesting when different kinds of edges occur in a graph. We will see examples and a more precise definition below.

2.2 Data Types, Object Types, and Tuple Types

Data Types. Let $(D, \leq)$ be a finite set whose elements are called *data types*, with a partial order "$\leq$" ("subtype") which is restricted to organize data types into trees (that is, $\forall\ a, b, c \in D: a \leq b \wedge a \leq c \Rightarrow b \leq c \vee c \leq b$). If two data types belong to the same tree, we call them *related*. If two data types are related, then a *smallest common supertype*, denoted $lub(a, b)$ exists and is uniquely defined. Figure 1 shows a collection of data types organized into several trees. Here, for example, INTEGER is a subtype of NUM (INTEGER $\leq$ NUM) and lub(POINTS, LINES) = GEO. Each data type has an associated domain of values given by a function *dom* (e.g. dom(BOOL) = {true, false}). If $a \leq b$ then $dom(a) \subseteq dom(b)$. The purpose of the data type hierarchy is to allow polymorphic functions to be defined, for example, an intersection test can be applied to any two geometric values in EXT.

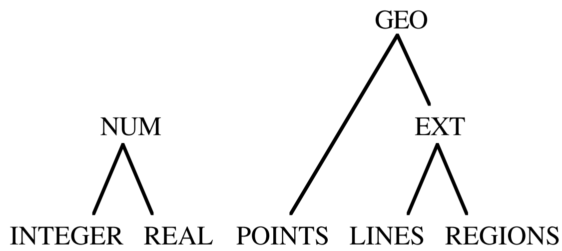

Figure 1

Object Types. There is a finite set $(OT, \leq)$ whose elements are called *object types* with a (tree) partial order "$\leq$". We will see below that there is a one-to-one correspondence between object types and classes in the database; in fact, an object type is nothing else than the name of a class. The partial order on object types corresponds to the class hierarchy. Similarly as for data types, two object types c, d may be related (belong to the same tree) in which case the smallest common supertype, $lub(c, d)$, is well-defined.

Each object type has an associated set of object identifiers which is a subset of a set of object identifiers *OID* (which contains the identifiers of all objects created so far). This set is given by a function *oids*: $OT \rightarrow \mathbf{P}(OID)$, where $\mathbf{P}(X)$ denotes the power set of X. If $c \le d$ ($c, d \in OT$), then $oids(c) \subseteq oids(d)$. On the other hand, if c and d are not related, then $oids(c) \cap oids(d) = \emptyset$. For an object type c, $oids(c)$ contains precisely the identifiers of objects created so far in the corresponding class c. Given an object identifier, we can determine its *immediate type* (the smallest object type in the hierarchy that it belongs to) by a function *itype*: $OID \rightarrow OT$ defined by:

$$itype(o) = c \iff (o \in oids(c) \wedge \forall\, d \in OT\colon\ o \in oids(d) \Rightarrow c \le d)$$

Tuple Types. Let A be a set whose elements are called *attributes* – a domain of attribute names that can be used in forming tuple types. The set of *tuple types*, denoted *TT*, is defined as follows:

$$TT = \{<(a_1, t_1), \ldots, (a_m, t_m)> \mid m \ge 0, \forall i \in \{1, \ldots, m\}\colon a_i \in A, t_i \in D \cup OT\}$$

That is, each tuple type in *TT* is a list of pairs where each pair contains an attribute name together with either a data type or an object type. The empty list <> is also a tuple type. For each tuple type, there is a domain of *tuple values* defined as follows. Let $T \in TT$, $T = <(a_1, t_1), \ldots, (a_m, t_m)>$.

$$values(T) = \bigcup_{i=m}^{\infty} V_i$$

where
$$V_i = \{<(v_1, u_1), (v_2, u_2), \ldots, (v_i, u_i)> \mid \forall j \in \{1, \ldots, i\}\colon\ u_j \in D \cup OT \wedge u_j \in D \Rightarrow v_j \in dom(u_j) \wedge u_j \in OT \Rightarrow v_j \in oids(u_j) \wedge j \le m \Rightarrow u_j \le t_j\}$$

In other words, a tuple value is also a list of pairs of some length i which must be at least m. Each pair is a value together with a type, and the value must belong to the corresponding data or object identifier domain. Furthermore, within the first m components the type in the tuple value must be a subtype of the corresponding data or object type in the tuple type T.

The subtype relationship on tuple types is defined as follows. Let $T = <(a_1, t_1), \ldots, (a_m, t_m)>$, $U = <(b_1, u_1), \ldots, (b_n, u_n)>$ be two tuple types.

$$T \le U \ :\Leftrightarrow\ m \ge n \ \wedge\ \forall i \in \{1, \ldots, n\}\colon t_i \le u_i$$

That is, tuple components (attributes) are matched by position. T must have at least as many components as U and in each of the first n positions the type in T must be a subtype of the one in U. Attribute names do not matter. For tuple types $T = <(a_1, t_1), \ldots, (a_m, t_m)>$, $U = <(b_1, u_1), \ldots, (b_n, u_n)>$ one can determine a *smallest common supertype* as follows:

$$lub(T, U) := <(a_1, lub(t_1, u_1)), \ldots, (a_k, lub(t_k, u_k))>$$

where $k \in \{0, \ldots, \min(m, n)\}$ such that for $1 \le i \le k$, t_i and u_i are related, and either $k \in \{m, n\}$, or t_{k+1} and u_{k+1} are not related. In other words, we take the longest common prefix of related types and within it for each pair of types their smallest common super data or object type. Of course, the result may be the empty tuple type. Note that attribute names are taken from the first operand, so the operation is not commutative. The purpose of these definitions is to allow for a "dynamic generalization" of collections of tuples. We will be able to form in queries any union of sets of tuples; for the resulting set, a new tuple type is derived such that all tuples in the union match this new type (see Section 3.3).

2.3 Classes and Database

A *database* is a pair $(C, \le)$ where C is a finite set of *classes*, and "≤" ("subclass") a tree partial order on C. A *class* $c \in C$ is a pair $(ctype(c), extension(c))$. The set of classes C is partitioned into three subsets: $C = SC \cup LC \cup PC$. Classes in SC, LC, and PC are called *simple classes*, *link classes*, and *path classes*, respectively. The subclass partial order respects this partition, that is,

$$a \le b \Rightarrow \{a, b\} \subseteq SC \vee \{a, b\} \subseteq LC \vee \{a, b\} \subseteq PC$$

The two components of a class, its *type* and its *extension*, are different for simple classes, link classes, and path classes. Informally, the type defines the structure of objects in the class, and the extension the collection of objects currently contained in it. In the following subsections we describe type and extension for the three kinds of classes, relating them to corresponding data definition commands. Subclasses are treated in the full paper [Gü94].

Simple Classes. A simple class is created by a command of the form

```
<class creation> ::= create class <class-
    name> [ = <attribute-list>] ;
<attribute-list> ::= <attr-name> : <type>
    | <attr-name> : <type> , <attribute-list>
```

The *type* of a simple class is a pair (c, T), if it was created by a command

```
create class c = T;
```

where c is the class name used in the definition and T the tuple type corresponding to the attribute list. If the optional clause is omitted, then the tuple type is the empty type <>. For brevity, we will speak in definitions simply of a class (c, T) instead of "a class with type (c, T)". As mentioned in Section 2.2, there is a one-to-one correspondence between classes and object types. Hence, the class creation command creates at the same time a new object type $c \in OT$. The *extension* of a simple class (c, T) is a subset of $oids(c) \times values(T)$, that is, a set of pairs consisting of an object identifier and a tuple value. Object identifiers are all distinct:

$$\forall\, (o_1, t_1), (o_2, t_2) \in extension(c)\colon o_1 = o_2 \Rightarrow t_1 = t_2$$

Link Classes. A *link class* is created by a command of the form

```
<class creation> ::= create link class
    <class-name> [ = <attribute-list>] from
    <class-name> to <class-name> ;
```

The *type* of a link class is a quadruple (c, T, d, e) if it was created by a command:

```
create link class c = T from d to e;
```

Here d and e must be the names of simple classes. The *extension* of a link class (c, T, d, e) is a set of quadruples which is a subset of $oids(c) \times values(T) \times oids(d) \times oids(e)$.

The Database Schema and Instance Graphs. Before we can define path classes, we need to understand the graph structure created by a collection of simple classes and link classes, which consists of a *database schema graph* and a *database instance graph*. We generally describe graphs as two sets (nodes and edges) together with two mappings *source* and *target* from the edges into the nodes. This is because multiple edges between the same two nodes are allowed in our data model.

The *database schema graph* is $SG = (S, L, source, target)$, where

(i) $S = \{ c \mid ((c, T), ext) \in SC\}$

(ii) $L = \{ c \mid ((c, T, d, e), ext) \in LC\}$

(iii) $source: L \rightarrow S$ is defined by
$source(c) = d \iff ((c, T, d, e), ext) \in LC$

(iv) $target: L \rightarrow S$ is defined by
$target(c) = e \iff ((c, T, d, e), ext) \in LC$

So for each simple class and each link class there is one node and one edge in the schema graph, respectively. These nodes and edges are also the object types corresponding to the respective classes.

The *database instance graph* is $IG = (S, L, source, target)$, where

(i) $S = \{ o \mid \exists\, c \in SC: (o, t) \in extension(c)\}$

(ii) $L = \{ o \mid \exists\, c \in LC: (o, t, p, q) \in extension(c)\}$

(iii) $source: L \rightarrow S$ is defined by
$source(o) = p \iff \exists\, c \in LC: (o, t, p, q) \in extension(c)$

(iv) $target: L \rightarrow S$ is defined by
$target(o) = q \iff \exists\, c \in LC: (o, t, p, q) \in extension(c)$

So the nodes and edges of this graph are object identifiers of objects in simple and link classes, respectively.

A *path type* is a quadruple (G, μ, s, F) where

(i) $G = (V, E, source, target)$ is a connected graph.

(ii) $\mu: V \cup E \rightarrow OT$ is a function labeling nodes and edges of G with object types such that
 (a) $v \in V \;\Rightarrow\; \exists\, ((c, T), ext) \in SC: \;\mu(v) = c$
 (b) $e \in E \;\Rightarrow\; \exists\, ((c, T, a, b), ext) \in LC:$
 $\mu(e) = c \;\wedge\; \mu(source(e)) = a \;\wedge\; \mu(target(e)) = b$

(iii) $s \in V$ (the *start node*)

(iv) $F \subseteq V$ (the *final nodes*)

Basically, a path type is nothing else than a finite automaton belonging to a regular expression over link class names, s is the start state, F the set of final states. The labeling function μ ensures consistency with the database schema graph. Each path in G from start node s to some node in F describes a corresponding set of paths in the database instance graph, defined below. In Figure 2 the path type corresponding to the regular expression "section+" from path class *highway* (Section 2.1) is shown (a circle around a node indicates the start node, a box one of the final nodes).

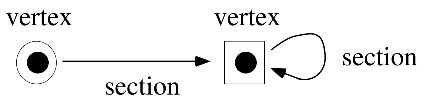

Figure 2

Path types are used in the definition of path classes, but also in queries, where graph traversal can be restricted to graphs of a desired form. A *path over IG* (the database instance graph), also called a *database path*, is a sequence of object identifiers

$$p = \langle v_0, e_1, v_1, \ldots, v_{n-1}, e_n, v_n \rangle$$

where for $0 \le j \le n$: $v_j \in S$ and for $1 \le j \le n$: $e_j \in L$, $source(e_j) = v_{j-1}$, and $target(e_j) = v_j$. This path *matches* a path type (G, μ, s, F) iff there exists a path $P = \langle V_0, E_1, V_1, \ldots, V_{n-1}, E_{n-1}, V_n \rangle$ in G such that:

(i) $V_0 = s$

(ii) $V_n \in F$

(iii) For $0 \le j \le n$: $itype(v_j) \le \mu(V_j)$ (*itype* yields the immediate type of object identifier v_j)

(iv) For $1 \le j \le n$: $itype(e_j) \le \mu(E_j)$

In other words, the database path p must have a corresponding path P in G such that each object in the path p is of a subtype of the one required in P. We denote by $paths(G, \mu, s, F)$ the set of all database paths matching path type (G, μ, s, F).

Path Classes. A *path class* is created by a command of the form:

```
<class creation> ::= create path class
   <class-name> [ = <attribute-list>] as
   <link-expression>
<link-expression> ::=  <class-name>+ |
   <class-name>
   | <link-expression> <link-expression>
   | (<link-expression> or <link-
   expression>) | (<link-expression>)*
```

Essentially a link expression *LE* is a regular expression over class names which must belong to link classes. The regular expression must be chosen in such a way that it defines a connected graph, that is, a path type. The correspondence between regular expressions and path types is straightforward. If a path class was created by a command

```
create path class c = T as LE;
```

then its *type* is the triple $(c, T, (G, \mu, s, F))$ where (G, μ, s, F) is the path type corresponding to *LE*. The *extension* of class $(c, T, (G, \mu, s, F))$ is a set of triples subset of $oids(c) \times values(T) \times paths(G, \mu, s, F)$.

2.4 The Public Transport Network

In this subsection we introduce a larger example which may give a better impression of the kind of applications this data model (and database system) is intended for. We shall also show some queries for this example in the next section. The application domain to be represented is public transport, e.g. bus, tram, or train lines and schedules. On closer inspection, this application is more complex than one might have expected. One can distinguish three levels of network, describing *the physical network*, *lines*, and *time schedules*. The lowest level represents the geometry of the network used for traveling. For example, for a railway network, at this level we find rails and switches, switches being the nodes and rail sections the edges of the graph. We call paths over this level *physical routes*. This level is modeled as follows:

```
class vertex = pos: POINT;
link class arc = route: LINE from vertex
   to vertex;
path class phys_route as arc+;
```

The next level introduces regular connections over the physical network usually called *lines*, for example, bus or underground lines traversing a certain path of the physical network. A line may be identified with a number or by giving the names of final destinations at both ends, and it contains a list of stops that we call *stations*. (A line is what is usually depicted on the wall within a bus or underground carriage.) Note that this level does not yet contain the time schedule for trips over lines.

```
class station = name: STRING, loc: vertex;
link class connection = travel_minutes:
   INT, way: phys_route from station to
   station;
path class line = line_type: STRING,
   line_no: INT as connection+;
```

Observe that this second level contains references to the first level, the physical network, associating *stations* by attribute *loc* with their physical positions (assuming that for each station a vertex has been established) and *connections* as a piece of the line between two stations by attribute *way* with a corresponding path over the physical network. *Lines* are paths of this level; the *line_type* attribute may be used to distinguish types of connections, e.g. fast long distance trains from slow local trains in a railway network.

The third level contains the actual time schedules. We model this as a collection of *departure* and *arrival events*, which will be the nodes of the third level graph. A departure event, for example, says that at a certain time a carrier (e.g. a train) of a specified line departs from a given station. A specific trip of a train over a line then corresponds to an alternating sequence of departure and arrival events (Figure 3).

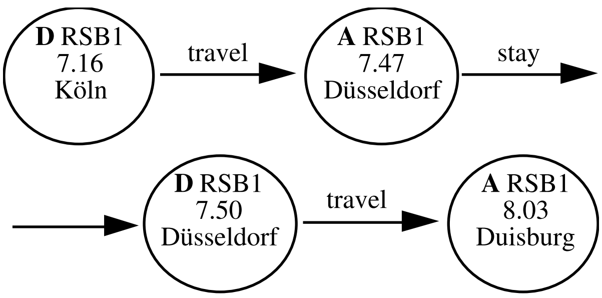

Figure 3

In Figure 3 **D** and **A** stands for departure and arrival, respectively; “RSB1” is the name of a particular line. The event nodes of a specific trip are connected by *travel* and *stay* edges. On the other hand, we can change at a station from one line to another (more precisely, from a trip of one line to a trip of another line). To model this, the arrival and departure events at one particular station are connected by *change* and *wait* edges, as shown in Figure 4.

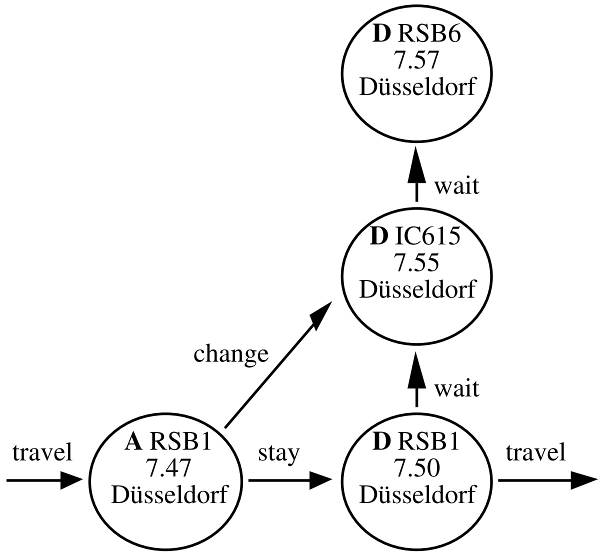

Figure 4

The idea is that a *change* edge connects an arrival event with the next departure one can reach in time at this station, and that all departure events are linked in the order of departure time. Hence changing at a station can be described by a sequence of change and wait edges of the form _change wait*_ (which is a path type). A complete trip of a traveler with possibly several changes of trains has a path type _travel (stay travel)* (change wait* travel (stay travel)*)*_.

So the third level is modeled as follows:

```
class event = time: INT, at_station:
   station, of_line: line;
event class arrival, departure;
link class travel = through: connection
   from departure to arrival;
link class stay from arrival to
   departure;
link class change from arrival to
   departure;
link class wait from departure to
   departure;
path class trip as travel (stay travel)*;
```

A graphical representation of this rather complex database schema is given in Figure 5. Here a path class is represented as a circle around its participating simple and link classes (omitting the more precise information in the path type); object-valued attributes are indicated by dashed lines.

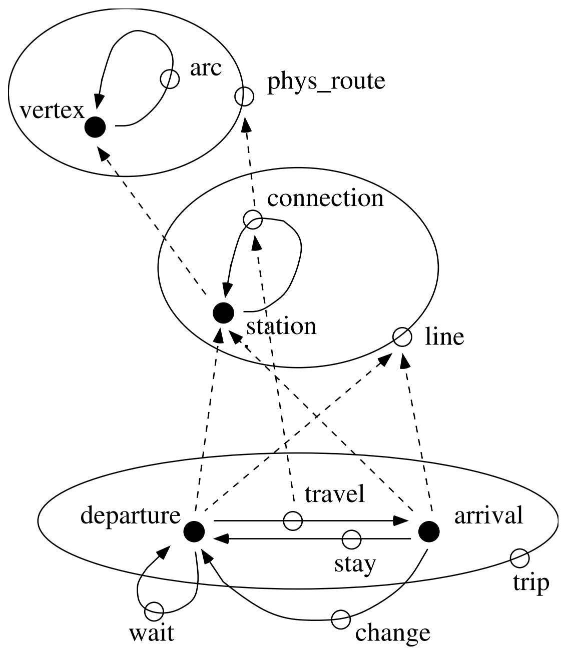

Figure 5

3 Queries

In this section we briefly discuss the concept of a query on a graph database, explain the structures the user manipulates in queries, and show some fundamental tools (statements, operations) for querying. We do not yet develop a complete query language but rather introduce some core elements. Also at this stage the semantics of operations are only described informally.

Query Concept. We would like to be able to conceptually modify the database graph in a query, for example, to add some edges computed by a query expression and then to apply a graph operation traversing old as well as new parts of the database graph, or to restrict the graph for consideration in a query. Therefore a query Q may consist of several *steps*, $Q = q_1; \ldots; q_m$. Each step may compute one or more classes of simple, link, or path objects. After each step, these classes are added to the database (and so, implicitly, extend the database graph). Or a step may express a restriction of the database graph for the following steps. Hence a graph operation used in step q_j "sees" the graph with the changes computed in steps $q_1, \ldots, q_{j-1}$. Examples of such *multistep queries* are given below.

Structures. What kind of structures at the conceptual level does a user create and manipulate in queries? Candidates might be graphs, sets of objects, nested relations, lists of object identifiers, etc. The design goal is to keep this collection simple but sufficiently expressive. It turns out that for our model four kinds of structures/objects suffice, namely a *uniform sequence of objects*, a *heterogeneous sequence of objects*, a (single) *object*, and a *value* of a data type.

A *uniform sequence of objects* contains a set of objects, usually from a single simple class, link class, or path class, in some, not necessarily specified, order. More precisely, the objects in the sequence may come from different classes but are all viewed under only *one* common tuple type. We use sequences rather than sets because it is then possible to offer operations in the query language making use of the order such as sorting, or taking head or tail of a sequence (see [GüZC89, MaV93]). Such a sequence of objects is the basic structure in formulating queries; since each object contains a tuple, it is the equivalent of a relation in the relational model. The most simple way to obtain a uniform sequence is just to write the name of a class. For example, writing "person" yields a sequence containing all person objects.

A *heterogeneous sequence of objects* may contain objects from several classes. These objects may have several different object as well as tuple types. For example, the node and edge objects forming a path in the database may be given as such a sequence. But more generally, heterogeneous collections of objects can be formed in queries and be manipulated in this form. The basic way to obtain a heterogeneous sequence is to write the name of several classes in angular brackets. For example, "<book, person, wrote>" yields a sequence of the corresponding objects from three different classes (Section 2.1).

What can one do with such a "mixed" collection of objects? First, there is a specialized and very powerful tool in the query language, called the *rewrite* operation, to deal with such sequences (Section 3.2). Second, in the same way as we have interpreted the set of simple, link, and path classes making up a database as a *database graph*, we will be able to interpret such collections as *graphs* or even as new parts of the database graph that are added in a query. Therefore, no specific types for graphs are needed in the user's conceptual model for querying. Third, one can apply a union operation in the query language to a heterogeneous sequence and so transform it into a uniform sequence (Section 3.3).

Tools for querying. The fundamental tools for querying a graph database are:

- The *derive statement*, which takes the role of the classical *select … from … where*, but has an extended meaning for graphs; it includes the functionalities of selection, join, projection, and function application;
- the *rewrite operation* as a basic tool for the manipulation of heterogeneous sequences; it allows to replace objects or subsequences by other (new) objects;
- the *union operation* for achieving "dynamic generalization", that is, for transforming a heterogeneous collection of objects into a homogeneous one, viewed under a common super tuple type;
- a collection of graph operations, e.g. *shortest path search.*

Additionally, the query language will contain further operations, e.g. for sorting, grouping, aggregate functions, data type operations, etc. In the following subsections we explain the four main tools listed above.

3.1 Derive

The *derive statement* is the most fundamental tool in the query language. Perhaps the best way to introduce it is to show a few examples. The first refers to the standard application from Section 2.1:

Q1. List the titles of all books written (coauthored) by Hopcroft in 1983!

```
on person wrote book
where person.name = "Hopcroft" and
   book.year = 1983
derive book.title
```

Here the on-clause says that each combination of *person*, *wrote*, and *book* objects should be considered where the *person* is connected by the *wrote* link to the *book*. From this collection of triples of objects the where-clause selects those fulfilling the two conditions. The derive-clause creates for each selected triple a new object with a single attribute called *title* whose value is taken from the attribute *title* of the *book* object in the triple. In this case, simple objects of an unnamed object type are created. It is also possible to create link objects, for example; other cases and the semantics in general are described below. The second example query is based on the public transport database from Section 2.4.

Q2. Make a listing of all departures from Dortmund main station in the form:

Time of departure	Type and number of train	End station and arrival time
6.13	IC 615	München 14.23
6.22	D 308	Wiesbaden 12.18
...		

The time of departure can be found within a departure event. Type and number of train correspond to a line type and a line number. The name of the final destination is either in the last node of a line path or can be found from the last event of a trip path. However, only the last event of a trip also contains the arrival time. The query can be formulated as follows:

```
on departure at_station station, departure
   of_line line, departure in trip
where station.name = "Dortmund"
derive departure.time, line.line_type,
   line.line_no,(trip end).at_station.name,
   (trip end).time
```

Here in the on-clause all combinations of *departure* events, *stations*, *lines* and *trips* are formed where (i) the *departure* object is connected through its object-valued attribute *at_station* with the *station* object (that is, *departure.at_station* = *station*), (ii) the *departure* object is connected through attribute *of_line* with the *line* object, and (iii) the *departure* object is a node in the path of the *trip* object. Note that in queries object-valued attributes can be used quite in the same way as link objects. There is some freedom here for the user to specify such connections in the on-clause or e.g. within predicates of the where-clause. In this example, it would be possible to reduce the on-clause to `departure in trip` and to require in the where-clause `departure.at_station.name = "Dortmund"` (and to access the `line` object similarly).

There is nothing new in the where- and derive-clauses except of the use of a function *end* to get from a path object its last node object.

Let us now consider syntax and semantics of the derive-statement in general. It has the following form:

```
<derive-statement> ::= [ <range
   declarations> ] on <subgraph-spec>
   [ where <condition> ] derive <object-
   spec>
```

Range declarations are needed to feed into a derive statement the result of a query expression. They are not further discussed here (see [Gü94]), but an example occurs in query Q5 below. A *subgraph specification* has the form:

```
<subgraph-spec> ::= <pattern> | <pattern> ,
   <subgraph-spec>
<pattern> ::= <var-intro>
   | <simple-var-intro> <link-var-intro>
   <simple-var-intro>
   | <simple-var-intro> in <path-var-intro>
   | <link-var-intro> in <path-var-intro>
   | <var-intro><attribute-name><var-intro>
<var-intro> ::= <object-type> | <object-
   type>(<newname>)
```

A subgraph specification is a list of *patterns*. A pattern either introduces just a single variable or it connects two or three variables in various ways, requiring that a simple object is connected through a link object to another simple object, a simple object occurs as a node within a path object, a link object occurs as an edge within a path object, or a simple object has another simple object as an attribute value, respectively. A variable is *introduced* by either writing the name of an object type (class name) which is then used as a variable, or by introducing a new name explicitly, for example, in the form "*state*(*s*1)". This is needed when several variables range over the same class.

In the evaluation of the on-clause all possible assignments of objects to the variables are considered and those tuples of objects determined that are simultaneously consistent with all patterns. In general, the patterns in the on-clause describe one or more connected graphs (if we draw an edge between two variables linked in a pattern). If two or more graphs are present, it means that the cartesian product of the possible object tuples for each graph will be formed. Therefore, if each pattern is just a variable name, then we have the classical cartesian product operation or a join, if there are connecting conditions in the where-clause. If there is only one pattern which is a single variable, then we have a simple selection. The where-clause contains just a condition, used to filter tuples of objects coming from the on-clause. The derive-clause specifies how a resulting set of objects is to be formed, in the following form:

```
<object-spec> ::= <variable>
   | [ <newname> = ] <attribute-spec-list>
   | <newname> [ = <attribute-spec-list> ]
   from <variable> to <variable>
<attribute-spec-list> ::= <attr-spec> |
   <attr-spec> , <attribute-spec-list>
```

```
<attr-spec> ::= <variable>.<attr-name> |
   <newname>: <expression>
```

The object-specification can either be one of the variables of the derive statement, which means that these objects are put into the result sequence; so the whole statement amounts to a more or less complex selection. The other case is that new objects are formed. For these a new object type (name) may be given by the "<newname> =" part; if it is omitted, a name will be selected internally by the system (which is obviously unknown to the user and can therefore not be used in the rest of the query). Next, attributes for the new objects are defined. The first form is "<variable>.<attr-name>" in which case the new attribute name as well as the value is taken from the object denoted by the "<variable>". The other possibility is to explicitly introduce an attribute name and to assign to it by an arbitrary expression a value of a data or object type. So far, if the from-to-part is omitted, simple objects will be created. If the from-to-part is present, then the variables must refer to simple classes. In this case link objects are created connecting the corresponding pairs of objects assigned to the from- and to-variables. Creation of path objects in the derive statement has so far not been provided in the design.

The following example illustrates several points, namely the use of explicit variables, the formulation of a classical join in the derive statement, the creation of link objects, and the dynamic modification of the database graph in a multistep query. We assume the following database to be given:

```
class state = sname: STRING; region:
   REGIONS;
```

Here REGIONS is a spatial data type describing a polygonal region. The query is:

Q3. How many countries must be traversed traveling (by land) from Germany to China?

```
on state(s1), state(s2)
where s1.region adjacent s2.region
derive neighbour_of = cblength:
   length(common_border(s1.region,
   s2.region) from s1 to s2;

state("Germany") state("China")
shortest_path[neighbour_of+]
rewrite[state -> , neighbour_of ->
   neighbour_of] count
```

This is a multistep query; the first step is the derive statement which constructs a set of *neighbour_of* edges and adds them to the database graph; the second step uses these edges to find a shortest path from Germany to China. Here we only discuss the derive statement, the second step will be explained below when *rewrite* and *shortest-path* operations have been introduced. Explicit variables are used because two variables range over the same class. The patterns in the on-clause describe two independent graphs. So a cartesian product is formed which together with a subsequent selection condition amounts to a join (*adjacent* is a geometric predicate applicable to two REGIONS values). The derive-clause creates a new link object class *neighbour_of* between any two qualifying *state* objects; as an attribute of such a link the length of the common boundary is computed, using two geometric data type operations *length* and *common_border*. This is just to give an example of creating attribute values for a derived link class, the attribute is not needed for this query.

3.2 Rewrite

The *rewrite* operation is a very powerful tool for dealing with heterogeneous sequences and in particular, to manipulate paths (which are heterogeneous sequences). One simple way to use it is similar to case-statements in programming languages, since one can specify a treatment separately for each object type that may come along in a heterogeneous sequence. But it is also possible to apply transformations to whole subsequences of a given sequence. In this case the order of elements in the sequence plays a crucial role; to understand this order, *path types* (or, more generally, *sequence types*) defined in Section 2.3 are essential. Again, let us introduce the rewrite operation by a few examples. First, consider the second step of query Q3:

```
state("Germany") state("China")
shortest_path[neighbour_of+]
rewrite[state -> , neighbour_of ->
   neighbour_of] count
```

Here the *shortest_path* operator (whose arguments will be explained below) computes a shortest path from one state object (Germany) to another one (China). The two state objects are determined by a special "object identification" notation (see [Gü94]). The result is a heterogeneous sequence of *state* and *neighbour_of* objects. The result has a *sequence type* (which is equivalent to a path type, but also mentions the types of node objects)

```
state neighbour_of state (neighbour_of
   state)*
```

We abbreviate this sequence type as SNS(NS)*. Such a sequence is input to the rewrite operation which contains a list of *transformations*. Each transformation has a *left side* and a *right side* (separated by an arrow). The left side of a transformation is a *pattern* which is a list of one or more variables denoting object types. The right side is either an expression which must evaluate to an object, or empty. The meaning is roughly that whenever a subsequence of objects is encountered matching one of the patterns, then the corresponding transformation is applied (a more precise definition is given in [Gü94]). Hence in our example, the effect is that all objects of type *state* in the sequence are thrown away whereas all *neighbour_of* objects are moved unchanged into the result sequence. So rewrite can be used to realize a *type restriction* on a heterogeneous sequence.

Applying rewrite, one should keep track of the manipulation of the sequence type that it implies. In our example, the result sequence will have type NN*, that is

```
neighbour_of (neighbour_of)*
```

The second example is again based on the public transport database from Section 2.4.

Q4. List all direct connections from Dortmund to München with the distance traveled. That means, provide a table of the form:

Departure time	Arrival time	Distance
6.13	14.23	610 kms
7.13	15.23	610 kms
7.43	16.26	578 kms

...

To answer this query, all levels of the public transport network are needed. The query can be formulated as follows:

```
on departure at_station station(s1),
   arrival at_station station(s2), departure
   in trip, arrival in trip
where s1.name = "Dortmund" and s2.name =
   "Muenchen"
derive dtime: departure.time, atime:
   arrival.time, distance:
   trip suffix(departure) prefix(arrival)
   rewrite
      [departure ->, arrival ->, stay ->,
      travel -> travel_dist = (dist:
         travel.through.way
         rewrite[vertex -> ,
            arc -> arc_length = (len:
            length(arc.route))]
         sum[len])]
   sum[dist]
order_by[dtime +]
```

This is already a fairly complex problem; it is still possible to formulate the query in a relatively concise way. The derive statement finds *trip* objects containing stops in Dortmund and München. In the derive-clause objects with three attributes are produced, called *dtime*, *atime*, and *distance*, where *distance* is computed as follows: Each trip path is reduced by operations *suffix* and *prefix* to the part between Dortmund and München. These are operations of the query language for the manipulation of sequences; the argument is besides the sequence an object, and the sequence is reduced to the part after and including the object in case of suffix, similarly the part before the object for prefix. An object of a path class can be treated directly as a sequence, hence these operations are applicable to *trip* objects. The remaining part of the trip sequence is handled by a rewrite: *departure*, *arrival*, and *stay* objects are thrown away; *travel* objects are transformed into new *travel_dist* objects with a single attribute called *dist*, whose value is computed as follows. From the *travel* object via its *through* attribute the underlying *connection* object is reached, from which via attribute *way* the corresponding *phys_route* path object is obtained. We are now at the level of the physical network. Here the path of the form

```
vertex arc vertex (arc vertex)*
```

is again treated by a rewrite; *vertex* objects are thrown away and for each *arc* object a new *arc_length* object with a single attribute *len* is created whose value is determined by applying a function *length* to the *arc* object's *route* attribute (of data type LINES). Hence the result of the inner rewrite is a uniform sequence of *arc_length* objects; *sum* is an aggregate function applicable to such a sequence. The result is a single number which is finally assigned to the *dist* attribute of the new *travel_dist* object. Again *sum* is applied to a uniform sequence of *travel_dist* objects to obtain a number which is then used as the *distance* attribute value in the objects created by the derive statement. In a final step, the sequence of (unnamed) objects returned from derive is sorted by departure time (*dtime*).

For lack of space, in this paper we cannot further elaborate on the rewrite operation. A definition of syntax and semantics can be found in [Gü94] where also an example of a more sophisticated manipulation of sequence types ("sequence rewrite programming") is shown. In that respect the examples of this section are trivial since all patterns in rewrite operations consist only of a single variable (no subsequences are replaced).

3.3 Union

The *union* operation makes it possible to transform a heterogeneous sequence of objects into a uniform one, so that all objects in the sequence are viewed under a common tuple type. It does that by computing the smallest common super (tuple) type for the tuple types of the heterogeneous sequence. Consider the following example database:

```
class city = name: STRING, region:
   REGIONS, pop: INTEGER;
class village = name: STRING, position:
   POINT, pop: INTEGER;
class river = name: STRING, way: LINE;
```

We can, for example, form the union of cities and villages:

```
<city, village> union
```

The result is a uniform sequence with a tuple type

<(name, STRING), (region, GEO), (pop, INTEGER)>

because this is the smallest common supertype of the tuple types of *city* and *village* objects (see Section 2.2). The new tuple type can be used in the rest of the query, as in the following example.

Q5. List the names of all cities, villages, and rivers within Bavaria! (We assume that Bavaria has been introduced before as the name of a REGIONS value.)

```
range of cvr is <city, village, river>
   union,
on cvr
where cvr.region inside Bavaria
derive cvr.name
```

In this case the tuple type resulting from the *union* operation is <(name, STRING), (region, GEO)>. The geometric predicate *inside* has a signature GEO × REGIONS → BOOL, hence it is applicable to *region* attribute values of type GEO. This "dynamic generalization" feature is of particular importance for spatial databases where often collections of objects need to be formed that are just related by their spatial attributes (e.g.

lie in the same area). For more motivation, see [Gü91, ErG91].

3.4 Graph Operations

In a way, we arrive now at the main goal of the development of the GraphDB data model: to be able to formulate graph operations and to integrate them in a clean way into querying. This is possible because the database has a well-defined and explicit graph structure. In this section we do not yet describe a comprehensive collection of useful graph operations – this is a major task left to future work – as the purpose of this paper is to develop the right environment for the integration of such operations. But we show two examples. The first is an operation for finding shortest paths which has already been used in example query Q3. It takes two simple objects, which are used as the start and target nodes of the search, respectively, and returns a shortest path from the start to the target node in the form of a heterogeneous sequence. Further parameters are given in square brackets behind the operator name:

- a *path type*, which identifies those parts of the database graph that may be used in the search and defines a precise structure for the resulting sequence (for rewriting manipulations),
- for each class of edges (link objects) that may occur in the path according to the path type, a function assigning a cost to this edge. If such a function is not given as a parameter, a constant edge cost of 1 is assumed as a default.
- for each class of nodes (simple objects) that may occur in the path according to the path type, a function giving an estimated distance from this node to the target node. The reason this parameter is needed is that for the implementation of *shortest_path* the A* algorithm (see [Ni80]) will be used, which needs to estimate the distance from the target for nodes encountered in the search. As a default the function yielding constantly 0 is used. For A* to work correctly it is required that such a function must *underestimate* the distance to the target; with this function that is trivially true in which case A* reduces to Dijkstra's algorithm.

A further example query with a shortest path search is given below. The *subgraph* operation restricts the database graph for the following steps of a query. The argument (in square brackets) is a list of restrictions of the form "<classname> **where** <condition>". One can mention simple classes, link classes, or path classes. The semantics is that for the following steps of the query for each class that is mentioned only the objects qualified by the condition are part of the database graph. If, for example, a node class is restricted, then also only the edges incident with these nodes are present (or rather, visible) within the database; if edges are restricted, also paths going through "invalid" edges disappear. There is an inverse operation called *fullgraph* which restores the complete database graph for further steps of a query. Both *subgraph* and *fullgraph* form separate steps of a query. An efficient implementation of these operations is described in [Gü94] – by no means is it intended to make copies of the database graph.

We illustrate the use of the *subgraph* operation in connection with a more interesting example of a shortest path search. Consider the following query on the highway network (Section 2.1):

Q6. Find a shortest path from exit 16 to exit 252, avoiding a fog area described by a REGIONS value (a collection of polygons) *fog*!

```
subgraph[section where not
   (section.route intersects fog)];
exit(nr = 16) exit(nr = 252)
shortest_path[section+,
   fun(s: section)
   length(s.route)/s.top_speed,
   fun(v: vertex, target:vertex)
   dist(v.pos, target.pos)/200]
```

Here we have in the first step restricted to edges free from fog and in the second step computed a shortest path over these edges with respect to traveling time (assuming lengths and distances are stored and computed in kms, and top-speed in kms/hour). The syntax for denoting function parameters (defined in [Gü93]) is a variant of typed lambda calculus.

4 Conclusions

We have presented a data model that integrates an explicit modeling of graph structures smoothly into a "standard" object-oriented modeling and querying environment. In particular, explicit path objects are offered, and graph operations can be defined whose argument graphs (subgraphs of the database graph) can be specified by regular expressions over link class names. The derive statement extends the familiar *select … from… where* to a convenient querying of relationships (link classes, edges of the graph). The rewrite operation is a powerful tool for the manipulation of sequences, especially paths, in queries. The model is coupled to an implementation concept which offers special data structures for the representation of graphs and efficient graph algorithms for the graph operations. System architecture and implementation strategy are described in the full paper [Gü94]. Besides being attractive for standard applications, the model is particularly suitable for a sophisticated modeling and manipulation of spatially embedded networks, as has been demonstrated by the public transport example.

We are currently developing a first partial prototype for GraphDB following the system architecture and implementation plan described in [Gü94]. The general extensible query processing environment will be offered by SECONDO – a system based on the second-order signature concept described in [Gü93] – which is just about to be finished. To reduce the implementation effort (that is, to make the task manageable at all) we are trying to use as much as possible modules from the Gral system [Gü89], for example, storage and buffer management, index structures, data types, implementations of query processing operations (e.g. join algorithms). In a first phase, we would like to arrive at a prototype version that demonstrates some interesting part of the query processing capabilities needed for GraphDB. To realize the GraphDB query language as such, it is necessary to implement a SECONDO optimizer (perhaps along the lines of [BeG92]) which is still a major open task. The development of the GraphDB model and prototype is part of the ESPRIT project AMUSING.

Other future work includes a more complete design of the query language – in this paper we have only introduced some key elements –, the design of a corresponding SOS model level signature, and a formal definition of the semantics of query language operations. Note that the extensible system architecture makes it possible to postpone a complete query language design even until the system is running; missing operations can always be added later.

Acknowledgments

Helpful discussions with Hieu-Thien Pham and Frank Schoppmeier and their work on implementing the GraphDB prototype are gratefully acknowledged. Thanks also to Martin Erwig, Hieu-Thien Pham and Markus Schneider for reading drafts of this paper, and to the referees for their comments.

References

[Ag87] Agrawal, R., ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries. Proc. IEEE Data Engineering Conf. 1987, 580-590.

[AmS92] Amann, B., and M. Scholl, Gram: A Graph Data Model and Query Language. Proc. ECHT'92, Milano, December 1992.

[Andr92] Andries, M., M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche, Concepts for Graph-Oriented Object Manipulation. Proc. 3rd EDBT 1992 (LNCS 580), 21-38.

[BeG92] Becker, L., and R.H. Güting, Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. *ACM Transactions on Database Systems 17 (1992)*, 247-303.

[BiRS90] Biskup, J., U. Räsch, and H. Stiefeling, An Extension of SQL for Querying Graph Relations. *Computer Languages 15 (1990)*, 65-82.

[CoM90] Consens, M., and A. Mendelzon, GraphLog: A Visual Formalism for Real Life Recursion. Proc. ACM Conf. on Principles of Database Systems 1990, 404-416.

[CoM93] Consens, M., and A. Mendelzon, Hy$^+$: A Hygraph-based Query and Visualization System (Video Demonstration). Proc. ACM SIGMOD 93, 511-516.

[CrMW87a] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, A Graphical Query Language Supporting Recursion. Proc. SIGMOD Conf. 1987, 323-330.

[CrMW87b] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, G$^+$: Recursive Queries Without Recursion. Proc. 2nd Intl. Conf. on Expert Database Systems, 1989, 355-368.

[CrN89] Cruz, I.F., and T.S. Norvell, Aggregative Closure: An Extension of Transitive Closure. Proc. of the 5th Intl. Conf. on Data Engineering, 1989, 384-391.

[ErG91] Erwig, M., and R.H. Güting, Explicit Graphs in a Functional Model for Spatial Databases. FernUniversität Hagen, Informatik-Report 110, 1991, to appear in *IEEE Transactions on Knowledge and Data Engineering*.

[GePTV93] Gemis, M., J. Paredaens, I. Thyssens, and J. van den Bussche, GOOD: A Graph-Oriented Object Database System (Video Demonstration). Proc. ACM SIGMOD 93, 505-510.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Data Bases, 1989, 33-44.

[Gü91] Güting, R.H., Extending a Spatial Database System by Graphs and Object Class Hierarchies. In: G. Gambosi, H. Six, and M. Scholl (eds.) Proc. Int. Workshop on Database Management Systems for Geographical Applications (Capri, May 1991), Springer, 1992, 34-55.

[Gü93] Güting, R.H., Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization. Proc. ACM SIGMOD Conf. (Washington, 1993), 277-286.

[Gü94] Güting, R.H., GraphDB: A Data Model and Query Language for Graphs in Databases. Fernuniversität Hagen, Report 155, 1994.

[GüZC89] Güting, R.H., R. Zicari, and D.M. Choy, An Algebra for Structured Office Documents. *ACM Transactions on Information Systems 7 (1989),* 123-157.

[GyPV90a] Gyssens, M., J. Paredaens, and D. van Gucht, A Graph-Oriented Object Database Model. Proc. ACM Conf. on Principles of Database Systems 1990, 417-424.

[GyPV90b] Gyssens, M., J. Paredaens, and D. van Gucht, A Graph-Oriented Object Model for Database End-User Interfaces. Proc. ACM SIGMOD Conf. 1990, 24-33.

[Kung86] Kung, R., *et al.*, Heuristic Search in Database Systems. In: L. Kerschberg (ed.), Expert Database Systems. Benjamin Cummings, 1986.

[MaS90] Mannino, M., and L. Shapiro, Extensions to Query Languages for Graph Traversal Problems. *IEEE Trans. on Knowledge and Data Engineering 2 (1990)*, 353-363.

[MaV93] Maier, D., and B. Vance, A Call to Order. Proc. ACM Symposium on Principles of Database Systems (Washington, 1993), 1-16.

[Ni80] Nilsson, N.J., Principles of Artificial Intelligence. Tioga Publ. Company, Palo Alto, CA, 1980.

[OrM88] Orenstein, J., and F. Manola, PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Trans. on Software Engineering 14 (1988)*, 611-629.

[RoFS88] Rossopoulos, N., C. Faloutsos, and T. Sellis, An Efficient Pictorial Database System for PSQL. *IEEE Trans. on Software Engineering 14 (1988)*, 639-650.

[Rose86] Rosenthal, A., S. Heiler, U. Dayal, and F. Manola, Traversal Recursion: A Practical Approach to Supporting Recursive Applications. Proc. SIGMOD Conf. 1986, 166-176.

[StR86] Stonebraker, M., and L.A. Rowe, The Design of POSTGRES. Proc. of the 1986 SIGMOD Conf. (Washington, DC, May 1986), 340-355.

[SvH91] Svensson, P., and Z. Huang, Geo-SAL: A Query Language for Spatial Data Analysis. Proc. SSD 91 (Zurich, Switzerland), 1991, 119-140.