Hecataeus: A Framework for Representing SQL Constructs as Graphs

George Papastefanatos[1], Kostis Kyzirakos[1], Panos Vassiliadis[2], Yannis Vassiliou[1]

[1] National Technical University of Athens,
Dept. of Electrical and Computer Eng.,
Athens, Greece
{gpapas, kkyzir, yv}@dbnet.ece.ntua.gr

[2] University of Ioannina,
Dept. of Computer Science,
Ioannina, Greece
pvassil@cs.uoi.gr

Abstract. Traditional modeling techniques typically focus on the static part of databases and ignore their dynamic part (e.g., queries or data-centric workflows). In this paper, we first introduce and sketch a graph-based model that captures relations, views, constraints and queries. We then present HECATAEUS, a tool for implementing and visualizing the above framework.

Keywords: Database Modeling and Query Languages, Meta-modeling

1. Introduction

Traditional database modeling techniques, like ER diagrams, UML, etc., have been widely used in modeling database entities and relationships between them. Most of them, however, restrict themselves in explicitly modeling the main database parts (e.g., entities, relationships) of an information system, while ignoring components that interface with the database, such as queries, views, stored procedures, applications, etc. An ER diagram, for example, can describe in a precise way how data is to be stored and treated within a database, but cannot tell what is happening "around" the database in terms of queries, or how information flows through the components that interface with the database. This kind of knowledge is valuable to database administrators and designers, since it can be used for several purposes, including (a) the forecasting of the impact of changes in the system (e.g., what happens if we delete a certain attribute of a table?), (b) the visualization of the workload of the system (e.g., which queries pose the heaviest load on the system?) and (c) the evaluation of the quality of the database design.

So far, research has provided visualization techniques for queries [BaOO02], [CCLB97], [JaTh03], [MuGP98], [HFLP89], [PaKi95]; nevertheless, to our knowledge, a graph-based, uniform way to represent tables and queries has not been introduced yet. In this paper, we provide a simple model for representing databases and queries in a coherent way. We employ a graph theoretic approach and we map the aforementioned constructs to graphs. Moreover, we describe the internal architecture of a tool that is currently under implementation, HECATAEUS, that visualizes the

aforementioned constructs in a user friendly way. Finally, we demonstrate the usage of the tool over the OLTP example of TPC-C [TPCC].

Our contributions can be listed as follows:

- We introduce and sketch a graph-based model for an extended system catalog, capturing relations, views, constraints and queries in a cohesive framework [PaVV04].
- We describe a tool for automating the analysis of a database system and representing and visualizing its characteristics to the aforementioned graph-based model.

This paper is organized as follows. In Section 2, we sketch the graph model for database systems. We present the main features and the architecture of HECATAEUS framework in Section 3 and model the TPC-C example in Section 4. Finally, in Section 5 we provide insights for future work.

2. A Graph-based Model for SQL Constructs

In this section, we present a graph modeling technique that uniformly covers relational tables, views, database constraints and SQL queries as first class citizens [PaVV04]. The proposed technique provides an overall picture not only for the actual database schema but also for the architecture of a database system as a whole, since queries are incorporated in the model. Moreover, we distinguish the following essential components, which are included in our model: *relations, conditions* (covering database constraints and query conditions)*, queries* and *views*. The proposed modeling technique represents all the aforementioned database parts as a directed graph. Graphs are employed as a modeling technique because they can address the large size and complexity that characterize a database schema. In the rest of this section we discuss how we model each of these constructs as well as the overall graph of the database.

Relations. Each relation $R(A_1,A_2,\ldots,A_n)$ in the database schema is represented as a directed graph, which comprises (a) a *relation node*, representing the relation schema, (b) n *attribute nodes*, one for each of the attributes, and (c) n *schema edges* directing from the relation node towards the attribute nodes, indicating that the attribute belongs to the relation. We call these relationships, *schema relationships*.

Conditions. Conditions refer both to constraints of the database schema and selection conditions of queries and views. We consider two classes of atomic conditions (a) `A op` constant and (b) `A op A′`, where `A`, `A′` are attributes of the underlying relations and `op` is a binary operator (i.e., <, >, =, ≤, ≥, `!=`, `IN`, etc.).

In our graph model, a *condition node* is used for the representation of the condition. The node is tagged with the respective operator and it is connected to the two *operand nodes* of the conjunct clause through the respective *operand edges*. Composite conditions are easily constructed by tagging the condition node with an `AND` or an `OR` symbol and the respective edges (possibly more than two) to the conditions composing the composite condition.

Well-known constraints of database relations are easily captured by this modeling technique. Foreign keys are subset relations of the source and the target attribute,

range constraints are simple value-based conditions. Unique-value constraints (hence, primary keys) require a different modeling – in the context of this paper we explicitly represent them through a dedicated node and a single operand node.

Queries and Views. Queries and views are mapped to graphs in a similar manner. The graph representation of the query involves a new node representing the query, named *query node*, and *attribute nodes* corresponding to the schema of the query. The query graph is therefore a directed graph connecting the query node with all its schema attributes, via *schema edges*. In order to represent the relationship between the query graph and the underlying relations, we make the convention that each query is decomposed into the following essential parts: the `SELECT`, `FROM`, `WHERE` and `GROUP BY` parts, each of which is eventually mapped to a subgraph. Based on the above, the graph representation of a query is the composition of the three subgraphs, which are defined as follows:

- Each query/view is assumed to own a schema that comprises the named attributes appearing in the `SELECT` clause. In this context, the `SELECT` part of the query maps the respective attributes of the involved relations to the attributes of the query schema through a *map-select edge*, directing from the query attributes towards the relation attributes.
- The `FROM` part of a query can be regarded as the relationship between the query and the relations involved in this query. Therefore, for each relation included in the `FROM` part, a *from edge*, directing from the query node towards the relation node, is used.
- We assume that the `WHERE` clause of a query is in conjunctive normal form. Having explained conditions already, we can construct the graph corresponding to the `WHERE` clause of a query by introducing a directed *where edge* starting from the query node towards the operator node corresponding to the conjunction of the highest level.
- For the representation of aggregate queries, we employ two special purpose nodes: (a) a new node denoted as `GB`, to capture the set of attributes acting as the aggregators and (b) one node per aggregate function (e.g., `COUNT`, `SUM`, `MIN`, etc.) labelled with the name of the employed aggregate function. For the aggregators, an edge is used, directing from the query node towards the `GB` node, and labelled `<group-by>`, indicating *group−by* relationship. Then, the `GB` node is connected with each of the aggregators through an edge tagged also as `<group-by>`, directing from the `GB` node towards the respective attributes. These edges are tagged according to the order of the aggregators (i.e., `1` for the first aggregator, `2` for the second, etc.). Moreover, for every aggregated attribute in the query schema, there exists an edge directing from this attribute towards the aggregate function node as well as an edge from the function node towards the respective relation attribute. Both edges are labelled `<map-select>`, as this relationship indicates mapping of the query attribute to the corresponding relation attribute through the aggregate function node.

The following example demonstrates the proposed graph representation. Assume a query `Q` on three self-explanatory relations from [GMRR01], involving employees and the projects they work for. The database system `S` comprising the following relations and the query are:

```
Emp(Emp#,Name,Address,Age,Salary)
Works(Emp#,Proj#,Hours)
Proj(Proj#,Projname,Leader#,Location,Budget)
```

```
Q:  SELECT E.Emp# as ID,Name as NAME,P.Projname as PRJNAME
    FROM Emp E,Works W,Proj P
    WHERE P.EMP#=W.EMP# AND W.Proj#=P.Proj#
```

In Figure 1, we present this graph representation for system S, comprising the relations, constraints and the internal constructs for the above query.

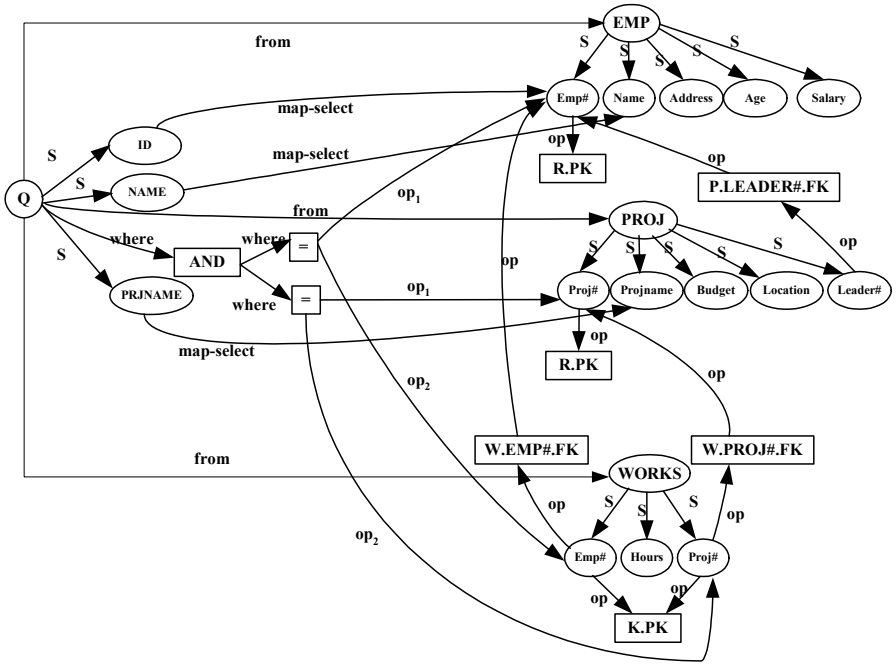

Figure 1: Graph representation of query Q

As far as modification queries are concerned, there is a straightforward way to incorporate them in the graph, too. Still, their behavior with respect to adaptation to changes in the database schema can be captured by SELECT queries. For lack of space, we simply mention that (a) INSERT statements can be dealt as simple SELECT queries and (b) DELETE and UPDATE statements can be treated as SELECT queries comprising a WHERE clause.

Definition 1. The *Database Graph-Model* is a directed graph G=(V,E) where V is a finite set of nodes and E a finite set of directed edges. The set V comprises the following kinds of nodes: *relation nodes, attribute nodes, query nodes, condition nodes, operator nodes, constant nodes* and *group by nodes.* The set E comprises the directed edges of the graph, classified as: *schema edges, map-select edges, where edges, operand edges, from edges* and *group by edges.*

3. Implementation of the Framework

In the context of the aforementioned graph model, we have prototypically implemented a framework for graphical representation, HECATAEUS. The user defines the file that contains the DDL definitions, and the file that contains the queries accessing the underlying schema. The tool creates and presents a graph that holds all the semantics of the nodes and edges of the aforementioned graph model. Moreover, the tool assists the user in several ways: apart from the zoom-in/zoom-out capability, HECATAEUS offers the ability to the user to isolate and highlight a part of a graph, in order to facilitate the accurate and complete understanding of the whole system. A distinctive feature of the HECATAEUS, is the ability to define graph algorithms and metrics to the graph, or to a specific sub-graph.

The framework's architecture consists of the coordination of HECATAEUS' four main components: the *Parser*, the *Graph Manager*, the *Metrics Manager* and the *catalog*.

- The *Parser* first parses the DDL file, provided as input, and passes each command to the database, where the tables are created, and then to the Graph Manager. Secondly, it parses the SQL script file, also provided as input, to the Graph Manager.
- The functionality of the *Catalog* is to maintain the schema of the relations as well as to validate the syntax of the queries parsed, before they are modeled by the Graph Manager.
- The *Graph Manager* component is responsible for representing the underlying database schema and the parsed queries in the proposed graph model. The Graph Manager holds all the semantics of nodes and edges of the aforementioned graph model, assigning nodes and edges to their respective classes. The Graph Manager communicates with the repository and the parser and constructs the node and edge objects for each class of nodes and edges (i.e., relation nodes, query nodes, etc.). It also processes each query to the database in order to validate it against its syntactical correctness and then creates its graph – constructing the respective subgraphs (i.e., select, from, where, group by subgraphs). Also, the Graph Manager allows distinct colorization for each set of nodes and edges, and offers the user the ability to edit the graph elements. The Graph Manager allows the direct manipulation of the graph components so that the user may change the way that the graph is presented (including labels for edges and nodes, structure of the graph etc.) in order to create different views/perspectives of the graph. Finally, the Graph Manager offers the ability for modularization or abstraction of the graph, by isolating and highlighting a part of the graph (i.e., a set of queries operating on a relation) or hiding specific classes of edges and nodes, respectively.
- Lastly, the *Metrics Manager* component is responsible for the definition and application of metrics and algorithms on the graph.

In Figure 2, we present the overall architecture of the proposed framework.

HECATAEUS is implemented in the MS Visual Studio .NET platform [V.NET], [TeOl03]. For the parser and the database engine, we have used SharpHSQL, an open source database engine [HSQL], whereas for the graph visualization we have used the LEDA C++ class library [LEDA].

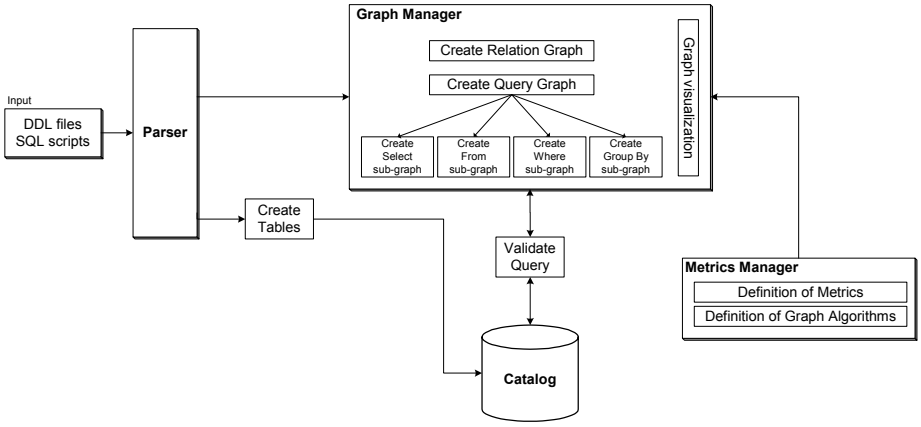

Figure 2: HECATAEUS architecture

4. Case study: Modeling TPC-C benchmark

In this section we present the representation of the TPC-C benchmark [TPCC] to the proposed graph model. TPC-C benchmark comprises five types of transactions, which are carried out through respective stored procedures in the database. Each of these transactions operates against a database of nine tables through a set of 27 queries, performing update, insert, delete and select operations. According to the proposed graph model, the TPC-C example is mapped to the graph shown in Figure 3.

The tables of the TPC-C benchmark are represented by relation nodes (displayed as blue oval-shaped nodes), labeled with the name of each table, and connected through *schema* edges (tagged with an S) to the respective attribute nodes (displayed as gray oval-shaped nodes). Queries are represented by query nodes (displayed as red circle-shaped nodes), labeled with the name of each query[1], which are connected through *from* edges (displayed as red directed edges) with the relation nodes. Finally, each query subgraph comprises, apart from the query node, the attribute nodes representing the attributes of the schema of the query (displayed as gray oval-shaped nodes), the *where* and *operand* edges as well as the *operand* nodes (displayed as green square-shaped nodes) representing the operands in the *where* clause of the query. In the top-most corner of Figure 3, a more detailed view of a part of the TPC-C graph (isolated and transformed by the framework) is presented, which represents the query: `SELECT * FROM HISTORY`.

The TPC-C graph of Figure 3 allows us to stress some useful remarks:

- The TPC-C graph consists of a set of non-connected subgraphs, (basically due to the fact that there are no foreign keys imposed on these relations).
- Each subgraph comprises the respective relation subgraph(s) along with the query subgraphs operating on these relations. Queries, doing join operations

[1] According to the order with which each query is parsed by the framework, it is automatically assigned a name, i.e., Q_1 for the first query parsed, Q_2 for the second query parsed, etc.

on relations, act as bridges between these relation subgraphs (e.g., right-most subgraph of Figure 3).

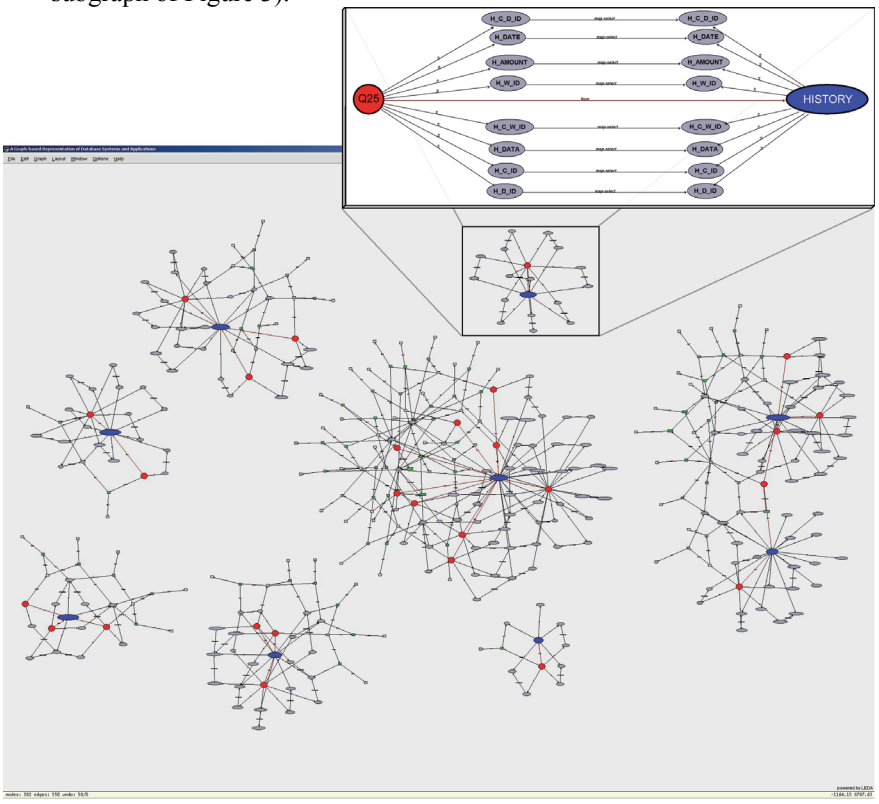

Figure 3: TPC-C graph

- Simpler subgraphs (i.e., comprising small numbers of nodes and edges) correspond to relations with smaller schemata (fewer attributes) on which a small number of queries operate (e.g., history relation which is accessed by only one query), whereas more complex subgraphs indicate relations being accessed by a large number of queries (e.g., customer relation which is accessed by 9 queries).

5. Discussion

In this paper, we have presented HECATAEUS, a tool for visualizing a graph-based model that that uniformly captures relations, views, constraints and queries. Modeling queries and relations through graphs can be used by database administrators and designers for several purposes, including (a) the visualization of both the structure and, interestingly, the workload of the system, (b) the forecasting of the impact of changes, and (c) the evaluation of the quality of the database design.

Simple visualization has great value, by its own. For example, highlighting highly used attributes is useful for workload balancing and impact forecasting in the presence of changes. As another example, observing the frequent correlation of two tables possibly might lead the database administrator to the definition of a new view.

The smooth evolution around the database of an information system can be supported by our framework through the detection of node *dependencies*, at all levels of abstraction (attribute, relation, or query). The graph model aids both visually and algorithmically to this end. Algorithms can easily be applied to prevent *attribute/table deletions* that would lead to semantic or syntactic inconsistencies of deployed, query-based modules (forms, reports, etc) around the database. Also, our work in progress includes algorithms that can be used to determine (or even, automate) the adaptation of such modules towards *additions* (or updates) to the database schema.

The definition of graph-based database metrics for the evaluation of the quality of a graph is also an issue that we currently investigate. For example, when in dilemma, the designer can isolate parts of the graph and test alternatives. The minimization of graph complexity possibly implies the minimization of the maintenance effort, too.

References

[BaOO02] N. H. Balkir, G. Ozsoyoglu, Z. M. Ozsoyoglu. A Graphical Query Language: VISUAL and Its Query Processing, IEEE Trans. Knowl. Data Eng. 14(5): 955-978 (2002)

[CCLB97] T. Catarci, M. F. Costabile, S. Levialdi, C. Batini: Visual Query Systems for Databases: A Survey. J. Vis. Lang. Comput. 8(2): 215-260 (1997)

[GMRR01] A. Gupta, I. S. Mumick, J. Rao, K. A. Ross. Adapting materialized views after redefinitions: Techniques and a performance study. In Information Systems, 26 (2001), p.323-362

[HFLP89] L. M. Haas, J. C. Freytag, G. M. Lohman, H. Pirahesh: Extensible Query Processing in Starburst. SIGMOD Conference 1989: 377-388

[HSQL] SharpHSQL - An SQL engine written in C#, http://www.c-sharpcorner.com/database/SharpHSQL.asp

[JaTh03] H. Jaakkola, B. Thalheim: Visual SQL - High-Quality ER-Based Query Treatment. ER (Workshops) 2003: 129-139

[LEDA] LEDA. C++ class library of efficient data types and algorithms, http://www.algorithmic-solutions.com/enleda.htm

[MuGP98] N. Murray, C. A. Goble, N. W. Paton: A Framework for Describing Visual Interfaces to Databases. J. Vis. Lang. Comput. 9(4): 429-456 (1998)

[PaKi95] A. Papantonakis, P. J. H. King: Syntax and Semantics of Gql, a graphical query language. J. Vis. Lang. Comput. 6(1): 3-25 (1995)

[PaVV05] G. Papastefanatos, P. Vassiliadis, Y. Vassiliou. A Graph-based Representation of Database Systems and Applications. Working Draft 2005. Available upon request.

[TeOl03] Templeman, J. and Olsen, A., Microsoft Visual C++ .NET Step By Step, Microsoft Press, 2003.

[TPCC] Overview of the TPC Benchmark C: The Order-Entry Benchmark. http://www.tpc.org/tpcc/

[V.NET] Ms Visual Studio .NET, http://msdn.microsoft.com/vstudio/