

©Copyright by Dmitry Zelenko, 1997.

QUERY MODELS IN DATABASE-ORIENTED SOFTWARE DEVELOPMENT

BY

DMITRY ZELENKO

M.S., Applied Mathematics, Belarusian State University, 1996

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

## ABSTRACT

We survey possible methodologies of database-oriented software development and investigate the ramifications of adopting the database-oriented approach. The *query model* formalism is introduced and analyzed. We give its pragmatic equivalent using the notion of query object from [11]. We present several efficient algorithms and exhibit power of reification. We also describe a graphical user interface for editing query models.

## **ACKNOWLEDGMENTS**

I am grateful to my advisor Ralph Johnson for his continual help and council. I would also like to thank my office-mates from the Business Modeling group Jeff Barcalow, John Brant, Jee Ku, and Joseph Yoder for creating a wonderful environment for work and research.

# TABLE OF CONTENTS

CHAPTER	Page
<b>1 INTRODUCTION</b> . . . . .	1
<b>2 MODELS IN DATABASE SOFTWARE DEVELOPMENT</b> . . . . .	2
2.1 Conceptual Modeling . . . . .	3
2.2 Database Modeling . . . . .	5
2.3 Application Modeling . . . . .	6
2.4 Model Interplay in System Engineering . . . . .	7
<b>3 QUERY MODEL</b> . . . . .	10
3.1 Relational Database Theory . . . . .	10
3.2 Query Model Architecture . . . . .	13
3.3 Reflective Query Model . . . . .	18
3.3.1 Computing Transitive Closure . . . . .	19
3.4 Query Object Framework . . . . .	20
3.5 Updates . . . . .	22
<b>4 EDITING QUERY MODELS</b> . . . . .	26
<b>5 CONCLUSIONS AND FURTHER RESEARCH</b> . . . . .	32
<b>REFERENCES</b> . . . . .	33

# LIST OF FIGURES

Figure	Page
2.1 Models and Dependencies . . . . .	8
3.1 Query Model Example . . . . .	14
3.2 Query to Be Added to Query Model . . . . .	17
3.3 Resultant Query Model . . . . .	18
3.4 Query Object Class Hierarchy . . . . .	21
3.5 Query Model and Updates . . . . .	22
3.6 Query Model for Saving . . . . .	23
4.1 Constructed Query Model . . . . .	27
4.2 TableQuery Creation . . . . .	27
4.3 JoinQuery Creation . . . . .	28
4.4 SelectionQuery Creation . . . . .	29
4.5 Expression Editor . . . . .	30
4.6 Expression Node Editor . . . . .	31
4.7 ProjectionQuery Creation . . . . .	31

## LIST OF ALGORITHMS

1	Adding Query to Query Model . . . . .	16
2	Saving Data through Query Qbjects . . . . .	25
3	Polymophic Saving Data through Query Objects . . . . .	25

# CHAPTER 1

## INTRODUCTION

Two important subfields of software engineering are databases[14] and data modeling[12]. The fields have developed their own methodologies, which often conflict with one another. For example, some of the data modeling methodologies are based on competing philosophical assumptions, and database design uses a number of formalisms.

Such a multitude of views and paradigms is a blessing, rather than a curse. Since our world, including the realm of software development, is not uniform, we can choose the pieces of approaches that would best suit the needs of a software project. We advocate such a relativistic approach and drift carefully among conflicting theories in describing both philosophical and mathematical foundations, as well as certain pragmatic issues of database-oriented software development.

The manuscript is structured as follows. Chapter 2 presents the philosophical underpinnings of database-based software development and examines several vying theories in the data modeling community. In particular, we adopt the subjectivist view of the world and follow the database-oriented approach of system engineering. Chapter 3 introduces a query model built atop a relational model, describes its existing pragmatic object-oriented equivalent, gives several efficient algorithms, and outlines a possible extension of relational algebra. Chapter 4 describes software for editing query models.

# CHAPTER 2

## MODELS IN DATABASE SOFTWARE DEVELOPMENT

This chapter divides information system development into three stages, each with its own model. These are the *conceptual model*, the *database model*, and the *application model*. The information system is supposed to model some particular problem domain, and each stage corresponds to a particular view of this problem domain. To illustrate the stages of system development and models used for each stage, we consider the following example.

A group of developers has embarked on a project to develop a warehouse information system. The warehouse can be used by different companies for storing their products. The information system should register all shipments to and from the warehouse as well as have information on the products and companies involved. The project begins with careful analysis of the domain, interviews with warehouse workers and representatives of the companies. In other words, the team of developers learns the domain and the requirements of the party needing the product. As a result, the developers and the customers produce an initial draft that embodies knowledge of the domain relevant to the project. This draft is the *conceptual model*. It is usually a combination of textual

and graphical representations. It is designed to provide a clear description of the domain and the goals of the project.

Next the draft is taken by database developers, who use it for designing a relational *database model* encompassing the data necessary for the project. It is a collection of tables, indexes, constraints, physical parameters and other database-specific information. The database also describes the warehouse; however, the description is directed towards efficient computer processing. The database model developers can also point out the ways to alter the conceptual model and, perhaps, reorganize the warehouse operations.

Having been given the database and conceptual model, the application developers maintain a close contact with customers to better understand their needs and design the application accordingly. The architecture is based upon the building blocks, which represent the working language of application developers. This can be a set of abstract data types, objects, existing frameworks, patterns, etc. Such a set along with relationships within it is termed an *application model*. Knowledge of application developers can “back-propagate” to database and conceptual models as well. Finally, part of the application model needs to be connected to the existing database. This can be accomplished using the ideas and tools of chapter 3.

We now elaborate each of the models.

## 2.1 Conceptual Modeling

Conceptual modeling is usually labeled “data modeling” in the computer literature. However, the “data model” term is ambiguous[9]. It refers to both the product of application modeling techniques represented in some notation, and the language used for constructing the model. We will employ the term in the former sense, to avoid confusion.

The goal for creating a conceptual model is to capture the meaning of a domain and represent it in a form that is amenable to computer processing.

There are two basic competing philosophical theories serving as a basis for conceptual modeling. The philosophical assumptions give different answers to the question, “What is the goal of the model?”.

- *Objectivist view*[6]. According to the objectivist view, the goal of the model is to represent truth of the domain. The view is based on the assumption that there is a true meaning of the modeled phenomenon(e.g. organization). In practice this view results in transferring responsibility for system specifications to project managers. Management is deemed most qualified to determine software project ends, and it is empowered to specify the objectives for database designers and application developers. The conceptual model constructed from system specifications is considered fairly static, and it is rarely modified in the process of system development.
- *Subjectivist view*[5] emphasizes dynamics of the environment and absence of the *true* meaning of the domain. The goal of the model is to represent common understanding of the modeled organization. The organization (and, hence, its model) are understood and endowed with meaning via collective perception. Since the understanding changes, the model is subject to change. People who hold this view stress communication between management, database, and application programmers. Project objectives are stated collectively, and they are usually altered as the project continues.

The second dichotomy in data modeling is between static and dynamic modeling[10]. The latter stresses the flow of information in the system, whereas the former is oriented toward depicting the system structure. Thus, the static modeling paradigm pushes information flow issues to further stages of software development.

The main disadvantage of the subjectivist view is that it leads to frequent updates of the conceptual model and its database model. Very often the updates of the database are not desirable because of their high cost. Therefore, the developers can change the application model leaving the database intact. A major goal of the thesis is to facilitate such changes by providing a unified framework and tools for performing the changes.

We are concerned primarily with static modeling. Our approach is structural, not behavioral. We believe that behavioral changes for the most part preserve structural invariants. Therefore, the term “model” refers to the structural composition with semantic meaning as given by its creator.

## 2.2 Database Modeling

Informally, a database is a pool of structured information. Databases existed since appearance of records and medium for their storage. Such medium could be clay tablets in Phoenicia, parchment scrolls in medieval Europe, or punch cards of the early twentieth century. In these examples, the representation of data is very closely associated with the physical objects holding them.

Computers allowed separation of data representation from the physical media. Surprisingly though, progress was slow. Up to the early seventies data were considered bound to their storage devices. Writing programs in most programming languages required knowing formats and parameters of storage medium for manipulating the data.

The 1970s were marked by the appearance of several competing proposals for languages for describing and manipulating data. Those were network, hierarchical, and relational models. The first two are record-based whereas the third one is set-oriented. The heated debate closed up in the early 1980s with a victory of the relational model and its widespread acceptance for production by commercial vendors[10].

The database model has differences from the conceptual model in several ways. As has been noted above, the conceptual model is intended to capture the meaning of a domain and be used by people for understanding, facilitating communication, and developing other kinds of models. This implies that it is informal and visual. The database model, in contrast, is used to represent data in a computer. Hence, it should have a sound formal basis. The database model also should provide for efficient data manipulation. Therefore, model clarity can be traded for model efficiency.

The differences in emphasis of the conceptual and database models are often overlooked by practitioners. The conceptual model is created for humans, it then should be transformed into a database model. Sadly, most available products aiming at combining conceptual and database modeling usually suffer a shift to the database side, often “helping” construct virtually incomprehensible conceptual models. One example is the popular IDEF1X notation[3], which equates a conceptual “entity” with a database table and provides pure database-oriented mechanisms for organizing and structuring entities.

## 2.3 Application Modeling

A goal of a software engineer is to create an application to satisfy specified needs. The application shall have its own vocabulary with both structural and behavioral properties. These properties are determined to some extent by the programming language chosen for implementation. However, we believe that the structural properties of the vocabulary can be abstracted and made language-independent. Different languages will provide different levels of abstraction and encapsulation, but most languages support abstract data types and are amenable to our investigation. We view the object-oriented paradigm as most appropriate for expressing an application model and choose an object-oriented language(Smalltalk) for implementing the framework described in the subsequent sections.

However, we do not wish to restrict ourselves to object-oriented languages because we are not concerned with behavioral properties of objects. Thus, the results can be applied to any language possessing abstract data types.

## 2.4 Model Interplay in System Engineering

This section considers two possible scenarios of application development: database-oriented and application-oriented.

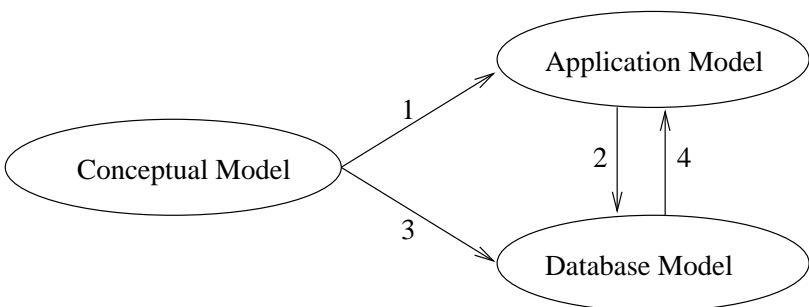
Producing a software component is an iterative process. It can be useful to distinguish the basic steps of designing the data models under consideration in the process.

Creating a conceptual model is usually one of the first steps of a software project. During that step developers acquire a language to communicate about the domain. The step can be repeated whenever new qualities or earlier neglected structure resurfaces to refine the conceptual model.

After developing the conceptual model developers move to the next step. Which model is to be created at that step? The answer to this question depends on the nature of the software product being developed.

Figure 2.1 depicts the relationship between the models as well as points two ways for engineering them.

- *Application-Oriented(1-2)*. This approach is commonly taken by a programmer, who needs a rich and flexible vocabulary for constructing a complex application with a medium-size database. The database is then created from the application with use of heuristic rules and patterns(see [2]). The derived model hardly lends itself to optimization or obvious association with the original conceptual model[1]. This perspective suits application developers the best, though it burdens database designers and administrators.



**Figure 2.1** Models and Dependencies

- *Database-Oriented(3-4)*. This direction is taken whenever database design and optimization is of primary importance. Several applications can be foreseen to be constructed on the database so neither shall be given preference in the database design. It is even quite common to design and maintain huge databases with no application in mind(for example, data warehouses, legacy systems). Thus, the database-oriented approach is then the only one possible. It involves two steps:
  1. Generate a database model from the conceptual model.
  2. Generate an application object model from the database model, perhaps, using the conceptual model.

The first step, as we pointed out above, is semi-automatic, and involves creating a formal representation for the “ideas” of the original human-oriented conceptual

schema. A database designer should be also aware of applications to be designed on top of the database and strive to optimize the database with respect to the applications. The major goal, however, is to adequately reflect the meaning and structure of the conceptual model in the database.

The second step assumes the existence of a stable database. The application structural model is to be constructed on the database to provide an intermediate level suited for that particular application.

# CHAPTER 3

## QUERY MODEL

### 3.1 Relational Database Theory

We sketch the foundations of the relational model using notational conventions close to Maier[13].

Consider a set of *attributes*  $\mathbf{A} = \{A_1, A_2, \dots, A_n\}$

Each attribute  $A_i$  has a domain  $D_i = D(A_i)$ ,  $1 \leq i \leq n$  of *values* it can take.

**Definition 3.1 Cartesian Product** of domains  $D_{i_1}, D_{i_2}, \dots, D_{i_k}$  written as  $D_{i_1} \times D_{i_2} \times \dots \times D_{i_k}$  is the set of all tuples  $(v_1, v_2, \dots, v_k)$  such that  $v_j \in D_{i_j}, i_j \in \{1, 2, \dots, n\}, 1 \leq j \leq k$ .

**Definition 3.2 Relation**  $R$  over attributes  $A_1, A_2, \dots, A_n$  is any subset of  $D_1 \times D_2 \times \dots \times D_n$  and denoted  $R(A_1, \dots, A_k)$ .

We assume that a relation is always *finite*.

We will denote a tuple  $t$  of a relation  $R(A_1, A_2, \dots, A_k)$  as  $\langle t_1, t_2, \dots, t_k \rangle$ , where  $t_j \in D_j, 1 \leq j \leq k$ .

**Definition 3.3** A Relational Schema  $S$  over attributes  $A_1, A_2, \dots, A_n$  is a set of relations  $R_1, R_2, \dots, R_m$  written as  $S[R_1, R_2, \dots, R_m]$ , where for each  $R_i(A_{i_1}, A_{i_2}, \dots, A_{i_{l(i)}})$  we have that  $A_{i_j} \in \{A_1, A_2, \dots, A_n\}, 1 \leq j \leq l(i)$ .

A relation  $R(A_1, A_2, \dots, A_k)$  as defined above can be viewed as an extension of a  $k$ -ary predicate symbol  $r$ . So  $R$  may be considered as a model of a first-order logic logical formula  $r(x_1, x_2, \dots, x_k)$ . This correspondence between first-order logic and relations is very useful and we will refer to it frequently.

The relational algebra consists of the following five operations on relations.

1. Let  $R_1$  and  $R_2$  be two relations over the same set of attributes. The *union* of  $R_1$  and  $R_2$ , denoted  $R_1 \cup R_2$ , is the set of tuples that are in  $R_1$  or  $R_2$  or both.
2. Let  $R_1$  and  $R_2$  be two relations over the same set of attributes. The *difference* of  $R_1$  and  $R_2$ , denoted  $R_1 - R_2$ , is the set of tuples that are in  $R_1$  but not in  $R_2$ .
3. Let  $R_1$  and  $R_2$  be relations of arity  $k_1$  and  $k_2$ , respectively, over possibly different sets of attributes. The *Cartesian product* of  $R_1$  and  $R_2$  denoted  $R_1 \times R_2$  is the set of all  $(k_1 + k_2)$ -tuples whose first  $k_1$  components form a tuple in  $R_1$ , and whose second  $k_2$  components form a tuple in  $R_2$ .
4. The *projection* of  $R(A_1, A_2, \dots, A_k)$  on  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$ , where  $A_{i_j} \in \{A_1, A_2, \dots, A_n\}, 1 \leq j \leq n$ , is a relation over attributes  $A_{i_1}, A_{i_2}, \dots, A_{i_n}$  denoted  $\Pi_{A_{i_1}, A_{i_2}, \dots, A_{i_n}}(R)$ , which is a set of  $n$ -tuples  $\langle a_{i_1}, a_{i_2}, \dots, a_{i_n} \rangle$  such that there is a  $k$ -tuple  $\langle b_1, b_2, \dots, b_k \rangle \in R$  for which  $a_{i_j} = b_{i_j}, 1 \leq j \leq n$ .
5. Let  $F$  be a formula in first-order logic including comparison operators  $<, >, \leq, \geq, \neq$  with atoms from  $\{A_1, A_2, \dots, A_n\}$  and respective domains. The *selection*  $\sigma_F(R(A_1, A_2, \dots, A_n))$  is a relation consisting of all the tuples in  $R$  satisfying  $F$ .

Additional operations:

1. *Intersection*:  $R_1 \cap R_2 = R_1 - (R_1 - R_2)$ .

2.  *$\theta$ -join*: Let  $R_1(A_1, A_2, \dots, A_n)$  and  $R_2(B_1, B_2, \dots, B_m)$  be two relations. Let  $\theta$  be a first-order logic formula including comparison operators  $<, >, \leq, \geq, \neq, =$  with atoms from  $\{A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m\}$  and respective domains. Then  *$\theta$ -join*  $R_1 \bowtie_{\theta} R_2 = \sigma_{\theta}(R_1 \times R_2)$ .

If a set of comparison operators in  $\theta$  includes only  $=$  then the operation is often called *equijoin*.

3. *Natural join*:  $R_1 \bowtie R_2$  — applicable only if  $R_1(A_1, A_2, \dots, A_n)$  and  $R_2(B_1, B_2, \dots, B_m)$  have a nonempty set  $I$  of common attributes. Let  $C = \{A_1, A_2, \dots, A_n\} \cup \{B_1, B_2, \dots, B_m\}$ . The fact that  $C$  is a set guarantees that it has only one “copy” of attributes from  $I$ . Then:  $R_1 \bowtie R_2 = \Pi_C(R_1 \bowtie_F R_2)$ , where  $F = \bigwedge (R_1.i = R_2.i), i \in I$ .

In the real world, relational algebra operations are applied to a database through a *SQL query interface*.

The standard SQL query:

```
select A1, A2, ..., An
from R1, R2, ..., Rm
where <condition >
```

maps to a relational algebra expression:

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_{\langle \text{condition} \rangle}(R_1 \times R_2 \times \dots \times R_m))$$

An example: Database schema:

*Student*(name, course, grade)

*Professor*(name, course)

An example SQL query returning students getting A's in classes of professor whose name is John:

```
select Student.name
from Student, Professor
where (Professor.name = 'John')
      and (Student.grade = 'A')
      and (Student.course = Professor.course)
```

## 3.2 Query Model Architecture

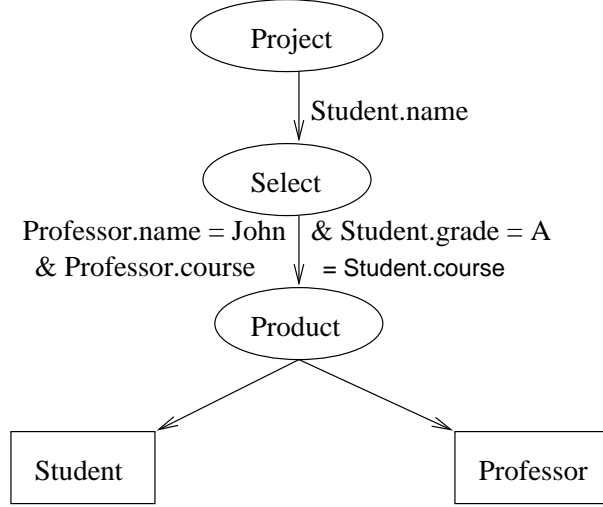
The query model is a result of query abstraction. Usually programmers use queries as declarative statements for extracting information from a database. Queries are used once to be discarded and later rewritten as the need for them arises. Some queries, however, are executed often enough to make such rewriting undesirable. The queries can be stored and retrieved for execution. As the number of such queries increases, disjointed storage of the queries becomes redundant. It is possible to use the structure of relational algebra to organize the queries into a graph-like structure, avoid redundancy and provide for their efficient use.

An example of the structure is shown in Figure 3.1. The example is a query model representation of the database schema and query introduced in section 3.1.

We now formally define the query model using the graph theory terminology of Cormen[7].

**Definition 3.4** *A query model  $QM$  is a five-tuple  $(S, G, q, e, t)$ , where*

- $S[R_1, R_2 \dots, R_m]$  is a relational schema over some set of attributes  $\mathbf{A}$ .



**Figure 3.1** Query Model Example

- $G = (V, E)$  is a finite directed acyclic graph with the set of vertices  $V$  and the set of edges  $E$ .
- $q : V \rightarrow \{Table, Union, Difference, Project, Select, Product\}$  is a label on each of the graph vertices.
- $e : E \rightarrow For \cup 2^{\mathbf{A}}$  is a label on the graph edges, a first-order formula over  $\mathbf{A}$  (an element of  $For$ ) or a subset of the set of available attributes (including the empty set).
- $t : \{v : v \in V, q(v) = Table\} \rightarrow \{R_1, R_2, \dots, R_m\}$  labels each  $Table$ -node with a relation from the schema  $S$ .

We also require a query model to satisfy the following constraints:

1. No edge leaves a  $Table$ -vertex:  $\forall v (q(v) = Table \rightarrow deg^+(v) = 0)$   
( $deg^+(v)$  is the number of edges leaving the vertex  $v$ ).
2. Edges leaving  $Select$ -vertices are labeled with first-order formulas:  
 $\forall (u, v) \in E (q(u) = Select \rightarrow e((u, v)) \in For)$ .

Edges leaving *Project*-vertices are labeled with attributes:

$$\forall(u, v) \in E \ (q(u) = \textit{Project} \rightarrow e((u, v)) \in 2^{\mathbf{A}} - \emptyset).$$

Other edges are not labeled:

$$\forall(u, v) \in E \ (q(u) \notin \{\textit{Select}, \textit{Project}\} \rightarrow e((u, v)) = \emptyset).$$

3. The connected component of a node  $v$  of the graph is a directed tree with  $v$  as its root. The leaves of the tree are *Table*-nodes. The tree is the abstract syntax tree of the node(Figure 3.1). It corresponds to a single query. Therefore, it is possible to generate a query for each node of the query model.

We now deal with changing a query model. A set of queries is rarely stable: new queries can be added to the query model. Therefore, we devise an efficient algorithm for adding queries. In the algorithm we assume that the query to be added is represented in the query model form. Some notational conventions employed in the algorithm:

- $V(QM)$ ,  $E(QM)$  refer to the set of vertices or edges of the query model  $QM$ , respectively.
- $parent_{QM}(v)$  gives the set of vertices with an edge to  $v$  in the graph of the query model  $QM$ . The function is also extended to work on sets of vertices.
- $\Leftarrow$  is an assignment operation.

The algorithm starts with the *Table*-nodes of the query model  $QM$  and the query  $Q$  to be added. It moves up the abstract syntax tree corresponding to the query and adds to  $QM$  the vertices and edges of  $Q$  that are not present in  $QM$ . For each iteration of the algorithm, *CurrentLevelNodes* is the set of nodes of the query  $Q$  whose children have already been added to the query model  $QM$ . *CurrentLevelNodes* are partitioned into the nodes that are embedded in  $QM$ (*EmbeddedNodes*) and the nodes that need to be added to  $QM$ (*NotEmbeddedNodes*). The steps 7-8 of the algorithm determine which

---

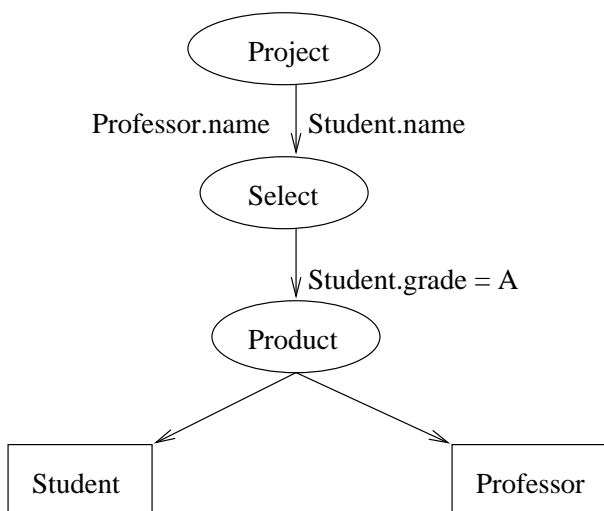
**Algorithm 1** Adding Query to Query Model

---

**Require:** a query model  $QM$ , a non-empty query  $Q$

**Ensure:**  $QM$  includes  $Q$

- $\{CurrentLevelNodes$  is the set of nodes of  $Q$  whose children have already been added to the query model  $QM\}$   
 $\{EmbeddedNodes$  is the subset of  $CurrentLevelNodes$  whose connected components are in  $QM\}$   
 $\{NotEmbeddedNodes$  is the complement of  $EmbeddedNodes$  in  $CurrentLevelNodes$ .  $NotEmbeddedNodes$  need to be added to  $QM\}$
- 1:  $ModelTableNodes \leftarrow \{v : v \in V(QM) \wedge q(v) = Table\}$
  - 2:  $QueryTableNodes \leftarrow \{v : v \in V(Q) \wedge q(v) = Table\}$
  - 3:  $EmbeddedNodes \leftarrow ModelTableNodes \cap QueryTableNodes$
  - 4:  $NotEmbeddedNodes \leftarrow QueryTableNodes - EmbeddedNodes$
  - 5:  $V(QM) \leftarrow V(QM) \cup NotEmbeddedNodes$  {update  $QM$ }
  - 6:  $CurrentLevelNodes \leftarrow QueryTableNodes$
  - 7: **while**  $parent_Q(CurrentLevelNodes) \neq \emptyset$  **do**
    - $EmbeddedNodes \leftarrow \{v : v \in parent_Q(EmbeddedNodes) \wedge$   
 $\exists w \in parent_{QM}(EmbeddedNodes) \wedge$   
 $(q_Q(v) = q_{QM}(w) \wedge$   
8:  $\forall (v, u) \in E(Q)$   
 $(u \in EmbeddedNodes \wedge$   
 $(w, u) \in E(QM) \wedge$   
 $e((w, u)) = e((v, u)))\}$
  - 9:  $NotEmbeddedNodes \leftarrow parent_Q(CurrentLevelNodes) - EmbeddedNodes$   
{add not embedded nodes and edges to  $QM$ :}
  - 10:  $V(QM) \leftarrow V(QM) \cup NotEmbeddedNodes$
  - 11:  $E(QM) \leftarrow E(QM) \cup \{(u, v) : (u, v) \in E(Q) \wedge u \in NotEmbeddedNodes\}$   
{move one level up:}
  - 12:  $CurrentLevelNodes \leftarrow parent_Q(CurrentLevelNodes)$
  - 13: **end while**
-



**Figure 3.2** Query to Be Added to Query Model

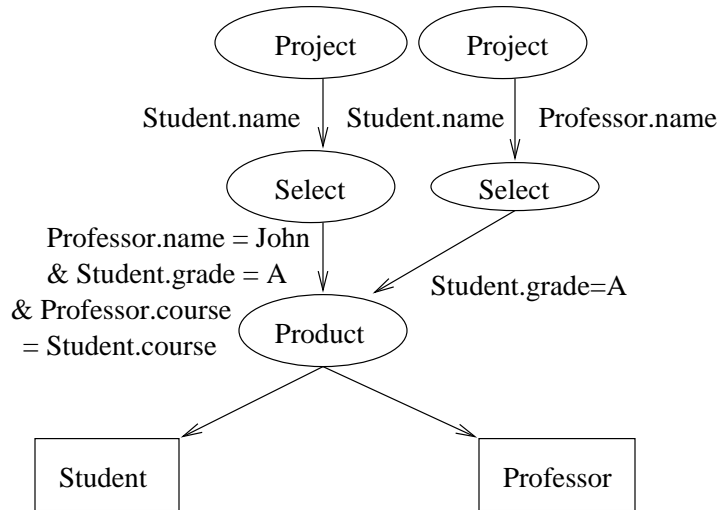
of *CurrentLevelNodes* are *EmbeddedNodes*. For each parent  $v$  of already embedded nodes of  $Q$  the algorithm looks for a corresponding node  $w$  of  $QM$  with the same label and children as those of  $v$ . If such a  $w$  is found, the node  $v$  is added to *EmbeddedNodes* and identified with  $w$ ; otherwise,  $v$  is one of *NotEmbeddedNodes* to be added to  $QM$ .

The running time of algorithm 1 is  $O(n * indegree_Q * indegree_{QM} * outdegree_Q)$ , where  $n$  is the number of relational algebra operations in  $Q$ ,  $indegree_Q(indegree_{QM})$  is the maximal entering degree of nodes in  $Q(QM)$ , and  $outdegree_Q$  is the maximal number of edges leaving a node in  $Q$ . It is easily seen from the observation that the number of iterations in the algorithm is bounded by the number of relational operators in  $Q$ , whereas the steps 7-8 have complexity  $O(indegree_Q * indegree_{QM} * outdegree_Q)$ .

Consider an example of adding a query to a query model using algorithm 1.

The base relational schema is defined on page 12. The initial query model is depicted in Figure 3.1. The query  $Q$  to be added to the query model is

$$\Pi_{Student.name, Professor.name}(\sigma_{Student.grade='A'}(Student \times Professor))$$



**Figure 3.3** Resultant Query Model

Its query model representation is shown in Figure 3.2. Execution of Algorithm 1 yields the query model in Figure 3.3. The rightmost *Project*-node corresponds to the query  $Q$ .

Special caution should be given to removing parts of a query model since a node in a model can serve as a part in more than one queries. More precisely, a node can be safely removed from a query model, if no edge enters the node.

In general, the query model representation appears to be efficient and easily organizable. We describe the visual interface for editing query models in chapter 4

### 3.3 Reflective Query Model

One of the advantages of representing an application model as query model is that we can ameliorate basic limitations of the relational algebra without degrading its mathematical elegance. The limitations of the relational algebra are well known. One cannot, for example, express transitive closure over relations in relational algebra.

An approach to overcoming the limitations lies in the structure of the query model described above. Since it represents a labeled graph, it is amenable to relational representation too. That implies that a query model itself can be represented as a relation and, consequently, stored in a database. In other words, the set of queries determined by an application programmer is reified[4]. Consequently, queries of the query model can be named, treated as functions mapping relations to relations, and referenced by a programmer in other queries.

In order to be stored in a database, a query model needs to be encoded in some way. A number of possible encodings are possible([8]). We do not wish to concentrate on the technical details of a query model encoding. Instead, we present a method for extending the functionality of relational algebra using reification of a query model.

### 3.3.1 Computing Transitive Closure

We will show an example of how reflection can help compute transitive closure of a relation.

Let  $Structure(Parent, Child)$  be a relation. We would like to find all of the descendants of a given parent. It is known that it cannot be done in ordinary relational algebra or SQL([14]). However, if a query defined on  $Structure$  is reified, it is possible to equate the query with a function that maps  $Structure$  into its subset.

Define the following query(function):

```
Desc(x) := select Desc(Child) from Structure
          where parent = x
          union
          select Child from Structure
          where parent = x
```

We will argue that  $Desc(x)$ , indeed, computes all descendants of a parent  $x$ . It may seem at the first glance that the definition of the query is not valid since it is recursive. However, we should keep in mind that  $Desc(x)$  is reified, that is, named, represented as a query object (see Section 3.4) or stored in a database and retrieved whenever necessary. Therefore, use of  $Desc(Child)$  in the select clause of the query would result in the query  $Desc(x)$  made concrete and being executed for the given value of  $Child$ .

The execution of the function  $Desc(A)$  proceeds as follows:

1. Get the records of the *Structure* table whose *Parent* field is equal to  $A$ .
2. If the resulted set of records is not empty, evaluate the function  $Desc$  with the value of the *Child* field for each of the records. Return the union of the evaluations.
3. Return the results of the previous step combined with the immediate children of  $A$  in the *Structure* table.

The finite relation assumption (see page 10) as well as transitivity and anti-reflexivity of the descendant relationship imply that the execution time of the function  $Desc$  is finite.

This approach preserves the declarative syntax of relational algebra. It only extends the range of functions that can be used in the relational algebra expressions with query functions. Of course, any application using such query functions is bound to implement their interpreter.

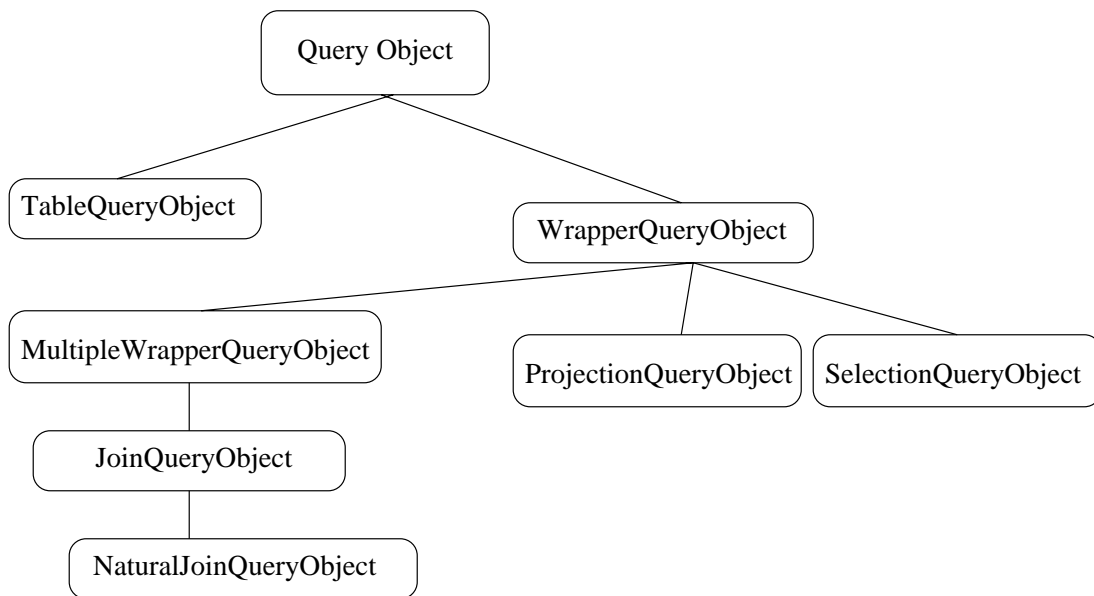
### 3.4 Query Object Framework

This section is based on the work of Brant and Yoder [11]. It is an object-oriented equivalent of the query model architecture described in section 3.2.

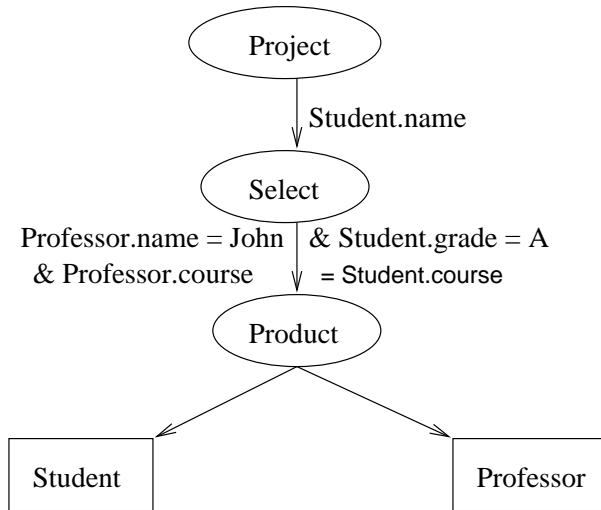
This framework has been born as a natural result of solving the problem of integrating object-oriented languages and relational databases. The solution follows the direction

suggested in [15], which considers a parallel between relational views[14] and objects. The idea is that each existing relational database model is given an object data model(a collection of objects and relationships among them) that equates each object in the model with a view performed upon the relational database. Since an object can be treated as an abstract data type, the set of its values becomes precisely the one associated with a view the object is defined upon. The set of operations, however, is unrestricted and determined by application requirements.

The next step is to free a programmer from the database query interface for creating objects. This can be done by providing a set of classes that operate on the meta-level for the object data model and serve as building blocks for constructing view-object mappings. We call them query objects. They are based on relational algebra operations(see Section 3.1). As classes in an object data model, they are embedded in the hierarchy of classes. A sub-hierarchy of classes corresponding to basic relational algebra operations is presented in Figure 3.4.



**Figure 3.4** Query Object Class Hierarchy



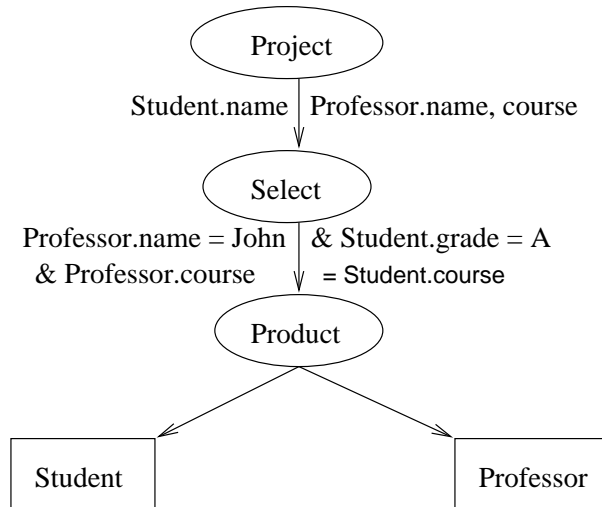
**Figure 3.5** Query Model and Updates

*QueryObject*, *WrapperQueryObject*, and *MultipleWrapperQueryObject* are abstract classes carrying basic components for mapping objects to views, applying relational algebra operations to single query, and multiple queries respectively. *TableQueryObject* relates to a single relation in a database. *ProjectionQueryObject*, *SelectionQueryObject*, *JoinQueryObject*, and *NaturalJoinQueryObject* are standard building blocks corresponding to relational algebra operations. Instantiation and composition of subclasses of *QueryObject* will allow us to construct descriptions of objects in an object data model that would map to relational views built upon an available relational database.

The framework has been successfully used to provide a dynamic object-oriented interface to relational databases.

## 3.5 Updates

Having designed a query data model, we can use it for accessing data in the relational database and performing necessary operations on the application level. A problem arises, however, when we attempt to use the data model for saving data in the database.



**Figure 3.6** Query Model for Saving

As an example, consider saving the tuple (*Mark*) in the *Student-Professor* database via the query depicted in Figure 3.5. Informally, we have to “propagate” the tuple from the the root to the leaves of the query tree of Figure 3.5 and end up with two tuples to be stored in the *Student* and *Professor* tables. However, none of the tuple fields is in the *Professor* table. Moreover, the field *name* of the *Student* table does not include the table key(see below). Consequently, the “propagated” tuples cannot be stored in the *Student* and *Professor* tables, and so the query in Figure 3.5 is inappropriate for saving. The algorithm presented below requires that the “propagated” records be saved in every table of the query; otherwise, the save fails.

To illustrate the case when a tuple can be saved, consider the following query for the same database:

```
select Student.name, Professor.name, course
from Student, Professor
where (Professor.name='John') and (Student.grade='A')
      and (Professor.course = Student.course)
```

The tuple to be saved via the query is (*Mark John CS497*). The query tree is shown in Figure 3.6. In this case, we can split the given tuple into two: (*Mark CS497*) and (*John CS497*). The latter is a *whole* tuple for the *Professor* table and can be stored right away. (*Mark CS497*) is not a *whole* tuple for the *Student* table; however, its fields (*name* and *course*) comprise the key of the table (see below), so the remaining field (*grade*) can be left undefined, and the tuple (*Mark CS497 NULL*) can be stored in the *Student* table.

The formal description of the algorithm below uses polymorphism extensively. That makes the algorithm distributed over the hierarchy of query objects depicted in Figure 3.4.

**Problem:** given a query object  $O$  and a tuple  $t$  for  $O$ , we need to save  $t$  in the database using  $O$ .

For simplicity assume the following:

- $t$  satisfies the description of  $O$  so no checking will be made.
- For each relation  $R \in RS$  there is a subset of its attributes  $\mathbf{K}_R \subseteq \mathbf{A}$ , such that  $\forall x, y ((x \in R) \wedge (y \in R) \wedge (\Pi_{K_R}(x) = \Pi_{K_R}(y)) \Rightarrow (x = y))$ ,  $\mathbf{K}_R$  is called a **key** of  $R$ .
- *TableQueryObject* associated with a relation  $R$  “knows” how to save a sub-tuple of  $t$  in  $R$ , if it includes all of the attributes that are in  $\mathbf{K}$  for  $R$ .
- For *JoinQueryObject* (and its subclass) the set of all its constituent queries is denoted as  $\mathbf{Q}$ . That is, if *JoinQueryObject*  $Q = Q_1 \times Q_2$ , then  $\mathbf{Q} = \{Q_1, Q_2\}$ . The attributes of a query  $Q_i$  are denoted as  $\mathbf{A}_i$ .
- For *WrapperQueryObject* (and its subclasses) the constituent query is denoted as  $Q$ . For example, *SelectionQueryObject*:  $\sigma_{\langle condition \rangle}(Q)$ .
- **rollback** rollbacks previous saves done by the algorithm.

In the algorithm below *FAIL* is a boolean variable that is assigned *true* whenever a save of a tuple fails.

---

**Algorithm 2** Saving Data through Query Objects

---

**Require:**  $O, t$   
**Ensure:**  $FAIL = true$  or  $t$  is saved  
 $FAIL \leftarrow false$   
 $FAIL \leftarrow$  Algorithm 3 in  $O$  with  $t$   
**if**  $FAIL$  **then**  
    rollback  
**end if**

---



---

**Algorithm 3** Polymorphic Saving Data through Query Objects

---

**Require:**  $t$   
**Ensure:**  $FAIL = true$  or  $t$  is saved

---

in *TableQueryObject*

**if**  $K_R \subseteq$  attributes of  $t$  **then**  
    save  $t$   
**else**  
     $FAIL \leftarrow true$   
**end if**

---

in *WrapperQueryObject*

$FAIL \leftarrow$  Algorithm 3 in  $Q$  with  $t$

---

in *JoinQueryObject*

**for all**  $Q_i \in \mathbf{Q}$  **do**  
     $FAIL \leftarrow$  Algorithm 3 in  $Q_i$  with  $\Pi_{A_i}(t)$   
    **if**  $FAIL$  **then**  
        exit  
    **end if**  
**end for**

---

Algorithm 3 “propagates” the tuple  $t$  down the query tree for  $O$  and saves the derived sub-tuples, if possible. If at least one save fails, *FAIL* is set to *true* and Algorithm 2 performs rollback.

# CHAPTER 4

## EDITING QUERY MODELS

As was noted in section 2.4 a programmer needs a vocabulary of entities to operate with. We assumed that the entities extensionally represent queries defined over the available database. Section 3.2 precisely specified the structure of the collection of queries and outlined efficient algorithms for working with it. This chapter describes a tool for helping programmer create and manipulate the application vocabulary represented as a query model.

We implemented a graphical user interface for query model manipulation. There are many visual interfaces for editing queries such as Query-By-Example and Smalltalk QueryEditor. Our interface, however, is based on the query model structure. Its main purpose is not creating *single* queries, but managing *collections* of queries.

We describe an example of editing a constructing and editing the query model depicted in Figure 4.1.

First we construct the *ProfessorQuery* corresponding to the *Professor* table. This is done by selecting the *TableQuery* type, choosing the *Professor* table in the list of available tables, and entering “ProfessorQuery” in the query name input field(see Figure 4.2). Clicking on the “New” button adds the query to the query model.

Building the *StudentQuery* for the *Student* table is done similarly.

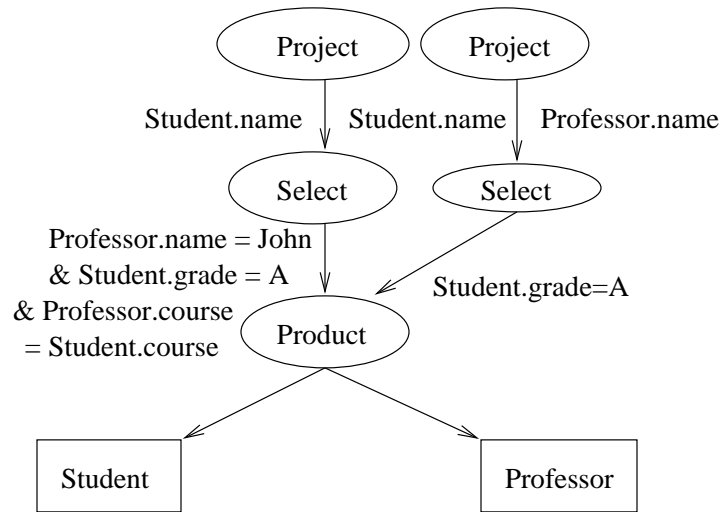


Figure 4.1 Constructed Query Model

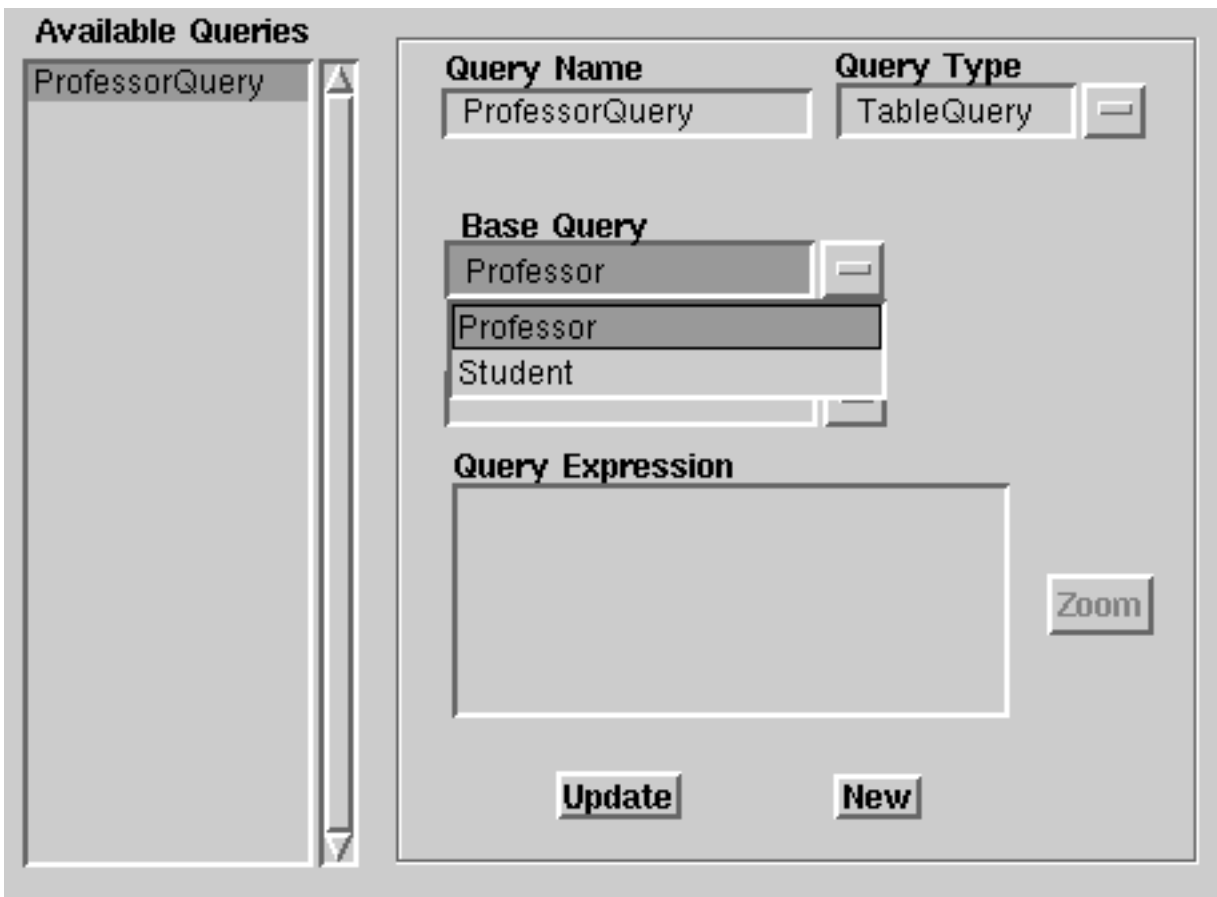


Figure 4.2 TableQuery Creation

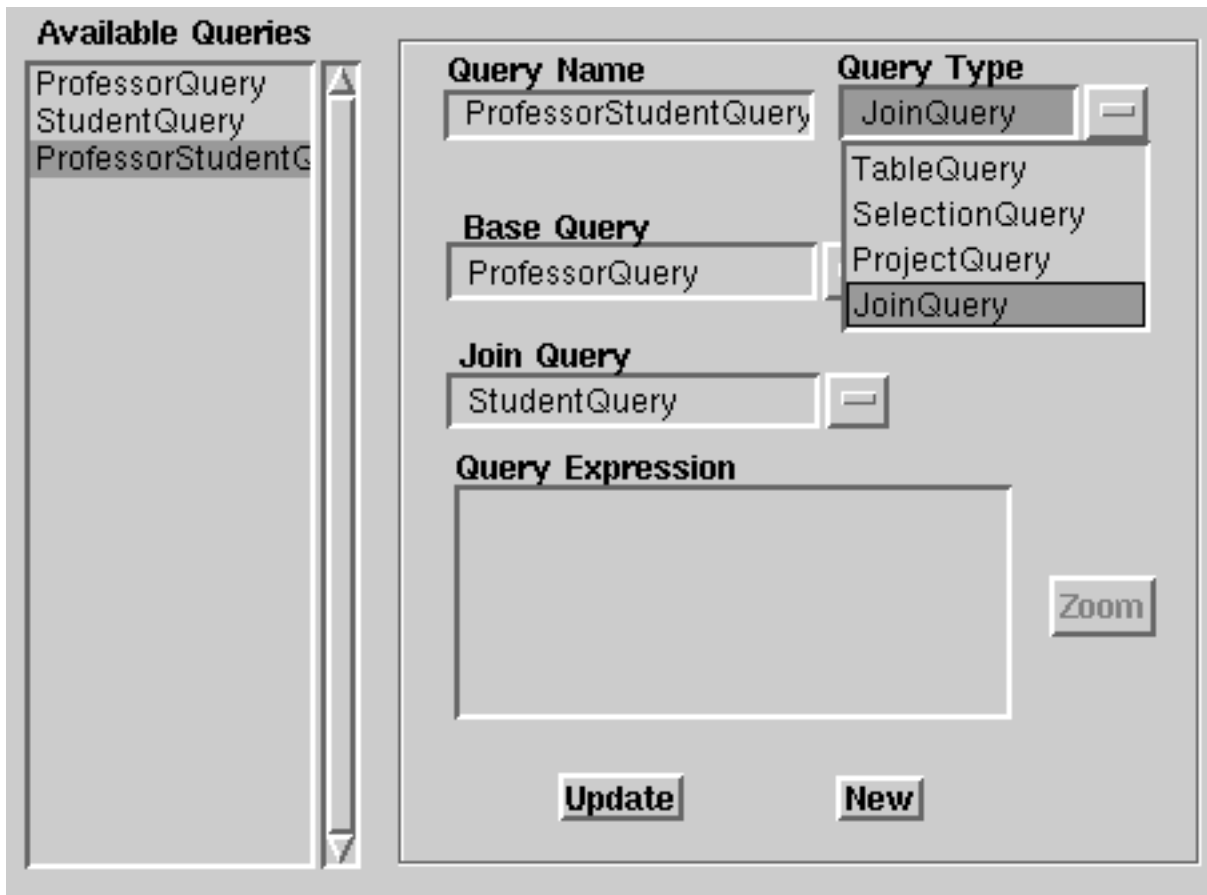


Figure 4.3 JoinQuery Creation

To create the *ProfessorStudentQuery*, which is a *JoinQuery* of the *ProfessorQuery* and *StudentQuery*, we select the *JoinQuery* type. Both base query and join query fields are active now, and we assign to them *ProfessorQuery* and *StudentQuery*, respectively (see Figure 4.3). We then add the query to the query model.

In order to construct a *SelectionQuery* over the *ProfessorStudentQuery* we perform similar steps. In addition to specifying the underlying queries we also have to specify the selection expression. This is done by building the tree representation of the expression. The resultant *ProfessorGradeQuery* is depicted in Figure 4.4.

The “Zoom” button creates a separate window with the expression tree representation (Figure 4.5). By double-clicking on the *Student.grade* node of the tree we bring up the

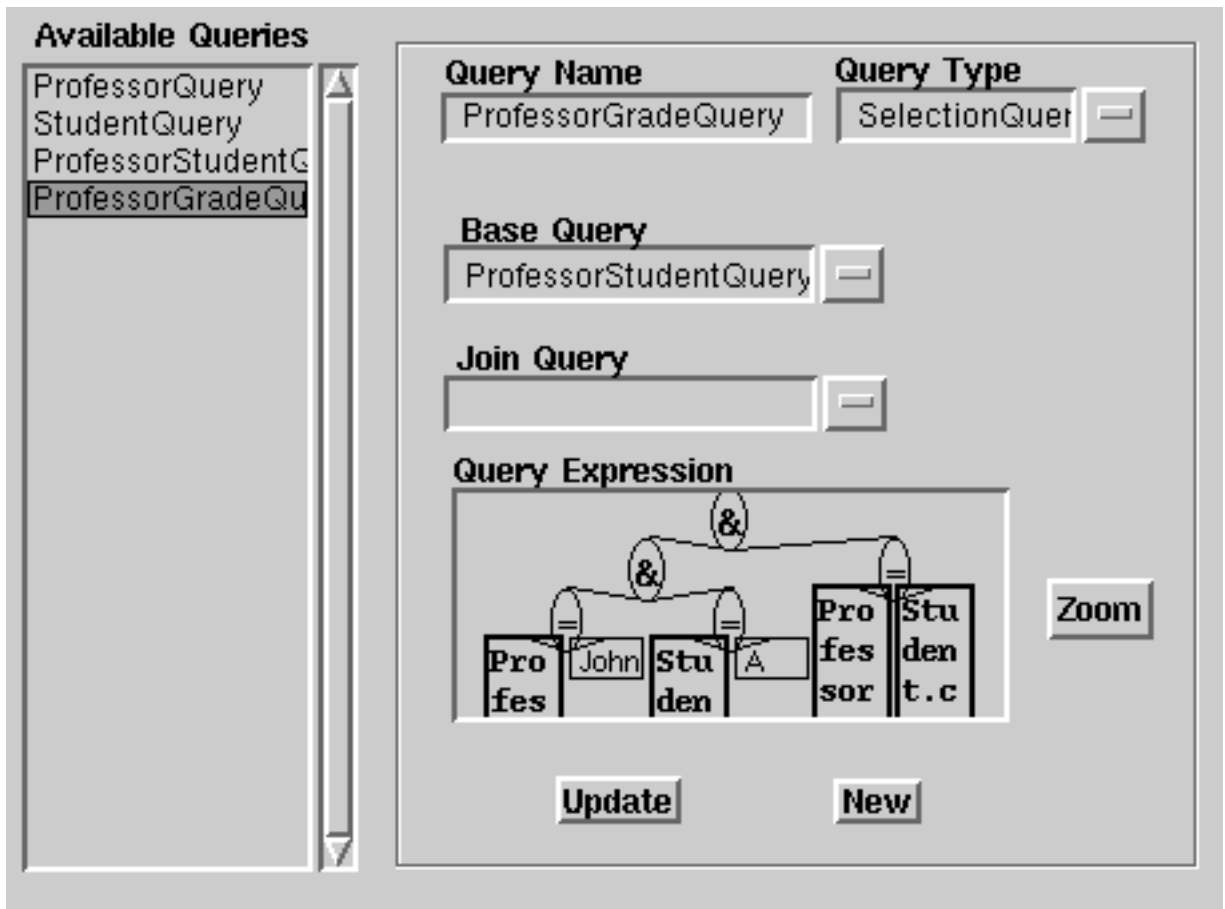


Figure 4.4 SelectionQuery Creation

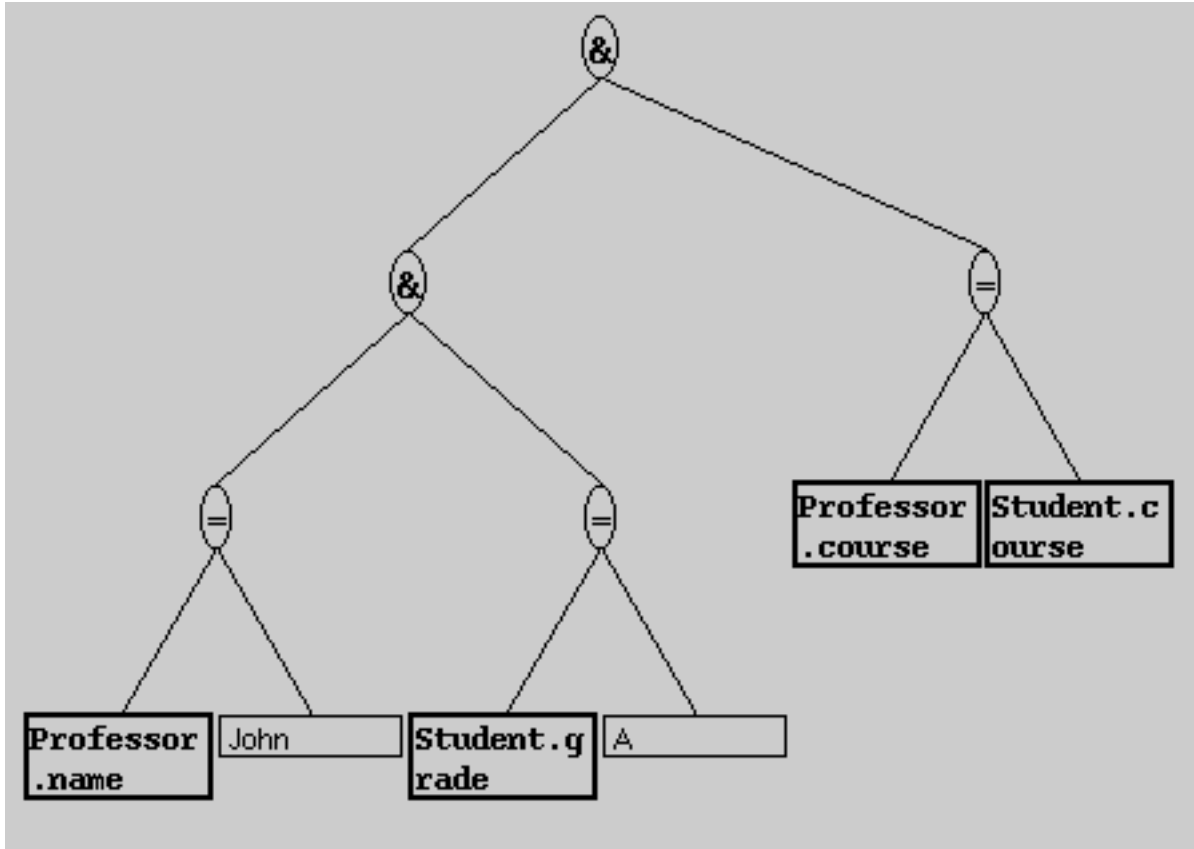


Figure 4.5 Expression Editor

expression node editor shown in Figure 4.6 and specify that the expression corresponding to the node is a field expression. A field expression is characterized by the query and field attributes, which are assigned *StudentQuery* and *name* values, respectively.

We continue by building the *GradeQuery* in a similar fashion.

Finally, we illustrate creation of a *ProjectionQuery* by projecting *name* field of both *Student* and *Professor* tables. The built *ProjectedGradeQuery* is shown in Figure 4.7. The expression part of the query is created in the same way as the selection expression of the *ProfessorGradeQuery* has been created.

We think that our example illustrates that the tool provides a clear and intuitive interface for creating query models.

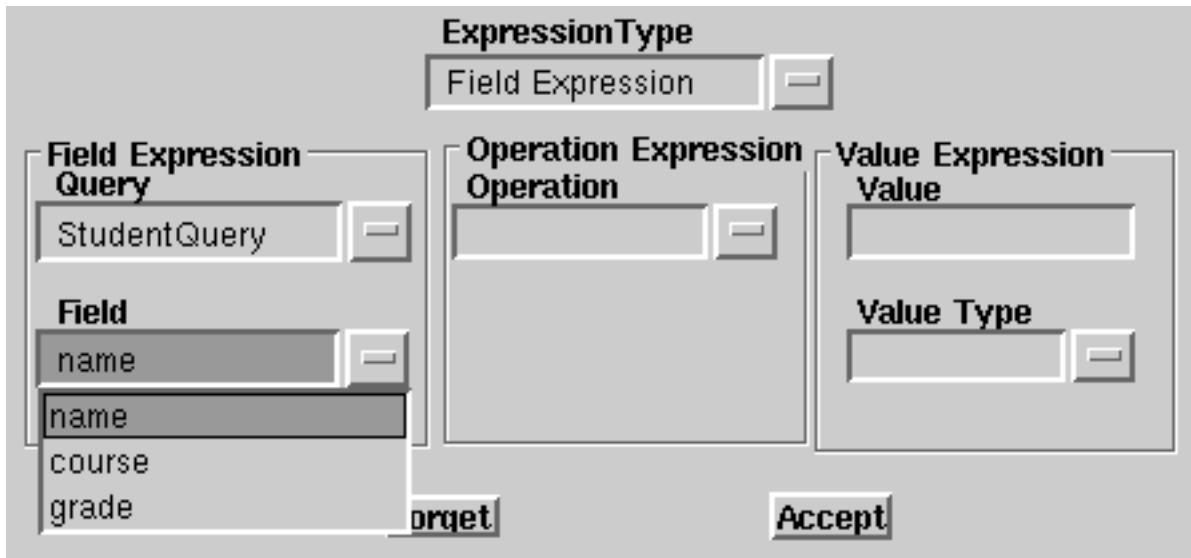


Figure 4.6 Expression Node Editor

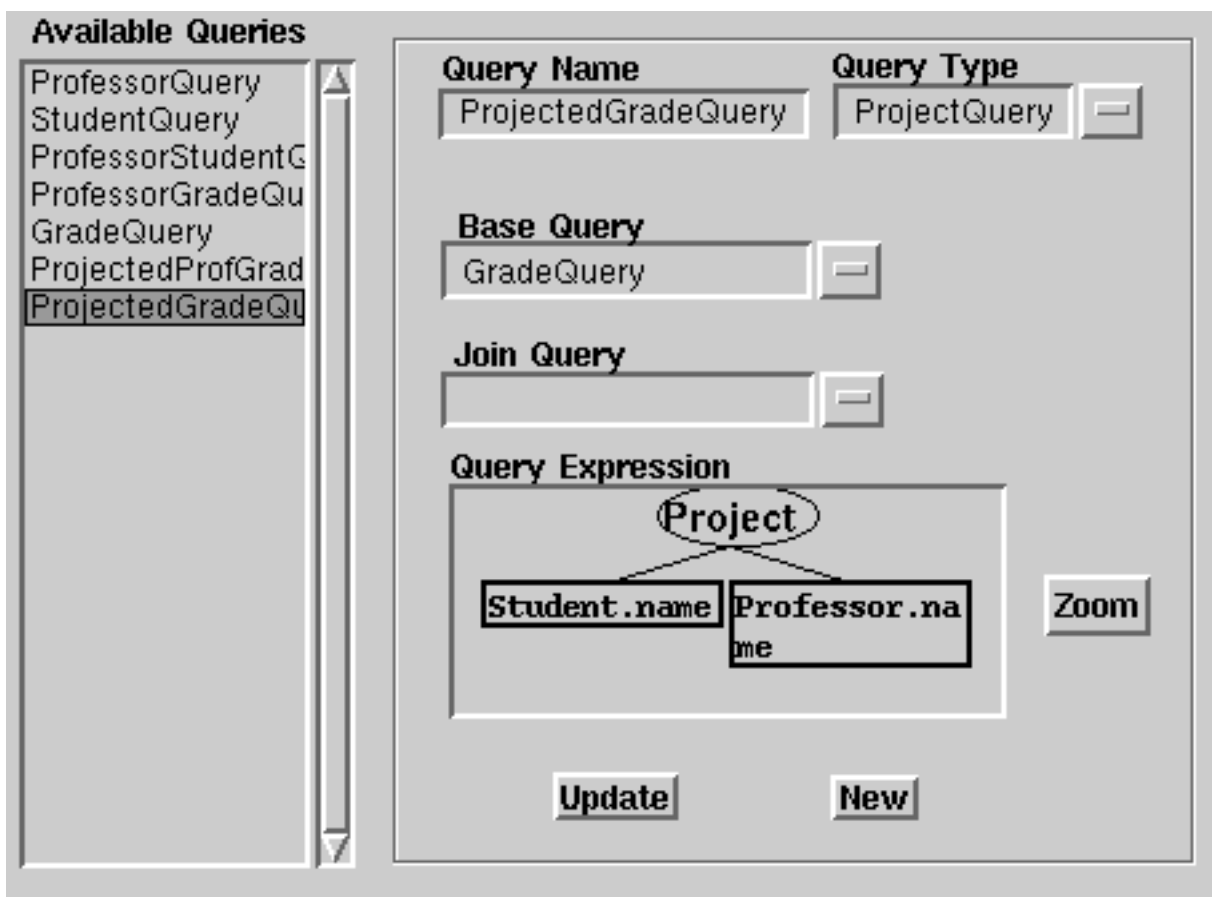


Figure 4.7 ProjectionQuery Creation

## CHAPTER 5

### CONCLUSIONS AND FURTHER RESEARCH

We investigated the process of database-oriented software development centered around an existing relational database and suggested a number of mechanisms for providing an applications programmer with a language for creating the vocabulary of application concepts. The language is expressed as a query model. Not only does it lend itself to efficient manipulation, but also its object-oriented implementation is easily adapted to increase the expressive power of relational algebra. Ease and intuitive interface for editing query models constitute one more attraction and provide a visual tool for accomplishing a significant part of application design.

The work herein was primarily concerned with the structural aspects of application modeling. We purposefully avoided behavioral issues. However, it would be interesting to consider a relationship between our purely extensional approach and the principles of inheritance and polymorphism of the object-oriented paradigm. Also, the reflection mechanism is clearly under-explored, and it should help endow query models with more expressive power. In general, the query model notion begs for further extension and refinement that may prove to be vital for approaching numerous problems of database-oriented software development.

## REFERENCES

- [1] Shailesh Agarwal and Arthur M. Keller. Architecting Object Applications for High Performance with Relational Databases. <http://www.persistence.com/persistence/pageTwo.pages/highperf.html>.
- [2] Kyle Brown and Bruce G. Whitenack. Crossing Chasms – a Pattern Language for Object-RDBMS Integration. Based on work done at Knowledge Systems Corp.
- [3] T. Bruce. *Designing Quality Databases with IDEF1X Information Models*. Dorset House Publishing, 1991.
- [4] Siva Challa and Dennis Kafura. Using Reflection for Implementing ICOM, An Interoperable Common Object Model. [http://actor.cs.vt.edu/siva/research/reflection\\_paper/new.html](http://actor.cs.vt.edu/siva/research/reflection_paper/new.html).
- [5] P. Checkland. *Systems Thinking, Systems Practice*. Wiley, 1981.
- [6] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.
- [7] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [8] M. Dalkitic. Design and Implementation of Reflective SQL. Technical report, Computer Science Department, Indiana University, 1996.
- [9] R. Hirschheim, H. Klein, and K. Lyytinen. *Information System Development and Data Modeling: Conceptual and Philosophical Foundations*. Cambridge University Press, 1995.
- [10] M. Jackson. *System Development*. Prentice Hall, 1983.
- [11] J. Brant and J. Yoder. Query and Formula Patterns. in preparation.
- [12] W. Kent. *Data and Reality*. North-Holland, 1975.
- [13] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
- [14] Jeffrey D. Ullman. *Principles of database systems*. Computer Science Press, 1982.
- [15] G. Wiederhold. Views, Objects, and Databases. *Computer*, 19(12):37–44, December 1986.