PIQUE: A Relational Query Language without Relations

David Maier
David Rozenshtein
Sharon Salveter
Jacob Stein
David S. Warren

Technical Report CS/E-87-003
9 April 1987

Oregon Graduate Center
19600 S.W. von Neumann Drive
Beaverton, Oregon 97006-1999

PIQUE: A Relational Query Language without Relations[1]

David Maier
Oregon Graduate Center

David Rozenshtein
Rutgers University

Sharon Salveter
Boston University

Jacob Stein
Servio Logic Corp. and Oregon Graduate Center

David S. Warren
SUNY at Stony Brook

[1]Some of the work presented in this paper was done while the first four authors were at the State University of New York at Stony Brook. Work supported in part by NSF grants IST 81-14622, MCS 82-07216, and IST 84-08970.

To appear in Information Systems.

Abstract

Relational database systems have gone far towards providing users with physical data independence. To use a relational database, users need not know the physical storage structures of relations, and are protected from changes in these structures. However, a user must still navigate among relations. In other words, if the information needed to answer a question spans several relations, he must explicitly specify how these relations are to be combined.

Physical data independence is not enough. A user should also be afforded some degree of structural data independence. More specifically, he should be able to pose queries without having to explicitly navigate among relations in the database. Instead, the system should do the navigation for him.

We consider universal scheme interfaces as a means for automatic database navigation, and introduce the concept of a *generator* as central to such navigation. We describe a particular generator based on the semantic notions of *decomposable* and *non-decomposable facts*, and present PIQUE, an attribute-based query language designed to work with this generator.

PIQUE is a concise, yet powerful, language with natural semantics. A distinguishing feature of PIQUE is that tuple variables in queries are bound implicitly and that the logical connectives "and," "or," and "not" can affect the binding, and, therefore, take on "semantic overtones." Furthermore, the semantic interpretation PIQUE gives to these connectives is more natural than the one given by most other query languages.

In the appendix, we present PIQUE's formal syntax and semantics, along with the proof that PIQUE is relationally complete.

1. Introduction

A database system for interaction with non-technical users should provide them with a "user oriented" query language with natural syntax and semantics, so that it is easy to learn and remember. By "user oriented" we mean that the language should support the user (semantic) point of view on the world being represented rather than the database (structural) one. In particular, the queries posed in the language should be phrased in terms of semantic notions and be free from representational artifacts.

At the very least, such a language should provide users with a certain degree of *physical data independence*. That is, it should allow users to pose queries without any knowledge of the physical structure of the database. We note that all existing relational query languages (such as SQL [CA], Quel [SWKH], QBE [Zl], etc.) possess this property. The user of a relational database sees the data as a collection of named relations, each over some set of attributes. Therefore, he does not have to be concerned with the details of physical implementation of these relations (such as pointers, indexing schemes, physical record structures, etc.). Likewise, since the user phrases his questions in a high level relational language, he does not have to know anything about the actual implementation of the operations on relations.

To illustrate the importance of achieving physical data independence, consider a simple network database modelling a university. A possible scheme for such a database is

```
record S(field STUDENT, ...)
record C(field COURSE. ...)
record F(field FACULTY, ...)
record R1(...)
record R2(...)
set    S-R1(owner S; member R1)
set    C-R1(owner C; member R1)
set    C-R2(owner C; member R2)
set    F-R2(owner F; member R2)
```

where sets S-R1 and C-R1 represent information about students taking courses. and sets

C-R2 and F-R2 represent information about faculty members teaching them. A query such as "Who takes some course taught by Smith?" directed at this database, can then be posed in the following way (the syntax is borrowed from Date [Da] and is somewhat simplified).

```
1.      move 'Smith' to FACULTY in F
2.      find any F using FACULTY in F
3.      find first R2 within F-R2
4.      perform until not fail
5.              find owner within C-R2
6.              find first R1 within C-R1
7.              perform until not fail
8.                      find owner within S-R1
9.                      get S
10.                     print STUDENT in S
11.                     find next R1 within C-R1
12.             end-perform
13.             find next R2 within F-R2
14.     end-perform
```

We shall not discuss this example in detail (a discussion on the design and use of network databases can be found elsewhere [Da]). However, it is clear that the structure of the query above is very much dependent on the specific physical structure that we have chosen for the database. In particular, lines 3, 5, 6, 8, 11 and 13 depend on faculty, course and student records being organized in particular ring structures, while line 2 refers to a specific method of accessing faculty records.

On the other hand, a typical relational database representing the same situation would contain the following relations.

```
S(STUDENT, ...)
C(COURSE, ...)
F(FACULTY, ...)
TAKES(STUDENT, COURSE)
TEACHES(FACULTY, COURSE)
```

The same question can then be posed in Quel as follows.

```
1.    range of x is TEACHES
2.    range of y is TAKES
3.    retrieve (y.STUDENT)
4.    where (y.COURSE = x.COURSE) and (x.FACULTY = 'Smith')
```

In this query, lines 1 and 2 specify the relations involved, line 3 describes the information to be retrieved, while line 4 refers to the conditions to be satisfied. Note that no reference to the internal structure of relations or to any particular access method need be made.

We note, however, that the user of a relational database must still navigate among relations. In other words, if the information needed to answer a question spans several relations, he must specify how these relations are to be combined to achieve the desired answer. In particular, in posing his queries, the user must refer explicitly to relation names (in algebraic languages) or explicitly declare and bind variables (in calculus languages), and explicitly specify natural joins. In the Quel query above, for example, relations TAKES and TEACHES are introduced into the query through explicit declaration of tuple variables x and y, while condition (y.COURSE = x.COURSE) serves only to specify the natural join between these relations.

In order to navigate within a relational database, the user must know its structure. This requirement may, in turn, present problems. First, the structure of the database in any real application can be quite complex. Second, databases tend to evolve in time. In particular, relations in the database may be decomposed or restructured for reasons of normalization or redundancy. Therefore, it seems unreasonable to expect all users to know the database structure in detail at all times. Finally, some users may be precluded from seeing all of the database for security reasons.

We, therefore, believe that having physical data independence is not enough. A user of a database should also be afforded some degree of *structural* (sometimes also called *logical) data independence.* More specifically, he should be able to pose queries without having to explicitly navigate among relations in the database. Instead, the system should do the navigation for him. As we shall see, this will allow a considerable simplification of the query language, since none of the navigational information will need to be included in queries. This, in turn, will make the language more natural in its appearance, and, therefore, more suitable for users.

The remainder of this paper is organized as follows. In Section 2, we consider *universal scheme interfaces* as a means for automatic database navigation. We discuss general principles behind providing such interfaces and introduce the concept of a *generator* as central to automatic navigation. In Section 3, we describe one particular universal scheme interface, based on the semantic notion of *fact*. In Section 4, we present PIQUE, an attribute-based query language designed to work with this interface. In PIQUE, the question "Who takes some course taught by Smith?" can be posed as follows.

retrieve (STUDENT) **where** (FACULTY = 'Smith')

In Section 5, we discuss various improvements to PIQUE. Finally, in the appendix, we present PIQUE's formal syntax and semantics, along with the proof that PIQUE is relationally complete.

2. Universal Scheme Interfaces

The traditional method of hiding structural details of a database from a user (thus, in effect, avoiding the problem of navigation) has been to offer the user a *view* of the database [Da,Ma,Ull]. A view in a relational database is a set of (derived) relations defined by some set of relational algebra (or relational calculus) expressions, usually explicitly referencing stored relations of the database. For the sake of simplicity, we shall assume a view to be a single relation, rather then a set of them.

We believe that there are several problems with views. First, views do not really free the user from the need to know the structure and semantics of the underlying database. For example, consider a user who is given some relation r as a view on a database. If the user is to make any use of this view, he must be able to correctly interpret the information contained in it; that is, he must be able to associate a correct semantic interpretation with tuples from r.

The only way the user can do this *precisely* is by looking at how view r was defined. This implies that he has to know the meaning of all stored relations involved in the definition of r. Furthermore, if there are several views that the user can choose from, he has to understand the meaning of all stored relations mentioned in any of them. And finally, for the user to realize that none of the available views suit him (and, consequently, to request the construction of a new one) he has to understand the meaning of all stored relations comprising the database.

As an example, consider a database consisting of a single relation TEACHES-TO(FACULTY,COURSE,STUDENT). Suppose a user has been given a view TEACHER(FACULTY,COURSE) defined as π_{FC}(TEACHES-TO). (π_{FC} stands for "projection onto attributes {FACULTY,COURSE}.")

If the user is to understand the meaning of this view, he has to be aware of the presence and the meaning of relation TEACHES-TO. Then, and only then, will the user be able to correctly interpret a tuple <f,c> from view TEACHER: namely, that there exists some student s who takes course c taught by faculty member f.

The second problem with views is that the user has to remember the names and the schemes of the available views. Since the number of views in any non-trivial database can be quite large, this approach does not seem to provide any real advantage.

Yet another problem with views is that they are database oriented. Since views are usually predefined by the database designer, they really represent his (overall) point of view on the database, which might not correspond to the interpretations that users have in mind.

In particular, the most intuitive interpretation of view TEACHER is that faculty members teach courses (regardless of the existence of any students taking them). Clearly, however, such an interpretation of view TEACHER is not correct. Furthermore, without the use of *placeholder nulls* [Sc], a view with the above interpretation cannot even be defined in terms of relation TEACHES-TO.

A more promising approach is the use of *universal scheme interfaces.* A universal scheme interface allows one to access the database solely through the attributes. The assumption is that attributes in a such an interface correspond naturally to the entities in the real world, and the attribute names are chosen in such a way as to give users an intuitive understanding of the relationships among these entities that is close to the semantics of the database.

Query processing in a universal scheme interface can be viewed as a two step procedure. First, a set of attributes X appearing in the query is determined. Then, on the basis of the state of the database, a relation r over X is generated. (If the query contains several variables, then attributes appearing with each variable are used to generate separate relations.) Second, further operations specified by the query (such as selections, projections and joins) are performed on the generated relation(s) to produce an answer.

These two steps are usually called *binding* and *evaluation* [MRW2].

The process of binding can, in turn, be divided into two sub-stages: *navigation* and *computation*. Given a set of attributes X, the universal scheme system first determines an appropriate navigation path. That is, the system establishes a relational expression that defines r over X in terms of stored relations. This relational expression is usually called a *window* (or a *connection*) for X and is denoted by [|X|] [MRW2]. Note that formally [|X|] can also be thought of as a function from database instances to relations over X. Therefore, [|X|] is sometimes referred to as a *window function* [MRW2]. During the computation stage, expression [|X|] is used to compute the relation r. While, properly, we should denote the value of [|X|] on database instance d as [|X|](d), in most cases d will be understood, and we shall simply write [|X|].

A mechanism for navigating within a database, or more precisely for generating windows is called a *window generator*, or simply a *generator* [MRS2]. Formally, a generator can be looked at as a mapping from a set of attributes and a database scheme to the window for that set in that scheme.

In general, there is more than one way to establish a navigation path covering a given set of attributes. Therefore, generators often use additional semantic information in choosing among possible paths. This semantic information usually falls into one of two categories: semantic tools provided by or added to the relational model (e.g., functional and join dependencies), and assumptions made about the state of the database (e.g., universal instance assumption, universal relation scheme assumption, weak and/or representative instance assumption).

Several universal scheme systems have been proposed or are under development. Among them are: APPLE [CK]; the system of Shenk and Pinkert [SP]; q [AK]; System/U [Ko,KKFGU,KU,MU,Ul2]; PIQUE [MRS1,MRS2,MRSSW,MW,Ro]; Parafrāse [KMRS,KS]; FIDL [Ba]; DURST [BB]; and the system of Arazi-Gonczarowski [A-G]. Maier, Rozenshtein and Warren [MRW1,MRW2] provide a detailed overview and a comparison of these systems. For the purpose of exposition, in the next section, we choose and briefly present the *object-based* (OB) generator adopted from the *association-object data model* [MW].

In concluding this section, we note that windows are similar to views. They differ from traditional views, however, in that generators provide a uniform discipline for defining and naming them. In addition, unlike an arbitrary set of views, windows in many (although not all) universal scheme interfaces display some manner of semantic consistency. In fact, the choice of the term "window" is intended to convey the image of a consistent set of views onto a single database world.

3. The Object-Based Generator

In presenting the OB generator, we take the view that the database scheme should be designed in such a way that the relations in the database would correspond to sets of *irreducible facts.* (A precise definition of *fact* for relational databases was introduced by Sciore [Sc].) A fact is considered irreducible if it cannot be derived from any of its partial subfacts. For example, a tuple <Smith,CS302> from relation TEACHES(FACULTY,COURSE) represents an irreducible fact, since we cannot posit that Smith teaches CS302 from knowing that Smith is a faculty member and CS302 is a course. Likewise, a tuple <Smith,CS302,Kirk> from relation TEACHES-TO(FACULTY,COURSE,STUDENT) corresponds to an irreducible fact, since again we cannot posit that Smith teaches CS302 to Kirk from knowing that Smith teaches CS302 and Kirk takes it.

Additional semantic information (to be used by the OB generator) is represented by means of semantic devices called *objects* and by the enforcement of the so-called *unique role assumption* (URA). We use the term *extended database scheme* to refer to a database scheme together with its objects.

Each object is a set of attributes. The set of objects is declared by the database designer. Declaring W to be an object corresponds to asserting that there exists some semantic relationship among attributes in W and that this relationship is decomposable. Thus, tuples over W have meaningful interpretations and correspond to *reducible facts.* These facts can be derived from their subfacts. We shall return to how this is done later in this section.

The URA is a simplifying assumption that basically states that the database represents at most one semantic relationship among any set of attributes. In other words, there is at most one way to interpret an existing tuple over any set of attributes. For example, a database with relations TAKES(STUDENT,COURSE) and

ASSISTS-WITH(STUDENT,COURSE) would violate the URA, since it contains two semantically distinct relationships between STUDENT and COURSE. One of them states that STUDENT takes COURSE, while the other states that STUDENT is a teaching assistant for COURSE. An obvious way to make this database satisfy the URA is to rename attribute STUDENT in one of the relations. For example, we can achieve satisfaction of the URA by renaming STUDENT into ASSISTANT in relation ASSISTS-WITH.

Note that the URA does not require that there exist a meaningful relationship among *any* set of attributes. It merely states that if a directly represented relationship does exist among the attributes in some set X, then it is unique. Also note that we do not construe the URA so strongly as to prohibit the very existence of multiple semantic relationships among a set of attributes. We only intend our system to consider one of them to be the most natural, and to establish that relationship automatically. Other relationships must be established by the user.

For example, consider a simple database modelling a university. A possible scheme for such a database is:

TEACHES(FACULTY,COURSE)
TAKES(STUDENT,COURSE)
ASSISTS-WITH(ASSISTANT,COURSE)
TEACHES-TO(FACULTY,COURSE,STUDENT)

with objects

{FACULTY,COURSE,ASSISTANT}
{STUDENT,COURSE,ASSISTANT}.

A possible instance for this database is:

TAKES =

STUDENT	COURSE
Kirk	CS120
Rabkin	CS120
Kirk	CS302
Ross	CS303

TEACHES =

FACULTY	COURSE
Smith	CS120
Smith	CS302
Smith	CS303

TEACHES-TO =

FACULTY	COURSE	STUDENT
Smith	CS302	Kirk

ASSISTS-WITH =

ASSISTANT	COURSE
Chung	CS112
Johnson	CS302
Ross	CS112

Note that this database does satisfy the URA, with each relation scheme corresponding to a *type* of irreducible facts.

One of the consequences of the URA is that a database may contain at most one stored relation for any set of attributes. Therefore, we do not have to use relation names to identify relations in a URA database. From now on, when convenient, we shall use r(R) to denote the stored relation over R.

We note that the OB generator allows *subscheme relations* to be present. It is a consequence of the URA, however, that these relations must satisfy the following *containment condition*: for any two relations r(R) and r(S) if $R \subseteq S$, then $\pi_R(r(S)) \subseteq r(R)$. For example, since the scheme of relation TEACHES is included in the scheme of relation TEACHES-TO, these relations must (and, in fact, do) satisfy the following containment condition.

$$\pi_{FC}(\text{TEACHES-TO}) \subseteq \text{TEACHES}$$

(In the expression above and in other expressions throughout this paper we often abbreviate attributes to their first letters.)

Recall that objects correspond to types of reducible facts. Relations for objects are not stored in the database, but rather are computed from stored relations. In particular, let W be an object. The semantics of a tuple w over W must follow from the semantics of those subtuples of w that correspond to irreducible facts. That is, the meaning given to any tuple w(W) is a combination of meanings given to tuples from all those stored relations whose schemes are included in W. Furthermore, by the URA, the interpretations assigned to all such relations are mutually consistent with each other.

Therefore, we define a relation for W (denoted by $\bar{r}(W)$ and called an *object view*) as the following natural join expression (called the *object-join*).

$$\bar{r}(W) = *_{\text{R is relation scheme, } R \subseteq W}(r(R))$$

We shall again abuse the notation and often write $\bar{r}(W)$ to mean $\bar{r}(W)(d)$ when the database state d is understood and the context is clear.

For example, the object view for object {FACULTY,COURSE,ASSISTANT} is defined as follows.

$$\bar{r}(FCA) = r(FC) * r(AC)$$

The actual relation for this object, which is computed by applying object-join $\bar{r}(FCA)$ to our database, is

FACULTY	COURSE	ASSISTANT
Smith	CS302	Johnson

The definition above makes sense only if W is equal to the union of some set of relation schemes [MRW1,MRW2,MW]. The OB generator makes this restriction on allowable objects. In addition, since every fact can always be considered to be reducible to itself, every relation scheme is automatically included in the set of objects. For the sake of succinctness, however, relation schemes do not have to be explicitly declared as objects.

Note that, for any two relations r(X) and r(Y), if $X \subseteq Y$ and $\pi_X(r(Y)) \subseteq r(X)$, then $r(X) * r(Y) = r(Y)$. Therefore, by the containment condition for relations, only relations over the *maximal* schemes need actually be considered in taking the join. Furthermore, as the next lemma shows, object views also satisfy the containment condition.

Lemma 1:

For any two objects V and W, if $V \subseteq W$, then $\pi_V(\dot{r}(W)) \subseteq \dot{r}(V)$.

Proof:

Immediate from the definition of object views.

□

We note that our generator, in effect, uses objects to navigate within the database. To the generator, each object represents a meaningful semantic relationship among a set of attributes, and therefore a meaningful navigation path. Note, however, that objects are the *only* tool used by this generator. Therefore, in our definition of objects there is an implicit assumption that they represent *all* meaningful relationships. Again we do not construe this assumption so strongly as to prohibit the possibility that there exists some semantic relationship among attributes in a set that is not declared as an object. However, this relationship will not be recognized as meaningful by our generator, and therefore will not be established automatically.

We also note that if W is an object, then there exists a meaningful relationship among any subset V of W. Furthermore, the semantics of this relationship is (at least partially) determined by the semantics of the object W.

Finally, we note that by the URA and by the requirement that objects be closed under nonempty intersection (this requirement is introduced later in this section) the semantic interpretations given to any two objects that share attributes are consistent.

These arguments motivate the following strategy for defining windows. Given a set of attributes X, the OB generator considers all objects whose attributes include X. Every such object represents a meaningful relationship relevant to the relationship among attributes in X; these objects represent all relevant relationships; and, finally, all of these objects have consistent semantic interpretations. Therefore, the OB generator defines $[|X|]$ as the following union.

$$[|X|] = \cup_{W \text{ is object},\ X \subseteq W}\ \pi_X(\tilde{r}(W))$$

For example, the window for {FACULTY,ASSISTANT} is defined as follows.

$$[|FA|] = \pi_{FA}(\tilde{r}(FCA)) = \pi_{FA}(r(FC) * r(AC))$$

The corresponding relation, which is obtained by applying the expression $[|FA|]$ to our database, is

FACULTY	ASSISTANT
Smith	Johnson

A tuple <Smith,Johnson> from this relation is interpreted as "faculty member Smith teaches some course for which Johnson is an assistant."

If the extended database scheme contains no objects that include X, then the generator will return *nothing* (i.e., the relational expression defining the empty relation over X) as the definition of $[\![X]\!]$. In fact, it should probably be made to return a message stating that it could not define $[\![X]\!]$ at all. This is appropriate, since the absence of objects that include X indicates that (as far as the database designer is concerned) the relationship among X is not semantically meaningful (or, at least, is not sufficiently meaningful to be defined automatically).

By the containment condition for objects, if objects W and V both contain X, and W $\subsetneq$ V, then V can be dropped from the union that defines $[\![X]\!]$. Thus, only the *minimal* objects need actually be considered in taking the union. In addition, as the next lemma shows, the windows computed by the OB generator also satisfy the containment condition.

Lemma 2:

For any X and Y, if X $\subseteq$ Y, then $\pi_X([\![Y]\!]) \subseteq [\![X]\!]$.

Proof:

Immediate from the definition of windows.

$\square$

It follows from Lemma 2 that windows defined by the OB generator are semantically consistent. Furthermore, the OB generator is *faithful*. That is, windows are defined in such a way that for any relation scheme R, r(R) = $\bar{r}$(R) = $[\![R]\!]$. We note that faithfulness is an essential and very much desired property of any generator. Unless the generator is faithful, it is possible to add a tuple to a stored relation and not be able to retrieve it. The next theorem describes the necessary and sufficient conditions for the faithfulness of the OB generator.

Theorem 1 [MRW2]:

The OB generator is faithful if and only if every relation scheme is also considered to be an object, and relations satisfy the containment condition.

$\square$

The final condition deals with the *integrity* of objects. The purpose behind this condition is to prevent little knowledge from being a dangerous thing. If someone knows the semantics of all relations whose schemes are contained within some object W, then he should be able to deduce the meaning of the window on any subset of W. Formally, object W is *integral* if for any $X \subseteq W$, $[|X|]$ can be computed from $\{r(R) \mid R \subseteq W\}$ alone.

Our example database violates integrity of objects, because, for example, $[|COURSE|]$ cannot be defined from relations in object {FACULTY,COURSE} alone. In fact, the OB generator would define it as

$$[|C|] = \pi_C(r(FC)) \cup \pi_C(r(SC)) \cup \pi_C(r(AC)).$$

The danger here is that if a user knows only that the database contains information about faculty members and courses (and does not know about students and assistants) his assumptions about the meaning of $[|COURSE|]$ will be incorrect.

The next theorem shows that integrity of objects is equivalent to the set of objects being closed under nonempty intersection. That is, if objects V and W have a nonempty intersection Z, then Z must also be an object. This closure property has an important computational advantage: for any X there is a *unique minimal* object W containing X. This, together with the containment condition for objects, implies that $[|X|]$ can be defined in terms of $\bar{r}(W)$ alone. In other words, no unions need to be taken to define $[|X|]$.

Theorem 2 (MRW2):

All objects are integral if and only if objects are closed under nonempty intersection.

□

The integrity of objects and the condition requiring every object to be the union of some relation schemes imply that relation schemes themselves must be closed under nonempty intersection. In accordance with this, we modify our example to include a relation over {COURSE}. A possible interpretation of relation r(COURSE) is to represent all existing courses (e.g., all courses listed in a catalogue). We note that a course may be listed in a catalogue independently from any students taking it, any faculty member teaching it, or any assistant assisting with it.

4. PIQUE: The Query Language for the OB Generator

Recall that query processing in a universal scheme system involves two steps: binding and evaluation. During binding, the generator establishes the necessary windows and computes the corresponding relations; during evaluation, additional operations specified in the query are applied to these relations to generate an answer.

While these steps do interact, they are loosely-coupled. Changes to the generator can be made without affecting the procedures at the evaluation step. Therefore, the PIQUE query language, while designed for use with the OB generator, can also be used with other types of generators. (We note, however, that changes to the generator will affect the way it defines windows and, therefore, may result in different answers to queries.)

PIQUE is a tuple calculus language similar to Quel [SWKH]. Thus, a query in PIQUE has the general form

retrieve <retrieve set> **where** <condition>,

where <retrieve set> is a list of tuple-attribute pairs and <condition> is a predicate of a form to be specified later. There are, however, two important distinctions between PIQUE and Quel. First, because the OB generator automatically navigates within the database, the tuple binding mechanism of Quel that explicitly binds tuple variables to relations becomes superfluous. Instead, tuple variables are bound to windows. (More precisely, they are bound to relations obtained by evaluating windows on the current state of the database.) The second distinction is that users do not have to specify the bindings explicitly. Instead, the system uses the syntax of the query to determine the range of tuple variables.

In this section, we present examples of queries that motivate the constructs necessary for a natural and powerful query language. We also show how these constructs can be used in determining the range of tuple variables within a query. All sample PIQUE queries discussed in this section will refer to the database with the following relation schemes

{COURSE}
{FACULTY,COURSE}
{STUDENT,COURSE}
{ASSISTANT,COURSE}
{FACULTY,COURSE,STUDENT}

the following objects

{FACULTY,COURSE,ASSISTANT}
{STUDENT,COURSE,ASSISTANT}

and the following instance

COURSE
CS112
CS120
CS302
CS303

FACULTY	COURSE
Smith	CS120
Smith	CS302
Smith	CS303

STUDENT	COURSE
Kirk	CS120
Rabkin	CS120
Kirk	CS302
Ross	CS303

ASSISTANT	COURSE
Chung	CS112
Johnson	CS302
Ross	CS112

FACULTY	COURSE	STUDENT
Smith	CS302	Kirk

This database is our example database of Section 3 with relation r(COURSE) added in. In the world reflected by this database, assistants are assigned to courses, not to faculty members or students. Therefore, an assistant is responsible for aiding every student taking his course regardless of which faculty member teaches it. Likewise, an assistant is responsible for helping every faculty member teaching his course.

Again, whenever convenient we shall abbreviate attribute names to their first letters. Also, we shall often use [|X|] to denote the actual relation corresponding to the window for

X, rather than the expression defining it.

4.1. Simple Queries

Consider a request for the list of faculty members and the courses they teach. Query

retrieve t.F, t.C **where** (Q_1)

models the above request in an intuitive manner. However, to make Q_1 return an intuitively correct answer, namely

{<Smith,CS120>, <Smith,CS302>, <Smith,CS303>}

tuple variable t has to be bound to [|FC|].

Note that condition in Q_1 is empty. For the sake of succinctness, if the condition of the query is empty, then keyword '**where**' may be omitted from the query.

Next consider a request for the list of students taking some particular course, say CS120. This information is contained in the tuples of [|SC|] that have C-values of CS120. Consider the query

retrieve t.S **where** (t.C = CS120). (Q_2)

Again, the query above intuitively corresponds to the request. If during its evaluation tuples with C-value CS120 are selected from [|SC|] and their S-values are returned, then query Q_2 returns

{<Kirk>, <Rabkin>}

which is intuitively correct. In order to do that, however, tuple variable t must be bound to [|SC|].

In both queries Q_1 and Q_2 above, tuple variable t is bound to [|men(t)|], where men(t) is the *mention set* of t: the set of attributes that appear with t in the query. We note that mention sets were used to determine tuple binding in System/U queries in the presence of maximal objects [MU]. However, the way men(t) is defined in PIQUE queries is different from the way it is defined in System/U queries. A formal definition of men(t) and the binding mechanism of tuple variables are provided in the appendix.

In the condition 't.C = CS120', attribute C is compared to a constant. PIQUE also provides the facility to compare two attributes and allows comparisons for inequality. In addition to that, for ordered domains, PIQUE provides the facility to use any of the binary comparators from {<, <=, >, >=}. For string domains, substring and regular expression matching can also be performed.

Consider a request for a set of courses that have assistants. Query

retrieve t.C **where** (t.A = t.A) (Q_3)

models this request if tuple variable t is bound to [|AC|]. Query Q_3 returns an intuitively correct answer, which is

{<CS112>, <CS302>}.

Note that condition 't.A = t.A' is included in Q_3 just to insure that tuple variable t is bound to [|AC|] rather than [|C|], and not for the sake of the comparison. Therefore, in PIQUE queries, 't.A' is written in place of 't.A = t.A' and is referred to as a *name drop*, indicating that A was "dropped" into men(t). The query then takes the form

retrieve t.C **where** (t.A). (Q_4)

The name drop 't.A' in Q_4 can then be interpreted as "assistant exists."

It is interesting to note the similarity in the use of name drops in PIQUE queries and in System/U queries evaluated in the presence of maximal objects. In both systems name drops are used to limit the range of the query. Formally, there is a difference however. In a System/U query, a name drop restricts a tuple variable to range over fewer maximal objects, while in a PIQUE query, it forces a tuple variable to range over larger objects.

In the queries above, 'P', 'C' and 'A' could have been used in place of 't.P', 't.C' and 't.A', if the existence of a *default* tuple variable was assumed. In PIQUE, this default tuple variable is referred to as the *blank* tuple variable and the above substitutions are allowed. For example, query Q_2 can be rewritten as

retrieve S **where** (C = CS120). (Q_5)

Now consider queries that involve several comparisons. Consider the following closely related English questions: "Which courses are being taken by Kirk from Smith?," and "Which courses are being taken by Kirk and taught by Smith?" Note that in answering the first question, information from [|FCS|] should be used. In PIQUE, this question is posed as

retrieve t.C **where** (t.F = Smith)*(t.S = Kirk). (Q_6)

Symbol '*' in the condition of Q_6 indicates that *both* F and S are to be included in men(t), thus forcing the query to be evaluated over [|FCS|]. From a semantic point of view, the '*' operator is interpreted as "and simultaneously." In other words, it indicates that both of the selection conditions are to be performed on the same tuple. The answer to Q_6 is

{<CS302>}.

In answering the second question, tuples from [|FC|] and [|SC|], rather than [|FCS|], should be considered. This is because a course can be taught by Smith and taken by Kirk, but not necessarily be taught by Smith to Kirk. The above request can be posed as

retrieve t.C **where** (t.C = u.C)*(t.F = Smith)*(u.S = Kirk). (Q_7)

By the binding convention, tuple variable t is bound to [|FC|] and tuple variable u is bound to [|SC|]. The answer to this query is

{<CS120>, <CS302>}.

It is obtained by taking the cross-product of [|FC|] and [|SC|], applying specified selections, and applying a projection onto t.C. There is, however, a more natural way of posing this request.

4.2. Complex Queries

Consider again a request for the courses taken by Kirk and taught by Smith. Consider the query

retrieve t.C **where** (t.P = Smith) **and** (t.S = Kirk). (Q_8)

The rule for interpreting Q_8 is motivated by the fact that semantics of '**and**' is almost invariably associated with intersection. The answer to this query is obtained by evaluating the queries

retrieve t.C **where** (t.P = Smith)

and

retrieve t.C **where** (t.S = Kirk)

and returning the intersection of the evaluation results. We note that this answer is, of course, the same as the answer to query Q_7.

We also note, that at different stages in the evaluation of query Q_8, tuple variable t is bound to different connections. In other words, men(t) takes on different values at different stages of evaluation. This situation does not occur in Quel queries, because tuple variables there are explicitly bound to named relations. Neither does it occur in System/U queries, since all tuple variables there are permanently bound to the *universal instance* [KU], or, in the presence of maximal objects, permanently bound to the union of maximal objects.

While queries with multiple tuple variables do occur in PIQUE, they are not usually used to specify joins on stored relations. Since the need to join stored relations occurs frequently, the ability to specify these joins using only one tuple variable is an advantage.

Note that in queries involving multiple tuple variables, the retrieve set may contain duplicate attributes. Therefore, a mechanism for renaming attributes is provided in the language. As an example of a query involving renaming. consider

retrieve t.STUDENT -> CLASS-MATE, u.STUDENT **where**
(t.COURSE = u.COURSE)*(t.STUDENT != u.STUDENT). (Q_9)

This query corresponds to the question "What are pairs of students taking the same course?" The presence of 't.STUDENT -> CLASS-MATE' in the retrieve set of Q_9 indicates that in the result of this query, which is

{<Kirk,Rabkin>, <Rabkin,Kirk>}

attribute STUDENT associated with tuple variable t is to be renamed to CLASS-MATE.

The language also contains '**not**' and '**or**'. *PIQUE interpretation of* '**not**' is motivated by the following example. Consider the query

retrieve t.C **where not** (t.F). (Q_{10})

It is reasonable to require that this query returns the list of courses that have no faculty members associated with them, i.e., the courses not included in $\pi_C([|FC|])$. On the other hand, it is also reasonable to require that the query returns only those courses that actually exist, i.e., the ones appearing in $[|C|]$. The above motivates the following interpretation.

First, compute the answer to the query

retrieve t.C. (Q_{11})

Then, compute the answer to the query

retrieve t.C **where** (t.F). (Q_{12})

Finally, return the relational set difference of the latter from the former as the answer. Thus, the answer to query Q_{10} is

{<CS112>}.

Note again that tuple variable t is bound to different connections, namely $[|C|]$ and $[|FC|]$, at different stages of the computation.

Consider another query involving '**not**'.

retrieve t.C **where not** (t.S = Kirk) (Q_{13})

In designing PIQUE, it was considered important that connectives be interpreted independently of conditions they connect. In particular, interpretation of '**not**' should not depend on the form of the condition in the query. If the query above is interpreted in a manner similar to that of the previous query, then it returns the existing courses that are not taken by any student, i.e., have no students enrolled, as well as those not taken by Kirk. Thus, it returns

{<CS112>, <CS303>}.

Query Q_{13} then corresponds to the question "What are the courses not taken by Kirk?"

Note that condition '**not** (t.S = Kirk)' is not the same as condition 't.S != Kirk', where symbol '!=' stands for "not equal." Consider the query

retrieve t.C **where** (t.S != Kirk). (Q_{14})

This query is evaluated in exactly the same manner as if it had an equality condition. In the course of its evaluation, tuple variable t ranges over ||men(t)||, in this case ||SC||. Those tuples from ||SC|| that satisfy condition 't.S != Kirk' are selected, and their C-values are returned. Therefore, this query returns all courses taken by at least one student other than Kirk, or

{<CS120>, <CS303>}.

Thus, query Q_{14} corresponds to the question "What are the courses that students other than Kirk are taking?"

In PIQUE, neither of the above two questions can be expressed in terms of the other. By distinguishing between '**not** (... = ...)' and '(... != ...)', more flexibility in posing questions is allowed to the user. Also, if θ is a binary comparator from {!=, <, <=, >, >=}, then in general conditions '**not** (... θ ...)' and '(... *not* θ ...)' are not the same. To emphasize this point, consider the queries

retrieve t.C **where not** (t.S <= Rabkin) (Q_{15})

and

retrieve t.C **where** (t.S > Rabkin), (Q_{16})

where '<=' and '>' indicate alphabetical order. The first query will return

{<CS112>, <CS303>},

i.e., the list of existing courses that are not taken by any student whose last name comes before or is equal to "Rabkin." The second query, on the other hand, will return

{<CS303>},

i.e., the list of courses that are taken by students whose last names come after "Rabkin."

There is a major difference between the use and interpretation of '**not**' in PIQUE queries and in Quel queries. In PIQUE, the use of '**not**', in effect, allows us to specify a relational set difference in a single query. In Quel, however, this is not the case. Taking a true relational set difference in Quel involves a creation of a temporary relation, and the use of the '**delete**' command.

The semantics of '**or**' in the language is analogous to that of '**and**' with union substituted for intersection. Thus, the query

retrieve t.C **where** (t.F = Smith) **or** (t.S = Kirk) (Q_{17})

corresponds to the question "Which courses are being taken by Kirk or taught by Smith?" The answer to this query is

{<CS120>, <CS302>, <CS303>}.

De Morgan's and Distributive Laws hold for the interpretations of '**and**', '**or**' and '**not**'. In PIQUE queries, '**not**' has precedence over '**and**', which in turn has precedence over '**or**'. These rules of precedence can be overridden with parentheses.

To illustrate the simplicity with which complex English questions can be posed, consider retrieving the courses which are either taught by Smith and taken by Kirk, but not taught by Smith to Kirk, or have no assistants. In Quel, the query posing this question requires the use of four tuple variables bound to appropriate relations over FC, SC, FCS, and AC. In PIQUE, the above question can be posed using only one tuple variable (which is conveniently chosen to be the blank tuple variable). The query is

retrieve C **where** (F = Smith) **and** (S = Kirk) **and**
not (F = Smith)*(S = Kirk) **or not** (A). (Q_{18})

The answer to Q_{18} is

{<CS120>, <CS303>}.

As another example consider the query

retrieve S **where** (C = CS120) **and** (C = CS302). (Q_{19})

This query corresponds to the question "Which students take both CS120 and CS302 courses?" and returns

{<Kirk>}

as the result. An analogous Quel query would always return the empty relation as the answer, since the condition '(C = CS120) **and** (C = CS302)' is always False when evaluated with respect to any *single* tuple.

4.3. Query Nesting

PIQUE also provides a facility for query nesting. Query nesting can be used in three ways: to restrict the answer of the query, to override the tuple binding mechanism, and to test for subset or superset inclusion.

A *semijoin*, denoted by symbol '*<', and an *antisemijoin*, denoted by symbol '!*<', are binary relational algebra operations, defined as follows.

$$r(R) *< s(S) = \pi_R(r(R) * s(S))$$

$$r(R) !*< s(S) = r(R) - \pi_R(r(R) * s(S))$$

While the operations of semijoin and antisemijoin are defined with respect to natural join, it is straightforward to extend them to arbitrary θ-joins [Ro].

Restriction of an answer is done by performing a semijoin or an antisemijoin of the intermediate result in the query evaluation, with respect to the answer to the subquery. In particular, a subquery that retrieves a single attribute can be substituted for one of the attributes everywhere a comparison between two attributes is allowed.

For the convenience of users, a subquery is allowed to have a retrieve set consisting of several attributes. In that case, since renaming of attributes is allowed in PIQUE, the user has to specify a map for the comparison. That is, the user has to specify how the attributes from the main query are to be compared with the attributes from the retrieve set of the subquery.

As an example of the use of the restriction facility consider the query

retrieve C **where** (S **in retrieve** S **where** (P = Smith)). (Q_{20})

Subquery

retrieve S **where** (P = Smith), (Q_{21})

which returns the list of all students taught by Smith is evaluated first. The answer to subquery Q_{21} is

{<Kirk>}.

Then, during the evaluation of the main query, tuples over SC are compared with the result of subquery Q_{21}. Keyword '**in**' indicates that the comparison is to be done by semijoin. The absence of a comparison map indicates that the semijoin is based on the natural join. Query Q_{20}, then, corresponds to the question "What are the courses taken by students who are taught by professor Smith?" The answer to Q_{20} is

{<CS120>, <CS302>}.

An equivalent query not using the restriction facility is

retrieve C **where** (S = t.S)*(t.P = Smith). (Q_{22})

While this query is shorter, its structure does not reflect the structure of the question, thus making it less intuitive than query Q_{20}. The difference in the clarity of the queries that use the restriction facility and those that do not increases rapidly as queries become more complex.

As another example of the use of the restriction facility consider the query

retrieve A **where** (A **notin** A = S **retrieve** S). (Q_{23})

Again, the subquery

retrieve S (Q_{24})

is evaluated first. It returns the list of all students, which is

{<Kirk>, <Rabkin>, <Ross>}.

Then, during the evaluation of Q_{23}, tuples from [|A|] are compared with the result of subquery Q_{24}. Keyword '**notin**' indicates that the comparison is to be done by antisemijoin, while 'A = S' provides a map for the comparison. In particular, it indicates that attribute A from [|A|] is to be compared for equality with attribute S in the answer to the subquery. Thus, this map indicates that the antisemijoin is based on the equijoin. Query Q_{23}, then, corresponds to the question "Who are the assistants that are not among the students?" The answer to Q_{23} is

{<Chung>, <Johnson>}.

While this question can be posed in PIQUE without the use of the restriction facility, the resulting query would not be natural.

Overriding tuple variable binding is performed using keyword '**from**'. As an example consider the query

retrieve t.PERSON **where** (t **from retrieve** STUDENT -> PERSON)
or (t **from retrieve** ASSISTANT -> PERSON). (Q_{25})

This query corresponds to the question "Who is either a student or an assistant?" The answer to this query is obtained in the following way.

First, compute the answer to the first subquery

retrieve STUDENT -> PERSON. (Q_{26})

Note that attribute STUDENT is renamed to PERSON. The answer to this query is

{<Kirk>, <Rabkin>, <Ross>}.

Then, compute the answer to the query

retrieve t.PERSON **where** (t **from retrieve** STUDENT -> PERSON), (Q_{27})

by binding tuple variable t to the answer of subquery Q_{26}. (The answer to Q_{27} is, of course, the same as the answer to its subquery.)

Second, compute the answer to the second subquery

retrieve ASSISTANT -> PERSON. (Q_{28})

Again, note the renaming. The answer to this query is

{<Chung>, <Johnson>, <Ross>}.

Then, compute the answer to the query

retrieve t.PERSON **where** (t **from retrieve** ASSISTANT -> PERSON). (Q_{29})

Again, the answer to Q_{29} is the same as the answer to its subquery.

Finally, return the union of the answers to Q_{27} and Q_{29} as the result. Thus, the answer to query Q_{25} is

{<Chung>, <Johnson>, <Kirk>, <Rabkin>, <Ross>}.

We note that this question cannot be asked at all in PIQUE without the use of the facilities for attribute renaming and binding overriding. We also note that posing the above question in Quel requires the creation of a temporary relation and the use of the '**append**' command.

Testing for subset inclusion is performed using keyword '**of**' in conjunction with keyword '**include**' that corresponds to superset inclusion, or keyword '**among**' that corresponds to subset inclusion. As an example of superset inclusion, consider the query

retrieve S **where** (C **of** S **include retrieve** C **where** (A = Johnson)). (Q_{30})

This query corresponds to the question "Who are the students that have taken all of the courses that are assisted by Johnson?" The answer to Q_{30} is computed in the following way.

First, compute the answer to the subquery

retrieve C **where** (A = Johnson). (Q_{31})

The answer to Q_{31} is

{<CS302>}.

Then, divide (in the relational algebra sense) [SC] by the answer to this subquery. The result, which is a relation over S, is

{<Kirk>}.

Finally, during the evaluation of the main query, restrict the answer to those tuples that are contained in the result of the division. In other words, perform a semijoin with respect to {<Kirk>}. The answer to query Q_{30} is

{<Kirk>}.

As an example of subset inclusion, consider the query

retrieve S **where** (C **of** S **among retrieve** C **where** (F = Smith)). (Q_{32})

This query corresponds to the question "Who are the students that have taken only those courses that are taught by Smith?" The answer to this query is computed in a manner similar to the previous computation, except that the division is taken in the opposite direction. (A formal description of how this is done is provided in the appendix.)

4.4 PIQUE Implementation

There exists a running compiler for a substantial subset of PIQUE. The only features of PIQUE that are not implemented are: the facility to test subset and superset inclusion, and the facility to perform regular expression matching. The compiler was written in the C language and runs under the UNIX operating system. The parser for PIQUE was created by the YACC system [Jo].

After a PIQUE query has been parsed, its intermediate form is stored in six system relations. These relations satisfy the URA, and thus can be made part of a database. This allows users to precompile their queries and to store their compiled form over a period of time. This is especially useful if several queries share the same subquery.

The algorithm for evaluating PIQUE queries, which corresponds to the examples discussed in this section and is consistent with the formal definition of PIQUE semantics described in the appendix, is presented below.

Algorithm

1. Expressions are constructed for the answers to all subqueries of a given query Q.
2. Query Q is decomposed into a set of *simple* queries $\{Q_1, ..., Q_n\}$, where the condition in each of these simple queries does not contain keywords '**not**', '**and**' or '**or**'.
3. All tuple variables in each simple query Q_i are appropriately bound. That is, if a tuple variable t appears with keyword '**from**' in the condition of Q_i, then an expression for the answer to the corresponding subquery is constructed; else, the expression for $[|men(t)|]$ is constructed.
4. For each simple query Q_i, a cross product of the expressions for all tuple variables is taken. Let the resulting expressions be $E_{i,1}$.
5. For each simple query Q_i, restrictions (i.e., semijoins and antisemijoins) specified

in the condition of Q_i are applied to $E_{i,1}$. Let the resulting expressions be $E_{i,2}$.

6. For each simple query Q_i, selections specified in the condition of Q_i are applied to $E_{i,2}$. Let the resulting expressions be $E_{i,3}$.

7. For each simple query Q_i, projections onto attributes specified in the retrieve set of query Q are applied to $E_{i,3}$; necessary renamings are performed. Let the resulting expression be $E_{i,4}$. Note that these expressions represent answers to simple queries $\{Q_1, ..., Q_n\}$.

8. Expressions $E_{i,4}$ are combined with relational set difference, intersection and union, according to '**not**', '**and**' and '**or**' connectives in the condition of query Q. The resulting expression is evaluated and the result is returned as the answer to query Q.

$\square$

Eris [Re2] is a relational database system developed at Brown University. It is based on the path model of relational database implementation proposed by Reiss [Re1]. The PIQUE compiler translates the intermediate form of queries into relational algebra supported by Eris. Because Eris optimizes relational algebra expressions, the cross products taken in Step 4 of the Algorithm above are not actually computed unless necessary.

5. Conclusions

5.1. Summary

This paper presents PIQUE, an attribute-based relational query language.

Section 1 motivates the need to go beyond physical data independence, towards structural data independence. Section 2 considers views, the traditional method of hiding the structural details of the database from the user. However, views do not always provide the user with a semantically consistent and intuitive image of the underlying database. Universal scheme interfaces, also considered in Section 2, are a viable and semantically consistent alternative for providing structural data independence.

The concept of a window generator, discussed in Section 2, is central to many universal scheme interfaces. Section 3 examines in detail the object-based (OB) generator, which is based on the semantic notions of decomposable and non-decomposable facts. PIQUE, an attribute-based tuple calculus-like language developed for the use with the OB generator is presented in Section 4 and the appendix. Section 4 provides examples motivating the syntax and the semantic interpretation of PIQUE queries. The appendix, on the other hand, contains PIQUE's formalization and the proof of its relational completeness.

The distinguishing characteristic of PIQUE is that the syntactic structure of queries together with the extended database scheme determines the range of tuple variables. We believe PIQUE to be a powerful, yet natural, language. Many queries can be posed in PIQUE without explicitly using any tuple variables at all. In particular, given the appropriate choice of objects, these queries include the important class of "project-select-join" queries of the form

$$\pi_{<\text{Set of attributes}>}(\sigma_{<\text{Condition}>}(r_1 * \dots * r_n)).$$

Also there exists a large class of queries that can be asked using many fewer tuple variables than in other tuple calculus-like languages.

5.2. Related and Future Work

Recall that the OB generator depends on the unique role assumption (URA) for navigation. While any database can be made to satisfy the URA by renaming of attributes, certain semantic information may be lost. For example, consider a database with relations HAS(EMPLOYEE,SALARY), WORKS-IN(EMPLOYEE,DEPARTMENT) and MANAGES(EMPLOYEE,DEPARTMENT). This database does not satisfy the URA because it contains two semantically distinct relationships between employees and departments.

The database can be made to satisfy the URA by renaming attribute EMPLOYEE in one of the relations; say, in relation MANAGES to MANAGER. However, certain semantic information, namely that EMPLOYEE and MANAGER represent entities from the same class, but in different *roles*, is lost from the database. Furthermore, nothing in the renamed database indicates how to establish a relationship between managers and their salaries (Note that in the original database it was the presence of the same attribute EMPLOYEE in both relations HAS and MANAGES that indicated how to compute salaries of managers.)

In [MRS1,MRS2] we explore means of recapturing this information by incorporating role hierarchies into the database scheme. We then show how the role information can be used by a generator to automatically navigate within the database in new ways. Unfortunately, even in the presence of role information, there may exist several semantically different ways of establishing a relationship among a given set of attributes. In that case, it may be possible for the user to disambiguate the request by explicitly specifying some role information in the query. We are currently investigating ways of incorporating the role information into PIQUE.

In this paper, we have presented a computational definition of the OB generator. Stein [St,SM] develops an equivalent definition based on ***representative instances*** [Ho2,Sa]. The data dependencies inherent in the model are then explored. These dependencies are similar to functional, join and template dependencies, except for their reliance on *no information nulls* [Za]. The dependencies developed are well behaved, possessing a complete, self contained axiomatization. We intend to use these dependencies to develop normal forms for the fact-based databases satisfying the URA.

Bibliography

[A-G]	Z. Arazi-Gonczarowsky. A high-level interface for users in a relational database. Manuscript, Dept. of Computer Science, Hebrew University, 1983.
[AK]	A.V. Aho, B.W. Kernighan. Program research!user/ava/q/README, 1980.
[Ba]	E. Babb. Joined normal form: A storage encoding for relational databases, *ACM Transactions on Database Systems*, 7:4, December 1982.
[BB]	J. Biskup, H.H. Bruggeman. Universal relation views: A pragmatic approach, Technical Report 150, University of Dortmund, March 1983.
[CA]	D.D. Chamberlin, et al. SEQUEL2: A unified approach to data definition, manipulation and control. *IBM Journal of Research and Development*, 20:6, pp. 560-675, November 1976.
[CK]	C.R. Carlson, R.S. Kaplan. A generalized access path model and its application to a relational database system. *Proceedings of the ACM-SIGMOD Conference*, June 1976.
[Da]	C.J. Date. *An Introduction to Database Systems.* 3rd edition. Addison-Wesley, 1982.
[Ho2]	P. Honeyman. Testing satisfaction of functional dependencies. *Journal of the ACM*, 29:3, pp. 668-677, July 1982.
[HU]	J.E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.
[Jo]	S.C. Johnson. YACC - yet another compiler compiler. Computing Science Technical Report 32, Bell Laboratories, July 1975.
[Ko]	H.F. Korth. System/U: A progress report. *Proceedings of the XP2 Workshop on Relational Database Theory*, June 1981.
[KKFGU]	H.F. Korth, G.M. Kuper, J. Feigenbaum, A. van Gelder, J.D. Ullman. System/U: A database system based on the the universal relation assumption. *ACM Transactions on Database Systems*, 9:3, pp. 331-348, September 1984.
[KMRS]	S.M. Kuck, D.A. McNabb, S.V. Rice, Y. Sagiv. The Parafrase database user's manual. Computer Science Technical Report 80-1046, University of Illinois, December 1980.
[KS]	S.M. Kuck, Y. Sagiv. A universal relation database system implemented via the network model. *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.
[KU]	H.F. Korth, J.D. Ullman. System/U: A database system based on the universal relation assumption. *Proceedings of the XP1 Workshop on Relational Database Theory*, June-July 1980, Stony Brook, N.Y.
[Ma]	D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.
[MRSSW]	D. Maier, D. Rozenshtein, S. Salveter, J. Stein, D.S. Warren. Towards

logical data independence: A relational query language without relations. *Proceedings of the ACM-SIGMOD Conference*, June 1982, Orlando, Fla.

[MRS1] D. Maier, D. Rozenshtein, J. Stein. Representing roles in universal scheme interfaces. Extended Abstract. *Proceedings of the IEEE Computer Data Engineering Conference*, April 1984, Los Angeles, Ca.

[MRS2] D. Maier, D. Rozenshtein, J. Stein. Representing roles in universal scheme interfaces. *IEEE Transactions on Software Engineering*, SE-11:7, pp. 644-652, July 1985.

[MRW1] D. Maier, D. Rozenshtein, D.S. Warren. Windows on the world. *Proceedings of the ACM-SIGMOD Conference*, May 1983, San Jose, Ca.

[MRW2] D. Maier, D. Rozenshtein, D.S. Warren. Window functions. *Advances in Computing Research*, vol. 3, JAI Press, Inc., Greenwich, Ct, 1986.

[MW] D. Maier, D.S. Warren. Specifying connections for a universal relation database scheme. *Proceedings of the ACM-SIGMOD Conference*, June 1982, Orlando, Fla.

[MU] D. Maier, J.D. Ullman. Maximal objects and the semantics of universal relation databases. *ACM Transactions on Database Systems*, 8:1, pp. 1-14, March 1983.

[Re1] S.P. Reiss. The path model of relational database implementation. Unpublished manuscript, Brown University, December 1981.

[Re2] S.P. Reiss. Eris reference manual. Unpublished manuscript, Brown University, January 1982.

[Ro] D. Rozenshtein. *Query and Role Playing in the Association-Object Data Model.* Doctoral Dissertation, State University of New York at Stony Brook, 1983.

[Sa] Y. Sagiv. Can we use the universal instance assumption without using nulls? *Proceedings of the ACM-SIGMOD Conference*, April - May 1981, Ann Arbor, Mich.

[Sc] E. Sciore. *The Universal Instance and the Database Design.* Doctoral Dissertation, Princeton University, 1980.

[SP] K.L. Schenk, J.R. Pinkert. An algorithm for servicing multirelational queries. *Proceedings of the ACM-SIGMOD Conference*, August 1977.

[St] J. Stein. *Constraints in the Association-Object Data Model.* Doctoral Dissertation in preparation, State University of New York at Stony Brook.

[SM] J. Stein, D. Maier. Relaxing the universal relation scheme assumption. *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1985.

[SWKH] M. Stonebraker, E. Wong, P. Kreps, G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1:3, pp. 189-222, September 1976.

[Ul1] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1980.

[Ul2] J.D. Ullman. The U.R. strikes back. *Proceedings of the ACM*

Symposium on Principles of Database Systems. March 1982.

[Zl] M.M. Zloof. Query-by-example: A database language. *IBM Systems Journal*, 16:4, pp. 324-343, December 1977.

[Za] C. Zaniolo. Database relations with null values. *Proceedings of the ACM Symposium on Principles of Database Systems*, 1982.

Appendix

In this appendix, we present a formal definition of the PIQUE query language.

A1. PIQUE Syntax

As usual, let U denote the universal set of attributes, and let A and B stand for attributes in U. Letters t and u are tuple variables. Symbol θ stands for a binary comparator from {=, !=, <, <=, >, >=, <<, =re}, where symbol '<<' denotes substring matching and symbol '=re' denotes regular expression matching. Letter c denotes a constant. Symbol ϵ represents the empty string. The notion of domain is extended to tuple-attribute pairs by letting dom(t.A) be equal to dom(A).

If t is a tuple over a set of attributes Y and $A \in Y$, then t(A) stands for the value of attribute A in t. Also, if $X \subset Y$, then t(X) stands for the restriction of t to X.

PIQUE grammar is presented below. For the sake of clarity, nonterminal symbols in the productions are indicated by enclosing them with angle brackets, while keywords are shown in boldface.

```
<term>     ::= t.A
<inset>    ::= <term> | <term><inset>
<terms>    ::= <term> | <term><terms>
<map>      ::= <term> θ B | <term> θ B, <map>
<retset>   ::= <term> -> B | <term> -> B, <retset>
<econd>    ::= ϵ | (<term>) | (c θ <term>) | (<term> θ c) |
                 (<term> θ <term>) | (t from <query>) |
                 (<inset> in <map><query>) |
                 (<inset> notin <map><query>)
                 (<terms> of <terms> include <query>) |
                 (<terms> of <terms> among <query>)
<scond>    ::= <econd> | <econd>*<scond>
<ccond>    ::= <scond> | (<ccond>) | not <ccond> |
                 <ccond> and <ccond> | <ccond> or <ccond>
<query>    ::= retrieve <retset> where <ccond>
```

In the grammar, nonterminal <term> stands for a tuple-attribute pair. The retrieve set of a query is denoted by nonterminal <retset>, which is a sequence of renamed terms. The symbol '->' in '<term> -> B' indicates that, in the answer to the query, '<term>' is to be renamed to 'B'. Since duplicate attributes are disallowed in the answer to a query (which is, of course, a relation), no two '<term> -> B' elements of <retset> may have the same B. Throughout this appendix, Bs in the retrieve set of the query are referred to as *new names* for the attributes. Also, if B is already an attribute in the database, then the domain of <term> must be equal to the domain of B. Finally, even though nonterminals <inset> and <terms> have structurally identical definition, they are used in the grammar for quite different purposes. Thus, for the sake of clarity, they are not combined into a single nonterminal.

There are three levels of complexity of conditions in PIQUE queries. The simplest of them correspond to nonterminal <econd> and are called *elementary conditions*. There are several type restrictions that apply to elementary conditions. In particular, θ must be defined on the domains of constituent <term>s, and constant c must be an element of the proper domain. Also, if θ denotes a regular expression match, then constant c is taken to be a regular expression pattern, expressed in a standard notation [HU].

There are three kinds of elementary conditions that deserve special attention. First, an elementary condition may involve keywords '**in**' or '**notin**'. In that case it specifies a restriction by semijoin or by antisemijoin. The <inset> specifies a list of tuple-attribute pairs to be used in the comparison with the answer to subquery <query>. The <map> specifies how the comparison is to be performed. If an elementary condition indeed involves a semijoin or an antisemijoin, then the following restrictions apply: the <term>s in <inset> must be the <term>s in <map>, and each new name B in <map> must be among new names in the <retset> of the subquery <query>.

Second, an elementary condition may involve keyword '**from**'. This condition specifies explicit binding for the tuple variable involved.

Third, an elementary condition may involve keyword '**of**'. This condition specifies a restriction that depends on whether the answer to subquery <query> is a subset (in the case of '**include**') or a superset (in the case of '**among**') of a particular projection of a certain intermediate result in the query evaluation. If an elementary condition describes this type of restriction, then the domain of the i-th <term> in <terms> (where <terms> is the nonterminal that appears just before '**of**') must be the same as the domain of the i-th new name B in <retset> of subquery <query>.

The slightly more complex conditions are referred to as *simple conditions.* They are constructed from elementary conditions using the '*' operator. Simple conditions correspond to nonterminal <scond>. There is only one restriction applicable to simple conditions: there may be at most one explicit binding for a particular tuple variable in a simple condition.

The most complex conditions are those involving '**not**', '**and**', and '**or**' operators connecting simple conditions. They correspond to nonterminal <ccond> and are referred to as *complex conditions.*

A query in PIQUE is specified by specifying a retrieve set and a condition.

Example 1:

As an example, consider the following three query derivation sequences, where symbol '$\Rightarrow$' stands for "derives in one step," symbol '$*\Rightarrow$' stands for "derives in one or more steps," and letters C, D, E, G, K, L, M denote attributes.

<query> ⇒ **retrieve** <retset> **where** <ccond> *⇒
retrieve <term> -> E, <retset> **where** <econd>*<scond> *⇒
retrieve t.C -> E, u.D -> F **where** (t.C)*(t.G = u.K)*(u.L > 5) (Q_1)

<query> ⇒ **retrieve** <retset> **where** <ccond> *⇒
retrieve <term> -> N **where** <econd> *⇒
retrieve t.M -> N **where** (t **from** <query>) *⇒
retrieve t.M -> N **where** (t **from retrieve** u.C -> M **where**) (Q_2)

<query> ⇒ **retrieve** <retset> **where** r<ccond> *⇒
retrieve <term> -> E **where** <econd> **or** <econd> *⇒
retrieve t.D -> E **where** (t.K = 9) **or** (t.L = 7) (Q_3)

[]

In concluding this section, we note that because of the incorporation of certain user-friendly features into the language, the grammar used in the actual implementation of PIQUE differs slightly from the grammar presented above. These features include the ability to drop keyword '**where**' from the syntax of the query if the condition is empty, and the ability not to rename attributes in the retrieve set unless necessary.

A2. Formal Semantics of PIQUE Queries

In this section, we present rules for semantic interpretation of PIQUE queries. To illustrate these rules, we shall use the three PIQUE queries derived in the previous section. Our example database shall consist of two relations r(CG) and r(DKL), with no additional objects. The instances for these relations are presented below.

C	G
c	2
e	4

D	K	L
d	9	3
f	4	7

Throughout this appendix, we shall often want to distinguish a restricted class of PIQUE queries: those whose conditions do not involve keywords '**not**', '**and**' or '**or**'. Each query from this class has the form

retrieve <retset> **where** <scond>

and is referred to as *simple query*.

Recall that, in PIQUE queries, tuple variables are bound automatically. In case of a simple query, a tuple variable t is bound to the window for the ***mention set*** of t: the set of attributes that appears with t in the query.

Function men(t, Q) that computes the mention set of tuple variable t in simple query Q is defined recursively as follows.

```
For <term>:
        men(t, u.A) = {A} if t = u,
                    = $\phi$ otherwise.

For <inset>:
        men(t, <term>) is defined above;
        men(t, <term>,<inset>) = men(t, <term>) $\cup$ men(t, <inset>).

For <terms>:
        men(t, <term>) is defined above;
        men(t, <term>,<terms>) = men(t, <term>) $\cup$ men(t, <terms>).

For <retset>:
        men(t, <term> -> B) = men(t, <term>);
        men(t, <term> -> B,<retset>) = men(t, <term>) $\cup$ men(t, <retset>).

For <econd>:
        men(t, $\epsilon$) = $\phi$;
        men(t, (<term>$_1$ $\theta$ <term>$_2$)) = men(t, <term>$_1$) $\cup$ men(t, <term>$_2$);
        men(t, (<term> $\theta$ c)) = men(t, <term>);
        men(t, (c $\theta$ <term>)) = men(t, <term>);
        men(t, (<term>)) = men(t, <term>);
        men(t, (<inset> **in** <map><query>)) = men(t, <inset>);
        men(t, (<inset> **notin** <map><query>)) = men(t, <inset>);
        men(t, (u **from** <query>)) = $\phi$, even if t = u;
        men(t, (<terms>$_1$ **of** <terms>$_2$ **include** <query>))
                             =men(t, <terms>$_1$) $\cup$ men(t, <terms>$_2$);
        men(t, (<terms>$_1$ **of** <terms>$_2$ **among** <query>))
                             =men(t, <terms>$_1$) $\cup$ men(t, <terms>$_2$).

For <scond>:
        men(t, <econd>) is defined above;
        men(t, <econd>*<scond>) = men(t, <econd>) $\cup$ men(t, <scond>).

For simple query Q:
        men(t, **retrieve** <retset> **where** <scond>)
                                   = men(t, <retset>) $\cup$ men(t, <scond>).
```

Example 2:

Consider tuple variable t from query Q_1. Its retrieve set contributes {C} to men(t), and its condition contributes {CG} to men(t). Thus, men(t) in Q_1 is {CG}. On the other hand, men(u) in Q_1 is {DKL}.

Consider tuple variable t from query Q_2. The retrieve set of Q_2 contributes {M} to men(t), while its condition contributes nothing to men(t). Thus, men(t) in Q_2 is {M}. On the other hand, men(u) in Q_2 is ϕ.

Note that the definition does not apply to query Q_3, since its condition contains keyword '**or**'.

[]

Again, let Q be a simple query of the form

retrieve <retset> **where** <scond>.

Let T_e be the set of *externally* defined tuple variables in simple query Q, i.e., those tuple variables that appear with keyword '**from**' in the condition of Q. Formally, T_e is defined as

T_e = {t | '(t **from** ...)' appears in condition of simple query Q}.

Let T_i be the set of *internally* defined tuple variables in simple query Q, i.e., those tuple variables that *do not* appear with keyword '**from**' in the condition of Q. Formally, T_i is defined as

$$T_i = \{t \mid t \notin T_e \text{ and } \mathrm{men}(t, Q) \neq \phi\}.$$

Example 3:

For query Q_1, T_e is empty and T_i is {t, u}. On the other hand, for query Q_2, T_e is {t} and T_i is empty. Again note that these definitions do not apply to query Q_3, since it is not simple.

[]

Let function ξ stand for some tuple assignment function. In other words, function ξ assigns a particular tuple over men(t, Q) to each tuple variable t in $T_i \cup T_e$ for simple query Q. However, not all of the tuple assignment functions can be considered satisfactory. In particular, let Ξ_Q be the set of those ξ that map internally defined tuple variables mentioned in Q to tuples contained in the database (more precisely, to tuples contained in corresponding windows), and externally defined tuple variables to tuples contained in the answers to the appropriate subqueries.

Formally, Ξ_Q is defined as

$$\begin{aligned}\Xi_Q = \{\xi \mid & \forall\, t \in T_i, \xi(t)(\text{men}(t, Q)) \in [|\text{men}(t, Q)|], \text{ and} \\ & \forall\, t \in T_e, \xi(t)(\text{men}(t, Q)) \in \text{ans}(\text{<query>}_t)\},\end{aligned}$$

where <query>_t is '<query>' that appears in 't **from** <query>' in the condition of Q. (Function ans that maps queries to their answers is defined formally later.)

Example 4

Consider again queries Q_1 and Q_2 evaluated against our example database.

Let ξ_1 be a tuple assignment function that maps tuple variable t from Q_1 to tuple <c,2> and tuple variable u from it to tuple <d,9,3>. Then ξ_1 is included in Ξ_{Q_1}. On the other hand, the tuple assignment function ξ_2 that maps t to <c, 6> (and u to any tuple from r(DKL)) is not included in Ξ_{Q_1}.

In fact, there are only four tuple assignment functions that are included in Ξ_{Q_1}. They are the ones that map t to <c,2> and u to <d,9,3>, or map t to <c,2> and u to <f,4,7>, or map t to <e,4> and u to <d,9,3>, or map t to <e,4> and u to <f,4,7>.

The determination of Ξ_{Q_2} is also straightforward. The answer to the subquery

retrieve u.C -> M **where**

is a relation {<c>, <e>} over scheme {M}. The tuple assignment function ξ_3 that maps t from Q_2 to <c> is included in Ξ_{Q_2}, while the tuple assignment function ξ_4 that maps t from it to <g>, where g is some constant distinct from both c and e, is not included in Ξ_{Q_2}.

We note that, since query Q_3 is not simple, the definition above again does not apply to it.

[]

The tuple assignment functions contained in Ξ_Q are not guaranteed, however, to satisfy the selection conditions in the condition of simple query Q. Therefore, let function η_Q select those ξ from Ξ_Q that do satisfy the selection conditions in Q.

Function η_Q is defined recursively for the condition of simple query Q as follows.

For <term>:

$\eta_Q(\xi,$ t.A$)$ = ξ(t)(A), thus mapping t.A to the value of ξ(t) for A.

For <terms>:

$\eta_Q(\xi,$ <term>) is defined above;

$\eta_Q(\xi,$ <term><terms>) = $\eta_Q(\xi,$ <terms>) appended to $\eta_Q(\xi,$ <term>).

For <econd>:

$\eta_Q(\xi, \epsilon)$ = True;

$\eta_Q(\xi$, (<term>$_1$ θ <term>$_2$)) =True if $\eta_Q(\xi$, <term>$_1$) θ $\eta_Q(\xi$, <term>$_2$),
= False otherwise;

$\eta_Q(\xi$, (<term> θ c)) = True if $\eta_Q(\xi$, <term>) θ c,
= False otherwise;

$\eta_Q(\xi$, (c θ <term>)) = True if c θ $\eta_Q(\xi$, <term>),
= False otherwise;

$\eta_Q(\xi$, (<term>)) = True;

$\eta_Q(\xi$, (<inset> **in** <map><query>)) = True
if $\exists$ tuple u $\in$ ans(<query>), such that
$\forall$ elements 't.A θ B' in <map>, ξ(t)(A) θ u(B),
= False otherwise;

$\eta_Q(\xi$, (<inset> **notin** <map><query>)) = True
if $\neg$ $\exists$ tuple u $\in$ ans(<query>), such that
$\forall$ elements 't.A θ B' in <map>, ξ(t)(A) θ u(B),
= False otherwise;

$\eta_Q(\xi$, (t **from** <query>)) = True;

$\eta_Q(\xi$, (<terms>$_1$ **of** <terms>$_2$ **include** <query>)) = True if ξ(<terms>$_2$) $\in$ r,
= False otherwise;

$\eta_Q(\xi$, (<terms>$_1$ **of** <terms>$_2$ **among** <query>)) = True if ξ(<terms>$_2$) $\in$ s,
= False otherwise.

For <scond>:

$\eta_Q(\xi$, <econd>) is defined above;

$\eta_Q(\xi$, <econd>*<scond>) = True
if $\eta_Q(\xi$, <econd>) = True and $\eta_Q(\xi$, <scond>) = True,
= False otherwise.

Relations r and s used in the previous definition are defined as follows. Let relation q be defined as q = ans(**retrieve** <terms>$_1$,<terms>$_2$), and relation p be defined as p = ans(<query>), where '<query>' is the nonterminal that appears after keywords '**include**' or '**among**'.

Then, relation r is defined as r = q / p, where symbol '/' denotes the relational algebra division operator [Da]. In other words, if q is a relation over X and p is a relation over Y, then r is the relation over (X-Y), where r = {z(X-Y) | $\forall$ w $\in$ p, $\exists$ v $\in$ q, such that v(X-Y) = z and v(Y) = w}.

Relation s is defined as s = q \ p, where symbol '\' denotes the *reverse division* operator [Ro]. In other words, if q is a relation over X and p is a relation over Y, then s is the relation over (X-Y), where r = {z(X-Y) | $\forall$ v $\in$ q, such that v(X-Y) = z. $\exists$ w $\in$ p, such that v(Y) = w}. Note, that s = q \ p can also be expressed as s = $\pi_{X-Y}(q)$ - $\pi_{X-Y}(q$ - $(\pi_{X-Y}(q)$ * p))

Thus, given a tuple assignment function $\xi \in \Xi_Q$, η_Q maps it to True if and only if the tuples assigned by ξ to all of the tuple variables used in simple query Q jointly satisfy the selection conditions specified in the condition of Q.

Example 5:

Consider two tuple assignment functions ξ_5 and ξ_6 from Ξ_{Q_1}. Let ξ_5 map t to <e,4> and u to <f,4,7>, and let ξ_6 map t to <c,2> and u to <d,9,3>. Then η_{Q_1} maps ξ_5 to True and maps ξ_6 to False.

[]

Function ans maps queries to their answers. It is defined in two steps.

Semantic Definition 1:

Let Q be a simple query of the form

retrieve <retset> **where** <scond>.

The answer to query Q is the relation over the new names for the attributes in <retset>. Function ans(Q) that maps simple query Q to the relation over the appropriate scheme is defined as follows.

ans(Q) = {u| u is a tuple over attributes B in <retset> of Q,
such that $\exists$ $\xi \in \Xi_Q$, such that $\eta_Q(\xi$, <scond>) = True,
such that $\forall$ t $\in$ $T_i \cup T_e$,
if 't.A -> B' appears in <retset> of Q, then u(B) = ξ(t)(A)}.

[]

Example 6:

The answer to query Q_1, which is a relation over {EF}, is computed as follows. As

Example 4 illustrates, there exist four tuple assignment functions, which are included in Ξ_{Q_1}. However, only one of them, namely ξ_5 from Example 5 that maps t to <e,4> and u to <f,4,7>, satisfies η_{Q_1}. Consider a tuple v from the answer to Q_1. Since 't.C -> E' appears in the retrieve set of Q_1, v(E) must be equal to $\xi_5(t)(C)$, which is, in turn, equal to e. Likewise, since 'u.D -> F' appears in the retrieve set of Q_1, v(F) must be equal to $\xi_5(u)(D)$, which is, in turn, equal to f. Therefore, the answer to Q_1 is {<e,f>}.

The answer to query Q_2 is computed as follows. First, the subquery

retrieve u.C -> M **where**

is evaluated. The answer to this subquery, which is a relation over {M}, is {<c>,<e>}. Then the main query itself is evaluated. The answer to it, which is a relation over {N}, is {<c>,<e>}.

[]

Semantic Definition 2:

Let Q be a (not necessarily simple) query of the form

retrieve <retset> **where** <ccond>,

where <ccond> is assumed to be fully parenthesized. The answer to query Q is obtained in the following way.

ans(**retrieve** <retset> **where** <scond>) is defined above.

ans(**retrieve** <retset> **where not** <ccond>) =
ans(**retrieve** <retset> **where**) -
ans(**retrieve** <retset> **where** <ccond>).

ans(**retrieve** <retset> **where** $<\text{ccond}>_1$ **and** $<\text{ccond}>_2$) =
ans(**retrieve** <retset> **where** $<\text{ccond}>_1$) $\cap$
ans(**retrieve** <retset> **where** $<\text{ccond}>_2$).

ans(**retrieve** <retset> **where** $<\text{ccond}>_1$ **or** $<\text{ccond}>_2$) =
ans(**retrieve** <retset> **where** $<\text{ccond}>_1$) $\cup$
ans(**retrieve** <retset> **where** $<\text{ccond}>_2$).

Example 7:

The answer to query Q_3, which is a relation over {E}, is computed as follows. First, Q_3 is broken into two queries,

retrieve t.D -> E **where** (t.K = 9)

and

retrieve t.D -> E **where** (t.L = 7)

The answer to the first of these queries, which is a relation over {E}, is {<d>}. The answer to the second, which also is a relation over {E}, is {<f>}. Therefore, the answer to Q_3 is a relation over {E}, and is equal to {<d>,<f>}.

$\square$

Even though the semantics of queries was not defined directly over the syntax, the answer is defined for any syntactically correct query Q, as the next theorem shows.

Theorem 1:

Function ans(Q) is defined for any expression Q of the form

retrieve <retset> **where** <ccond>.

Proof: (by induction on n: the number of **not**, **and**, and **or** connectives in <ccond>)

Basis: n = 0. Then Q is of the form

retrieve <retset> **where** <scond>

which is defined by Semantic Definition 1 above.

Induction: Let ans(Q) be defined for any query Q whose condition contains fewer than n connectives. Then one of the following three cases applies.

Case 1: Q is of the form

retrieve <retset> **where not** <ccond>.

Then <ccond> has n - 1 connectives. Therefore, ans(**retrieve** <retset> **where** <ccond>) is defined. Therefore, ans(**retrieve** <retset> **where not** <ccond>) is defined.

Case 2: Q is of the form

retrieve <retset> **where** <ccond>$_1$ **and** <ccond>$_2$.

Then each of <ccond>$_1$ and <ccond>$_2$ has at most n - 1 connectives. Therefore, ans(**retrieve** <retset> **where** <ccond>$_1$) is defined, and ans(**retrieve** <retset> **where** <ccond>$_2$) is defined. Therefore, ans(**retrieve** <retset> **where** <ccond>$_1$ **and** <ccond>$_2$) is defined.

Case 3: Q is of the form

retrieve <retset> **where** <ccond>$_1$ **or** <ccond>$_2$.

Then each of <ccond>$_1$ and <ccond>$_2$ has at most n - 1 connectives. Therefore, ans(**retrieve** <retset> **where** <ccond>$_1$) is defined, and ans(**retrieve** <retset> **where** <ccond>$_2$) is defined. Therefore, ans(**retrieve** <retset> **where** <ccond>$_1$ **or** <ccond>$_2$) is defined.

[]

The PIQUE query language is relationally complete. Moreover, unlike Quel [SWKH], PIQUE is *single-query* complete. Any relational algebra expression can be posed by a single PIQUE query (with subqueries, of course). The next theorem shows that relational algebra that uses renaming, union, difference, product, selection and projection, is reducible to PIQUE.

Theorem 2:

Assume a relational database satisfying the URA. Let r(R) identify the stored relation over scheme R. Furthermore, assume that no additional objects have been declared for this database. Thus, [|R|] = r(R) for any stored relation r(R).

Let E be a relational algebra expression posed with respect to this database. Then there is a PIQUE query Q equivalent to E (written $Q \equiv E$).

Proof: (by induction on n: the number of connectives in E)

Basis: $n = 0$. Then E is just r(R) for some relation r(R), where R is a set of attributes $\{A_1, ..., A_n\}$. Construct Q to be the following query.

retrieve $A_1, ..., A_n$ **where**

Then $Q \equiv E$.

Induction: Let the theorem be true for all E with fewer than n connectives. Then one of the following six cases applies.

Case 1 (Renaming): E is of the form $\delta_{(B_1 \text{->} C_1, ..., B_k \text{->} C_k)}(E_1)$, where E_1 results in a relation over $\{B_1, ..., B_k, B_{k+1}, ..., B_n\}$ and E results in a relation over $\{C_1, ..., C_k, B_{k+1}, ..., B_n\}$. By the inductive hypothesis there exists query $Q_1 \equiv E_1$. Let this query Q_1 be

retrieve <retset>$_1$ **where** <ccond>$_1$.

Construct <retset> to contain 't.A -> C' if 't.A -> B' appears in <retset>$_1$ and 'B -> C' appears in the renaming, and contain 't.A -> B' if 't.A -> B' appears in <retset>$_1$ and 'B -> C' does not appear in the renaming. Construct Q to be the following query.

retrieve <retset> **where** <ccond>$_1$

Then $Q \equiv E$.

Case 2 (Union): E is of the form $E_1 \cup E_2$. By the inductive hypothesis there exist queries $Q_1 \equiv E_1$ and $Q_2 \equiv E_2$. Let query Q_1 be

retrieve <retset>$_1$ **where** <ccond>$_2$

and let query Q_2

retrieve <retset>$_2$ **where** <ccond>$_2$.

For $E_1 \cup E_2$ to make sense the schemes of E_1 and E_2 have to be the same. Construct <retset> to contain 't.B -> B' if new name 'B' appears in <retset>$_1$. Construct Q to be the following query.

retrieve <retset> **where** (t **from** Q_1) **or** (t **from** Q_2)

Then $Q \equiv E$.

Case 3 (Difference): E is of the form $E_1 - E_2$. For $E_1 - E_2$ to make sense the schemes of E_1 and E_2 have to be the same. Let $Q_1 \equiv E_1$ and $Q_2 \equiv E_2$ be of the form shown in Case 2. Construct <inset> to contain 't.A' if 't.A -> B' appears in $<retset>_1$. Construct <map> to contain 't.A = B' if 't.A -> B' appears in $<retset>_2$. Construct Q to be the following query.

retrieve $<retset>_1$ **where** $<ccond>_1$ **and** (<inset> **notin** <map> Q_2)

Then $Q \equiv E$.

Case 4 (Product): E is of the form $E_1 \times E_2$. Let $Q_1 \equiv E_1$ and $Q_2 \equiv E_2$ be of the form shown in Case 2. Assume that Q_1 and Q_2 do not share any tuple variables; also for $E_1 \times E_2$ to make sense these queries should not share any new names. Construct <retset> to contain 't.C -> C' if new name C appears in $<retset>_1$ and contain 'u.D -> D' if new name D appears in $<retset>_2$. Construct Q to be the following query.

retrieve <retset> **where** (t **from** Q_1)*(u **from** Q_2)

Then $Q \equiv E$.

Case 5 (Projection): E is of the form $\pi_Y(E_1)$. Let $Q_1 \equiv E_1$ be of the form shown in Case 1. Construct <retset> to contain 't.B -> B' if $B \in Y$ and new name B appears in $<retset>_1$. Construct Q to be the following query.

retrieve <retset> **where** (t **from** Q_1)

Then $Q \equiv E$.

Case 6 (Selection): E is of the form $\sigma_{B=c}(E_1)$. Let $Q_1 \equiv E_1$ be of the form shown in Case 1. Construct <retset> to contain 't.C -> C' if new name C appears in $<retset>_1$. Construct Q to be the following query.

retrieve <retset> **where** (t **from** Q_1)*(t.B = c)

Then $Q \equiv E$.

$\square$