A Gateway from a DBPL to Ingres:

Modula-2, DBPL, SQL+C, Ingres

Florian Matthes Andreas Rudloff Joachim W. Schmidt
Kazimierz Subieta [*]

University of Hamburg
Department of Computer Science
Vogt-Kölln-Straße 30
D-2000 Hamburg 54
e-mail:subieta@dbis1.informatik.uni-hamburg.de

Abstract

A gateway from DBPL (being a superset of Modula-2) to the commercial Ingres is described. DBPL extends Modula-2 in several ways, in particular, it introduces a new bulk and persistent data type "relation", and high-level relational expressions (queries) based on the predicate calculus. The gateway enables the user to write normal DBPL programs addressing Ingres databases. In contrast to typical implementations that embed SQL statements into a programming language, the interface becomes fully transparent to DBPL programmers: they need not to be familiar with SQL and Ingres. In this way the impedance mismatch problem is avoided. DBPL queries and other statements referring to Ingres tables are automatically converted into corresponding SQL statements, and the output from Ingres automatically becomes the property of the DBPL program. The gateway supports queries that refer both, Ingres and DBPL relations. This paper presents design assumptions of the gateway and implementation methods. In addition, we discuss design and implementation difficulties.

1 Introduction

This paper presents motivations, design assumptions and implementation methods of the gateway from the database programming language DBPL [ScMa92, MRSS92] to the commercial database management system Ingres [IngrA89, IngrB89].

The coupling of programming languages with relational database systems is usually based on embedding a query language into a programming language. The border distinguishing querying and programming languages has become, however, more and more fuzzy.

[*]On leave from Institute of Computer Science, Polish Academy of Sciences

Many functionalities typical for programming languages and programming environments have been fixed in the SQL standard as capabilities of the "query language". Besides the infamous impedance mismatch, this approach involved another disadvantage (less discussed in the literature) known as *bottom-up evolution*, i.e. extending incrementally and *ad hoc* the functionalities of query languages. In the result many positive features that were the motivation of the initial development are lost. This concerns mainly SQL which evolved in the direction of programming languages (which is especially striking in the Oracle PL/SQL [Orac91]).

Initial trends in the development of query languages were different from that of programming languages. A basic assumption was simplicity and naturalness of the whole query interface, called *user-friendliness*. The positive aspects of user-friendliness include data independence, declarativity, simplification of notions concerning data views, macroscopic operations allowing the user to determine extensive computations in a compact form, and a syntax similar to a natural language. However, user-friendliness also means the restriction of the language's functionality and power, as well as non-orthogonality of the language's constructs (e.g.due to syntax). Real applications may consist of a large number of queries and other constructs, thus other aspects of user-friendliness are of vital importance, such as preventing the user from his/her own errors, computational completeness, and supporting various programming abstractions.

In comparison to query languages, Database Programming Languages (DBPL-s) have the following positive features: strong and static type checking (what gives the possibility to remove many errors before a program is executed), syntactic and semantic orthogonality (what reduces the number of necessary primitives in the language, as well as the size of manuals), full computational and pragmatic universality with respect to the processing of all data types, clean semantics, and the support for various programming abstractions (modules, procedures, functions, etc.)

DBPL-s adopt the concept of a query language as a powerful construct of a programming language. This is the case in DBPL [ScMa92, MRSS92], which extends Modula-2 in several directions. In particular, it introduces persistence, bulk data (relations) and high-level relational expressions (equivalent to queries), which make possible an associative access to relations.

Despite various advantages of DBPL, we are aware that it has little chance of success in the commercial world. There are several reasons for this. As observed in [Banc92], products of research activity suffer from the "new programming language syndrome": very few organizations are ready to adopt a new programming language or a new system. DBPL is a new product working with its own database format. Clients of database systems usually prefer the long existence of the databases, since investment in gathering data, writing programs, education of staff, organization of technological routines of data processing, etc. is high. DBPL was produced at a university as a scientific project and no company is willing to support it long term. In general, commercial systems are equipped with a large family of useful and necessary utilities which have no special scientific interest, thus they are not implemented in the DBPL environment. We do not expect, therefore, that potential clients of database systems will decide to use DBPL as the only tool for the full development of database applications.

University software, such as DBPL, can be available in the commercial world as a supplement to popular systems. Many professionals who are dissatisfied with the capabilities and programming style offered by languages such as embedded SQL are potential DBPL clients. This is an important reason for creating a gateway from DBPL to commercial systems. Furthermore, the implementation of the gateway will give the DBPL community - students and researchers - the opportunity to access large commercial database systems and thereby use, within the DBPL programming environment, their various capabilities.

Below we describe five possible approaches in making a gateway; there are, however, strong arguments against the first four listed here.

1. A precompiler for SQL embedded in DBPL
   This approach is typical and the easiest to implement. However, almost all advantages of DBPL, such as strong typing, style of programming, abstraction capabilities, avoidance of the impedance mismatch, etc. would be lost.

2. Packages of procedures calling Ingres routines
   The procedures will allow full access to Ingres databases through dynamic SQL. This approach shares the disadvantages of the previous approach. Both approaches would mix two very different languages, thus the impedance mismatch problem cannot be avoided.

3. Package of procedures on the level of Ingres files
   The procedures organize the access to Ingres database files without an intermediate language. The main disadvantages of this approach are that the SQL optimizer, concurrency mechanisms of Ingres, indices, catalogs, etc. will not be used. Additionally, implementors must deal with the physical organization of Ingres databases which is probably not easy and not well specified.

4. A gateway from DBPL to Ingres on the level of Ingres files
   In contrast to the previous approaches an Ingres database is not seen as a specific object served by special procedures but as a normal DBPL database. As a result a 1:1 mapping between Ingres and DBPL data is stored and used during the access. As in approach 3 above, the disadvantage of this approach is that the SQL optimizer and Ingres concurrency control, etc. will not be used. These last two options may imply difficulties in achieving concurrency with other Ingres applications acting on the same database.

5. Interface from the DBPL run-time system to the Ingres SQL machine
   All references from DBPL programs to Ingres databases are transformed into dynamic SQL statements during run-time. This permits use of the SQL optimizer and all capabilities of Ingres that are "below" the SQL machine (concurrency, indices, views, Ingres/Star, gateways, etc.).

In this project we implemented the fifth approach. The interface is fully transparent to DBPL programmers: knowledge of SQL and Ingres is not necessary. DBPL statements referring to Ingres databases are automatically converted into corresponding SQL statements. The output from Ingres automatically becomes the property of the DBPL program. This approach does not support the opinion that SQL should be the "intergalactic language" [SRLG+90] for the next database era. We do not believe that SQL, as a programming language, presents the maturity that should be continued (despite fixing it in huge standards).

The paper is organized as follows. In Section 2 we present briefly similarities and differences between DBPL and Ingres SQL. In Section 3 we discuss possible methods of mapping DBPL constructs into SQL statements and present the methods chosen. In Section 4 the conversion of low-level features of DBPL into dynamic embedded SQL is described. In Section 5 we present architectural assumptions of the gateway in connection with the architecture of DBPL. In Section 6 implementation difficulties and open or unsolved problems are discussed.

2 Similarities and Differences of DBPL and Ingres SQL

2.1 Data Structures

The differences in data structures supported by Ingres and DBPL are as follows:

1. DBPL permits the full orthogonality of type constructors which results in the possibility of defining nested relational structures. Ingres allows flat relations only.

2. Primary keys of relations in DBPL are an important conceptual issue. This determines the semantics of some operators. In Ingres primary keys are declared and internally used, but information about primary keys is irrelevant for querying and programming.

3. Atomic data types in DBPL and Ingres are slightly different: DBPL uses a machine-independent convention for naming and understanding types (cardinal, integer, char, ...), while Ingres uses another convention with roots in traditional data processing. The following table presents atomic data types supported by Ingres and shows possible DBPL equivalents. Variable-length data types have no equivalents in DBPL; they can be represented by long fixed-length strings.

Ingres type	Format and size	Assumed DBPL type	Comments
char(1)-char(2000)	fix-length char string	ARRAY[..] OF CHAR	
c1 - c2000	fix-length printable char strings	ARRAY[..] OF CHAR	
varchar(1)-varchar(2000)	var-length char string		can be mapped to fixed-length arrays
text(1)-text(2000)	another var format		
integer1 integer2 integer4	1-byte integer 2-byte integer 4-byte integer	CHAR INTEGER LONGINT	
float4 float8	4-byte floating 8-byte floating	REAL LONGREAL	
date	12-byte date	ARRAY[0..11] OF CHAR	require special functions
money	8-byte money value	ARRAY[0..7] OF CHAR	
table-key	8 bytes	ARRAY[0..7] OF CHAR	
object-key	16 bytes	ARRAY[0..15] OF CHAR	

4. Ingres deals with null-values through special features, while DBPL does not introduce this concept at all.

5. DBPL does not allow duplicate tuples either in stored relations or in intermediate query outputs. Ingres allows duplicate tuples in both cases. Since these Ingres capabilities cannot be utilized in DBPL, the programming of some tasks in DBPL may prove impossible and, in addition, will be inconvenient for those users familiar with SQL.

2.2 Levels of Database Interfaces

Both Ingres SQL and DBPL make the distinction between querying a database and processing a database. The consequences of this distinction are different for DBPL and Ingres SQL.

Formerly, SQL was designed for simple retrieval only. Later extensions involved some programming capabilities, that is, inserting, deleting and updating. The assumption concerning semantic domains of relational query languages (so-called "value-orientation") makes it difficult to express updating operations through queries. In SQL updating can be accomplished by the highly-specialized "update" construct or by a more powerful method employing cursors. Programming difficulties result from the impossibility of storing the value of a cursor (i.e. a pointer to a tuple) in a separate variable, and vice-versa. This approach to cursors is inconsistent w.r.t. some updates, e.g. updating a tuple during a cursor loop may cause the loop to process this tuple twice. Some semantic effects are surprising for the programmer, e.g. the opening of a cursor for a query, which returns an empty result, causes a run-time error.

In DBPL the main reason for the distinction between queries and other constructs of the language is query optimization. DBPL is based on the assumption that queries problematic for query optimizers should be forbidden syntactically. As a consequence, functional symbols and operators are not allowed within DBPL predicates. This restriction results in a potential for good performance, however, it violates the orthogonality principle. For example, "Get employees having tax less than 300", where tax is given by a function of the salary, cannot be expressed as

```
EACH X IN EMPLOYEES: tax( X.SALARY ) < 300
```

The user must change her/his way of thinking, for example, s/he can reformulate this query using the procedural construct:

```
FOR EACH X IN EMPLOYEE: TRUE DO
   IF tax(X.SALARY) < 300 THEN
      some processing of X
   END;
END;
```

The distinction between querying and processing a database in SQL and DBPL corresponds to the distinction between high-level and low-level programming. High-level programming assumes the "many-data-at-a-time" principle, while low-level works "one-datum-at-a-time", as found in classical programming languages. DBPL integrates both levels while SQL deals with the high level only: the low level is delegated to the host language. In DBPL and SQL these levels do not match each other: some high-level constructs of DBPL cannot be expressed by constructs of SQL and vice versa. Mapping low-level constructs of DBPL into high-level constructs of SQL causes difficulties in establishing general rules. This means that an interface connecting DBPL and SQL cannot utilize the full power of these systems. For example, it is impossible to express in SQL such DBPL queries that address nested relations. On the other hand, it is practically impossible to utilize such SQL capabilities as arithmetic operators, aggregate functions, grouping, etc.in DBPL.

2.3 High-level Constructs of DBPL

By high level constructs we denote such programming capabilities which support data independence and follow the "many-data-at-a-time" principle. We list all high-level constructs of DBPL that may concern Ingres databases with short comments concerning their semantics and possible equivalents in SQL. All examples refer to the classical supplier-part database presented in the Appendix.

The following high-level constructs are introduced in DBPL:

1. Selective access expressions, for example:

```
EACH X IN supp: X.city = "London"

EACH X IN supp: SOME Y IN sp (X.sno = Y.sno)
```

Selective access expressions can be used inside the FOR iterator; in this case the range variable has the status of an updatable programming variable. Selective access expressions have a direct counterpart in SQL.

2. Quantified boolean expressions, for example:

```
ALL X IN part ( SOME Y IN sp (X.pno = Y.pno))
```

Quantifiers have several counterparts in SQL; we will discuss them later.

3. Constructive access expressions, for example:

```
{X, Y} OF EACH X IN supp, EACH Y IN sp:
                              (X.sno = Y.sno) AND (Y.qty > 200)

{X.sname,Y.pno} OF EACH X IN supp, EACH Y IN sp:
                              (X.sno = Y.sno) AND (Y.qty > 200)
```

They have a direct counterpart in SQL.

4. Aggregate expressions used to construct tuple and relation values, for example:

```
partRel{ { "P7", "bolt", "green", 65, "London"},
         { "P8", "nut",  "red",   11, "Rome"  } }

partRel{}
```

They have no counterpart in SQL.

5. Relation expression for materializing the relations described by access expressions, for example:

```
JoinRelType{ {X.sname,Y.pname} OF EACH X IN supp, EACH Y IN part:
    SOME Z IN sp(
          (X.sno = Z.sno) AND (Y.pno = Z.pno) AND (Z.qty > 200))}
```

Relational expressions can be used in all contexts allowed for stored relations, i.e. they follow the orthogonality principle. SQL does not allow expressions as range relations under a *from* clause.

6. Union operator, for example:

```
suppRel{ EACH X IN supp: X.city = "London",
         EACH Y IN supp: Y.status > 10,
         { "S8", "Miller", 20, "Paris"}    }
```

Ingres SQL also supports union, but only on the top nesting level.

7. Relational operators =, #, <, <=, >, >= denoting relation equality, non-equality and set-theoretic inclusions, for example:

   ```
   suppRel{EACH Y IN supp: Y.status > 30} <=
                    suppRel{EACH X IN supp: X.city = "London"}
   ```

   SQL does not support these comparisons.

8. Assignments on relations realizing all updates: := (assign), : − (delete), : + (insert), and :& (update), for example:

   ```
   mysupp := suppRel{ EACH Y IN supp: Y.status > 20 };
   supp   :- suppRel{ "S1", "", 0, "" };              (* Delete *)
   supp   :+ suppRel{ "S1", "Schmidt", 25, "Berlin"}; (* Insert *)
   supp   :& suppRel{ "S1", "Schmidt", 30, "Berlin" } (* Update *)
   ```

   All operators follow the "many-data-at-a-time" principle. They can be implemented in Ingres by two methods: by using high-level "update", "insert", "delete" statements of SQL, or by fetching required tuples from Ingres tables to DBPL buffer, doing the required operations and shipping them back to Ingres.

9. Standard DBPL functions: CARD (the number of relation elements), EXCL (exclude a tuple), INCL (include a tuple). CARD has a direct counterpart in SQL. Semantics of EXCL and INCL is based on primary keys, which presents a problem; we will return to it later.

2.4 High-level Abstractions of DBPL and SQL

SQL views are special objects with independent existence in the Ingres database. They can be dynamically created and deleted. In DBPL similar notions are called "constructors" and "selectors" [ERMS91]. They are not, however, properties of the database but rather properties of the source text of programs. They are first-class objects and may exist in the database as values of variables, but only when proper assignments are executed in the user program.

The capabilities of either selectors and constructors are different from SQL views. Selectors can be considered as updatable views and may have parameters, thus they have no equivalent in Ingres. Constructors are closer to the traditional view concept but they extend it in two ways: they can be recursive (with a stratified fixed-point semantics) and may have parameters. On the other side, some capabilites embedded in the SQL *select* block (e.g. arithmetic and aggregate operators), which is used to create views, cannot be expressed in DBPL selectors and constructors.

Thus selectors and constructors have a common conceptual intersection with Ingres views but in general they are different objects. Since the mapping of selectors and constructors into views implies problems, we have chosen to construct the gateway between DBPL and Ingres on architectural levels that are below these abstractions.

Ingres database procedures, integrities and access permissions are not considered in the first stage of this project (and in this paper) since DBPL has no equivalent notions.

2.5 Null-values

Null-values are not implemented in DBPL. In Ingres SQL null-values are associated with special facilities (a comparison operator *is [not] null* and *indicator variables* in embedded SQL) and with a special treatment in aggregate functions. We partly agree with the criticism of the null-value concept, see e.g. [Date87]. Null values are captured in DBPL by variant records. However, there is no simple systematic mapping from existing SQL databases to semantically equivalent DBPL type definitions.

2.6 Concurrency Control

Both DBPL and Ingres are multi-user database systems and employ their own methods for dealing with transactions, locks, deadlocks, logs, etc. There is no danger of improper interference of these mechanisms since from the point of view of Ingres, a DBPL application is one of its clients. Since an Ingres application cannot be a client of DBPL, undetected deadlocks cannot occur. Some problems may be connected with transaction processing and recovery. Inside a DBPL transaction it is necessary to initialize an Ingres transaction. This leads to nested transactions and two commits, one inside Ingres and the second inside DBPL. Again, this problem seems to be minor since the time period between commits is short (thus the probability of a crash between commits is low), and since we do not expect that DBPL databases will be large this decreases the danger of fatal crashes.

2.7 Dynamic SQL and Data Interchange Between SQL and DBPL

Dynamic SQL is an extension of the capabilities in embedded SQL allowing the user to write generic programs. In dynamic SQL, statements are strings of characters which can be manipulated during run-time while in normal embedded SQL statements are pieces of the source program. (Their form is, therefore, fixed before compilation.) Since the gateway from DBPL to Ingres must be generic (i.e. must work for all types of relations and for any DBPL high-level expressions), we are forced to use dynamic SQL.

SQL uses two methods for data communication. In the first method, the programmer explicitly declares host variables for the attributes s/he wants to process. This method is inapplicable in our case since in the generic and transparent interface it is impossible to predict the number and types of host variables. We must apply another method implemented in dynamic SQL. It introduces the so-called SQL Description Area (SQLDA), which allows communication through pointers. SQLDA is a dynamically created data structure essentially consisting of the following information:

sqln		Integer indicating the size of allocated table sqlvar
sqld		Integer indicating the number of columns in the result table Usually sqld should be equal or smaller than sqln
sqlvar		An sqln-size array of records:
	sqltype sqllen sqldata sqlind sqlname	Integer indicating the type of the attribute Integer indicating the length of the attribute Pointer to the attribute value Pointer to a variable storing information about null-values String denoting the attribute name
	sqltype sqllen sqldata sqlind sqlname	
	.....	

The *sqldata* pointers are counterparts of the host variables in embedded SQL. They should be filled in by an application program, and point to values, which are stored elsewhere. They have two kinds of applications. In the first case (used by *select* statements) they determine places, where the attributes of a retrieved tuple will be written. In the second case, they determine actual parameters of an SQL statement. This technique assumes application of statements containing question marks as "formal parameters". E.g., for inserting a tuple in an Ingres table we must prepare the statement:

```
INSERT INTO relation name ( list of attributes ) VALUES ( ?, ?,..,? )
```

The number of question marks should be equal to the number of attributes. For example, insertion into the *supp* relation requires preparing the statement

```
INSERT INTO supp ( sno, sname, status, city ) VALUES ( ?, ?, ?, ? )
```

In this case the *sqldata* pointers point values, which will be substituted fot the question marks when the statement is executed. In many cases, this technique is inconvenient; some problems with it are presented later.

2.8 Low-level DBPL Capabilities

The low-level caoabilities of DBPL include the standard procedures LOWEST, HIGHEST, THIS, NEXT and PRIOR. They enable processing of DBPL relations in a tuple-by-tuple fashion. The programmer can use them to find tuples having the *lowest* respectively *highest* primary keys in the given relation. Given the primary keys of a tuple the programmer can find *this*, *next*, and *prior* tuples. Proponents of novel approaches to database programming neglect these operators, considering them to be too implementation-oriented. Some tasks, however, cannot be programmed without them, for example, merging of relations, or building a browsing utility (allowing the user to look over the relation through a window by moving forward and backward). SQL gives no equivalents of these operators thus we have so far no idea how to implement them. This concerns in particular the PRIOR operator.

3 Mapping DBPL Constructs into SQL

3.1 Unproblematic Cases

- DBPL expressions without quantifiers ALL and SOME.
  Consider the following DBPL expressions:

  `EACH X IN Rel:  p(X)`

  { *projection list* } `OF EACH` X_1 `IN` Rel_1`,...,` `EACH` X_n `IN` Rel_n`:` `p(`X_1`,...,`X_n`)`

  If the predicate p does not contain any quantifiers and all comparisons in p are available in SQL, then these expressions are (respectively) equivalent to the following SQL queries:

  `select X.* from Rel X where p(X)`

  `select` *projection list* `from` Rel_1 X_1`, ...  ,` Rel_n X_n
  `where p(`X_1`,...,`X_n`)`

- Changing DBPL predicates returning boolean values.
  Since SQL has no semantic domain with boolean values, a DBPL predicate p can be converted into the following SQL statement:

  ```
      select * from AuxRel where p
  ```

  where AuxRel is the name of an auxiliary Ingres table containing exactly one tuple. We need only a simple procedure returning TRUE if the select statement will return a non-empty result, and FALSE otherwise. Below we give an example of this conversion. The DBPL predicate ("Do all suppliers have a status higher than 10?")

  ```
     ALL X IN supp ( X.status > 10 )
  ```

is converted to the following SQL query:

```
select * from AuxRel
where not exists ( select * from supp X1
                   where not( X1.status > 10 ) )
```

- DBPL expressions with quantifier SOME in the prenex form. If a DBPL expression contains only existential quantifiers in the prenex form, the conversion is also simple: the DBPL expression

  { *projection list* } OF EACH X_1 IN Rel_1,...,EACH X_n IN Rel_n:
  SOME Y_1 in Rel_{n+1},...,SOME Y_m in Rel_{n+m} (p)

  where p contains no quantifiers, can be directly mapped to the following SQL query:

  select *projection list*
  from Rel_1 X_1, ... , Rel_n X_n, Rel_{n+1} Y_1, ..., Rel_{n+m} Y_m
  where p

The above cases show also how we map DBPL projection lists into SQL equivalents. There are no changes, since syntax in both cases is the same.

3.2 Predicates with Universal Quantifiers

SQL supports several methods for expressing queries which require the use of universal quantifiers when formulated in other languages [Frat91]. These are the following:

- **Through quantified comparisons**
  By a quantified comparison we denote a normal comparison followed by the key words "all" or "any", for example, "=all", "<any", ">=all", etc. The syntactic rule and semantics of these operators are the following. Let v be a value (or an attribute) and X be a set of values. Syntactically, only predicates *(v θ any X)* and *(v θ all X)* are allowed, where θ is a normal comparison. The meaning of the first predicate is $\exists x \in X(x\theta v)$, and the meaning of the second is $\forall x \in X(x\theta v)$. A simple example shows that quantified comparisons are not as powerful as quantified formulas. Consider $\forall x \in N(x > 2 \vee x < 3)$. Obviously, the predicate is true. Now observe that both predicates $\forall x \in N(x > 2)$ and $\forall x \in N(x < 3)$ are false. Hence decomposition of the former predicate into quantified comparisons does not work. Because of the lack of universality of quantified comparisons the automatic conversion of DBPL's universal quantifiers is problematic, because DBPL uses them without restrictions.

- **Through function "count"**
  The predicate `ALL X IN R (p(X))` can be expressed in embedded SQL as

```
(select count(*) from R X where not ( p( X ))) = 0
```

(This is not a correct SQL statement, but such a query can be easily expressed in embedded SQL.) Since function "count" requires materialization of its argument, this method may lead to performance problems.

- **Through operator "exists"**
  The predicate `ALL X IN R (p(X))` can be expressed in SQL as

  ```
  not exists ( select * from R X where not p( X ) )
  ```

  The method seems to be the most promising because of its universality and potential for optimization; thus it is used in the implementation.

Existential quantifiers are also implemented through the operator "exists". Below we give an example of the conversion. The DBPL expression ("Suppliers supplying all parts")

```
EACH X IN supp: ALL  Y IN part ( SOME Z IN sp (
               (X.sno = Z.sno) AND (Y.pno = Z.pno) ) )
```

is converted into the following SQL query:

```
select  * from supp X1
where  not exists (
             select * from part X2
             where  not exists (
                          select  * from sp X3
                          where X1.sno = X3.sno
                          and   X2.pno = X3.pno ) )
```

3.3 Escape Methods

Since in some cases there is no convenient solution for converting DBPL constructs into SQL, we need escape methods. Although being not very efficient, they allow the completion of computations. We note the following methods:

1. Copying DBPL relation(s) to the Ingres side. It is likely that DBPL relations will be used as auxiliary objects, thus we can expect they will not be too large. By copying them to the Ingres side we give more freedom to the SQL optimizer, thus allowing better performance. Once copied, the resulting expression will contain no reference to DBPL relations. At the end of a transaction the relations should be copied back to DBPL. This action can be optimized: only relations that have been updated needed to be copied back; others may be destroyed on the Ingres side.

2. Copying Ingres table(s) to the DBPL side. During the processing of Ingres tables copies on the DBPL side access to the original Ingres tables should be forbidden for other Ingres transactions. After processing, the tables are copied back from DBPL to Ingres (again with the possibility of optimization). However, we may expect that Ingres tables will be rather large, thus this method may lead to low performance.

3. Tuple substitution according to Ingres or DBPL relation(s). This method assumes that Ingres or DBPL relation(s) will be processed tuple-by-tuple, thus there will be no references to them inside a DBPL predicate. The method, although well-known, implies implementation problems, and still, there is no evidence that it will yield good performance.

3.4 DBPL Expressions Mixing DBPL and Ingres Relations

Two kinds of mixing can be distinguished. In the first case, a DBPL statement contains sub-statements independent of external variables. As an example, consider the expression

```
EACH X IN supp: SOME Y IN sp ( Y.pno = "P1" )
```

Assume that supp is an Ingres table and sp is a DBPL relation. Since the internal sub-predicate does not reference the external variable X, we can evaluate it on the side of DBPL, and then generate a proper SQL statement. This method is applied in the implementation: using a procedure that recursively scans a predicate tree, discovers independent subpredicates, evaluates them, and then modifies the tree by reducing it and inserting the calculated truth values resp. temporary relations.

In other cases mixing requires escape methods. For example, for the following DBPL expression

```
EACH X IN supp: SOME Y IN sp ( (X.sno = Y.sno) AND (Y.pno = "P1") )
```

there is no possibility to separate calculations. In such a case the DBPL relations participating in the expression are copied to the Ingres database. This method cannot be applied to nested relations. In order to avoid copying full Ingres relations to the DBPL side a more efficient method can be applied. It assumes, that only tuples which are necessary to evaluate the DBPL expressions, are copied from Ingres databases. The problem is similar to the query processing in distributed databases, and some known methods (e.g. employing the semi-join concept) can be applied. This method is not general, since it requires some "monotonicity" or "additivity" of DBPL expressions (i.e. a larger argument table implies that the result of an expression is not smaller). This property does not hold e.g. for expressions containing universal quantifiers.

3.5 Predicates with a Range Relation Given by a Subpredicate

SQL does not allow *select* blocks in the *from* clause, thus direct mapping of predicates with range subpredicates (range relations described by relation expressions) is impossible. We can use several methods:

1. Unnesting: transformation of a SQL query with a *select* block nested under *from* clause into an equivalent statement without such a nesting. Although there are simple examples when such a method can be applied, there are difficulties in establishing general rules.

2. Creating a temporary range relation on the Ingres side.

3. Creating a temporary range relation on the DBPL side.

4. Creating a view on the Ingres side.

The last method seems to be the most promising, since it leaves freedom for the standard SQL optimizer. However, because of additional operations on data dictionaries, creating, using and deleting the view may be more expensive than materializing the range relation . In the first stage of the project we implemented temporary range relations on the Ingres side. All presented methods will not work if the expression determining the range relation is parameterized by an external variable. In this case shipping tuples of participating relations from Ingres to the DBPL side seems to be the only method.

4 Conversion of DBPL Capabilities into Dynamic SQL

4.1 Implementation of the FOR EACH Construct

The FOR EACH construct of DBPL leads to one of the most difficult implementation problems. The semantics of the construct

```
FOR EACH variable IN relation :  predicate DO
         sequence of statements
END
```

can be explained as follows. The *sequence of statements* is executed for each tuple in *relation*, for which *predicate* is true. The *sequence of statements* may contain arbitrary DBPL statements, in particular other "FOR EACH" statements. The *variable* inside the *sequence of statements* is considered as a normal programming variable. In particular, all updates of the *relation* can be done by this variable. The updating semantics is, however, not straightforward: the variable contains a main memory copy of the processed tuple and all updates modify the copy only. At the end of each loop the original tuple in the relation is modified according to the values of this eventually modified copy.

An example of the FOR EACH construct:

```
FOR EACH X IN supp : X.city = "London" DO
   X.status := X.status + 10;
   FOR EACH Y IN sp : X.sno = Y.sno DO
```

```
        WriteString(X.sname); WriteString(Y.pno); WriteInt(Y.qty);
        WriteLn;
      END;
    END;
```

The above semantics has a direct counterpart in SQL relying on the application of cursors. Dynamic SQL assumes that the buffer for fetching/flushing a tuple is organized through SQLDA areas. Since DBPL allows nested FOR EACH and other constructs refering to database relations, SQLDA areas must be managed by a stack. The same is true for names of cursors and names of SQL *PREPARE* statements.

The dynamic SQL version is neither clear nor well specified in SQL manuals. We also discovered a few bugs. Thus the final solution is the result of a frustrating sequence of experiments rather than careful reading of manuals. To serve the FOR EACH construct we need the following steps:

1. Generate a SQL query from the argument of the FOR EACH statement. The statement presented above will produce the query

   ```
   select X1.* from supp X1 where X1.city = "London"
   ```

2. Execute *PREPARE* and *DESCRIBE* SQL commands with the argument being the query generated in the previous step. This step is necessary to obtain attribute names of the relation.

3. Generate an extended SQL query

   ```
   <previous query> FOR DIRECT UPDATE OF <list of all attributes>
   ```

   The statement presented above will produce the query

   ```
   select X1.* from supp X1 where X1.city = "London"
               FOR DIRECT UPDATE OF sno, sname, status, city
   ```

4. Generate a new statement name and push it on the stack.

5. Create a new SQLDA and push it on the stack.

6. Execute *PREPARE* and *DESCRIBE* SQL statements for the given extended query, statement name and SQLDA.

7. Generate a new cursor name and push it on the stack.

8. Declare the cursor.

9. Open the cursor. This is the preparation step for fetching tuples from the Ingres relation.

10. Extract the relation name from the query.

11. Generate the SQL update statement

    ```
    UPDATE relation name SET attribute₁ = ?,..., attributeₙ = ?
          WHERE CURRENT OF cursor name
    ```

    For the example above we generate the statement

    ```
    UPDATE supp SET sno = ?, sname = ?, status = ?, city = ?
          WHERE CURRENT OF dbcursᵢ
    ```

12. Generate a new statement name and push it on the stack.

13. Execute the SQL *PREPARE* statement w.r.t. the generated UPDATE statement and the new statement name.

Now, on the top of the stack we have two statements, one cursor and one SQLDA. Fetching tuples requires the SQL *FETCH* command (with the cursor addressing the first statement), while flushing the tuple requires the SQL *EXECUTE* command addressing the second statement.

The above procedure is complicated, although the task is typical. In our opinion, design solutions concerning cursor processing in embedded SQL were burdened by attempts to hide the fact that cursors are pointer-valued variables. In effect, this programming interface is hard to accept.

4.2 Implementation of High-level Relational Assignments

For each of the four kinds of high-level relational assignments in DBPL (assign, insert, update, delete) we must consider four cases:

1. On the left hand side of assignment there is a DBPL relation, and on the right hand side is an expressions refering to DBPL relations only. This case is already implemented in DBPL.

2. On the left hand side of the assignment there is a DBPL relation, and the right hand side expressions refers to Ingres relations. This case, dependingly on the kind of assignment and the kind of expression, is implemented by two methods: (1) creating a temporary relation from the expression and applying the previous case; (2) generating SQL statement from the expression and then making the proper operations on the right hand side relation on-the-fly.

3. On the left hand side of assignment there is an Ingres relation, and the right hand side refers to DBPL relations only. In this case an intermediate relation is created inside DBPL, then proper actions are done on the external relation.

4. On the left hand side there is an Ingres relation, and the predicate on the right hand side refers to Ingres relations only. In principle all processing can be done on the side of Ingres through a sequence of SQL statements. However, SQL updating operations are not sufficiently powerful. Thus we decided to make some operations on the DBPL side. For the *delete* operator we recognize the case when the left hand side relation is also referenced on the right hand side and for this case we generate a normal SQL *delete* statement. For other cases we create a temporary table from the expression on the DBPL side and then apply the second case.

The *delete* operator of DBPL presents an example of problems that appear in SQL. In DBPL semantics of the operator is based on primary keys. The construct $R_1 : - R_2$ means removing from R_1 all those tuples whose primary keys are the same as for one of the R_2 tuples; values of other attributes of R_2 are not taken into account. Assume R_1 is an Ingres relation and R_2 is a DBPL relation. The SQL method of passing parameters to statements through question marks requires filling in values of *all* attributes. But, as follows from DBPL semantics, some of attributes of R_2 tuples may be meaningless, thus we must somehow ignore them. The solution that we have found is tricky. We generate the SQL statement

$$\texttt{DELETE FROM } R_1 \texttt{ WHERE } p_1 \texttt{ AND } p_2 \texttt{ AND } \ldots \texttt{ AND } p_n$$

where n is the number of attributes. Predicate p_i has the form
$\texttt{attribute}_i = \texttt{?}$
for key attributes, and the form
$(1 = 1 \texttt{ OR attribute}_i = \texttt{?} \ )$
for non-key attributes. For example, for the DBPL statement

```
supp :- suppRel{ {"S1","",0,""}, {"S2","",0,""} };
```

we generate the SQL dynamic statement

```
DELETE FROM supp WHERE sno = ?
  AND (1=1 OR sname = ?) AND (1=1 OR status = ?) AND (1=1 OR city = ?)
```

This statement is executed for each tuple of the right hand side relation.

We think that a programming language that requires such tricks cannot be appreciated either from technical and scientific points of view.

4.3 Relational Comparisons

Since DBPL does not allow duplicate tuples, all comparisons can be performed by one operator *contains* (denoted >=). Equality and strong comparisons are obtained by comparison of the numbers of tuples and *contains*. Semantics of relational comparisons in DBPL assumes that only primary keys are taken into account. That is, the DBPL predicate $R_1 >= R_2$ means

$\pi_{primarykeys}(R_1) \supseteq \pi_{primarykeys}(R_2)$

where π denotes projection, and $\supseteq$ is an inclusion of sets. As in the previous case, we must consider four cases of relational comparisons, dependingly wether left hand side and right hand side relations are on the DBPL or Ingres side. In the case when one relation is DBPL and another is Ingres, we apply a sequential scan through the right hand side relation and check if the primary key of the tested tuple is present in the left hand side relation. If both relations are from the side of Ingres, we change predicate $R_1 >= R_2$ into a quantified predicate

```
ALL Y IN R₂( SOME X IN R₁((X.key₁ = Y.key₁) AND ...
                          AND (X.key_last = Y.key_last)))
```

and then generate the corresponding *select* statement:

```
SELECT * FROM AuxRel WHERE NOT EXISTS(
  SELECT * FROM R₂ Y WHERE NOT EXISTS(
    SELECT * FROM R₁ X WHERE X.key₁ = Y.key₁ AND ...
                          AND X.key_last = Y.key_last ))
```

As before, AuxRel is the name of auxiliary Ingres table containing exactly one tuple and key_i is the name of i-th primary key attribute. If the result of the *select* is non-empty, it means that the original query returns *TRUE*; otherwise *FALSE*

5 Architecture of the Gateway

The general architectural view of the gateway, DBPL and Ingres is presented in Figure 1.

The entry Ingres interface is embedded SQL. It is standardized by the so-called X-Open Standard. This is an agreement between companies supporting SQL concerning the source of EXEC SQL statements. These statements are processed by a precompiler producing source code in one of the popular programming languages. We considered applying another interface consisting of procedure calls in C (normally created by the precompiler). For the generic programming point this more convenient. However, it is not standardized, not specified in the documentation and there is no guarantee that it will be supported in the future in the same version; hence we did not follow this idea. We wrote a package of procedures in embedded SQL + C capable of mapping all DBPL constructs. The procedures are also available as a normal DBPL module. They allow the user to write SQL statements inside DBPL programs, an advantage for some kinds of applications.

Exit points in DBPL implied more problems. We assumed that the DBPL compiler should not be changed; all connections to Ingres should be done from the existing run-time system. The DBPL run-time system consists of several layers and features of DBPL are

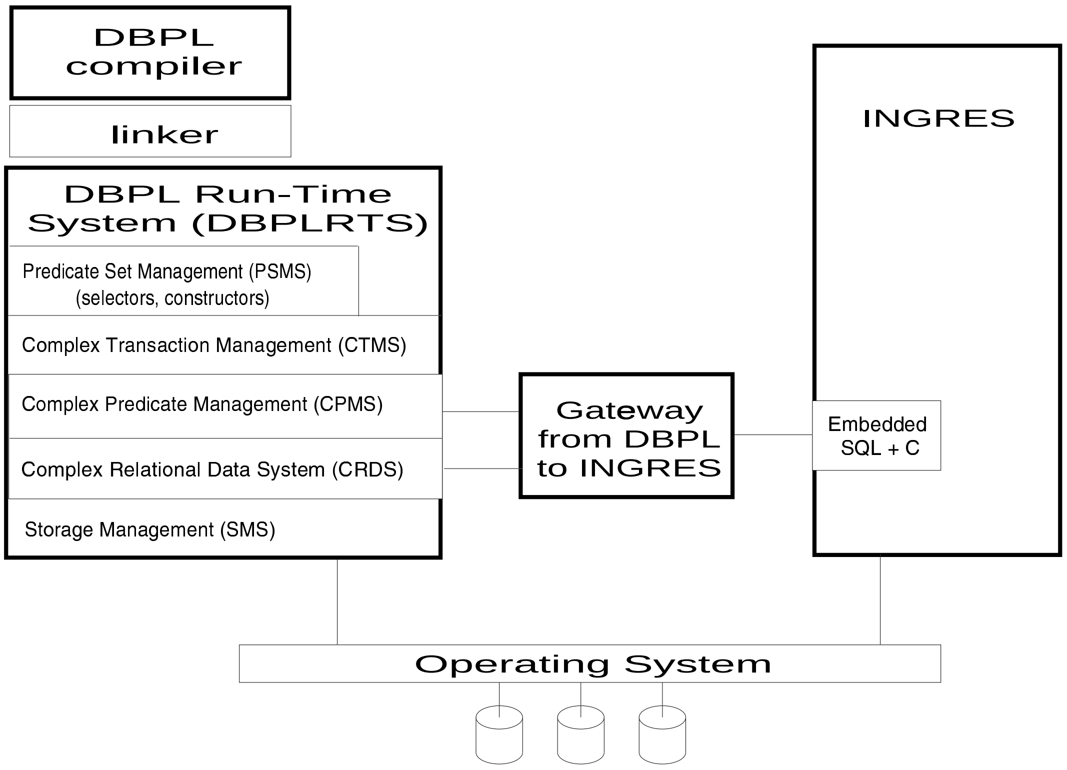
DBPL compiler
linker
DBPL Run-Time System (DBPLRTS)
Predicate Set Management (PSMS) (selectors, constructors)
Complex Transaction Management (CTMS)
Complex Predicate Management (CPMS)
Complex Relational Data System (CRDS)
Storage Management (SMS)
Gateway from DBPL to INGRES
INGRES
Embedded SQL + C
Operating System

Figure 1: General view on the gateway, DBPL and Ingres

tailored to parts of different layers. Some work was necessary to make the architecture of the run-time system cleaner. Afterwards it was possible to determine exit points "below" the transaction processing system (thus the gateway does not deal with locking, unlocking, log, recovery, etc.) and "below" the system responsible for evaluation of DBPL selectors and constructors (thus the gateway also does not deal with them). Exit points to Ingres are in the module responsible for evaluation of DBPL predicates and sometimes in the lower layer responsible for tuple-oriented processing of relations.

We introduced a change to the DBPL data dictionary allowing the distinction between DBPL and Ingres relations. Since the compiler is unchanged, a DBPL program processing Ingres relations is exactly the same as that required for DBPL relations. This means that each Ingres relation which has to be processed by DBPL should have a "twin" relation on the DBPL side. Normally this twin is empty and not used but it must be declared and stored. Twin DBPL relations allow the user to receive typing information and sometimes they are internally used for storing intermediate results.

The architecture of the gateway is presented in more detail in Figure 2. A special utility is written to change the status of DBPL relations. This utility compares types of corresponding DBPL and Ingres twin relations. If the types are fully compatible, it allows the user to change the status of the DBPL relation so that further processing will

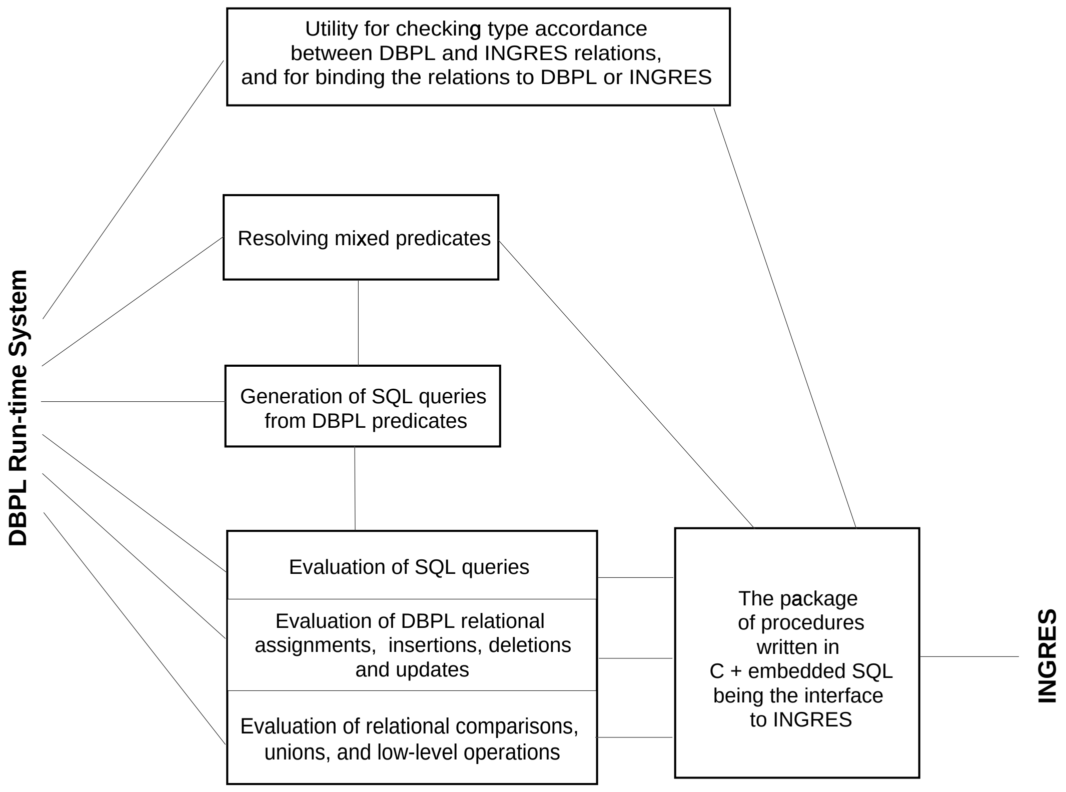
Utility for checking type accordance between DBPL and INGRES relations, and for binding the relations to DBPL or INGRES
Resolving mixed predicates
Generation of SQL queries from DBPL predicates
Evaluation of SQL queries
Evaluation of DBPL relational assignments, insertions, deletions and updates
Evaluation of relational comparisons, unions, and low-level operations
The package of procedures written in C + embedded SQL being the interface to INGRES
DBPL Run-time System
INGRES

Figure 2: Architecture of the gateway

be performed on the Ingres relation. As a result of the above, it is impossible to change the status of a relation which is not in first normal form.

Generation of SQL queries from DBPL predicates is done by a recursive scan of the DBPL predicate tree. During the scan a list of SQL lexicals is built. Roots of the tree corresponding to access expressions cause pushing lexicals *select*, *from* and *where* to the list. Then, projections in the tree insert proper lexicals after *select*, range relations in the tree insert proper lexicals after *from*, and conditional expression insert proper lexicals after *where*. The list works as a stack: to take into account nested select blocks, the insertions are done after this *select*, *from* or *where*, which is the nearest from the top of the list. When the select block is completed it is "masked" (so it is not seen by further insertions). This algorithm is modified fot *exists* and other lexicals to take into account all situations that can occur in DBPL predicate trees. The final SQL query is obtained by direct generation of the query text from the list of lexicals.

Some difficulties were connected with inventing procedures for testing and evaluating DBPL predicates containing mixed references to DBPL and Ingres relations. The procedures recursively scans a predicate tree and produces one of the following answers: *pure dbpl, pure ingres, top dbpl, top ingres, mixed joins, badly mixed.* The answer *pure ingres* means that the predicate contains only references to Ingres relations and can be converted into a SQL query. *Top dbpl* means that the predicate contains independent subpredicates that are *pure ingres*; thus they can be evaluated internally, and then, the whole predicate becomes *pure dbpl*. Similarly, *top ingres* means that the predicate contains subpredicates that are *pure dbpl*; they can be internally evaluated on the side of DBPL, then the resulting temporary relations are copied to the Ingres side, thus the whole predicate becomes *pure ingres*. In the case of *mixed joins* we send the participating DBPL relations to the Ingres side (the escape method), thus the predicate becomes *pure ingres*. The result *badly mixed* means that all good methods fail, and the only method is copying Ingres relations to the DBPL side. For performance reasons we prefer to generate in such a case a run-time error in the current implementation .

6 Implementation Difficulties and Open Problems

6.1 Difficulties Recognized from the Side of DBPL

1. The DBPL run-time system is programmed in Modula-2, thus all advanced data structures are stored in dynamic memory. This concerns, in particular, syntactic trees for high-level expressions and predicates, type descriptors, and data dictionary. Receiving information from these structures requires navigation via pointers, a feature which is cumbersome and error-prone. In Modula-2 there is no alternative solution. This is an argument in favour of languages having the possibility to define bulk types.

2. The construction and semantics of internal DBPL structures is not always well specified and clear.

3. Small optimizations concerning syntactic trees introduced additional difficulties in recognizing their semantics and in traversing them.

4. No possibility of using arithmetic and other operators inside DBPL high-level expressions and predicates limits the full power of SQL

5. The semantics of range variables inside FOR EACH loop is based on copying, which in some cases leads to semantic anomalies (unexpected effects).

6. The semantics for relational comparisons takes into account primary keys only. This also may lead to unexpected effects.

6.2 Difficulties Recognized from the Side of SQL

1. Communication of dynamic SQL with the external world is based on prepared statements, cursors, and SQL Description Areas. This interface is not well prepared for nested and recursive processing, which is inherent for languages like DBPL. Therefore

we have to implement a special stack of statements, cursors and SQLDA-s together with operators acting on this stack.

2. Although the concept of primary key is basic to the relational model, SQL has no direct possibility of updates based on primary keys. In contrast, in DBPL all updates are based on primary keys. This causes problems in how to express some DBPL updates in SQL.

3. No truth values: as a substitute we must generate formulas 1=1 or 1=0.

4. No nested unions; they must be done on the DBPL side.

5. No queries returning boolean values.

6. No comparison of tuples for equality, and no (officially supported) explicit tuple identifiers and operations on them.

7. No syntactic orthogonality of the *count* function with the *select* block. We would prefer the syntax `count( select ... from ...)` instead of `select count(*) from ...`. The current syntax is illogical and makes difficulties in the automatic generation of SQL queries.

8. There are difficulties in retrieving all information about Ingres tables; in particular, this concerns recognizing which attributes are forming the primary keys.

9. SQL dynamic statements use question marks as "formal parameters". This is inconvenient and error-prone.

10. SQL gives poor testing capabilities for programmers, e.g. about names of available relations, about their ownership, status, number of tuples, etc.

11. The system of navigation through cursors gives no possibility to navigate to the *prior* tuple, what makes implementing browsing capabilities extremly difficult.

12. There are some not well-justified syntactic features of SQL statements; for example, an *update* statement through a cursor requires the relation name despite the fact that it was determined previously during declaration of the cursor.

13. Direct update through cursors may change the order of rows, what means that the processing may lose consistency (e.g. the same row will be updated two times).

14. In programming of generic applications we need to "capture" some system reactions to errors. These reactions are not well specified in the documentation of Ingres. For example, we can recognize that a possible message is -40400, but we do not know all the situations in which it is issued, and we are not sure if it is a standardized element of the interface (which cannot be changed in next releases of the system).

15. Automatic generation of the *SQL STOP* statement, in all possible places where the error is expected, is controversial. STOPs after errors are frequently unacceptable, because before the stop some operations must be performed. This forces use to use the statement *WHENEVER SQLERROR CONTINUE* after each SQL statement, which makes the text of the program longer and less readable. Besides, this statement does not work in all cases (bugs?), what caused the necessity to correct the C program resulted from precompilation.

16. To open a cursor for updating in dynamic SQL, the programmer must generate the statement *select ... from ... FOR DIRECT UPDATE OF ...* which is not described in the manual (we invented it by experiments).

17. An attempt to open a cursor for a query returning an empty result causes a run-time error. Normally it is impossible to predict if the result of the user query is empty or non-empty. This means that empty results of queries must be handled in a special way.

18. Ingres SQL makes no differences between lower and upper cases for names of relations and names of attributes, what creates some problems in DBPL.

6.3 Open and Unsolved Problems

1. Mapping DBPL nested relations into two or more Ingres relations connected by referential integrities. The problem lies in generating SQL statements from DBPL predicates addressing a nested relation, and in composing DBPL nested tuple from several tuples returned by Ingres.

2. Optimization of the case when a DBPL predicate mixes references to DBPL and Ingres relations. For example, when a DBPL relation is nested, the only currently available possibility is copying the Ingres relation(s) to the DBPL side.

3. Processing null-values by DBPL.

4. Mapping of DBPL selectors and constructors into SQL views.

5. The full power of SQL cannot be utilized from DBPL since it does not allow operators and functional symbols within predicates. Two approaches to this problem are possible: extending the DBPL language, compiler and run-time system to handle such cases, or recognizing this cases inside low level DBPL capabilities so that they could be mapped as high-level capabilities to SQL.

6. Utilization various capabilities of Ingres in DBPL: the form management, rules, database procedures, and so on.

7 Conclusion

Implementation of the gateway from DBPL to the commercial Ingres achieves several results. A direct pragmatic result is that Ingres databases are now available within DBPL programs. From the technical and scientific points of view we considered it important that it was possible to implement such a gateway. This work has also uncovered some disadvantages of both DBPL and SQL. Considering DBPL, we recognized limitations of high-level constructs, which may produce problems for users especially if they come from the SQL world. The advantage of DBPL - strong typing - may become a great disadvantage for some applications requiring generic procedures. The interface between DBPL and persistent data is complex and the specification of it refers to low-level data structures.

The majority of problems were connected, however, with SQL. In contrast to the enthusiasm found in popular database textbooks our experience with SQL as a programming language indicated that SQL is below the state-of-the-art. Many ad-hoc solutions, irregularity of syntax and semantics, limitations, unclear rules of use, an approach to user-friendliness which forbids untypical (but still reasonable) situations, lack of programming abstractions, etc. make programming of generic programs difficult and frustrating. We feel that SQL has yet to achieve the maturity necessary for next-generation databases. It is our hope that this paper helps clarify some design pitfalls in database languages and opens the way to provide solutions for their correction.

References

[Banc92] F. Bancilhon. Understanding Object-Oriented Database Systems. Advances in Database Technology - EDBT '92, Proc. of 3rd International Conference on Extending Database Technology, Vienna, Austria, March 1992, Springer LNCS 580, pp.1-9, 1992

[Date87] C.J. Date. A Guide to Ingres. Addison-Wesley 1987.

[Frat91] C. Fratarcangeli. Technique for Universal Quantification in SQL. SIGMOD RECORD Vol.20, No.3, Sep. 1991, pp.16-24

[IngrA89] Ingres/SQL Command Summary for the UNIX and VMS Operating Systems. Release 6, Relational Technology Inc., August 1989

[IngrB89] Ingres/SQL Reference Manual. Release 6, Relational Technology Inc., August 1989

[Orac91] PL/SQL, User Guide and Reference, Version 1.0, June 1991. Oracle Corporation 1991.

[ERMS91] J. Eder, A. Rudloff, F. Matthes, J.W. Schmidt. Data Construction with Recursive Set Expressions. Next Generation Information System Technology.

Proc. of 1st East/West Database Workshop, Kiev, USSR, Oct.1990, Springer Lecture Notes in Computer Science 504, 1991, pp.271-293

[ScMa92] J.W. Schmidt, F Matthes. The Database Programming Language DBPL, Rationale and Report. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/46, 1992

[MRSS92] F. Matthes, A. Rudloff, J.W. Schmidt, K. Subieta. The Database Programming Language DBPL, User and System Manual. FIDE, ESPRIT BRA Project 3070, Technical Report Series, FIDE/92/47, 1992

[SRLG+90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech: The Committee for Advanced DBMS Function. Third-Generation Data Base System Manifesto. Memorandum No. UCB/ERL M90/28, 9 April 1990, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720 (also) Proc. IFIP WG 2.6 Conf. on Object-Oriented Databases (DS-4) Windrmere, July 1990, (also) ACM SIGMOD Record 19(3), pp.31-44, 1990.

8 Appendix: An Example Database (DBPL Syntax)

```
TYPE
  supptype = RECORD  sno      : ARRAY[0..2] OF CHAR;
                     sname    : ARRAY[0..19] OF CHAR;
                     status   : INTEGER;
                     city     : ARRAY[0..19] OF CHAR; END;

  parttype = RECORD  pno      : ARRAY[0..2] OF CHAR;
                     pname    : ARRAY[0..19] OF CHAR;
                     color    : ARRAY[0..9] OF CHAR;
                     weight   : INTEGER;
                     city     : ARRAY[0..19] OF CHAR; END;

  sptype   = RECORD  sno      : ARRAY[0..2] OF CHAR;
                     pno      : ARRAY[0..2] OF CHAR;
                     qty      : INTEGER; END;

  suppRel  = RELATION  sno OF supptype;
  partRel  = RELATION  pno OF parttype;
  spRel    = RELATION  sno, pno OF sptype;
```

```
VAR
  supp   : suppRel;
  part   : partRel;
   sp    : spRel;



PROCEDURE CreateSuppPartDB;
BEGIN
  supp := suppRel{ {'S1', 'Smith', 20, 'London' },
                   {'S2', 'Jones', 10, 'Paris'  },
                   {'S3', 'Blake', 30, 'Paris'  },
                   {'S4', 'Clark', 20, 'London' },
                   {'S5', 'Adams', 30, 'Athens' } };

  part := partRel{ {'P1', 'Nut'  , 'Red'  , 12, 'London' },
                   {'P2', 'Bolt' , 'Green', 17, 'Paris'  },
                   {'P3', 'Screw', 'Blue' , 17, 'Rome'   },
                   {'P4', 'Screw', 'Red'  , 14, 'London' },
                   {'P5', 'Cam'  , 'Blue' , 12, 'Paris'  },
                   {'P6', 'Cog'  , 'Red'  , 19, 'London' } };

  sp   :=  spRel{  {'S1', 'P1', 300 }, {'S1', 'P2', 200 },
                   {'S1', 'P3', 400 }, {'S1', 'P4', 200 },
                   {'S1', 'P5', 100 }, {'S1', 'P6', 100 },
                   {'S2', 'P1', 300 }, {'S2', 'P2', 400 },
                   {'S3', 'P2', 200 }, {'S4', 'P2', 200 },
                   {'S4', 'P4', 300}}, {'S4', 'P5', 400 } };
END CreateSuppPartDB;
```