A Gateway from DBPL to 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:rudloff@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.

KEY WORDS DBPL Modula-2 Ingres SQL Gateway Interoperability

---

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

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. Many functionalities typical for programming languages and programming environments have been fixed in the SQL standard as capabilities of the "query language". Besides the impedance mismatch, this approach involved another disadvantage 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 recently evolved in the direction of programming languages (which is especially striking in the INGRES/Windows/4GL [Ingr90] and 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) are strongly and statically type checked (what gives the possibility to remove many errors before a program is executed), syntactically and semantically orthogonal (what reduces the number of necessary primitives in the language, as well as the size of manuals), they support full computational and pragmatic universality, clean semantics, and various programming abstractions. DBPL-s adopt the concept of a query language as a powerful construct of a programming language. The language DBPL [ScMa92, MRSS92] 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. Commercial systems should be equipped with a large family utilities which are not implemented in DBPL since (however very useful) they do not present scientific problems. 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 the gateway. 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.

There are several possible approaches in making a gateway. In this project we implemented an 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.). 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, and 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 we present architectural assumptions of the gateway in connection with the architecture of DBPL. In Section 5 implementation difficulties are discussed.

2 Similarities and Differences of DBPL and Ingres SQL

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. Variable-length data of Ingres types have no equivalents in DBPL; they can be represented by long fixed-length strings. Ingres types such as *date, money, table-key*, and *object-key* can be represented in DBPL but they require special functions to serve them.

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.

Both Ingres SQL and DBPL make the distinction between querying a database and processing a database. SQL was designed for retrieval and then extended by some programming capabilities, that is, inserting, deleting and updating. It is not computationally complete: more complex (but still typical) programming tasks require a classical (host) programming language. 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;
```

In DBPL and SQL high-level and low-level programming do not match each other: some high-level constructs of DBPL cannot be expressed by constructs of SQL and vice versa. 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 in DBPL such SQL capabilities as arithmetic operators, aggregate functions and grouping.

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 containing the following relations:

```
supp( sno, sname, status, city);
part( pno, pname, color, weight, city );
sp( sno, pno, qty );
```

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.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 SQL high-level "update", "insert", "delete" statements, 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 discuss it later.

SQL views are special objects with independent existence in the 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 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 of SQL view definitions 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.

DBPL does not deal with null-values. 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. 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.

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, and a Ingres application cannot be a client of DBPL.

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 have been forced to use the dynamic SQL. It is an extension of the capabilities in embedded SQL allowing to write SQL statements as strings of characters, which can be manipulated during run-time.

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 it is impossible to predict the number and types of host variables. Another method introduces the so-called SQL Description Area (SQLDA), which allows communication through pointers. SQLDA is a dynamically created data structure consisting of explicit typing information and pointers to data. The pointers are counterparts of the host variables in embedded SQL. They have two kinds of applications. In the first case (used by *select* statements) they determine places, where the attributes of a retrieved tuple are to 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 into the *supp* relation requires preparing the statement

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

In this case the pointers point values, which will substitute the question marks when the statement is executed.

DBPL capabilities include the low-level standard procedures LOWEST, HIGHEST, THIS, NEXT and PRIOR. They enable processing of DBPL relations in a tuple-by-tuple fashion. Some tasks cannot be programmed without them, for example, merging of relations, or 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.

3 Mapping DBPL into SQL

In this section we present basic methods that we have used to map DBPL constructs referring to Ingres relations into dynamic SQL + C constructs. For most DBPL constructs such a mapping exists, but sometimes it is not easy. In some cases there is no convenient solution for converting DBPL constructs into SQL, thus we needed escape methods. Although being not very efficient, they allow the completion of computations. There are several such methods; in this project we use only one of them, namely copying DBPL relations to the Ingres side.

3.1 Unproblematic Cases

- DBPL expressions without quantifiers. 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 quantifiers and all comparisons in p are available in SQL, then these expressions are 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`)`

- DBPL predicates returning boolean values. SQL has no semantic domain with boolean values, thus we convert a DBPL predicate p 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. 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 SOME quantifiers in the prenex form, the conversion is 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

DBPL projection lists into SQL equivalents with no change, 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:

- **Quantified comparisons** They are normal comparisons followed by the key words "all" or "any", for example, "=all", "<any", ">=all", etc. Because of the lack of universality of quantified comparisons the automatic conversion of DBPL's universal quantifiers is problematic.

- **Function "count"** Since it requires materialization of its argument, this method may lead to performance problems.

- **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.

For uniformicity in this stage of the project existential quantifiers are also implemented through the operator "exists". 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 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 subpredicate 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.

3.4 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. There are several methods of solving this problem, in particular unnesting and create a view on the Ingres side. In this project we decided to create a temporary range relation on the Ingres side.

3.5 FOR EACH Construct

Mapping of the FOR EACH construct of DBPL was the most difficult implementation problem. 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 follows:

```
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. DBPL allows nested FOR EACH and recursion thus SQLDA areas, names of cursors, and names of SQL *PREPARE* statements must be managed by a special stack, which we implemented in SQL+C.

The dynamic SQL version is neither clear nor well specified in SQL manuals; there are also some bugs. Thus the final solution is the result 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 attribute$_1$ = ?,..., attribute$_n$ = ?
    WHERE CURRENT OF cursor name

    For the example above we generate the statement

    UPDATE supp SET sno = ?, sname = ?, status = ?, city = ?
    WHERE CURRENT OF dbcurs$_i$

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 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 more difficult than necessary.

3.6 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, dependingly on which is the status of relations on left-hand and right-hand sides (DBPL/DBPL, DBPL/Ingres, Ingres/DBPL, Ingres/Ingres). Each case implied specific methods. Even in the Ingres/Ingres case not all operations can be executed by SQL because of the lack of power. An example of these problems is the *delete* operation for the Ingres/DBPL case. In DBPL 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. The SQL method of passing parameters to statements through question marks requires filling in values of *all* attributes. As follows from DBPL semantics, some of attributes of R_2 tuples may be meaningless. To ignore them, 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 $\texttt{attribute}_i = ?$ for key attributes, and the form $(1 = 1 \texttt{ OR } \texttt{attribute}_i = ?)$ for non-key attributes. For example, the DBPL statement

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

generates the SQL 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. The example clearly shows the disadvantage of the SQL *ad hoc* approach to the generic programming.

3.7 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 R_2( SOME X IN R_1((X.key_1 = Y.key_1) AND ...
                         AND (X.key_last = Y.key_last)))
```

and then generate the corresponding *select* statement:

```
SELECT * FROM AuxRel WHERE NOT EXISTS(
  SELECT * FROM R_2 Y WHERE NOT EXISTS(
    SELECT * FROM R_1 X WHERE X.key_1 = Y.key_1 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*

4 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, which is an agreement between companies supporting SQL concerning source EXEC SQL statements. 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 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

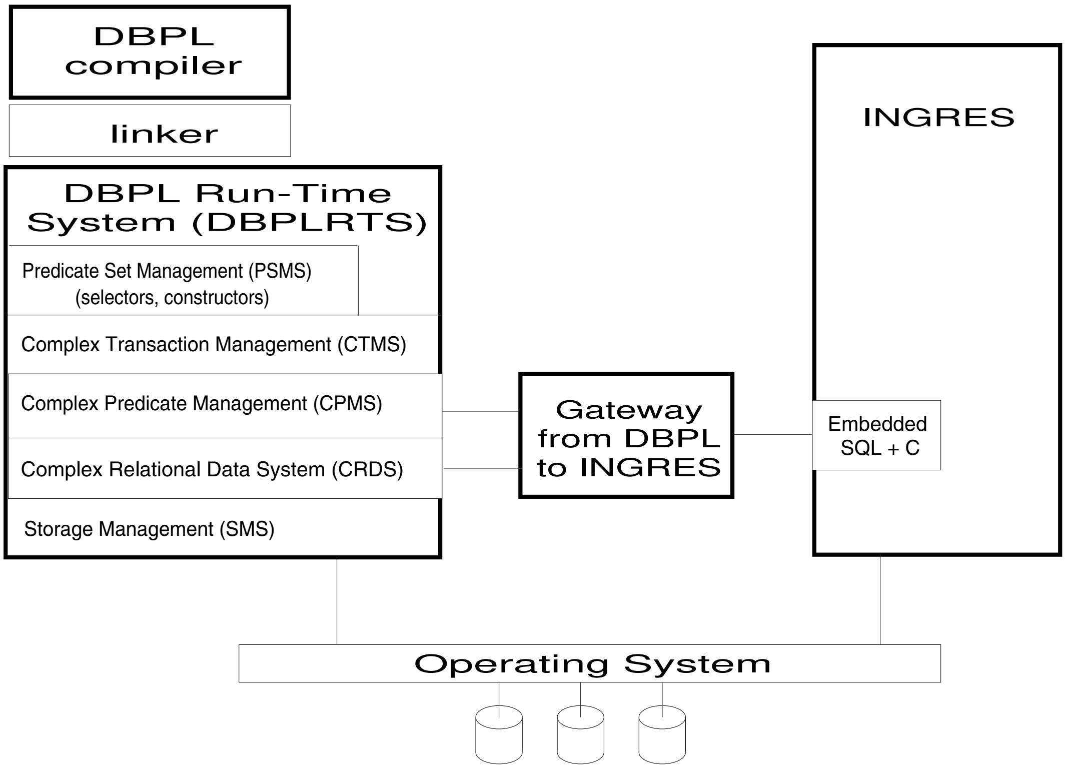
DBPL compiler
linker
DBPL Run-Time System (DBPLRTS)
Predicate Set Management (PSMS) (selectors, constructors)
Complex Transaction Management (CTMS)
Complex Predicate Management (CPMS)
Complex Relational Data System (CRDS)
Storage Management (SMS)
Gateway from DBPL to INGRES
INGRES
Embedded SQL + C
Operating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

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

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 be performed on the Ingres relation.

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 for *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 a procedure for testing and evaluating

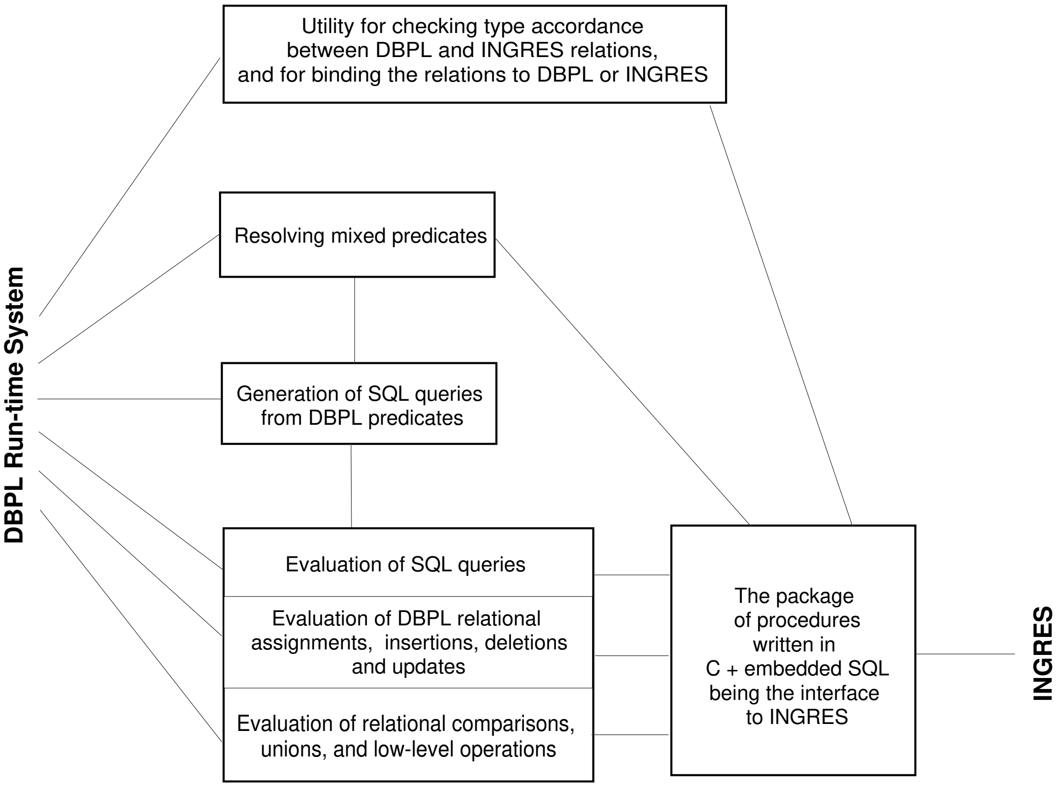
Utility for checking type accordance
between DBPL and INGRES relations,
and for binding the relations to DBPL or INGRES
Resolving mixed predicates
Generation of SQL queries from DBPL predicates
Evaluation of SQL queries
Evaluation of DBPL relational assignments, insertions, deletions and updates
Evaluation 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

DBPL predicates containing mixed references to DBPL and Ingres relations. The procedure 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 .

5 Implementation Difficulties

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

3. No possibility of using arithmetic and other operators inside DBPL high-level constructs limits utilization of the full power of SQL

5.2 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. For example, we would prefer the syntax `count( select ... from ...)` instead of `select count(*) from ...`. The current syntax is sometimes 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 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. 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

[Ingr90] Language Reference Manual for INGRES/Windows 4GL for the UNIX and VMS Operating Systems. INGRES Release 6, Ingres Corporation, August 1990.

[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. ACM SIGMOD Record 19(3), pp.31-44, 1990.