Appeared in: *Lecture Notes in Computer Science* Vol. 819, pp. 365–380, 1994
(Proceedings of First International Conference on Applications of Databases, ADB-94)

A Gateway from DBPL to Ingres [*]

Florian Matthes[1], Andreas Rudloff[1], Joachim W. Schmidt[1], Kazimierz Subieta[2]

[1] University of Hamburg, Dept. of Computer Science, Vogt-Kölln-Straße 30, D-22527 Hamburg, Germany, e-mail: J_Schmidt@informatik.uni-hamburg.de
[2] Polish Academy of Sciences, Institute of Computer Science, Ordona 21, PL-01-237 Warszawa, Poland, e-mail: subieta@wars.ipipan.waw.pl

Abstract. A gateway from DBPL (being a superset of Modula-2) to the commercial database system Ingres is described. DBPL extends Modula-2 by a new bulk data type constructor "relation", persistence, and high-level relational expressions (queries) based on the predicate calculus, thereby maintaining the basic concepts of the language like strong typing and orthogonality. The gateway enables the user to write *normal* DBPL programs for accessing Ingres databases. This is in contrast to typical implementations that embed SQL statements into a programming language and results in a fully transparent interface for DBPL programmers. DBPL queries and statements referring to Ingres tables are automatically converted into corresponding SQL statements, are evaluated by the Ingres database server and the results are transferred back under the control of the DBPL program. This procedure also resolves queries referring to both Ingres and DBPL tables. The design assumptions of the gateway and the used implementation methods are presented as well as design and implementation difficulties.

1 Introduction

The coupling of programming languages with relational database systems is based conventionally 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 were fixed in the SQL standard as capabilities of the "query language". Besides the impedance mismatch, this approach involved yet another disadvantage known as *bottom-up evolution*, i.e. extending incrementally and *ad hoc* the functionalities of query languages with the result that many positive features that were the reason for the initial development were lost. This concerns mainly SQL which recently evolved in the direction of programming languages (which is especially striking in the INGRES/Windows4GL [Ingr90] and Oracle PL/SQL [Orac91]).

[*] This research was supported by ESPRIT Basic Research (FIDE)

Initial motivation and trends in the development of query languages were different from those of programming languages. One 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 with the consequence that other interpretations of user-friendliness are of vital importance such as preventing the user from his/her own errors, computational completeness, and support of various programming abstractions.

Database Programming Languages (DBPL-s) are to be distinguished from these pure query languages. DBPL-s are strongly and statically type checked (allowing the removal of many errors before a program is executed) as well as syntactically and semantically orthogonal (leading to a reduction of the number of necessary primitives in the language, as well as the size of manuals). They support full computational and pragmatic universality as well as various programming abstractions and have clean semantics. DBPL-s adopt the concept of a query language as a powerful construct of a programming language. The language DBPL [ScMa91, 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 allow declarative and associative access to relation variables.

Despite the various advantages of DBPL, we are aware that it has little chance of success in the commercial world as a complete system. 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 produced at a university. 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. Commercial systems are equipped with a large family of 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 and their concepts, however, can be transferred to the commercial world as a supplement on top of popular and widely distributed systems. Many professionals who are dissatisfied with the capabilities and the programming style offered by languages such as SQL embedded in some host language delivered with commercial database systems are potential DBPL clients. Direct use of the high level language with its clean concepts is at hand with storage and access of the data, for example, in an Ingres database, thereby allowing the use of all the tools supported by the system. This was the main reason for deciding to create the gateway. Vice versa, the implementation of the gateway allows the DBPL community - students and researchers - to access large commercial database systems and thereby use, within the DBPL programming environment, their various capabilities.

There are several possible approaches in designing a gateway. In this project we decided to couple the DBPL run-time system with the Ingres SQL machine [IngrA89, IngrB89]. All references from DBPL programs to Ingres databases are transformed during run-time into dynamic SQL statements. This permits the 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. This approach does not support the opinion that SQL should be the "intergalactic language" [SRLG+90] for the next database era, but only accepts a widely used standard. We do not believe that SQL, as a programming language, has reached its maturity (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

There are significant differences in the available types for structuring data offered by Ingres and DBPL. Full orthogonality of type constructors is a leading principle in DBPL which results in the possibility of defining nested relational structures whereas Ingres allows flat relations only. On the level of queries orthogonality of DBPL f.e. allows range relations of queries to be described by queries itself, in SQL this is not possible. Persistence in DBPL is also introduced as a orthogonal property of modules (**DATABASE MODULE**s) allowing variables of every type (except pointer-variables) to become persistent. INGRES only supports persistent relations.

Primary keys of relations in DBPL are considered as a structural property of a relation and as such they are declared in the definition of the relation type. The semantics of some operators depends on them. In contrast to this primary keys in Ingres are defined for relation variables and used only internally (for creating an index structure). For querying the information about primary keys is irrelevant. As a consequence DBPL does not allow duplicate tuples either in stored relations or in intermediate query outputs, but Ingres does so. So efficient programming of some tasks in DBPL may prove impossible.

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, functions and operators are not allowed within DBPL predicates. This restriction results in a potential for good performance, however, it violates the orthogonality principle. As consequence it is practically impossible to utilize in DBPL queries requiring capabilities available in SQL like 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 defined as follows (whereby the primary key is determined by the attributes following the `RELATION`-keyword in the type definition):

```
TYPE suppRel = RELATION sno OF
          RECORD sno:...; sname:...; status:...; city:... END;
     partRel = RELATION pno OF
          RECORD pno:...; pname:...; color:...;
                 weight:...; city:... END;
     spRel   = RELATION pno OF
          RECORD sno:...; pno:...; qty:... END;
VAR supp: suppRel; part: partRel; sp: spRel;
```

1. 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.
2. Selective access expressions, for example:

   ```
   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.
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 have no counterpart in SQL. One example is: for example:

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

5. Relation expressions for describing relation values by combining an access expression with a compatible relation type (which determines the primary

key), 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))}
```

Relation 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 the insert operation:

```
supp   :+ suppRel{{"S1", "Schmidt", 25, "Berlin"}};
```

   All operators follow the "many-data-at-a-time" principle (both operands are relation-valued; in the example the right-hand side expressions is a relation value of cardinality one). They can be implemented by SQL high-level "update", "insert", "delete" statements, or by fetching the required tuples from Ingres tables to a DBPL buffer, doing the required operations and shipping them back to Ingres.

The standard DBPL functions CARD (the number of relation elements) has a direct counterpart in SQL whereas the "one-data-at-a-time" functions EXCL and INCL (exclude/include one tuple) present a problem, because their semantics is based on primary keys (discussed later). No SQL equivalent exists for 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 for visualizing the contents of a database.

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. Both, selectors and constructors, may have parameters and so they are different from SQL views. Beside this selectors could be considered as updatable views whereas the constructors (pure query expressions) could be recursive (with a fixed-point semantics). 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 (still allowing to evaluate recursive queries on Ingres tables).

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. Ingres types such as *date, money, table-key,* and *object-key* could be represented in DBPL but they require special functions to serve them and are currently not available.

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.

3 Mapping DBPL into SQL

Since the gateway from DBPL to Ingres is a generic application which must work for all types of relations and for any DBPL high-level expressions the use of dynamic SQL was necessary. It is an extension of the capabilities in embedded SQL allowing to write SQL statements as strings which can be manipulated during run-time. Basic component is the so-called SQL Description Area (SQLDA), which allows a communication between the application and the Ingres server through pointers. It 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" which will be substituted by the values referenced through the pointers at execution time.

In the following the basic methods that have been used to map DBPL constructs referring to Ingres relations are presented. For most DBPL constructs such a mapping exists, but in some cases there is no convenient solution, thus we needed some 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 expression:

```
{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 this expression is equivalent to the following SQL query:

```
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 X where not( X.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. The DBPL access expression and the corresponding SQL statement for the query ("Suppliers supplying all parts") are

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

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 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 the application of 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 creation of 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 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);
  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 SQLDAs. DBPL allows nested FOR EACH and recursion, thus SQLDAs, names of cursors and 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 the query generated in the previous step as argument. This step is necessary to obtain the 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. Then create a new SQLDA and push it also on the stack.
5. Execute *PREPARE* and *DESCRIBE* SQL statements for the given extended query, statement name and SQLDA.
6. Generate a new cursor name and push it on the stack. Then declare the cursor.

7. Open the cursor. This is the preparation step for fetching tuples from the Ingres relation.
8. Extract the relation name from the query and 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
   ```
9. Generate a new statement name and push it on the stack. Then 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 it should be.

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, depending on the status of the left-hand and right-hand side relations (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

```
DELETE FROM R_1 WHERE p_1 AND p_2 AND ... AND p_n
```

where n is the number of attributes. Predicate p_i has the form `attribute`$_i$ = ? for key attributes, and the form (1 = 1 `OR 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 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

$\pi_{primarykeys}(\texttt{R}_1) \supseteq \pi_{primarykeys}(\texttt{R}_2)$

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 whether the 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 transform it in a corresponding *select* statement (see sec. 3.2).

4 Architecture of the Gateway

The general architectural view of the gateway, DBPL and Ingres is presented in Fig. 1. The entry Ingres interface is embedded SQL. We wrote a package of procedures in embedded SQL + C capable of mapping all DBPL constructs (unfortunately the dynamic embedding is not standardized until now). The procedures are also available in 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.

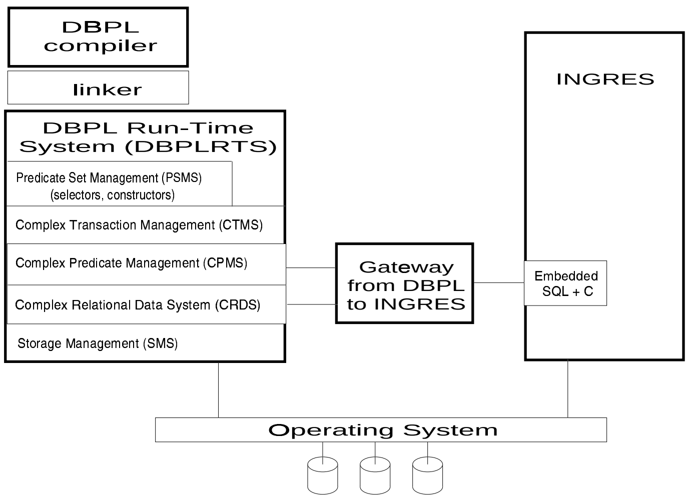
DBPL compiler
linker
DBPL Run-Time System (DBPLRTS)
Predicate Set Management (PSMS) (selectors, constructors)
Complex Transaction Management (CTMS)
Complex Predicate Management (CPMS)
Complex Relational Data System (CRDS)
Storage Management (SMS)
Gateway from DBPL to INGRES
INGRES
Embedded SQL + C
Operating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

Fig. 1. General view on the gateway, DBPL and Ingres

We introduced a change to the DBPL data dictionary allowing to distinguish 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 retrieve typing information and sometimes they are internally used for storing intermediate results. The architecture of the gateway is presented in more detail in Fig. 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.

For evaluating DBPL predicates containing mixed references to DBPL and

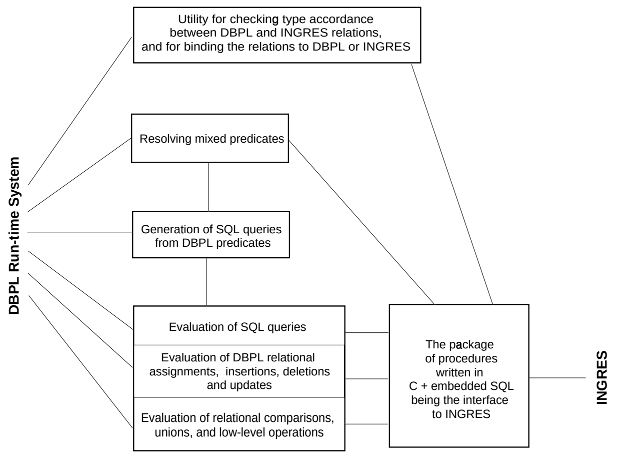
Utility for checking type accordance between DBPL and INGRES relations, and for binding the relations to DBPL or INGRES
Resolving mixed predicates
Generation of SQL queries from DBPL predicates
Evaluation of SQL queries
Evaluation of DBPL relational assignments, insertions, deletions and updates
Evaluation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

Fig. 2. Architecture of the gateway

Ingres relations the corresponding predicate tree is recursively scanned and one of the following answers is produced: *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.* They can be evaluated internally and 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

A few problems were caused on the side of DBPL. On the language level the missing possibility of using arithmetic and other operators inside the high-level constructs limits the utilization of the full power of SQL. On the system level the use of Modula-2 as system-programming language requires that all advanced data structures are stored in dynamic memory. This concerns, in particular, syntactic trees for high-level expressions and predicates, type descriptors, and the 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. 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.

Most of the problems had their origin on the SQL side especially in bad design assumption of SQL and its (dynamic) embedding concept.

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 SQLDAs together with operators acting on this stack.
2. Although the concept of primary keys is a fundamental part of 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 expressing some DBPL updates in SQL.
3. The missing orthogonality of the language leads to problems in the automatic generation of SQL queries. For example there is no nested union, there are no range expressions described by queries, all this must be done on the DBPL side. Another example is the `select count(*) from ...` statement where we would prefer the syntax `count( select ... from ...)`.
4. No queries returning boolean values, they must be simulated by counting the number of elements of a corresponding select-statement. The missing truth values requires the substitution by formulas like `1=1` or `1=0`.

Sometimes it is also difficult to retrieve internal information from the system which is necessary for writing generic applications. May be the ongoing standardization work (f.e. for the data dictionary) will lead to a better situation.

1. There are difficulties in retrieving all information about Ingres tables; in particular, this concerns recognizing which attributes are forming the primary keys.
2. No comparison of tuples for equality, and no (officially supported) explicit tuple identifiers and operations on them.
3. In programming of generic applications we need to "capture" some system reactions to errors. These reactions and error codes are not well specified in the documentation of Ingres. In this context the automatic generation of the *SQL STOP* statement, in all possible places where an 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. SQL itself gives poor testing capabilities for programmers, e.g. about names of available relations, about their ownership, status, number of tuples, etc.

With new releases (and realizations) of the SQL-standard, which will also cover dynamic SQL, some of the following problems will hopefully become obsolete.

1. The system of navigation through cursors gives no possibility to navigate to the *prior* tuple, making the implementation of browsing capabilities ex-

tremely difficult.

2. SQL dynamic statements use question marks as "formal parameters". This is inconvenient and error-prone.
3. 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).
4. 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.
5. 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).

Surprisingly there also have been technical problems with bugs in the used version of the Ingres system. So opening a cursor for a query returning an empty result causes a run-time error. Because normally it is impossible to predict whether the result of a user query is empty or non-empty a special handling of empty query results is necessary. Another example was the *WHENEVER SQLERROR CONTINUE* statement that does not work in all cases, requiring the manual correction of the C programs generated by the ESQL-precompiler.

6 Conclusion

The implementation of the gateway from DBPL to the commercial Ingres system achieves several results. A direct pragmatic result is that Ingres databases are now transparently accessible within DBPL programs. This allows the user to build up modular designed applications using a type-safe language with orthogonal language constructs thereby significantly reducing the risk of run-time errors and increasing the maintainability and extensibility of the application while using the reliability and effectiveness of a commercial database system developed and improved over years. It has also uncovered some limitations and disadvantages of both DBPL and SQL. When 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 hindrance to the development of some applications requiring generic procedures mainly because of its monomorphic type system. This shows directly the need for new languages with polymorphic type systems as a base for persistent programming and system construction [ScMa93].

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 the programming of generic programs difficult and frustrating.

This clearly emphasizes the necessity and usefulness of our approach to view database systems simply as external servers which can be accessed via powerful languages like DBPL. 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 offers a way to solutions for avoiding these problems.

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.

[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 LNCS 504, 1991, pp.271-293

[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.

[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

[ScMa91] J.W. Schmidt, F. Matthes. The Rationale behind DBPL. 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, Springer LNCS 495, 1991.

[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

[ScMa93] J.W. Schmidt, F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. Proceedings if the IEEE International Workshop on Research Issues in Data Engineering RIDE'93 Vienna, Austria, April 1993

[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.

This article was processed using the LaTeX macro package with LLNCS style