On Integration of Relational and Object-Oriented Database Systems*

Maria E. Orlowska\textsuperscript{1} and Hui Li\textsuperscript{1} and Chengfei Liu\textsuperscript{2}

\textsuperscript{1} School of Information Technology, The University of Queensland, Brisbane, Qld 4072, Australia
\textsuperscript{2} CRC for Distributed Systems Technology, Brisbane, Qld 4072, Australia

Abstract. The converging trend of relational database technology and object-oriented database technology results in object-relational database systems which extend the relational systems with add-on object features. Efforts from both academic research and industry have been directed into extended data type processing by fully functioned database systems. In this paper, we put forward an approach for realizing object-relational systems by deploying current distributed database technology. The paper first discusses the object-relational data model and its query language. Then a heterogeneous database architecture is presented for realizing object-relational technology by integrating relational and object-oriented database systems. Two main functions of the architecture: schema transformation and query translation are further discussed and correspondent algorithms are proposed.

1 Introduction

Traditional relational database systems (RDBMSs) have demonstrated a strong capability for most database applications, but suffer from almost no support for complex data. All data are stored in tables, and every attribute in the table is defined on one of the few atomic basic types (e.g. integer, float, boolean, character, string). It is currently being accepted that RDBMSs should allow type extensibility, i.e., new data types can be added to the system without significant changes to any part of the existing code. Unfortunately, in traditional RDBMSs complex data can only be stored as uninterpreted BLOBs (Binary Large Objects), and the interpretation of this data relies solely on the application. However, many specialised databases, such as engineering DBs, spatial DBs, multimedia DBs, scientific and statistical DBs, require more complex structures for data and nonstandard application-specific operations. It is desirable to extend the relational model to accommodate these features.

About a decade ago, researchers began to investigate general methods to introduce objects into database systems. A number of different ways were explored

* The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the Department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.
extended relational database systems, object-oriented (OO) databases [7, 3], toolkits for constructing special-purpose database systems, and persistent programming language. As claimed by Carey [2], the extended relational database systems, as they are called object-relational (OR) database systems now, appear likely to emerge as the ultimate winner in terms of providing objects for mainstream enterprise database applications.

OR systems start with the relational model and its query language, SQL, and are built from there. As such, it has a strong base: conceptual simplicity, open standard query language, powerful transaction management and recovery facilities, efficiency, availability, scalability, rich set of applications. All of these are what an OO database system lacks. Besides, OR systems also accommodate many object features. Among them are abstract data types (ADTs), row types, references, multivalued attributes and inheritance. ADTs are user-defined base types. Their role is to enable the set of built-in data types of the DBMS to be extended with new data types such as text, image, polygon, etc. Row types are a direct and natural extension of the type system for tuples. In addition to base type attributes, row objects are permitted to contain reference-valued attributes and multivalued attributes. The introduction of reference-valued attributes [8] will not only provide shareability of type instances, but also will enable us to support OO capabilities for rows in existing tables. A multivalued attribute value can be a set, a bag, a list or an array of base type elements. Finally, inheritance is also supported to enable natural variations among row types or ADTs to be captured in the schema.

Currently, different approaches are being taken by vendors to provide OR database products: native implementation such as Informix’s Illustra — commercialized version of Postgres [12], Fujitsu’s ODB II – commercialized version of Jasmine [5], Omniscience and UniSQL; incremental evolution taken by CA-Ingres, DB2/6000 C/S, Oracle 8, etc.; wrapper approach taken by HP’s Odapter. Building an ORDBMS is a complex and time-consuming task, requiring hundreds of man years of effort. It is always psychologically difficult for people to discard their investment in an old system.

For the purpose of providing new technology without giving up old systems, we put forward a new approach [10], utilising a heterogeneous database architecture as a vehicle for OR database system implementation. A relational DBMS and an OODBMS are integrated together to provide an ORDB environment. This approach preserves an enterprise’s current investment on relational database systems and applications, while still offering the benefits of new add-on OO features. Compared with the wrapper and gateway approaches discussed by Stonebraker [13], our approach is more biased towards using existing resources.

In our proposed heterogeneous database architecture, all the data are actually stored as underlying relational tables and objects. The OR table on the top is virtual. When an OR schema of a database application is defined, it is necessary to translate the schema into local relational and OO database schemas. When an query is issued against global OR schema, it must be translated to a set of local queries against corresponding local relational schema and OO schema
for execution. These local queries are either pure relational queries or pure OR queries. The local partial results are then sent back to the top level. The RDB engine at the global level is responsible for merging the partial results into the final result. In the rest of the paper, the object-relational data model and its query language is discussed in section 2. In section 3, a heterogeneous database architecture is presented for implementation of OR systems. Two kernel functions of the architecture, the schema transformation and query partition, are further studied in section 4 and section 5, respectively. Section 6 concludes the paper.

2 Object-Relational Databases

In this section, an object-relational data model is introduced, with emphasis on type extensibility, and corresponding extended query features.

2.1 Object-relational data model

There is still no agreement on how the relational model should be extended to have the modelling power of object-oriented systems while keeping the simplicity of relational systems. Current SQL3 draft [14] only supports unnamed row types, ADTs and collection types. In the separate "SQL/Object" part of SQL3 [8], named row types (NRTs) are introduced with polymorphism, identity, no inheritance, and no encapsulation. Reference types are also introduced but only references to row types are allowed. In contrast, an ADT supports polymorphism, inheritance, encapsulation, but no identity. Beech [1] suggests that a possible future simplification of SQL3 is a combination of ADTs and NRTs. However, this may compromise the simplicity of relational systems. In this paper, we prefer to keep the relational flavour in OR systems, i.e., to keep a distinction between ADTs and NRTs.

The basic components of our object-relational model are types. An OR database schema consists of a set of row types, and each attribute in a row type is defined on a certain type, which can be a built-in type, an abstract data type (ADT), a collection type, a reference type or another row type. Therefore, the types can be defined recursively as follows:

**Base types** are the system built-in types including integer, float, date, string, boolean and day-time, which are supported by SQL92.

**Abstract Data Types (ADTs)** are user defined and implemented types. The implementation of objects, and their attributes and behaviours, is invisible to the query system. All accesses to the instances of an ADT are through the interface defined for the type. Therefore, for the purpose of our discussion here, we can describe an ADT by a collection of named functions, \( T(f_1 : T_1, \cdots, f_n : T_n) \) where \( T \) is the name of the ADT, \( f_i (1 \leq i \leq n) \) represents a method of the ADT \( T_i \), the function type of \( f_i \). A function type has the form \( Fun(T_0, T_1, \cdots, T_n) \) where \( T_1, \cdots, T_n \) is a list of input types, and \( T_0 \) is the output type of the function.
Row types have the form $T(A_1 : T_1, \cdots, A_n : T_n) : (T_r, \cdots, T_m)$, where $T$ is the name of the row type, $A_i (1 \leq i \leq n)$ is the name of an attribute, $T_i$ is the data type of $A_i$, and $T_j (r \leq j \leq m)$ is a supertype of $T$. A row type has a name and a set of attributes. An instance of a row type $T$ is an element of the Cartesian product of the value sets of the types that define $T$. A row type may be defined with or without a name. The former is called a named row type (NRT), and the latter is called an unnamed row type.

Reference types have the form $ref(T)$ where $T$ is a row type. In current OR models, tables are the only top-level named entities that can be stored persistently. In other words, only tuples in a table are treated as independent objects with identity and thus can be referenced.

Collection types have the form $C(T)$ where $T$ can be a base type, ADT, row type, reference type or another collection type. $C$ represents one of the built-in collection type constructs, including set, bag, list, tree, etc.. They represent different ways to group up $T$’s instances. For the brevity of discussion, we only discuss set type in the paper.

2.2 Query language extension

Due to type extensibility, SQL92 is no longer sufficient for querying OR databases. In supporting type extensibility, we extend SQL92 with the following query features which are supported in our SQL3-like language.

method invocations – Since ADT have methods defined, method invocations are allowed to appear in the select-clause and where-clauses. A method invocation may take zero or more values as input and one value as output. The data type of every value is either a base type or an ADT.

path expressions – Since NRTs and reference types are introduced, path expressions are used to navigate the complex structures of objects and their relationships to other objects via references. A deref() function is used to dereference an object identity to get its object. In the query, we adopt the dot notation to represent a path expression. For example, $V.A_1.A_2.\cdots A_k$ is a path expression in the query where $V$ is a tuple variable, $A_i (1 \leq i \leq k - 1)$ is either an attribute or a dereferenced attribute in the OR table of $V$, and $A_k$ can be an attribute or a dereferenced attribute of type($A_{k-1}$), or a method invocation of $A_{k-1}$.

set operations – Since collection types are introduced, set operations are also allowed in the where-clause. The predicates such as membership ($IN$) and inclusion ($ISSUB$) are allowed.

The followings are examples of OR schema definition and queries.

Example 1. In the following we define an OR database schema for a company. It consists of one ADT point, two NRTs emp, dept, and two tables emp and dept.

create ADT point (  
x.coordinate float,  
y.coordinate float;  
distance(point) float);
const CENTRAL_POINT = new point(0, 0);

create NRT emp_t {
    name varchar(30),
    salary decimal(9,2),
    interest set varchar(40)),
    location point,
    dept ref dept_t)
    friend set(REF(emp_t));

create table emp of emp_t
    scope for dept is dept,
    scope for friend is emp;

create NRT dept_t {
    dname varchar(30),
    budget float,
    location point,
    manager ref(emp_t);
    dept ref dept_t);
    friend set(REF(emp_t));

create table dept of dept_t
    scope for dept is dept,
    scope for manager is emp;

Example 2. Find the names and research interests of all employees who have interest in ORDB and work for the department which is located in central area (within 2 kilometers from the central point) and has budget more than 1 million dollars.

```
sel e.name, e.interest
from emp e
where e.deref(dept).budget > 1,000,000 and
  "ORDB" in e.interest and
  d.location.distance(CENTRAL POINT) < 2;
```

Example 3. Find the names of managers who have friends working in the same department and their salaries are more than $100,000.

```
sel d.deref(manager).name
from dept d, emp e
where d = e.deref(dept) and e in d.deref(manager).friend
  and e.salary >= 100,000;
```

Notice in the above example queries, we have used method invocations, set operations and path expressions to represent the queries. All these query features are used to support the type extensibility, therefore, they can not be found in SQL92.

3 HDB Architecture for ORDBMS

In this section, we address how a heterogeneous database (HDB) architecture can be used as a vehicle for realizing OR technology. Given a SQL3-like language, we need an ORDB engine to implement it. The HDB engine shown in Figure 1 can be used for this purpose. In fact, the HDB engine is a virtual ORDB engine, it is built based on local RDB engine and OODB engine. We provide so-called functional transparency. In the architecture, we are only interested in how a global ORDB request is functionally transformed into local RDB and OODB subrequests. By functional transparency we mean users need not know such functional transformation. Given a SQL3-like request, the HDE engine interprets the request in terms of supporting RDB and OODB engines. In the following, we describe each component in the architecture.
**Interface** - It receives users’ SQL3-like requests, does syntactical check and hands them to corresponding components for processing. It is also responsible to return the results of the requests back to users.

**Schema Transformer** - It handles SQL3-like requests for schema definition. The main function of this component is to keep the information of OR schema into the global directory and to transform the schema into schemas of both local RDB and OODB engines. A transformation plan is generated by this component which is delivered to the executor for execution. The schema mapping information is also kept in the global directory. The detail of this component is further discussed in section 4.

**Query Partitioner** - This component is responsible for translating SQL3-like queries which are issued against global OR schema into local queries on both local RDB and OODB engines. The schema mapping information is used for such transformation. All local queries form a query plan which is handed over to the executor. This component will be discussed in detail in section 5.

**Executor** - The executor is responsible for coordinating the distributed execution of the execution plan (both transformation and query plan). There is a RDB engine employed at the global level to hold temporary results which may pass between local relational system and OO system, to merge the results. Since
users are on the top of an ORDB interface, they may require the result in a form which is more than a flat table. Therefore, some functions are added for this purpose, such as, nest for reconstructing set-valued column values, deref for obtaining the objects, etc.

**RDB-Agent** – The RDB-agent is responsible for monitoring the execution of local queries against transformed local relational schema and returning the results.

**OODB-Agent** – The functions of this agent include: (a) translating SQL3-like queries to ODMG-93 OQL [3] queries which are supposed to be supported by the OODB engine; (b) monitoring the execution of local queries against transformed local OO schema; (c) returning results to the executor.

### 4 Schema Transformation

C. Yu et al. [15, 11] have studied translations between relational systems and object-oriented systems. In providing relational (OO) frontend to an OO (relational) system, two tasks with opposite directions are necessary: one is the transformation from OO (relational) schema to its relational (OO) equivalent schema, the other is translation of relational (OO) queries to OO (relational) equivalent. Our work is different, we are supporting top-down OR database design, both schema transformation and query translation tasks are one-directional, i.e., from OR to both relational and OO. Besides, in [11], only structural part of OC schema is translated to relational. In this approach we consider both structural and behavioral aspects of an OR system.

In this section, we show how a global OR schema can be transformed into corresponding local relational and OO schemas. Our criteria is to maximise the transformation of OR schema to relational schema as a trivial transformation always exists from OR schema completely to OO local schema.

In the following, we first informally describe the steps of an algorithm for mapping OR schema to its corresponding relational schema and OO schema. Then, we summarize with a correspondence between an OR table and its transformed relational tables and OO classes.

#### 4.1 Transformation Algorithm

**Input** An OR schema

**Output** A relational schema and an associated OO schema.

We use two lists to keep useful intermediate information. $\mathcal{NRT}$ for definitions of NRTs, $\mathcal{TAN}$ for table structures to be transformed. Initially, they are empty.

**Step 1:** For every NRT defined in the OR schema, create a structure in $\mathcal{NRT}$ to record the NRT name and its field definitions

**Step 2:** For every ADI, output a definition of a class called *deputy class* which is used to implement the ADT. The definition of the deputy class is formed by including definitions of attributes and methods of the ADT definition and the definition of an extra object identity attribute *oid*. 
Step 3: For table named ORT defined in the OR schema, create a (frame) structure in TRAV to record the table name, copy the field definitions of the NRT in NRT on which it is defined as the field definitions of the table, and if scopes of referenced fields are defined with the table, keep the names of scope tables within the field definition. A table called frame table with name ORT..R will be created later in the relational schema based on this frame structure.

Step 4: Loop for each table structure in TRAV until TRAV becomes empty.

Step 5: For each table structure in TRAV, let its table name TBL, create a relational table definition with table name TBL..R. For each field definition F : T in TBL, we classify 8 cases for different processing. Remove the structure of TBL after all its field definitions have been processed.

Case 1: If T is a built-in type, output a column definition F : T for TBL..R

Case 2: If T is an ADT, output a column definition F : long for TBL..R, we use type long to represent object identity.

Case 3: If T is a NRT, replace the field definition with all field definitions of the NRT T for processing. The newly added field definitions can be found in NRT. their field names need to be renamed with the prefix _.

Case 4: If T is a reference type ref(RNRT), there should be a scope clause declaring the table, say R1..BL, it really references to. Let RPK : TPK its primary key definition, then output a column definition F : TPK for TBL..R.

Case 5: If T is a set type set(ET) and ET is a built-in type, output a definition for a table called auxiliary table TBL..F..R(PK : TPK, TBL..F : ET), where PK : TPK is the primary key definition for TBL..R.

Case 6: If T is a set type set(ET) and ET is an ADT, output a definition of a class called auxiliary class TBL..F..C(oid : long, TBL..F : set(ET))

Case 7: If T is a set type set(ET) and ET is a NRT, create an auxiliary structure in TRAV with TBL..F..R as table name and (PK : TPK, EF1 : ET1, ..., EFm : ETm) as its field definition, where PK : TPK is the primary key definition for TBL..R and (EF1 : ET1, ..., EFm : ETm) is the definition for ET which can be retrieved from NRT.

Case 8: If T is a set type set(ET). ET is a reference type ref(RNRT), there should be a scope clause declaring the table, say R1..BL, it really references to. Let PK : TPK and RPK : RPK be primary key definitions of TBL and R1..BL, respectively, then output a definition of a table called auxiliary table TBL..F..R(PK : TPK, F..R..PK : RPK).

The above algorithm transforms an OO schema into a relational schema and an OO schema, if any. For every table in the OR schema, a frame table definition is always generated in the relational schema, together with definitions of auxiliary tables and classes, if any. A deputy class definition is always generated in the OO schema which implements an ADT in the OR schema. The definitions of NRTs in the OR schema are embedded in the transformed relational and OO schemas. In section 4.2 we will show how the transformed local relational and OO schemas correspond to the global OR schema. An example of schema transformation is given below.

Example 4. The transformed relational schema and OO schema of the OR schema of Example 1 are given below.
create class point {
    oid: long,
    x-coordinate: float,
    y-coordinate: float;
    distance(point): float;
}

create table emp_R {
    name varchar(30),
    salary decimal(9,2),
    location long,
    dept varchar(30),
    primary key is name;
}

create table dept_R {
    dname varchar(30),
    budget float,
    location long,
    manager varchar(30),
    primary key is dname;
}

create table emp_interest_R {
    name varchar(30),
    emp_interest varchar(40),
    primary key is name;
}

create table emp_friend_R {
    name varchar(30),
    friend_name varchar(30),
    primary key is name;
}

4.2 Global and Local Schema Correspondence

As shown in above section, an OR table TBL can be transformed into a frame relational table TBL_R, together with a set of auxiliary relational tables and a set of auxiliary classes. An ADT can be transformed into a deputy class Suppose TBL is defined on the NRT NRT_0(F_0 : BT_0, F_1 : BT_1, F_2 : ADT_2, F_3 : NRT_3, F_4 : ref(NRT_4), F_5 : set(BT_5), F_6 : set(ADT_6), F_7 : set(NRT_7), F_8 : set(ref(NRT_8))) where definitions of F_1 (1 ≤ i ≤ 8) are in correspondence with 8 cases discussed above. Let DC_5, DC_6 are deputy classes for ADT_2, ADT_6; AC_4 is auxiliary class for set(ADT_6); TBL_5, TBL_7, TBL_8 are auxiliary tables for set(BT_5), set(NRT_7), set(ref(NRT_8)); RTBL_4 and RTBL_8 are scope tables for F_4 and F_8. Suppose NRIs, existing tables and transformed tables have the following definitions:

The global table TBL can be represented as

select TBL.R.F_1 , TBL.R.deref(f_2) , TBL.R.F_3 , TBL.R.F_4 ,
nest(TBL.L.BT_1 , TBL.L.deref(f_2) , TBL.L.F_3 , TBL.L.R.F_4 ,
from I.BL.R.I.BL_1 , I.BL_2 , I.BL_4 , DC_5 , AC_4 , RI.BL_1 , RI.BL_4
where TBL.R.F_2 = DC_5.oid and TBL.R.F_4 = RTBL_4.F_3 and TBL.R.F_0 = TBL_5.F_0
and I.BL.R.F_0 = AC_4.oid and I.BL.R.F_0 = I.BL_2.F_0 and I.BL.R.F_0 = I.BL_4.F_0 and
TBL.R.F_0 = RTBL_8.F_0

The predicates used in bridging relational side to OO side are called bridge constraints, e.g., TBL.R.F_2 = DC_5.oid and TBL.R.F_0 = AC_4.oid in the above where clauses are bridge constraints.

Example 5. The correspondence between OR table emp in Example 1 and its transformed representation in Example 4 is given below

select emp.R.name , emp.R.salary , nest(emp.R.emp_interest , emp.R.emp_interest.R.emp_interest)
from emp.R.emp_interest.R.emp_interest.R.emp_interest.R.emp_interest

from emp.R.emp_interest.R.emp_interest.R.emp_interest.R.emp_interest

where emp.R.name = emp.interest.R.name and emp.R.name = emp.friend.R.name
and emp.R.location = point.oid and emp.R.dept = dept.R.dname

5 Query Partition

Given a query against global OR schema, it must be translated to local queries against its transformed local relational and OO schemas for execution. We design a query partition algorithm which has three major steps: substitution, decomposition and final result processing. The result consists of a group of relational queries in SQL92 [6, 4] format and a group of OO queries in OQL [3] format.

To simplify the discussion, we assume that all the constraints in the where-clause are connected by only “AND” operators. This is reasonable because for any where-clause C which contains an “OR”, it can always be transformed into the disjunctive form, “C1 OR C2”. The original query Q can then be translated into “Q1 UNION Q2”, where Q1 and Q2 take C1 and C2 respectively in the where-clause.

5.1 The substitution process

To process the OR query, the first task is to transform the OR style representations into suitable local forms. The OR table and attribute names should be substituted by local table and attribute names. As the set-valued attributes are flattened, some related predicates, including membership and inclusion, should be rewritten. More importantly, as the navigational access is not supported by relational systems, many path expressions V.A1, ... , A_n need to be translated into V.A form and a set of join predicates to record the traversal information. After the substitution, all the data elements from the local relational tables have the strict V.A form, and those from OO classes may have arbitrarily long paths starting from an OO variable.

An important concept used in the following algorithm is the cluster of path expressions. A cluster of path expressions in a query is a set of path expressions which start with the same tuple variable and have same first attribute. For example, V.A1, V.A1.A2 and V.A1.A1' are in the same cluster, while V.A1 and V.A1' belong to two other clusters.

Input: An OR query in the select · · · from · · · where · · · form, and there is nc “OR” operation in where-clause.

Output: An intermediate form. It is an integrated query against local schemes.

Step 1: (Initialization) Define k as the variable counter, and initialise with value 1. k will be used to name the variables in the query. Each time a variable being renamed or a new variable being created, k is increased by 1.

Step 2: (Translation of the from-clause and variable names) For each tuple variable V of OR table TBL in the from-clause of an OR query, replace the table name with the corresponding local frame table name TBL.R so that V becomes a relational tuple variable on TBL.R. Rename V and its all occurrence in the query with “VAR.k” so that all the variable names have a standard format. In
the rest of this algorithm we will still use $V$ to represent the variables wherever the
name format is not concerned.

**Step 3:** (Translation of the path expressions) In select-clause and where-clause,
for each cluster of path expressions in the form $V.A_1 \cdots$ or $V.deref(A_1) \cdots$, do
the following process based on the type of $A_1$, say $type(A_1)$. Repeat this step
until no more change can be applied

**Case 1:** If $type(A_1)$ is a base type, $A_1$ should be the end of the path expressions.
Nothing need to be done. Or if $V$ is defined on an OO class, the path expressions
need not to be changed either.

**Case 2:** If $type(A_1)$ is an ADT, add a new variable definition in the from-clause,
"ADTC VAR.k", where $ADTC$ is the OO class of the ADT. In the where-clause,
add a new predicate, "AND (V.A_1 = VAR.k.OID)". Change the original path expressions into "VAR.k \cdots".

**Case 3:** If $type(A_1)$ is a NRT, for each path expression in the cluster, there are
following subcases.

3.1 If there is a node, say $A_2$ after it, then based on the previous schema
transformation algorithm, $A_2$ should now be an attribute of $TBL.R$ and be renamed
as $A_1 \cdot A_2$. Here $TBL.R$ is the relation on which $V$ is defined. Therefore, we can
translate the original path expressions to "V.A_1 \cdot A_2 \cdots".

3.2 If $A_1$ is the last attribute in the path expression, and the expression appears
in the select-clause, then change the path expression with a group of expressions,
"V.A_1 \cdot A_1^1, \cdots, V.A_1 \cdot A_1^n", where $A_i^1 (1 \leq i \leq n)$ are the all attributes of
$type(A_1)$.

3.3 If $A_1$ is the last attribute in the path expression, and the expression appears
in the where-clause, then change the path expression with "V.A_1 \cdot P.K", where
$P.K$ is the primary key of $type(A_1)$

**Case 4:** If $type(A_1)$ is a reference type ref(RNRT), where $RNRT$ is a NRT,
there should be a table $RIBL.R$ to hold the referenced tuple. Add a new variable
definition in the from-clause, "RIBL.R VAR.k". In the where-clause, add
a new predicate, "AND (V.A_1 = VAR.k.RPK)"; where $RPK$ is the primary key
of $RNRT$. For each path expression in the cluster, there are following subcases
regarding the modification of the expression itself.

4.1 If the form is like $V.deref(A_1).A_2 \cdots$ (there is another node after $A_1$), change
the original path expressions into "VAR.k.A_2 \cdots".

4.2 If the form is like $V.deref(A_1)$ ($A_1$ is the last node), and appears in
the select-clause, then change the path expression with a group of expressions,
"VAR.k.A_1^1, \cdots, VAR.k.A_1^n", where $A_i^1 (1 \leq i \leq n)$ are the all attributes of
$RNRT$.

4.3 If the form is like $V.deref(A_1)$, and appears in the where-clause, rewrite the
path expression into $V.A_1$

4.4 If the form is like $V.A_1$, it should only appear the where-clause. No change
need to be done.

**Case 5:** If $type(A_1)$ is a set type $set(ET)$ and $ET$ is a built-in type, there
should be no more node after it in the path expressions. Add a new variable
definition in the from-clause, "TBL.A_1.R VAR.k", where $TBL.A_1.R$ is the re-
lation that hold the set values. In the where-clause, add a new predicate, “AND (V.PK = VAR.k.PK)”, where PK is the primary key for TBL.R, on which V is defined. Change the original path expressions into “VAR.k.TBL..A1”. If the path expression appears in the select-clause, add a new part in select-clause, “VAR.k.PK”.

Case 6: If type(A1) is a set type set(ET) and ET is an ADT, add a new variable definition in the from-clause, “TBL..A1.C VAR.k”, where TBL..A1.C is the OO class of the ADT set. In the where-clause, add a new predicate, “AND (V.A1 = VAR.k.OID)”. Change the original path expressions into “VAR.k.TBL..A1,...”.

Case 7: If type(A1) is a set type set(ET) and ET is a NRT, add a new variable definition in the from-clause, “TBL..A1.R VAR.k”, where TBL..A1.R is the table of the NRT set. In the where-clause, add a new predicate, “AND (V.PK = VAR.k.PK)”, where PK is the primary key of TBL.R, on which V is defined. For each path expression in the cluster, there are following subcases regarding the modification the expression itself.

7.1 If the form is like V.A1,A2,... (there is another node after A1), change the path expression into “VAR.k.A2,...”. If the path expression appears in the select-clause, add a new part in select-clause, “VAR.k.PK”.

7.2 If the form is like V.A1 (A1 is the last node) and appears in the select-clause, then change the path expression with a group of expressions

“VAR.k.A1..A1,...,VAR.k.A1..A0” , where A’i (1 ≤ i ≤ n) are the all attributes of type(A1).

7.3 If the form is like V.A1, and appears the where-clause, change the original path expression into “VAR.k.A1..PK”, where A1..PK is the primary key of type(A1).

Case 8: If type(A1) is a set type set(ET) and ET is a reference type ref(RNRT) there should be a table R1.BL.K to hold the referenced tuples and another table TBL..A1.R to hold the set information. Add a new variable definition “TBL..A1.R VAR.k” into from-clause. In the where-clause, add a new predicate, “AND (V.PK = VAR.k.PK)”, where PK is the primary key of TBL.R, on which V is defined. For each path expression in the cluster, there are following subcases regarding the modification the expression itself.

8.1 If the form is like V.deref(A1),A2,... (there is another node after A1), add a new variable definition “RNRT..R VAR.k’” (k’ = k + 1) into from-clause. In the where-clause, add a new predicate, “AND (VAR.k.RPK = VAR.k’.RPK)” where RPK is the primary key of R1.BL.R. Change the original path expression into “VAR.k’.A2,...”.

8.2 If the form is like V.deref(A1) (A1 is the last node), and appears in the select-clause, add a new variable definition “RNRT..R VAR.k’” (k’ = k + 1) into from-clause. In the where-clause, add a new predicate, “AND (VAR.k.RPK = VAR.k’.RPK)”, where RPK is the primary key of R1.BL.R. Change the original path expression into a group of expressions, “VAR.k’.A1,...,VAR.k’.A0” where A’i (1 ≤ i ≤ n) are the all attributes of RNRT.

8.3 If the form is like V.deref(A1), and appears the where-clause, add a new variable definition “RTBL..R VAR.k’” (k’ = k + 1) into from-clause. In the
where-clause, add a new predicate, “AND (VAR.k.RPK = VAR.k'.RPK)”, where RPK is the primary key of RBL. Change the original path expression into “VAR.k'.RPK”.

8.4 If the form is like $V.A_1$, it should only appear in the where-clause. Change the original path expression into “VAR.k.A_1.RPK”, where RPK is the primary key of RBL.

**Step 4:** (Translation of where-clause) After Step 3, all the data elements in the query are either constants, in the $V.A$ form if $V$ is a relational tuple variable, or in $V.A_\ast$ form if $V$ is OO variable. This step will process the predicates in the where-clause, specifically those that have collection types involved. The following are the processes on the membership and inclusion predicates.

**Case 1:** (Membership of ADT set) The predicate should appear like $P_1 \text{ IN } P_2$ where $P_1$ and $P_2$ are path expressions. The type of $P_1$ is an ADT and that of $P_2$ is the set of that ADT. In this case, nothing need be done.

**Case 2:** (Inclusion predicate between ADT sets) The predicate should appear like $P_1 \text{ ISSUB } P_2$, where $P_1$ and $P_2$ are two path expressions. Both of their types are same, an ADT set. Change the predicate into “for all $x$ in $P_1$ : $x$ in $P_2$”.

**Case 3:** (Other membership predicate) The predicate should appear like $E_1 \text{ IN } E_2$. $E_1$ can be either (1) a constant; (2) path expression $V.A$, where $V$ is relational tuple variable and $A$ is a built-in typed attribute; or (3) $V.A_1, \ldots, A_n$ where $V$ is an OO variable and $A_n$ is a built-in typed attribute or an ADT method invocation returning a built-in typed value. $E_2$ is in the form $V.A$ where $A$ is an attribute in the relational table, say TBL.R, to hold the flattened information of the set. If $E_2$ has no other occurrence in the membership predicates of the query, simply change the operator “IN” into “=”. Otherwise, create a new variable in from-clause, “TBL.R \text{ VAR.k}”. Change the predicate into “$E_1 = \text{VAR.k.A}$”. Add a new predicate “AND (VAR.k.PK = $V'.PK$)” in the where-clause. Here $PK$ is the primary key of TBL.R.

**Case 4:** (Other inclusion predicate between base type sets) The predicate should appear like $V_1.A_1 \text{ ISSUB } V_2.A_2$, where $A_1$ and $A_2$ are attributes respectively in the relational tables, say TBL1.R and TBL2.R (not necessarily distinct), that hold the flattened information of the sets. Change the original predicate into following script.

```sql
not exists
    select *
    from TBL1.R \text{ VAR.k}
where \text{VAR.k.PK1 = V1.PK1 and not exists}
    select *
    from TBL2.R
where \text{PK2 = V2.PK2 AND VAR.k.A1 = A2}
```

**Example 6.** After substitution, the query in Example 2 is transformed into the following format.

```sql
select \text{VAR.1.name, VAR.3.name, VAR.3 intéress}
from emp_R \text{ VAR.1, dept_R VAR.2, emp_interest_R VAR.3 VAR.5, point VAR.4}
```
where \( \text{VAR}_2\text{.budget} > 1,000,000 \) and
\( \text{VAR}_1\text{.dept} = \text{VAR}_2\text{.dname} \) and
\( \text{VAR}_3\text{.name} = \text{VAR}_5\text{.name} \) and
\( \text{VAR}_b\text{.emp\_interest} = \text{"URDB\"} \) and
\( \text{VAR}_1\text{.name} = \text{VAR}_3\text{.name} \) and
\( \text{VAR}_4\text{.distance(CENTRAL POINT)} < 2 \) and
\( \text{VAR}_1\text{.location} = \text{VAR}_4\text{.mid} \)

5.2 Variable graph for decomposition

After the substitution process, the query is formulated on the local schemas. However, we need to decompose the integrated form into several parts so that each part can be executed by the relative local db engine. To decompose the query into local queries, we need to identify the boundary between the relational system and the OO system. A variable graph is used to assist the process. The major job is to find the bridge constraints. A bridge constraint is either a predicate (called bridge predicate) in the where-clause that involves both OO variables and relational tuple variables, or a method invocation (called bridge invocation) that takes relational elements as input parameters. The where-clause is split based on the definition of the variables. In general, all the predicates that have only relational variables involved are transferred into one local relational query, all the predicates that have only OO variables are translated into a set of local OO queries, all the predicates that belong to bridge constraints are translated into one top level query. The select-clause and from-clause are also split accordingly. The decomposition result includes a local relational query, a top level relational query, and a set of local OO queries.

The variable graph is an extension of the relational predicate graph in [11]. Not only predicates but also method invocations are taken into consideration. Because the methods may appear in select-clause, we draw the graph from the whole query instead of the where-clause.

**Definition 1.** For a given query \( Q \), we define its variable graph: \( VG(Q) \) as an annotated undirected graph: \( VG(Q) = (V, E) \). Each vertex \( v \) in \( V \) represents \( v \) (relational or OO) variable used in \( Q \), and each edge \( e \) between vertices \( V_1 \) and \( V_2 \) in \( E \) represents either a predicate in \( Q \) that involve \( V_1 \) and \( V_2 \), or there is a method invocation \( v_1, \ldots, \text{method()} \) that takes \( v_1, \ldots \) as an input parameter \( (i, j \in \{1, 2\} \land i \neq j) \). Each edge is annotated with the predicate or the method invocation.

Compared with predicate graph, variable graph emphasises on the relationship among variables, and does not contain all the constraints in \( Q \). Rearrange the vertices so that two disjoint circles can be drawn to enclose all the relational tuple variables and OO variables respectively. All the edges that go cross the circles’ borders are called bridge edges, which correspond to the bridge constraints. An example is shown in Figure 2, which is the variable graph of Example 6. Three nodes are in the relational side, \( \text{VAR}_1 \), \( \text{VAR}_2 \) and \( \text{VAR}_3 \). One node is in the
Fig. 2. The variable graph of example ε

OO side, VAR_4. The only bridge constraint is the predicate, VAR_1.location = VAR_4.OID.

By removing all the bridge edges, we get a disconnected graph VG', called local partition graph. All the vertices of relational variables are always in one connected component of VG'. On the other hand, those OO variables may distribute in several connected components, called connected OO components (COC). For each connected component, there should be a corresponding local query. In the example there is only one COC that contains one node, VAR_4.

5.3 The decomposition and final result processing

OR queries may have set valued attributes in the select-clause, which are flattened in relational side. Unfortunately, the top relational query engine is incapable of restoring the flat data back into the nested format. Therefore, an external procedure is needed to do the final job. We slightly extend the syntax of the top level relational query to include a special function call nest().

Input An integrated query against local schemas

Output A set of relational queries in SQL form and a set of OO queries in OQL form.

Step 1: (Query decomposition) Draw the variable graph VG(Q) of the substituted query Q. Identify the bridge constraints. Identify the connected OC components. If there is no bridge constraints, no decomposition need to be done and jump to the last step.

Step 2: (Create local relational query) Create the local relational query LRQ in the “create table TEMP_R as select … from … where …” form, where “TEMP_R” is the table to hold the partial result. In the from-clause, copy all the relational tuple variable definitions from from-clause of Q. In the select-clause, copy all the relational attributes from select-clause of Q. For each relational attribute mentioned in bridge constraints, add it into select-clause unless it is already there. In the where-clause, copy all the predicates that have no OO variable involved.
Step 3: (Create local OO queries) Initialize the local OO query counter $i$ and new attribute counter $j$ to 1. These counters work just like the variable counter $k$. For each COC (if any), create the local OO query $LOQ_i$ in the form "define TEMP.O_i as select distinct struct(...) from ... where ...". In the from-clause, copy the definitions of all OO variables in the COC. In the where-clause, copy all the predicates mentioning only the variables in the COC. In the select-clause, for each data element "V..." in the select-clause of Q or in the bridge predicates, where V is in COC, add a new element "name : V...". The naming rule is, if the data element is in "V.A" form, then use A as the name, otherwise create a new name A,j. If the element is in "V.oid" form, change it into "A\*V" which represents the identity of the object referenced by V. In the select-clause, if there is any method invocations that take an attribute from relational side as input parameter, then add "VR in TEMP.R" in from-clause and replace each occurrence "V.A" with "V.R.A" for each relational variable V.

Step 4: (Create top level query) Create the top level relational query $TRQ$. The from-clause has the format "FROM TEMP.R VR, TEMP.O_1 VO_1, ..., TEMP.O_n VO_n", supposing there are n local OO queries. In the select-clause, include all the contents in the select-clause of Q. In the where-clause, copy all the predicates acting as bridge constraints. Change all the data elements in select-clause and where-clause into right form. For each variable V, find out the connected component it belongs to in $VG'(Q)$ and then the corresponding local query. Suppose the query results in a temporary table TEMP.T, replace all occurrence of V with V.T. Any method invocation long path expression should be assigned to a new attribute in a certain local OO query, and therefore substitute the invocation or long path with that attribute name.

Step 5: (Final result processing) To restore the flattened set values, the standard SQL syntax need to be extended a little. The nest operation is expressed as "nest A_1, ..., A_n on PK", where PK is the key regarding to the operation. The result is a set, $\{A_1, ..., A_n\}$. This expression can be nested. Therefore, we can apply this operation in the select-clause of $TRQ$ when the set valued attributes need be restored.

Example 7. To continue Example 6, we have the following queries as final result:

```sql
/* Relational local query:
create table temp_R as
  select VAR_1.name as name, VAR_1.location as location,
        VAR_3.name as name_2, VAR_3.emp_interest
  from emp_R VAR_1, dept_R VAR_2, emp_interest_R VAR_3, VAR_5
  where VAR_1.dept = VAR_2.deptname and VAR_2.budget > 1,000,000 and
    VAR_1.name = VAR_3.name and VAR_3.name = VAR_4.name and
    VAR_5.emp_interest = "URDB";
/* OO local query:
create table temp_U as
  select distinct struct(oid: &VAR 4)
  from VAR_4 in point
  where VAR_4.distance(CENTRAL POINT) < 2;
```
/* Relational top query:
select VR.name, nest VR.emp_interest on VR.name
from temp_R VR, temp_O VO_O
where VR.location = VO_O.oid;

Generally speaking, an OR query can be partitioned into three parts, a relational local query, a set of OO local queries, and a relational top query. Temporary tables are created to store the results of the local queries. The top query is applied on the temporary tables to merge the partial results. The following example shows a special situation. All the information used by the query is stored in the relational side. Therefore, the OO local queries are not created, and neither is the the top query. The original query is translated into one relational query applied on the local relational database.

Example 8. The queries in Example 3 can be transformed into the following query (only local relational one):

select VAR_3.name
from dept_H VAR_1, emp_H VAR_2 VAR_3
where VAR_1.dname = VAR_2.dept and VAR_2.salary >= 100,000 and
    VAR_3.name = VAR_1.manager

6 Conclusion

The Object-relational data model opens up type system of traditional relational model, allowing more complex data structures. This requires new facilities to manage the data and handle the queries. Instead of building an object-relational DBMS from scratch, we proposed an approach to build the OR system by integrating existing relational and OO database systems. The heterogeneous database architecture which is used for this purpose has been presented in this paper. In particular, we focused on two kernel components of the architecture, schema transformer and query partitioner. Algorithms for implementing these two components have been proposed. Up to now, a prototype has been built at DSTC/UQ (CRC for Distributed Systems Technology/University of Queensland). Currently, we are improving the prototype with other functions such as query optimization, integrity control [9].

References


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