Object-Relational Query Translation on Heterogeneous Database Architecture

Hui Li[†] Chengfei Liu[‡] Maria E Orlowska[†]

[†]CRC for Distributed System Technology

Level 7, GP South Building, The University of Queensland

St. Lucia, Qld 4072, Australia

Email: {huili, maria}@dstc.edu.au

[‡] School of Computing Sciences

University of Technology, Sydney

PO Box 123, Broadway NSW 2007, Australia

Email: liu@socs.uts.edu.au

Abstract

The converging trend of relational database technology and object-oriented database technology results in object-relational (OR) database systems, which extend the relational systems with add-on object features. Efforts from both academic research and industry have been directed into object-relational database systems. In this paper, we adopt an approach for implementing object-relational database systems by using a heterogeneous database system architecture. We propose a procedure to partition a global OR query into a group of relational and object-oriented sub-queries on local databases so that it can be processed without a specialised query engine. The correctness of this query translation is also shown.

1 Introduction

In traditional relational database systems (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: extended relational database systems, object-oriented (OO) databases [Kemper and Moerkotte, 1994, Cattell, 1994], toolkits for constructing special-purpose database systems, and persistent programming language. As claimed by Carey and DeWitt [Carey and DeWitt, 1996], 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.

Currently, different approaches are being taken by vendors to provide object-relational database solutions: native implementation such as Informix's Illustra [Stonebraker et al., 1990], Fujitsu's ODB II [Ishikawa et al., 1996], 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 OR database system is a complex, time-consuming task, requiring hundreds of man years of effort. It is always psychologically and economically difficult for people to discard their investment in an old systems.

For the purpose of providing new technology without giving up old systems, we put forward a new approach [Liu et al., 1997], utilising a heterogeneous database architecture as a vehicle for OR database system implementation. A relational DBMS and an OODBMS are combined 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 [Stonebraker, 1996], our approach is more biased towards using existing resources.

This paper extends the previous work discussed in [Liu et al., 1997], which mainly deals with the shcema translation algorithm. Here we emphasis on the query translation algorithm and its theory foundation. In the rest of the paper, the system environment will be introduced in section 2. In section 3 the query partition algorithm will be described. Before showing the correctness of our query processing approach in section 5, the OR tuple calculus is introduced in section 4 as the theory foundation.

2 System Overview

2.1 Object-relational database 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 [Melton, 1995] only supports unnamed row types, ADTs and collection types. In the separate "SQL/Object" part of SQL3 [Kulkarni et al., 1995], *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 [Beech, 1997] 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, and keep a distinction between ADTs and NRTs.

The basic components of our object-relational data 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 current SQL-3 standard draft.
- *Abstract Data Types (ADTs)* are user defined and implemented types. The implementation of

objects, and their attributes and behaviour, are invisible to the query systems. 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, $t(f_1 : T_{f_1}, \cdots, f_n : T_{f_n})$, where t is the name of the ADT, $f_i (1 \leq i \leq n)$ represents a method of the ADT, T_{f_i} is the function type of f_i. A function type have 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 R is an element of the Cartesian product of the value sets of the types that define R. 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 unnamed row type.

- *reference (row) types* have the form *ref(t)*, where t is a row type. In current OR model, 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.

2.2 Query language extension

The query language we utilise here is based on SQL92, with extended features to support type extensibility in OR systems. The new query features we focus on include:

- **method invocation** – Since ADT have methods defined, method invocations are allowed to appear in the Select-clause and Where-clause. 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 an tuple variable, A_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 subset(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 definitions of one ADT* point, *two NRTs* emp_t *and* dept_t *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 (                    create NRT dept_t (
       name varchar(30),                      dname varchar(30),
       salary decimal(9,2),                   budget float,
       interest set(varchar(40)),             location point,
       location point,                        manager ref(emp_t));
       dept ref(dept_t),
       friend SET(REF(emp_t)));
create table emp of emp_t             create table dept of dept_t
       scope of dept is dept,                 scope of manager is emp;
       scope of friend is emp;
```

As we can see here, tables are created based on certain NRTs. In addition, we need to specify the scopes of reference-typed attributes in that table.

Example 2 *Find the names of all employees and their research interests who have interest in ORDB*

and work for the department which is located in central area (within 2 kilometres from the central point) and has budget more than 1 million dollars.

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

2.3 Heterogeneous Database Architecture for ORDB

As shown in figure 1, an HDB engine is built on the top of a local RDB engine and an OODB engine. This HDB engine can be used as a virtual ORDB engine. In the following section, we describe each component in the architecture. Although it looks similar to UniSQL's multi-database management system (UniSQL/M), the purpose are different. UniSQL/M is used to integrate legacy systems, however our purpose is to utilize functionalities of the existing DBMSs to implement those of ORDBMS.

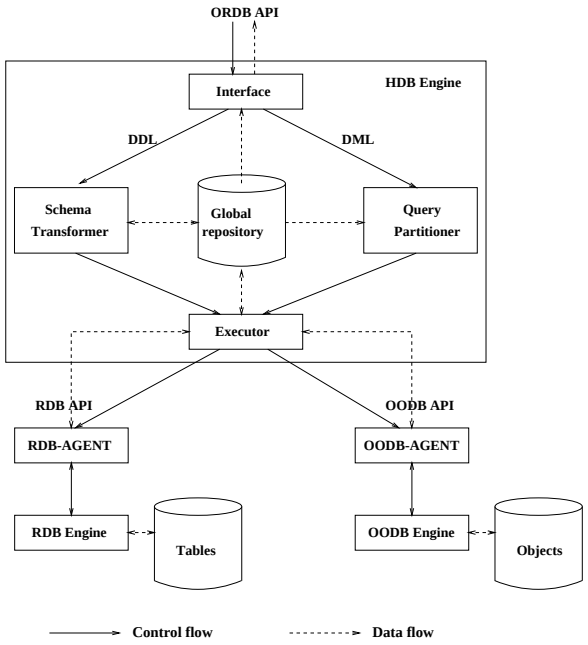

Figure 1: HDB Architecture for ORDB

- **Interface** – It receives users' OR requests, does syntactical check and hands them to cor-

responding components for processing. It is also responsible for returning the results of the requests back to users.

- **Schema Transformer** – It handles requests for schema definition.The main function of this component is to keep the information of the OR schema in the global repository, and to transform the schema into local schemas of both relational and OO systems. A transformation plan is generated by this component which is delivered to the executor for execution. The local mapping information is also kept in the global repository.

- **Query Partitioner** – This component is responsible for translating OR queries which are issued against global OR schema into local queries on both local relational and OO systems. The local mapping information is used for such transformation. All local queries form a query plan which is handed over for the executor to execute.

- **Executor** – The executor is responsible for coordinating the distributed execution of the execution plan (both transformation and query plan). There is an RDB engine employed at the global level to hold temporary results which may pass between the local relational system and the 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, simulation of an ORDB interface is necessary. Some functions are provided at this level for the final result processing, 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 OODB-agent is responsible for monitoring the execution of local queries against transformed local object-oriented schema and returning the results.

2.4 Schema transformation

Ideally, the HDB structure is transparent to users. That is, when designing an ORDB application, they can use the system just like an integrated, pure ORDBMS, not knowing how the underly-

ing RDBMS and OODBMS are combined together to implement the ORDBMS. Given an ORDB schema definition, it is the system's task to map the schema down to the RDBMS and OODBMS. Therefore, we need a process to automatically transform the OR schema into a set of relational schemes and OO schemes, while preserving the semantics. Since this part has been discussed in detail in [Liu et al., 1997], we just briefly recite the algorithm here.

`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, $\mathcal{TRAN}$ for table structures to be transformed. Initially, they are empty.

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

Step 2: For every ADT, output a definition of a *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*. The value of *oid* will be systematically assigned by the local OODBMS.

Step 3: For table named ORT defined in the OR schema, create a (frame) structure in $\mathcal{TRAN}$ to record the table name, copy the attribute definitions of the NRT in $\mathcal{NRT}$ on which it is defined as the attribute definitions of the table, and if scopes of referenced attributes are defined with the table, keep the names of scope tables within the attribute definition. A *frame table* named ORT_R will be created later in the relational schema based on this frame structure.

Step 4: Loop for each table structure in $\mathcal{TRAN}$ until no structure exists.

Step 5: For each table structure in $\mathcal{TRAN}$, let its table name TBL, create a relational table definition with table name TBL_R. For each attribute definition $A : T$ of TBL, we classify 8 cases for different processing. Remove the structure of TBL after all its attribute definitions have been processed.

Case 1: If T is a built-in type, output a column definition $A : T$ for TBL_R.

Case 2: If T is an ADT, output a column definition A : *long* for TBL_R, we use type *long* to represent object identity.

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

Case 4: If T is a reference type $ref(RNRT)$, there should be a scope clause declaring the table, say $RTBL$, it really references to. Let $RPK : T_{RPK}$ be its primary key definition, then output a column definition $A : T_{RPK}$ for TBL_R.

Case 5: If T is a set type $set(ET)$ and ET is a built-in type, output a definition for an *auxiliary table* $TBL\_A\_R(PK : T_{PK}, TBL\_A : ET)$, where $PK : T_{PK}$ is the primary key definition for TBL_R.

Case 6: If T is a set type $set(ET)$ and ET is an ADT, output an *auxiliary class* definition $TBL\_A\_C(oid : long, TBL_A : set(ET))$.

Case 7: If T is a set type $set(ET)$ and ET is a NRT, create an auxiliary structure in $\mathcal{TRAN}$ with TBL_A_R as table name and $(PK : T_{PK}, EA_1 : ET_1, \cdots, EA_m : ET_m)$ as its attribute definition, where $PK : T_{PK}$ is the primary key definition for TBL_R and $(EA_1 : ET_1, \cdots, EA_m : ET_m)$ is the definition for ET which can be retrieved from $\mathcal{NRT}$.

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 $RTBL$, it really references to. Let $PK : T_{PK}$ and $RPK : T_{RPK}$ be primary key definitions of TBL and $RTBL$, respectively, then output a definition of the auxiliary table $TBL\_A\_R(PK : T_{PK}, A\_RPK : T_{RPK})$.

3 Query Partition

Given a query against global OR schema, it must be translated to local queries against its transformed local relational schema and OO schema for execution. We design a query partition algorithm which has three major steps: substitution, decomposition and final result reconstruction. The result consists

of a group of relational queries in SQL92 [ISO, 1992, Date and Darwen, 1993] format and a group of OO queries in OQL [Cattell, 1994] 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 containing an "OR" operator, it can always be transformed into the disjunctive form, "C_1 OR C_2". The original query Q can then be translated into "Q_1 UNION Q_2", where Q_1 and Q_2 take C_1 and C_2 respectively in the where-clause.

3.1 The substitution process

To process the OR query, the first task is to transform the OR schema representations in the query into suitable local forms. The OR table and attribute names should be substituted by corresponding 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 $V.A_1.\cdots.A_n$ need be translated into $V.A$ form plus 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, where V is a relational tuple variable, and those from OO classes may have arbitrarily long paths starting from a 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 the same first attribute. For example, $V.A_1, V.A_1.A_2$ and $V.A_1.A_2'$ are in the same cluster, while $V'.A_1$ and $V.A_1'$ belong to two other clusters.

`Input:` An OR query in the $SELECT\cdots FROM\cdots WHERE\cdots$ form, and there is no "OR" operation in Where-clause.

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

Step 1: (Initialisation) 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 is renamed or a new variable is 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 whereever the name format is not concerned.

Step 3: (Translation of the path expressions) In the 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, $type(A_1)$. 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 be done. Or if V is defined on an OO class, the path expressions need not 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 A_1, then based on the previous schema transformation algorithm, A_2 should now be a attribute of TBL_R and be renamed as $A_1\_A_2$. Here TBL_R is the relation on which V is defined. Therefore, we can translate the original path expressions to "$V.A_1\_A_2.\cdots$".

3.2. If A_1 is the last attribute in the path expression, and the expression appears in the Select-clause, then replace the path expression with a group of expressions, "V.PK, $V.A_1\_A^1, \cdots, V.A_1\_A^n$", where $A^i, (1 \leq i \leq n)$ are the all attributes of $type(A_1)$ and PK is the primary key of TBL_R.

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 "$V.A_1\_PK$", where PK 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 $RTBL\_R$ to hold the referenced tuple. Add a new variable definition in the From-

clause, "$RTBL$_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 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_$k.A^1, \cdots,$ VAR_$k.A^n$", where $A^i, (1 \leq i \leq n)$ 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 relation 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_$k.TBL_A_1$". If the path expression appears in the Select-clause, add a new part in Select-clause, "VAR_$k.PK$".

Case 6: If $type(A_1)$ is a set type $set(ET)$ and ET is an ADT, add a new variable definition in the From-clause, "TBL_A_1_C VAR_k", where TBL_A_1_C is the OO class of the ADT set. In the Where-clause, add a new predicate, "AND ($V.A_1$ = VAR_k.OID)". Change the original path expressions into "VAR_$k.TBL_A_1.\cdots$".

Case 7: If $type(A_1)$ is a set type $set(ET)$ and ET is a NRT, add a new variable definition in the From-clause, "TBL_A_1_R VAR_k", where TBL_A_1_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 $V.A_1.A_2.\cdots$ (there is another node after A_1), change the path expression into "VAR_k.$A_2.\cdots$". 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 $V.A_1$ (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_$A^1, \cdots,$ VAR_k.A_1_A^n", where $A^i, (1 \leq i \leq n)$ are the all attributes of $type(A_1)$ plus the primary key of TBL_R.

7.3. If the form is like $V.A_1$, and appears in the Where-clause, change the original path expression into "VAR_k.A_1_PK", where A_1_PK is the primary key of $type(A_1)$.

Case 8: If $type(A_1)$ is a set type $set(ET)$ and ET is a reference type $ref(RNRT)$, there should be a table $RTBL\_R$ to hold the referenced tuples and another table TBL_A_1_R to hold the set information. Add a new variable definition "TBL_A_1_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 $V.deref(A_1).A_2.\cdots$ (there is another node after A_1), 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 $RTBL$_R. Change the original path expression into "VAR_k'.$A_2.\cdots$".

8.2. If the form is like $V.deref(A_1)$ (A_1 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 $RTBL$_R. Change the original path expression into a group of expressions, "VAR_k'.$A^1, \cdots,$ VAR_k'.A^n", where $A^i, (1 \leq i \leq n)$ are the all attributes of $RNRT$ plus the primary key of TBL_R.

8.3. If the form is like $V.deref(A_1)$, and appears in 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 $RTBL$_R. 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 $RTBL$_R.

Step 4: (Translation of Where-clause)

Case 1: (Membership of ADT set) The predicate appears like P_1 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 appears like P_1 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 appears like E_1 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). $V.A_1.\cdots.A_n$ 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 VAR_k". Change the predicate into "E_1 = 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 appears like $V_1.A_1$ ISSUB $V_2.A_2$, where A_1 and A_2 are attributes respectively in the relational tables, say TBL_1_R and TBL_2_R (not necessarily distinct), that hold the flattened information of the sets. Change the original predicate into following script.

```
NOT EXISTS (
    SELECT *
```

FROM TBL_1_R VAR_k

WHERE VAR_k.PK_1 = V_1.PK_1 AND NOT EXISTS (

SELECT *

FROM TBL_2_R

WHERE PK_2 = V_2.PK_2 AND VAR_k.A_1 = A_2

)

)

Example 3 *After substitution, the query in example 2 is transformed into the following format.*

```
SELECT VAR_1.name, VAR_3.name, VAR_3.interest
FROM emp_R VAR_1, dept_R VAR_2, emp_interest_R VAR_3 VAR_5, point VAR_4
WHERE VAR_2.budget > 1,000,000 AND
      VAR_1.dept = VAR_2.dname AND
      VAR_3.name = VAR_5.name  AND
      "ORDB" = VAR_5.emp_interest AND
      VAR_1.name = VAR_3.name  AND
      VAR_4.distance(CENTRAL_POINT) < 2 AND
      VAR_1.location = VAR_4.OID
```

3.2 Variable graph for decomposition

After the substitution process, the query is now over the local schemes. However, we need to decompose the integrated form into several parts so that each part can be executed by the related local db engine. 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 transfered into local relational

queries. Those which have only OO variables are in local OO queries, and those belonging to bridge constraints are in the top level query. The Select-clause and From-clause are also split accordingly. The decomposition result include a local relational query, a top level relational query, and a set of local OO queries.

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 variable graph is an extension of the relational predicate graph in [Meng et al., 1993]. Not only predicates, but also method invocations are taken into consideration. Because the methods may appear in the 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 a (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 $v_i.\cdots.method()$ that takes $v_j.\cdots$ as an input parameter $(i, j \in \{1, 2\} \wedge i \neq j)$. 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 corresponds to the bridge constraints. An example is shown in figure 2, which is the variable graph of example 3. Four nodes are in the relational side, VAR_1, VAR_2, VAR_3 and VAR_5. One node is in the 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.* The relational variables and OO variables may distribute in several connected components, called *connected relational components (CRCs)* and *connected OO components (COCs)* respectively. For each connected component, there should be a corresponding local query. In the example there is only CRC that contains four nodes, VAR_1, , VAR_2, VAR_3 and VAR_5, and one COC that containts one node, VAR_4.

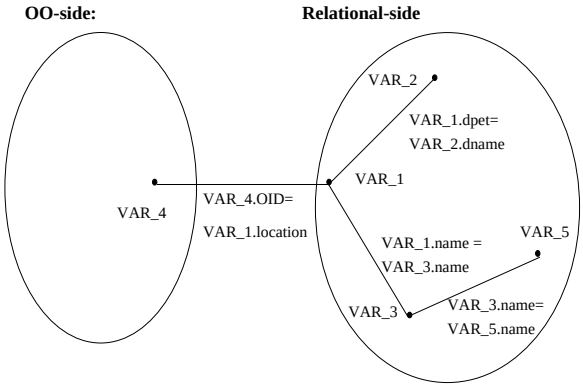

Figure 2: The variable graph of example 3

3.3 The decomposition and final process

OR queries may have multi-valued attributes in the Select-clause, which are flattened on 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 need to do the final job. We have slightly extended the syntax of top level relational query to include a special function call nest().

`Input:` An integrated query against local schemes

`Output:` A set of relational queries in SQL92 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 relational and OO components. If there is no bridge constraints, no decomposition need be done and jump to the last step.

Step 2: (Create local relational queries) Initialize the local relational query counter ri to 1. The counter works just like the variable counter k. For each CRC, create the local relational query LRQ_{ri} in the form "CREATE TABLE TEMP_R_ri AS SELECT $\cdots$ FROM $\cdots$ WHERE $\cdots$". In the From-clause, copy the definitions of all relational variables in the CRC. In the Where-clause, copy all the predicates mentioning only the variables in the CRC. In the Select-clause, copy each data element "$V.A$" which is in the Select-clause of Q or in the bridge predicates and V is in the CRC.

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_OO_i as select distinct struct($\cdots$) from $\cdots$ where $\cdots$". 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 "$V.\cdots$" which is in the Select-clause of Q or in the bridge predicates, and V is in COC, add a new element "$name : V.\cdots$". The naming rule is, if the data element is in "$V.A$" form, then use A as the name, otherwise create a new name ATTR_j. If the element is in "V.OID" form, change it into "&V" which represents the identity of the object referenced by V.

In the Select-clause and Where-clause, if there are any method invocations that take an attribute from relational side as input parameter, then add "VR_i in TEMP_R_i" in From-clause and replace each occurrence "$V.A$" with "$VR\_i.A$" for each relational variable V. Add "$VR\_i.A$" into the Select clause.

Step 4: (Create top level query) Create the top level relational query TRQ. The From-clause has the format "FROM TEMP_R VR_1, $\cdots$, TEMP_R_m, TEMP_OO_1 VOO_1, $\cdots$, TEMP_OO_n VOO_n", supposing there are m local relational queries and 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 CRC or COC 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 VT. Any method invocation or long path expression should be assigned to a new attribute in Select-clause of a certain local OO query, and therefore substitute the invocation or long path with that attribute name. If there is a bridge invocation connection between VR_i and VOO_j, add a new predicate, "$VR\_i.A = VOO\_j.A$", into the Where-clause for each $VR\_i.A$ used as input parameter in the j-th local OO query.

Step 5: (Final result reconstruction) To restore the flattened values, the standard SQL syntax need be extended a little. The nest operation is expressed as "NEST $A_1, \cdots, A_n$ ON PK", where PK is the key regarding to the operation. The result is a set of row type values, $\{(A_1, \cdots, A_n)\}$. 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 4 *To continue example 3, we have the following queries as final result.*

```
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.dname AND VAR_2.budget > 1,000,000 AND

              VAR_1.name = VAR_3.name AND VAR_3.name = VAR_4.name AND

              "ORDB" = VAR_5.emp_interest;


OO local query:

create table temp_OO 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_2

FROM temp_R VR, temp_OO VOO_1

WHERE VR.location = VOO_1.oid;
```

4 Extended Tuple Calculus for ORDB

Before showing the correctness of our query partition process, we need a query system to model the ORDB queries.

In [Li et al., 1997], a tuple calculus and a query algebra is proposed for object-relational data models. They are supersets of their relational counterparts, with new features to handle complex

data. The equivalence in expressive power for the calculus and algebra is also proved. Like other declarative query languages, our SQL-like OR language is based on the OR calculus. That is, we can easily translate an OR query in to an OR calculus expression.

The OR calculus adopts a 3-tiered structure. All the basic data elements are *terms*. An *atom* can be either a tuple variable declaration or a θ-comparison between two terms. A *formula* is a logic expression which takes atoms as operands. Formally, we have the following defintion.

Definition 2 A term in the tuple calculus expression can be either a constant, a variable, or a path expression.

The following definitions of atom and formula are similar to those of RTC [Maier, 1983].

Definition 3 An atom is the basic logical building block of formulas. It can be defined in the following way,

- For any OR table r in $\mathcal{D}$, and for any tuple variable x, $r(x)$ is an atom, standing for $x \in r$;
- For any comparator $\theta \in \Theta$, any terms s and t (not necessarily distinct) that are θ-comparable, $s\,\theta\,t$ is an atom.

Definition 4 The formula in tuple calculus can be either an atom, or one or more atoms connected by logical operators.

Definition 5 An OR tuple calculus $\mathcal{ORTC}$ is a penta-tuple $(\mathbf{U}, \mathcal{T}, typing, \mathcal{D}, \Theta)$ where $\mathbf{U}$ is the universe of attributes, $\mathcal{T}$ is the set of types, *typing* is a mapping from $\mathbf{U}$ to $\mathcal{T}$, $\mathcal{D}$ is a database on the row types in $\mathcal{T}$, and Θ is a set of comparators that includes at least equality and inequality for for every type in $\mathcal{T}$.

5 Correctness of the Query Partitioning

THEOREM 1 *The partitioned queries can be interpreted by a single $\mathcal{ORTC}$ expression E_{res}.*

PROOF. After substituting the path expressions of the global OR schema in the query, we get an integrated query Q against the local schema. The corresponding $\mathcal{ORTC}$ expression is,

$$\{V(A_1,\cdots,A_m)|\exists V_1,\cdots V_t(V_1 \in T_1 \wedge \cdots \wedge V_t \in T_t \wedge V.A_1 = V^1.A^1 \\ \wedge \cdots \wedge V.A_m = V^m.A^m \wedge P_1 \wedge \cdots \wedge P_n)\} \quad (1)$$

Here, we have $V^i \in \{V_1,\cdots,V_t\}, (i \leq i \leq m)$ and $T_j (1 \leq j \leq t)$ is either a local table or class.

The decomposition process divides Q into a top level query QT, a set of local relational queries $QLR_1,\cdots,QLR_m$, and a set of local OO queries $QLO_1,\cdots,QLO_n$. Because $\mathcal{ORTC}$ is extended from relational tuple calculus, any valid relational tuple calculus expression is also a valid $\mathcal{ORTC}$ expression. Besides, each local OO query simply takes part of Q and rewrites it to meet the syntax requirement of the OODB engine. Therefore, the semantics of the result queries can always be interpreted by a set of $\mathcal{ORTC}$ expressions as follows,

$$QT : RES = \{V(A_1,\cdots,A_k)|\exists VR_1,\cdots,VR_m,VO_1,\cdots,VO_n(VR_1 \in temp\_R_1 \\ \wedge \cdots \wedge VR_m \in temp\_R_m \wedge VO_1 \in temp\_O_1 \wedge \cdots \wedge VO_n \in temp\_O_n \\ \wedge V.A_1 = V^{T1}.A^{T1} \wedge \cdots \wedge V.A_k = V^{Tk}.A^{Tk} \wedge p_{T1} \wedge \cdots \wedge p_{Ts})\}, \\ (V^{Ti} \in \{VR_1,\cdots,VR_m,VO_1,\cdots VO_n\}, 1 \leq i \leq k) \quad (2)$$

$$QLR_1 : temp\_R_1 = \{V(A_{R_1 1},\cdots,A_{R_1 k})|\exists V_{R_1 1},\cdots,V_{R_1 t}(V_{R_1 1} \in T_{R_1 1} \wedge \cdots \wedge V_{R_1 t} \in T_{R_1 t} \\ \wedge V.A_{R_1 1} = V^{R_1 1}.A^{R_1 1} \wedge \cdots \wedge V.A_{R_1 k} = V^{R_1 k}.A^{R_1 k} \wedge p_{R_1 1} \wedge \cdots \wedge p_{R_1 s})\}$$

$$\cdots$$

$$QLR_m : temp\_R_m = \{V(A_{R_m 1},\cdots,A_{R_m k})|\exists V_{R_m 1},\cdots,V_{R_m t}(V_{R_m 1} \in T_{R_m 1} \wedge \cdots \\ \wedge V_{R_m t} \in T_{R_m t} \wedge V.A_{R_m 1} = V^{R_m 1}.A^{R_m 1} \wedge \cdots \wedge V.A_{R_m k} = V^{R_m k}.A^{R_m k} \\ \wedge p_{R_m 1} \wedge \cdots \wedge p_{R_m s})\}$$

$$QLO_1 : temp\_O_1 = \{V(A_{O_1 1}, \cdots, A_{O_1 k}) | \exists V_{O_1 1}, \cdots, V_{O_1 t}(V_{O_1 1} \in T_{O_1 1} \wedge \cdots \wedge V_{O_1 t} \in T_{O_1 t}$$
$$\wedge V.A_{O_1 1} = V^{O_1 1}.\cdots.A^{O_1 1} \wedge \cdots \wedge V.A_{O_1 k} = V^{O_1 k}.\cdots.A^{O_1 m} \wedge p_{O_1 1} \wedge \cdots \wedge p_{O_1 s})\}$$

$$\cdots$$

$$QLO_n : temp\_O_k = \{V(A_{O_n 1}, \cdots, A_{O_n k}) | \exists V_{O_n 1}, \cdots, V_{O_n t}(V_{O_n 1} \in T_{O_n 1} \wedge \cdots$$
$$\wedge V_{O_n t} \in T_{O_n t} \wedge V.A_{O_n 1} = V^{O_n 1}.\cdots.A^{O_n 1} \wedge \cdots \wedge V.A_{O_n k} = V^{O_n k}.\cdots.A^{O_n k}$$
$$\wedge p_{O_n 1} \wedge \cdots \wedge p_{O_n s})\}$$

Please note that $A_{O_i j}$ may be either an attribute or a method. We do not distinguish them here because they will be handled in the same way.

Comparing the expressions of Q and QT, we can see that they have the same result type, which is the row type $(A_1, \cdots, A_k)$. To prove their equivalence, we need to further show their constraints are the same. A constructive approach is adopted here.

First, we substitute all the occurrence of variables defined on temporary tables, say $temp\_R_i$ and $temp\_O_j, (1 \leq i \leq m, 1 \leq j \leq n)$, with those on local tables/classes. Based on [Quine, 1969], any predicate like $y \in \{x | F(x)\}$ can be substituted with $F(y)$. Therefore, we can substitute the predicate $VO_i \in temp\_O_i, (1 \leq i \leq n)$ with

$$\exists V_{O_i 1}, \cdots, V_{O_i t}(V_{O_i 1} \in T_{O_i 1} \wedge \cdots \wedge V_{O_i t} \in T_{O_i t} \wedge VO_i.A_{O_i 1} = V^{O_i 1}.\cdots.A^{O_i 1}$$
$$\wedge \cdots \wedge VO_i.A_{O_i k} = V^{O_i 1}.\cdots.A^{O_i k} \wedge p_{O_i 1} \wedge \cdots \wedge p_{O_i s})\}$$

Because of $VO_i.A_j = V^{O_i j}.\cdots.A^{O_i j}, (1 \leq j \leq k)$, we can always replace the occurrence of $VO_i.A_{O_i j}$ in every predicate $p_{Th}, (1 \leq h \leq s)$ and $V.A_{O_g} = VO_i.A_{O_i j}, (1 \leq g \leq k)$ with $V^{O_i j}.\cdots.A^{O_i j}$. After the replacement, the tuple variable VO_i only appears in predicates like $VO_i.A_{O_i j} = V^{O_i j}.\cdots.A^{O_i j}$.

Further, because $dom(VO_i.A_{O_i j}) = dom(type(A_{O_i j})) \supseteq dom(V^{O_i j}.\cdots.A^{O_i j})$, the predicates

like $\exists VO_i(VO_i.A_{O_ij} = V^{O_ij}.\cdots.A^{O_ij})$ is always true. Therefore we can remove all of this kind of predicates and the tuple variable VO_i.

Given a local OO query QLO_i, there may exist a method invocation, $V_{O_ik}.\cdots.fun()$, that takes an attribute in the result of a local relational query, say QLR_j, as its input parameter. After the above substitution the expression of top level query will contain more than one variables defined on temporary table $temp\_R\_j$, which represents the result of QLR_j. However, these variables can always be combined together. From Step 4 of the decomposition and final process we can see that in QT there is a predicate "$VR_j.A = VO_i.A$" for each bridge invocation. After substituting VO_i, the expression of QT now has the following format,

$$\{V(A_1,\cdots,A_k)|\exists VR_1,\cdots,VR_j,VR'_j,\cdots(VR_j \in temp\_R_j \wedge VR'_j \in temp\_R_j \wedge \cdots$$
$$\wedge VR_j.A = VR'_j.A \wedge \cdots \wedge V_{O_ik}.\cdots.fun(\cdots,VR'_j.A,\cdots) \wedge \cdots)\}$$

Therefore we can always substitute $VR'_j.A$ by $VR_j.A$, and then remove the predicate $VR_j.A = VR'_j.A$ and the definition of VR'_j without affecting the value of the expression. After the combination process, there is one and only one variable defined on each QLR result table.

Similar to the treatment on the variables defined on $temp\_O_i$, we can always substitute the variables defined on $temp\_R_j$ with corresponding local relational variables. After the whole replacement process, the expression of top level query is expanded into the following form.

$$\{V(A_1,\cdots,A_m)|\exists V'_1,\cdots,V'_l(V'_1 \in LTB_1 \wedge \cdots \wedge V'_l \in LTB_l \wedge V.A_1 = V'^1.A'^1 \wedge$$
$$\cdots \wedge V.A_m = V'^m.A'^m \wedge p'_{T1} \wedge \cdots \wedge p'_{Tn} \wedge p_{R1} \wedge \cdots p_{Rn} \wedge p_{O_i1} \wedge \cdots \wedge p_{O_kn})\},$$
$$(V'^i \in \{V_1,\cdots V_l\}) \qquad (3)$$

Given a variable V_i in Q, its definition in (1) is like $\exists V_1,\cdots,V_i,\cdots(V_1 \in LTB_1 \wedge \cdots \wedge V_i \in LTB_i \wedge \cdots)$, where LTB_i is the name of a local table/class. After decomposition, it participates in one and only one decomposed local query. In the expression of that local query, there is $\exists V'_i \cdots (V'_i \in LTB_i \cdots)$. For each result of local OO query, there is one and only one predicate of the form $VO_j \in$

temp_O_j in (2), which is to be expanded. For each result of local relational query, as shown before, we can always combine $VR_j \in temp\_R\_j$ and $VR'_j \in temp\_R\_j$ into one definition and therefore the expression of QLR_j needs only to be expanded once in the top level query expression. Therefore in (3) there is one and only one variable definition, $\exists V'_i \cdots (V'_i \in LTB_i \cdots)$, which corresponds to V_i. For a different variable V_k in Q, there exists a different variable definition V'_j in local query, and thus is still a different one in (3). Further, there is no new variables created in local queries, therefore in (3) there is no variable which has no correspondence in (1). That is, there exists one to one correspondence between the variables in (1) and those in (3).

For the same reason, there also exists one to one correspondence between the predicates in (1) and those in (3). For each ordinary predicate $P_i(V^1.\cdots.A^1, V^n.\cdots.A^n)$, where $V^1 \in LTB^1 \wedge \cdots \wedge V^n \in LTB^n$, it takes part in a local query which has the form

$$\{V(A_1, \cdots, A_m) | \exists V'_1, \cdots, V'_n (V'^1 \in LTB^1 \wedge \cdots \wedge V'^n \in LTB^n \wedge \cdots \\ \wedge P_i(V'^1.\cdots.A^1, V'^n.\cdots.A^n) \cdots\}$$

and V'^j corresponds to V^j in (1). After substituting the corresponding temporary table with the expression, P_i and all the related variables are copied into the top query expression without any change. Therefore we have

$$P_i(V'^1.\cdots.A^1, V'^n.\cdots.A^n) = P_i(V^1.\cdots.A^1, V^n.\cdots.A^n)$$

If P_i is a bridge constraint, in (1) it appears like$V^1.A^1\,\theta\,V^2.\cdots.A^2$, where V^1 a relational variable and V^2 is an OO variable. After decomposition, the predicate is in the top query and has the form

$$VR^1 \in temp\_R^1 \wedge VO^2 \in temp\_O^2 \wedge \cdots \wedge VR^1.A^1\,\theta\,V^2.A^2 \cdots$$

and the corresponding temporary table definitions are,

$$temp_R^1 = \{V(A^1, \cdots) | \exists V'^1, \cdots (V'^1 \in TBL_R^1 \wedge \cdots \wedge V'^1.A^1\,\theta\,V.A^1 \cdots)\}$$

$$temp_O^2 = \{V(A^2, \cdots) | \exists V'^2, \cdots (V'^2 \in TBL_O^2 \wedge \cdots \wedge V'^2. \cdots .A^2 \, \theta \, V.A^2 \cdots)\}$$

After substitution, there exist the following definitions in (3),

$$\{V(A_1, \cdots, A_n) | \exists V'^1, \exists V'^2, \cdots (V'^1 \in TBL_R^1 \wedge V'^2 \in TBL_O^2 \wedge \cdots \wedge V'^1.A^1 \, \theta \, V'^2.A^2 \cdots)\}$$

Because of the correspondence between V^1 and V'^1, and V^2 and V'^2, we have

$$(V^1.A^1 \, \theta \, V^2. \cdots .A^2) \equiv (V'^1.A^1 \, \theta \, V'^2.A^2)$$

Similarly, if P_i contains a bridge invocation, we also have

$$P_i(V'^1. \cdots .A^1, V'^n. \cdots .A^n) \equiv P_i(V^1. \cdots .A^1, V^n. \cdots .A^n)$$

where the left side in the equation is from (3) and right side is from (1).

Here we can see that we can get (3) by equivalent transformation from (2), and the constraints of (3) is equivalent to that of (1). Therefore, the constraints of (2) is also equivalent to that of (1). That is, the execution of decomposed queries is equivalent to the execution of Q, and we can use (1) to represent the decomposed queries.

The result reconstruction process is to add *nest* operations on the result of top level query. Because of the equivalence between OR algebra and $\mathcal{ORTC}$ [Li et al., 1997], there always exists a equivalent $\mathcal{ORTC}$ expression $\{x(\cdots) | \exists y(y \in RES \wedge \cdots)\}$. When we substitute $y \in RES$ with (1), we get a integrated calculus expression which represents the result of the partitioned queries. □

THEOREM 2 *Any query over an OR schema is equivalent to its substituted queries over the corresponding transformed local OO and relational schemes.*

By *equivalent* we mean that the two queries always produce same result regardless the population of tables being queried.

PROOF. We have proven that the integrated query over the transformed local OO and relational

schemes is equivalent to the partitioned query. Here we need to show that the query over OR schema is equivalent to the one over local schema. The main point is to show how the equivalence is preserved through the path expression substitution process.

Given a OR query QOR, its calculus expression and that of the substituted query over local schema can be written as,

$$\{V(A_1, \cdots, A_m) | \exists V_1(TBL_1), \cdots, V_n(TBL_n)(P(V, V_1, \cdots, V_n)\}, \tag{4}$$

$$\text{and } \{V(A_1, \cdots, A_m) | \exists V_1'(LTB_1), \cdots, V_s'(LTB_s)(P'(V, V_1', \cdots, V_s')\} \tag{5}$$

Here, $TBL_i, (1 \leq i \leq n)$ and $LTB_i, (1 \leq i \leq s)$ represents OR tables and local tables/classes respectively.

To prove the equivalence between the two expressions, we need to show that for each interpretation $I(P(T/V, T_1/V_1, \cdots, T_n/V_n))$, there exists one and only one interpretation $I(P'(T/V, T_1'/V_1', \cdots, T_s'/V_s'))$ so that $I(P) = I(P')$ is always true regardless the format of P.

We prove the theorem by induction on the number of variables excluding V in the expression (4).

Basis No variable. P has the form $V.A_1 = C_1 \wedge \cdots \wedge V.A_m = C_m$, where $C_i, 1 \leq i \leq m$ are constants. In this case, the algorithm makes no modification on the query, and thus P and P' are exactly the same. Therefore $I(P) = I(P')$ always holds, that is (4) and (5) are equivalent.

Induction Assume the theorem holds for any calculus expressions with fewer than k variables. Let EOR have k variables. We substitute the variables $V_2, \cdots, V_k$ with corresponding variables defined on local tables/classes, and get the following expression.

$$\{V(A_1, \cdots, A_m) | \exists V_1(TBL_1), V_1'(LTB_1), \cdots, V_t'(LTB_t)(P_0(V, V_1, V_1', \cdots, V_t')\} \tag{6}$$

(6) is equivalent to (4) because for each tuple $T_1 \in TBL_1$, $P(V, T_1/V_1, V_2, \cdots, V_n)$ has $k-1$

variables, and its substituted formula is $P_0(V, T_1/V_1, V_1', \cdots, V_t')$. Based on the assumption that the theorem holds for $k-1$ variables, there exists exactly one interpretation $I(P_0(T/V, T_1/V_1, T_1'/V_1', \cdots, T_t'/V_t'))$ that is equal to each interpretation of P, $I(P(T/V, T_1/V_1, \cdots, T_n/V_n))$. Therefore $I(P) = I(P_0)$ always holds. Now we have to concentrate on how the substitution of path expressions like $V_1.A.\cdots$ will affect the equivalence by comparing the interpretations of P' and P_0. We categorise these path expressions into eight cases and study the cases respectively.

CASE 1 There is a path expression having the form $V_1.A$, and the type of A is a built-in type. The definition of $V_1(TBL_1)$, is substituted into $V_k'(TBL_1\_R)$. From the schema transformation algorithm we can see that, for the OR table TBL_1, there is a relational table TBL_1_R that holds all the built-in typed attributes of TBL_1. For each tuple T_1 of TBL_1, there exists a tuple T_k' in TBL_1_R that for each built-in attribute A, $T_1.A = T_k'\_R.A$. Therefore for each interpretation $I(P_0(T/V, T_1/V_1, T_1'/V_1', \cdots, T_s'/V_s'))$, there always exactly exactly one interpretation $I(P'(T/V, \cdots, T_k'/V_k', \cdots, T_t'/V_t'))$ so that $T_1.A = T_k'.A$.

CASE 2 There is a path expression having the form $V_1.A.\cdots$, and the type of A is an ADT. The instances of that ADT is stored in a class $ADTC$, and the column A of TBL_1_R stores the OIDs of the ADT instances. After substitution, a new variable $\exists V_c'(ADTC)$ is introduced, the path expression is substituted with $V_c.\cdots$, and an additional predicate $\wedge(V_c'.OID = V_k'.A)$ is included in P'. Therefore, for each tuple T_1 of TBL_1, there exists exactly one instance T_c' in $ADTC$ corresponding to $T_1.A$, and its OID equals $T_k'.A$. Therefore for each interpretation $I(P_0(T/V, T_1/V_1, T_1'/V_1', \cdots, T_t'/V_t'))$, there exists exactly one interpretation $I(P'(T/V, T_1'/V_1', \cdots, T_c'/V_c', \cdots, T_k'/V_k', \cdots, T_s'/V_s'))$ so that both $T_1.A.\cdots = T_c'.\cdots$ and $T_c'.OID = T_k'.A$ are true.

CASE 3 There is a path expression having the form $V_1.A.\cdots$, and the type of A is a NRT. In the schema transformation process, all the attributes of $type(A)$ are expanded in TBL_1 and thus there is a set of virtual attributes in TBL_1, $A\_A_1', \cdots, A\_A_k'$, where $A_1', \cdots, A_k'$ are attributes of $type(A)$. Therefore, for each tuple T_1 of TBL_1, there is $TBL_1.A\_A_i' = TBL_1.A.A_i', 1 \leq i \leq k$. Three subcases are considered as following,

If the path expression is like $V_1.A.A'_i.\cdots$, the algorithm rewrites it as $V_1.A\_A'_i.\cdots$.

If the path expression is like $V_1.A$ and is used in the predicate like $V.A_i = V_1.A$, it is changed into a set of predicates, $V.A_i.A'_1 = V_1.A\_A'_1 \wedge \cdots \wedge V.A_i.A'_1 = V_1.A\_A'_n$. For each $I(P_0(T/V, T_1/V_1, T'_1/V'_1, \cdots, T'_s/V'_s))$ that makes $T.A_i = T_1.A$ true, the formula $T.A_i.A'_1 = T_1.A\_A'_1 \wedge \cdots \wedge T.A_i.A'_1 = T_1.A\_A'_n$ is also true, and vice versa.

If the path expression is like $V_1.A$ and is used in other kind of predicates like $S\,\theta\,V_1.A$, the predicates are changed into $S.PK\,\theta\,V_1.A\_PK$. Here, PK is the primary key of $type(A)$, S is either a constant or a path expression. If S is a constant, $S.PK$, denotes the component that corresponds to the attribute PK; if S is a path expression, $S.PK$ is also a path expression. Because the primary key always identifies a tuple, there is one to one correspondence between $V_1.A$ and $V_1.A\_PK$. Therefore, the transformed predicates are always equivalent to their original ones.

After the substitution, the newly created path expressions are in the form $V_1.A\_A'_i.\cdots$. They are further processed based on the type of $A\_A'_i$, just like normal path expressions of V_1. The final result $V'.A'.\cdots$ in P' is equivalent to $V_1.A\_A'_i.\cdots$ as long as each further iteration, which applies one of the eight cases, keeps the equivalence.

CASE 4 There is a path expression having the form $V_1.A.\cdots$ or $V_1.deref(A).\cdots$, and the type of A is a reference type. Assume the referenced row type instances are stored in TBL'_r. For each tuple T_1 of TBL_1, there exists a tuple T'_k in TBL_1_R such that $T'_k.A$ stores the value of the primary key of a tuple T'_r in TBL'_r, to which $T_1.A$ refers. Three subcases are considered as following,

If the path expression is like $V_1.deref(A).A'_i.\cdots$, a new variable $V'_r(TBL'_r)$ is introduced, the path expression is changed into $V'_r.A'_i.\cdots$, and a new predicate $V'_k.A = V'_r.PK$ is included. For each $I(P_0(T/V.T_1/V_1, T'_1/V'_1, \cdots, T'_s/V'_s))$ there exists one and only one tuple T'_r in TBL'_r and T'_k in TBL_1_R, such that $T_1.deref(A).A'.\cdots = T'_r.A'_i.\cdots$ and $T'_k.A = T'_r.PK$.

If the path expression is like $V_1.deref(A)$ and is used in the predicate like $V.A_i = V_1.deref(A)$, it is changed into a set of predicates $V.A_i.A'_1 = V'_r.A'_1 \wedge \cdots \wedge V.A_i.A'_1 = V'_r.A'_n \wedge V'_k.A = V'_r.PK$. For each $I(P_0(T/V.T_1/V_1, T'_1/V'_1, \cdots, T'_s/V'_s))$ there exists one and only one tuple T' in TBL'_r and T'_k in TBL_1_R, such that $T'_k.A = T'_r.PK$. If $T.A_i = T_1.deref(A)$, then the formula $T.A_i.A'_1 =$

$T'_r.A'_1 \wedge \cdots \wedge T.A_i.A'_1 = T'_r.A'_n$ is also true, and vice versa.

If the path expression is like $V_1.deref(A)$ or $V_1.A$, it participates in the predicates like $S\,\theta\,V_1.deref(A)$ or $S\,\theta\,V_1.A$. The predicates can be rewritten to $S.PK\,\theta\,V'_k.PK$ or $deref(S).PK\,\theta\,V'_k.PK$ correspondingly. Here, PK is the primary key of $type(A)$, and S may either be a constant or a path expression. If S is a constant $S.PK$ denotes the component that represents the value of attribute PK in S. If S is a path expression with the form $V_S.A_1.\cdots.A_i$, $S.A_j$ denotes a path expression $V_S.A_1.\cdots.A_i.A_j$, and $deref(S).A_j$ denotes a path expression $V_S.A_1.\cdots.deref(A_i).A_j$. Because primary keys uniquely identify tuples in a table, the rewritten predicate is true as long as the original one is true.

After the substitution, some newly created path expressions are in the form $V'.A'.\cdots$. They need further process based the type of A'. The final path expression $V'.A'.\cdots$ in P' is equivalent to $V'_r.A'_i.\cdots$ as long as each further iteration, which applies one of the eight cases, keeps the equivalence.

CASE 5 There is a path expression having the form $V_1.A$, and the type of A is a set of built-in type. The values of the attribute A is now stored in a relational table $TBL_1\_A\_R$. For each tuple T_1 of TBL_1, there is one and only one tuple T'_A in $TBL_1\_A\_R$ that corresponds to each value in $T_1.A$ so that $T'_A.A \in T_1.A$ and $T_1.PK = T'_A.PK$. In P, $V_1.A$ participates in three kinds of predicates, $S \in V_1.A$, $S \subseteq V_1.A$ or $V_1.A \subseteq S$, and $S = V_1.A$, where S is either a constant or path expression.

For $S \in V_1.A$, the algorithm changes the predicate into $\exists V'_A(TBL_1\_A\_R)(S = V'_A.A \wedge V'_k.PK = V'_A.PK)$. Therefore for each interpretation $I(P_0(T/V, T_1/V_1, T'_1/V'_1, \cdots, T'_t/V'_t))$, there exists exactly one interpretation $I(P'(T/V, T'_1/V'_1, \cdots, \cdots, T'_k/V'_k, \cdots, T'_s/V'_s))$ that corresponds to it. If $I(S) \in T_1.A$ is true in $I(P_0)$, then there exists one tuple $T'_A \in TBL_1\_A\_R$ so that $T'_k.A = I(S)$ and $T'_A.PK = T'_k.PK$. Otherwise, there is no such T'_A that makes both $T'_k.A = I(S)$ and $T'_A.PK = T'_k.PK$ true.

For $V_1.A \subseteq S$, the predicate is changed into $\neg\exists V'_A(TBL_1\_A\_R)(V'_k.PK = V'_A.PK \wedge V'_A.A \notin S$. For $S \subseteq V_1.A$, it is changed into $S \subseteq \{V(A) | \exists V'_A(TBL_1\_A\_R)(V.A = V'_A.A \wedge V'_k.PK = V'_A.PK)\}$. Clearly for each interpretation $I(P_0(T/V.T_1/V_1, T'_1/V'_1, \cdots, T'_s/V'_s))$, there exists exactly one interpretation $I(P'(T/V, T'_1/V'_1, \cdots, \cdots, T'_k/V'_k, \cdots, T'_s/V'_s))$ that corresponds to it. If the inclusion predicate is true for $I(P_0)$, the transformed predicate is also true in $I(P')$, and vice versa.

For $S = V_1.A$, it is changed into $S = \{V(A) | \exists V'_A(TBL_1\_A\_R)(V.A = V'_A.A \wedge V'_k.PK = V'_A.PK)\}$.

Like last subcase, we can show the transformed predicate is equivalent to the original one.

CASE 6 There is a path expression having the form $V_1.A.\cdots$, and the type of A is set of an ADT. The instances of that ADT set is stored in a class $TBL_1\_A\_C$, and the column A of TBL_1_R stores the OIDs of the ADT set instances. After substitution, a new variable $\exists V_c'(TBL_1\_A\_C)$ is introduced, the path expression is substituted with $V_c.\cdots$, and an additional predicate $\wedge(V_c'.OID = V_k'.A)$ is included in P'. Therefore, for each tuple T_1 of TBL_1, there exists exactly one instance T_c' in $TBL_1\_A\_C$ corresponding to $T_1.A$, and its OID equals $T_k'.A$. Therefore for each interpretation $I(P_0(T/V, T_1/V_1, T_1'/V_1', \cdots, T_t'/V_t'))$, there exists exactly one interpretation $I(P'(T/V, T_1'/V_1', \cdots, T_c'/V_c', \cdots, T_k'/V_k', \cdots, T_s'/V_s'))$ so that both $T_1.A.\cdots = T_c'.\cdots$ and $T_c'.OID = T_k'.A$ are true.

CASE 7 There is a path expression having the form $V_1.A.\cdots$, and the type of A is set of a NRT. The schema transformation process outputs an intermediate OR table TBL_A. The columns in TBL_A include all the attributes of $type(A)$ and the primary key of $type(TBL_1)$, say PK. Therefore, for each tuple T_1 of TBL_1, there exists a set of tuples in TBL_A such that for each tuple in the set, we have $T_{Ai}.PK = T_1.PK$. Further, there exists one to one correspondence between values in $T_1.A$ and the tuples in the set, such that $I.A_1' = T_{Ai}.A_1' \wedge \cdots \wedge I.A_k' = T_{Ai}.A_k'$, where $I \in T_1.A$, T_{Ai} is a tuple of TBL_A, and $A_1', \cdots, A_k'$ are all the attributes of $type(A)$. Two subcases are considered here. Like CASE 5, the path expression participates in three kinds of predicates, membership, inclusion, and equivalence.

If the path expression is like $V_1.A.A_i'.\cdots$, the following transformations are applied based on the predicate type it involves in. For $S \in V_1.A.A_i'.\cdots$, it is changed into $\exists V_A'(TBL_A)(V_A'.PK = V_k'.PK \wedge V_R'.A_i'.\cdots = S)$. For $S \subseteq V_1.A.A_i'.\cdots$ or $V_1.A.A_i'.\cdots \subseteq S$, it is changed into $S \subseteq \{V(TBL_A.A_i')|V.PK = V_k'.PK\}$ or $\neg\exists V_A'(TBL_A)(V_A'.PK = V_k'.PK \wedge V_A'.A_i'.\cdots \notin S)$ correspondingly. For $S = V_1.A.A_i.\cdots$, it is changed into $S = \{V(TBL_A.A_i'.\cdots)|V.PK = V_k'.PK\}$. Here, if S is a path expression, then $S.A_i'$ is also a path expression; if S is a constant, then $S.A_i'$ denotes the components in S representing the values of attribute A_i'.

If the path expression is like $V_1.A$, the following transformations are applied on the predicate type it involves in. For $S \in V_1.A$, it is changed into $\exists V_A'(TBL_A)(V_A'.PK = V_k'.PK \wedge V_R'.A_1' = S.A_1' \wedge$

$\cdots \wedge V'_R.A'_k = S.A'_k$), where $A'_1, \cdots, A'_k$ are all the attributes of $type(A)$. For $S \subseteq V_1.A$ or $V_1.A \subseteq S$, it is changed into $S.PK \subseteq \{V(TBL_A.PK) | V.PK = V'_k.PK\}$ or $\neg\exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_A.PK \notin S.PK)$ correspondingly. For $S = V_1.A$, it is changed into $S.PK = \{V(TBL_A.PK) | V.PK = V'_k.PK\}$. Here, if S is a path expression, then $S.A'_i$ is also a path expression; if S is a constant, then $S.A'_i$ denotes the components in S representing the values of attribute A'_i.

Like CASE 5, we can show the transformed predicates are equivalent to the original ones. After the substitution, the newly created path expressions are in the form $V'_A.A'_i.\cdots$. They are further processed based on the type of A'_i. The final result $V'.A'.\cdots$ in P' is equivalent to $V'_A.A'_i.\cdots$ as long as each further iteration, which applies one of the eight cases, keeps the equivalence.

CASE 8 There is a path expression having the form $V_1.A$ or $V_1.deref(A).\cdots$, and the type of A is a set of a NRT reference. Assume that the table holding the tuples referenced by A is TBL'. The schema transformation process outputs an intermediate OR table TBL_A. The table has two columns, one (PK) stores the primary key of TBL_1, the other (A_RPK) stores the primary key of TBL'. Therefore, for each tuple T_1 of TBL_1, there exists a set of tuples in TBL_A such that for each tuple in the set, we have $T_{Ai}.PK = T_1.PK$. Further, there exists one to one correspondence between values in $T_1.A$ and the tuples in the set, such that $deref(I).RPK = T_{Ai}.A\_RPK$, where $I \in T_1.A$, RPK is the primary key of TBL', and T_{Ai} is a tuple of TBL_A. Three subcases are considered here. Like CASE 5, the path expression participates in three kinds of predicates, membership, inclusion, and equivalence.

If the predicate is like $V_1.deref(A).A'_i.\cdots$, it may participates in three kinds of predicates, $S \in V_1.deref(A).A'_i.\cdots$, $S \subseteq V_1.deref(A).A'_i.\cdots$ or $V_1.deref(A).A'_i.\cdots \subseteq S$, and $S = V_1.deref(A).A'_i.\cdots$, where S is either a constant or another path expression. The first kind of predicate is changed into $\exists V'_A(TBL_A), V'_R(TBL')(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK \wedge V'_R.A'_i.\cdots = S)$. The second kind of predicate is changed into $S \subseteq \{V(TBL'.A'_i.\cdots) | \exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK)\}$ or $\neg\exists V'_A(TBL_A)(\exists V''_R(TBL')(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK \wedge V'_R.A'_i.\cdots \notin S))$. The third kind of predicate is changed into

$$S = \{V(TBL'.A'_i.\cdots) | \exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK)\}.$$

If the predicate is like $V_1.deref(A)$, the three kinds of predicates are transformed as following. If the predicate is like $S \in V_1.deref(A)$, it is changed into $\exists V'_A(TBL_A), V'_R(TBL')$ $(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK \wedge V'_R.RPK = S.RPK)$. If the predicate is like $S \subseteq V_1.deref(A)$ or $V_1.deref(A) \subseteq S$, it is changed into

$$S.RPK \subseteq \{V(TBL'.RPK) | \exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK)\}, or$$

$$\neg\exists V'_A(TBL_A)(\exists V'_R(TBL')(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK \wedge V'_R.RPK \notin S.RPK))$$

correspondingly. If the predicate is like $S = V_1.deref(A)$, it is changed into

$$S.RPK = \{V(TBL'.RPK) | \exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_R.RPK = V'_A.A\_RPK)\}.$$

Here, if S is a path expression, then $S.RPK$ is also a path expression; if S is a constant, then $S.RPK$ denotes the components in S representing the values of attribute RPK.

If the predicate is like $V_1.A$, the three kinds of predicates are transformed as following. If the predicate is like $S \in V_1.A$, it is changed into $\exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_R.RPK = deref(S).RPK)$. If the predicate is like $S \subseteq V_1.A$ or $V_1.A \subseteq S$, it is changed into $deref(S).RPK \subseteq \{V(TBL_A.RPK) | V'_A.PK = V'_k.PK\}$ or $\neg\exists V'_A(TBL_A)(V'_A.PK = V'_k.PK \wedge V'_A.A\_RPK \notin deref(S).RPK)$ correspondingly. If the predicate is like $S = V_1.deref(A)$, it is changed into $deref(S).RPK = \{V(TBL_A.A\_RPK) | V'_A.PK = V'_k.PK\}$. Here, both S and $deref(S).RPK$ are path expressions. Assume S is $V.A_1.\cdots.A_i$, then $deref(S).A_j$ denotes a path expression $V.A_1.\cdots.deref(A_i).A_j$.

Like CASE 5, these transformed predicates are equivalent to their original ones. After the substitution, some newly created path expressions are in the form $V'_R.A'_i.\cdots$. They are further processed based on the type of A'_i. The final result $V'.A'.\cdots$ in P' is equivalent to $V'_A.A'_i.\cdots$ or as long as each further iteration, which applies one of the eight cases, keeps the equivalence.

Therefore, we can see that for each interpretation $I(P_0(T/V, T_1/V_1, T'_1/V'_1, \cdots, T'_t/V'_t))$, there exists exactly one corresponding interpretation $I(P'(T/V, T'_1/V'_1, \cdots, T'_s/V'_s))$. Comparing the two interpretations, all the newly added predicates in P', which are introduced when substituting all

$V_1.A.\cdots$ like path expressions, are always true. All those predicates in P' that simply have $V_1.A.\cdots$ substituted with $V_i'.A'.\cdots$ produce same value as their counterparts in P_0 because $T_1.A.\cdots = T_i'.A'.\cdots$ always holds. All those predicates in P' that are transformed from predicates of P_0 produce the same values of those of P_0, as described above case by case. Therefore, we have $I(P_0(T/V, T_1/V_1, T_1'/V_1', \cdots, T_t'/V_t')) = I(P'(T/V, T_1'/V_1', \cdots, T_s'/V_s'))$, and further

$$I(P(T/V, T_1/V_1, \cdots, T_n/V_n)) = I(P'(T/V, T_1'/V_1', \cdots, T_s'/V_s'))$$

That is, the theorem holds for any calculus with k variables. □

Here, we have shown that any query over the integrated OR schema is equivalent to the substituted queries over transformed local OO and relational schemes, and the substituted query is equivalent to the partitioned queries. Therefore, the query over OR schema is equivalent to the set of queries produced by the partitioning algorithm. That is, given a query over OR schema, the execution of partitioned queries can always produce the same result as if there were an OR query engine.

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. In this paper, we focused on the design of query partitioner in this heterogeneous architecture. The algorithm for implementing this functionality has been proposed. The correctness of the process has also been proven here.

Comparing with the native implementation approaches, the performance of our approach is not as good. However, it provides an easy way to build an experimental ORDBMS, and therefore gives us a convenient platform to further study OR database issues.

Up to date, we have implemented the two algorithms of schema transformation and query parti-

tion, and will implement the full fledged heterogeneous system in the future. We have also observed that the performance of generated queries need further optimization. For example, it is desirable to reduce the number of join operations in the output query, and to reduce the interactions across the two local DBMSs. Further research work will include the OR query optimisation issues, especially in this heterogeneous environment.

References

[ISO, 1992] (1992). *ISO/IEC 9075:1992, Database Language SQL- July 30, 1992.* ISO/IEC.

[Beech, 1997] Beech, D. (1997). Can SQL3 be simplified? *Database Programming and Design*, pages 46–50.

[Carey and DeWitt, 1996] Carey, M. J. and DeWitt, D. J. (1996). Of objects and databases: A decade of turmoil. In *Proceedings of the 22nd International conference on VLDB*, pages 3–14, Mumbai (Bombay), India. VLDB, Morgan Kaufmann.

[Cattell, 1994] Cattell, R. G. G., editor (1994). *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers.

[Date and Darwen, 1993] Date, C. J. and Darwen, H. (1993). *A Guide to The SQL Standard, 3rd ed.* Addison-Wesley, Reading, MA.

[Ishikawa et al., 1996] Ishikawa, H., Yamane, Y., Izumida, Y., and Kawato, N. (1996). An object-oriented database system Jasmine: Implementation, application, and extension. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):285–304.

[Kemper and Moerkotte, 1994] Kemper, A. and Moerkotte, G. (1994). *Object-Oriented Database Management: Application in Engineering and Computer Science.* Pretice-Hall.

[Kulkarni et al., 1995] Kulkarni, K., Carey, M., DeMichiel, L., Mattos, N., Hong, W., Ubell, M., Nori, A., Krishnamurthy, V., and Beech, D. (1995). *Introducing Reference Types and Cleaning up SQL3's Object Model.* ISO DBL LHR-077 and ANSI X3H2-95-456 R2.

[Li et al., 1997] Li, H., Liu, C., and Orlowska, M. E. (1997). Extended algebra and calculus for object-relational databases. *information & Computation*. Submitted.

[Liu et al., 1997] Liu, C., Orlowska, M. E., and Li, H. (1997). Realizing object-relational databases by mixing tables with objects. In *Proceedings of 4th International Conference on Object-Oriented Information Systems*. Springer-Verlag.

[Maier, 1983] Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press.

[Melton, 1995] Melton, J. (1995). *(ISO/ANSI Working Draft) Database Language SQL3*. ISO DBL YOW-004 and ANSI X3H2-95-084.

[Meng et al., 1993] Meng, W., Yu, C., Kim, W., Wang, G., Pham, T., and Dao, S. (1993). Construction of a relational front-end for object-oriented database systems. In *Proceeding of 9th International Conference on Data Engineering*, pages 476–483.

[Quine, 1969] Quine, W. V. O. (1969). *Set Theory and Its Logic*, chapter 2. Virtual Classes, pages 15–21. The Belknap Press.

[Stonebraker, 1996] Stonebraker, M. (1996). *Object-Relational DBMSs: The Next Great Wave*. Morgan Kaufmann.

[Stonebraker et al., 1990] Stonebraker, M., Rowe, L. A., and Hirohama, M. (1990). The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142.