1994

Supporting Complex Data Types in a Relational Distributed Database System

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Yin-he Jiang

Report Number:
94-027
SUPPORTING COMPLEX DATA TYPES
IN A RELATIONAL DISTRIBUTED
DATABASE SYSTEM

Bharat Bhargava
Yin-he Jiang
Shalab Goel

CSD TR-94-027
March 1994
Supporting Complex Data Types in a Relational Distributed Database System

Bharat Bhargava  Yin-he Jiang
Shaab Goel
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

*To submit for publication*

We present the design and implementation of a distributed object-based database system, called O-Raid. O-Raid design embodies the extension of an existing distributed relational database system called Raid, to provide support for abstract data types. Our design and implementation reuses relational database system software while providing the client/server paradigm. The O-Raid SQL++, an extension to the relational query language SQL, supports the object-oriented data manipulation capabilities such as subobject referencing, method invocation, navigational queries, and “implicit joins” while maintaining the closure property of relational model. These features provide better expressiveness and functionality in user queries equivalent to SQL-3. We describe the features of SQL++, and design and implementation of its components, namely, the data definition language (DDL), and the data manipulation language (DML). O-Raid supports the complex objects by storing the subobject itself (embedded object approach) or by storing a pointer to it (pointer referencing approach) within the composite object. O-Raid allows partial and full replication of objects as well as selective replication of fragmented objects. We discuss the overheads for supporting objects in various phases of query execution: method execution, subobject referencing, and format translation, and compare the results with equivalent queries with relations on the underlying Raid. We show that for insert and select queries, the additional overheads are below 15%.

*This research is supported in part by a grant from AIRMICS and UNISYS.*
1 Introduction

To support the non-traditional applications such as Computer Aided Design, Geographical Information Systems, Multimedia, etc., in database systems, researchers have taken two approaches. In the first approach, the revolutionary approach, the database management system for supporting these applications is built from the scratch. For example, the object oriented database management system (OODBMS) design may entail extending the persistence of object oriented language such as Smalltalk or C++, as in ObjectStore [1], or designing a new semantic data model by combining the OO and database features, as in O2 [2]. Other systems following this design philosophy include GemStone [3] and Ontos. In the second approach, the evolutionary approach, an existing relational database system is extended with OO capabilities to support the abstract data types with method code, type inheritance and complex objects. Systems supporting this design methodology include POSTGRES [4], Starburst [5], Iris [6] and O-Raid [7]. We discuss the support for complex data in these systems in the next section. Most of the systems following this approach allow the co-existence [8] of the flat relations and the complex data. This means that the conventional applications, like banking need not be modified to run on more extensible database system supporting multimedia, for example, as they still can access the existing relational data. Furthermore, since the existing data will not be re-formatted usually, the porting of existing applications to the newer system is easy and fast. These database systems serving the non-conventional applications have been referred to as "Next-Generation" database systems in [9].

Complex data is expressed by the user defined types or classes in programming languages or query languages. In the class definition, the common features of objects are defined by instance variables (members) and methods (functions). The state of an object, i.e., the values of the instance variables in the object, can only be modified through the execution of methods defined for the class. Supporting complex data in a relational database system will require extension of both the Data Definition Language (DDL) and Data Manipulation Language (DML).

In Postgres, the query language POSTQUEL supports constructs for declaring new base types, and complex data types (referred to as constructed types). A class of functions, implemented as a set of POSTQUEL statements, returns a constructed type, and implement the non first normalised
relations. Procedures can be dynamically loaded and executed, as also the access methods for user defined basic types. Thus, most of the extended functionality of the data model is provided by extending the POSTQUEL. Postgres assigns each record a unique identifier (OID). Postgres supports multiple inheritance, by allowing a constructed type to inherit data elements from other constructed types. However, the two relations corresponding to these two constructed types will be stored separately and in their entirety. For example, a subobject is stored as an attribute in the relation corresponding to the superobject. Note, that this subobject is stored in a separate relation corresponding to its constructed type also. This storage model for data is compatible with the non first normal form relational model used in Postgres.

Starburst supports hierarchies of user-defined types and functions, large unstructured and structured complex objects. Unstructured objects can store objects up to 1.5 Gigabytes long. This size is moderately large to store video, audio and image objects. DBMS does not model the contents or structure of these objects. Structured complex objects comprise of views on the data stored in relation tables. Efficient techniques for parent-child joins are designed for constructing these views (objects). In Starburst’s eXtended Normal Form (XNF), an object is defined by an XNF view, which extends the relational views from single resulting table to multiple tables, organised as a structured, heterogeneous set of interrelated rows. These views are defined as queries and stored in database catalog. This approach is in contrast to the approaches of Postgres and O-Raid which use nested relation and navigational approaches respectively. This scheme provides more flexibility in defining new interrelationships (and hence new structured objects) on the existing relations. We think that efficiency of such “views” will be very dependent on the performance of joins, defining the parent-child relationships of the complex object.

Our design has been based on the belief that extending relational database systems to support object oriented system capabilities is a promising way to support the new applications. One motivation for making this choice, was the practical applicability of the extended systems. Most of the database system technology that has been developed and has costed many man years can be reused, in extending the systems. Furthermore, the vast amount of accumulated knowledge for relational data model for managing large amounts of data, recovery, concurrency control, reorgani-
zation of data, etc., can be directly applied to provide support for complex data in new-generation applications. We believed that it might be beneficial to leverage, and not reinvent the relational technology for supporting object-oriented systems built from the scratch. Based on these reasons, we designed and built a distributed object database system called O-Raid [7, 10] following the evolutionary approach. O-Raid extends a distributed relational database system prototype called RAID [11], also designed and implemented at Purdue. As we describe in the following sections, O-Raid uses an extended SQL version called SQL++. The extensions support the object method execution, the object creation and object referencing. The underlying storage model is still the normalized relational model. A novel technique for referencing the tuples corresponding to subobjects in the relation corresponding to the super object has been engineered. O-Raid integrates the relational and object model by retaining the simplicity of the relational data model while providing the functionality and modularity of the object data model. Finally, since O-Raid is built on top of RAID, it provides all the distributed services, provided by the underlying Raid system. We have conducted experiments with distributed transaction processing in O-Raid and selective replication schemes for composite objects [12]. It provides users with a view of a distributed object database system (See Figure 5).

The rest of the paper is organized as follows. First in section 2 we describe the interfaces that the O-Raid user interface uses to interact with the underlying database system. Section 3 describes the design and implementation of the the data definition facility. Section 3.3 discuss the design and implementation of the SQL++. We will also show how users can define classes in C++ and register these complex data types into the database system. We then show through examples how to define database schemas, create database (insert data), query database (select data), and modify database (update data). Finally, we discuss our experiences and outline the future work.

2 Assumptions and Environments

Interface O-Raid assumes only two things from the underlying system. One is the capability to handle transaction processing with ACID property. The other is the ability to organize meta data in the system. The O-Raid user interface interacts with the underlying RAID database system.
through a set of function/procedure calls. These function/procedure calls are:

- **Get.Unique.TS()**: return a *unique* timestamp value within the transaction manager.

- **init_transaction()**: get the timestamp for a transaction, initialize the write set for update actions of the transaction.

- **get_tuples()**: given a transaction identifier return all the tuples of a relation. If index information is given, it returns all the tuples satisfying the index condition.

- **exec_wropn()**: Execute write operation for a transaction by appending an updated relation to a write set. Note, the update actions are delayed until the commit point.

- **commit_transaction()**: commit a transaction of a transaction identifier along with the updated relations in the transaction.

- **abort_transaction()**: abort the transaction of a given transaction identifier.

- **cleanup_transaction()**: it is actually a call to *commit_transaction()* with zero number of updated relations. It is used to release resources occupied by a transaction.

Each query in O-Raid is usually regarded as a *single* transaction. A transaction is processed as follows:

- Get transaction identifier and initialize a write set for the transaction.

- Read data, including reading meta data, such as attribute type, size; reading relations, including reading tuples satisfying index values on an attribute or matching given tuple identifiers.

- Process data, including evaluating attributes, predicates, constructing objects, executing functions on objects etc.

- Write data, including inserting tuples, update tuples etc. All the updated data are accumulated in a write set instead of immediately updating into the database.
• Commit/Abort a transaction. If it is a commit, the write set of the updated relations will be sent to the transaction manager; else the transaction is aborted and resources occupied by the transaction are released back to the transaction manager.

All the interactions between query processor and the underlying transaction manager are through message passing as follows.

• The query processor sends a request message to the transaction manager and waits in a loop of blocking receive call.

• When the query processor receives a message, it checks if this message is for the current transaction by matching the tid. If it is, it breaks out the loop of waiting; else it loops back for the blocking receive and ignore the message.

• Based on the message types information received, perform the appropriate actions.

To avoid dead locks among multiple transactions, we adopt a simple solution of timeout. Users can specify a timeout values in seconds for a transaction to execute, usually much larger than needed, if the time expires before the transaction completes, the transaction is aborted and the transaction manager is cleaned up. Users can retry the same queries later. Based on this policy query processor may receive messages of the previous aborted transactions, therefore we need to check the validity of the message.

RAID DBMS Features In our approach for supporting complex data types in a relational system, we only require that the underlying transaction manager to have the following novel features:

• Organize data using relational data model.

• Possess ACID property for transaction processing.

• Organize relational information as relations, i.e., meta data.

The transaction manager RAID [11] is relational system that satisfy the above properties. In addition the RAID DBMS has more other properties:
• It is a distributed system. It supports replication of data and provides users a logical view of one copy data. The replication algorithms supported are Read-One-Write-All (ROWA) and Quorum Consensus (QC).

• It is adaptable. The concurrency controller supports a set of algorithms, such as two-phase locking (2PL), timestamp ordering (T/O) and optimistic concurrency control (OPT).

• It is robust. The atomicity controller supports two-phase commit (2PC) and more failure-resistant but expensive three-phase commit (3PC) protocols.

• Communication facility is based on message passing mechanism. The high level, layered communication package provides clean, location independent interface between RAID servers.

3 Design and Implementation

Outline of O-Raid Design  For simplicity and modularity, we separate Data Definition Language facility from Data Manipulation Language facility.

The transaction processing part of the figure presents the Raid relational database system. It divides the functions of transaction processing into software modules called servers. This architecture provides for modularity and extensibility. The modularity of RAID facilitates adding support for objects, because much of the servers' code can be reused. It takes message passing as communication scheme. A high level, layered communication package provides a clean, location independent interface between servers.

The roles of the servers in the RAID system are:

• User Interface (UI): a front end invoked by the user to process SQL++ queries.

• Action Driver (AD): accepts a parsed query from the UI, formats the query as a transaction (read and write actions), and executes the transaction.

• Access Manager (AM): provides access to the local database and ensures that updates are posted atomically to stable storage.
- **Atomicity Controller (AC):** manages the commit phases of transaction processing to ensure that a transaction commits or aborts globally.

- **Replication Controller (RC):** maintains consistency of replicated copies of the database in the event of failures.

- **Concurrency Controller (CC):** maintains serializability among concurrent transactions.

The data manipulation language part of the figure shows the four modules of the O-Raid data manipulation facility. *UI* is the O-Raid teletype user interface. *SUITE-UI* is graphical user interface for O-Raid. It is a window based system that provides buttons, pop up display windows and icons. *AD* module implements each query supported in O-Raid. It provides a set of function calls to the *UI*. *DLD* is a dynamic loader that allows O-Raid *UI* to load user-defined procedures on demand, do the linking and execute the procedure.

The data definition language part the figure shows three components of the O-Raid data defi-
nition facility. "C++PARSER" takes C++ class definition header files and generates class information for "DBEDIT". "DBEDIT" takes the database schema information and store it as meta relations. "DEMANGLER" is for translating mangled method symbols into method signatures. It is used in UI when it needs to dynamically execute user procedures.

To support complex data types in a relational database system we investigate the following problems:

- how to store user-defined class definition into a database.
- how to process queries involving attributes of user-defined types (classes).
- how to construct objects, execute user-defined functions, and handle object referencing.

We first describe an overview of the O-Raid design, and then discuss each item in details. In O-Raid we support a static user-defined types in O-Raid. That is, if users want to change the complex data types and schemas, they need to exit the query interface and recreate the user-defined types and schemas. Similar to OZ+ described in Chapter 13 of the book [13], we organize the class definition information as meta relations in the database. This method allows us to make use of the relational database system to insert and retrieve the class information.

We decompose our task of incorporating complex data types into two subtasks. One is to register the class definitions and create database schemas using the registered classes as the complex data types. The other subtask is to access object member, and methods of complex data types in queries.

3.1 Incorporate Abstract Data Types

Define Abstract Data Types To incorporate a new type in O-Raid, users need to first write a C++ program to define classes. A C++ program consists of a header file (.h) and a source file (.cc). The header file declares members (fields) and methods (functions) for the user-defined classes. In Figure 2, we define classes Grade, Student, and Grad_Student. Notice that in class Student, we have a member grade of user-defined type class Grade; in class Grad_Student, we have a member pStudent being a pointer to a user-defined class Student.
The source file contains the implementation of the class methods. Users then start a data definition session, in which the class definition files are parsed and registered, tables with columns of user-defined types can be created. After schema specification users can start a user interface to insert objects into tables, invoke methods on objects for selection of data and construct composite objects.

Register user-defined types  The first step of building O-Raid DBMS is to construct a front end subsystem for creating database schemas using complex data types. This requires an extension to the data definition facility for the following features:

- Register user-defined types
- Relations containing objects
- A table mapping between method signatures and symbol names
- Spanning multiple-site objects

A C++ parser is created for parsing the C++ header files, such as the one in Figure 2. For each class in the header file, the parser generates a list of items. Each item contains a name, type, and size in bytes of a member field in a class. If a member field in a class is itself of a type class, it will be recursively parsed and substituted by its members until all the members become of simple types. This is because the underlying relational DBMS can only handle relations of simple types.

Our design constraint is that:

We have to flatten out the user-defined class hierarchies into plain relations of simple types because the underlying database system is a relational one.

A facility called dbedit was developed to support the schema specification requirements. The dbedit facility adopts the standard SQL data definition commands [14] and extends it with commands for registering classes; specifying configuration of distributed database; specifying replication of relations, etc. The procedure of registering classes consists of the following:
class Grade {
    int midterm;
    int final;
    int hw[NUM_HWS];
public:
    Grade(int mid, int fin, int hw1, int hw2, int hw3);
    int get_exams(float mid_percent);
    ...
};

class Student {
    char name[NAME_LEN];
    class Grade grade;
public:
    Student(char *nm, int mid, int fin,
            int hw1, int hw2, int hw3);
};

class Grad_Student {
    Student *pStudent;
    char support[STLEN];
    Professor *pAdvisor;
public:
    Grad_Student(char *s);
};

Figure 2: A C++ Header file for user-defined classes Grade etc.
• **Parse the class definitions.** A parser equivalent to the front end of a C++ compiler was built. It takes C++ header files as input, recursively generate the information about each class. The information includes member names, member types, and their size in bytes, method name, method signatures.

• **Preprocess class implementation code and generate meta relations.** We use GNU G++ compiler to compile the C++ programs (written by users for implementing the class definitions) to generate object code (.o file). We then use the UNIX command nm to print the symbol names, and select the ones for method functions of which the second field is of character T (text segment symbol). We can generate method signatures from the method symbol names by a demangler program. A table mapping from method signatures to method symbol names is built. The table is used for dynamic execution of methods in DML during query processing. The details about the use and how it is created are discussed later.

**Meta Relations for Storing Class Definitions** To preserve the class hierarchy information, we need to construct some meta relations. They are `CLASSRELATION`, `CLASSATTRIBUTE`, `CLASSMETHOD`, and a class relation for each class. In the underlying relational DBMS we have the meta relations `RELATION`, `ATTRIBUTE`. We discuss them in details.

• The meta relation `RELATION` is a table of pairs of a relation name and its relation identifier. The relations include meta relations, and tables created by users.

• The meta relation `ATTRIBUTE` contains all the attribute information of all the relations. Each relation is described by a portion of tuples in the meta relation `ATTRIBUTE`. Each tuple from top down describes an attribute of a relation from left to right.

The interesting part is that there is a portion of tuples in `ATTRIBUTE` relation that describe itself, see Table 1. The Table 1 from top down says that the 1st column of the table of `ATTRIBUTE` is `TUPLE_ID`, a tuple identifier; the 2nd column is `VERSION`, a version number; the 3rd column is `USED`, a flag of use; the 4th column is `RNAME`, a relation name; the 5th column is `ANAME`, an attribute name; the 6th column is `COLUMN`, a column starting from
Table 1: The portion of tuples for ATTRIBUTE meta relation describing itself.

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>0</th>
<th>1</th>
<th>ATTRIBUTE</th>
<th></th>
<th></th>
<th></th>
<th>TUPLE.ID</th>
<th>0</th>
<th>3</th>
<th>4</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>VERSION</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>USED</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>RNAME</td>
<td>3</td>
<td>2</td>
<td>64</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>ANAME</td>
<td>4</td>
<td>2</td>
<td>64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>COLUMN</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>ATYPE</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>ASIZE</td>
<td>7</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>ATTRIBUTE</td>
<td>AINDEX</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0; the 7th column is ATYPE, attribute type encoded as integers, e.g., 2 as character string, 3 as integer; the 8th column is ASIZE, attribute size in bytes; the 9th column is AINDEX, flag for attribute index, 0 means no index.

- The meta relation CLASSRELATION contains a set of tuples. Each describes a class name, class relation identifier, class size in bytes, object module file name (.o file). The size information is needed for in memory construction of an object.

- The meta relation CLASSATTRIBUTE is a table. Each tuple (row) describes an attribute name, a relation name for the attribute, a type name (i.e. class name) for the attribute, a flag n, (if n > 0, it means the attribute is a pointer and n is the level of indirection; if n = 0, it means the attribute is an embedded object in the relation), and starting and ending columns in the relation for which the attribute spans in the relation. If the attribute is a pointer, it always spans three columns.

- The meta relation CLASSMETHOD is a table mapping method signatures to method symbol names. The use of the method signature table is for dynamically executing user-defined functions in queries. When a user invokes a function (method) in her query, the method
Method Signatures | Method Symbol Names
--- | ---
Grad.Student::Grad.Student(char *) | __12Grad.StudentPc
Grade::Grade(int, int, int, int) | __5Gradelili

Table 2: A meta relation CLASSMETHOD for Grade classes.

signatures 1 can be derived based on the arguments used in calling the method. But the dynamic loader needs the method symbol name 2 of the precompiled method code to load the object module into memory and link it with the executing process. Thus the Method Signature Table is consulted to convert a method signature to a method symbol name before dynamically executing a method during query processing. The table is generated during the "Register Class" stage of the data definition phase as follows:

- invoke the UNIX command `nm` on the class object file (.o) to generate a list of output lines. Select the lines containing mangled names for the methods by testing that the second field indicates text segment symbol.

    ```
    nm object_file | awk '$2=="T" { print $3 }'
    ```

- process them with a demangler program and build the desired table. The demangler program generates the method signature from a mangled method symbol.

An example of the meta relation CLASSMETHOD for Grade classes is shown in Table 2.

### 3.2 DDL Commands

The list of DDL commands are:

1. A method signature consists of number of parameters and the type of each parameter for that method.
2. A method symbol name is name in the symbol table for method functions in an object module file (.o file).
CREATE TABLE IDENT ( declares )
DELETE TABLE IDENT;
DISPLAY REPLICATE;
REPLICATE;
CONFIG ( HOST_MACHINE DB_DIRECTORY );
INSTALL;
CREATE INDEX IDENT ON IDENT ( IDENT );
DROP INDEX IDENT;
DISTRIBUTE TABLE IDENT ( HOST_MACHINE DB_DIRECTORY );
CLASSREAD;
REGISTERCLASS optdir IDENT;

Relation containing objects  O-Raid supports both inter-object referencing (or pointer referencing) and intra-object referencing (or embedded object referencing, where an object is stored within another object). In queries persistent pointers to user defined types in attributes of a relation is allowed. A persistent pointer in O-Raid is represented by three integers, object identifier (OID), relation identifier (RID), and a offset (OFFSET) [15] that uniquely identify the object and its class. The pointed objects are stored in their corresponding class relations. OID is the Object Identifier; it is used for finding the object within the class relation. RID uniquely identifies a class relation in which the pointed objects are stored. It also serves as a key to find class information for an object in the CLASSATTRIBUTE meta relation. OFFSET is the tuple offset in that relation for the object; it is used for fast access. An example using pointers as attributes in relation grad_students is shown in Figure 3.

For example the command

CREATE TABLE PersonalAddresses (char uid[32], Address a);

creates a relation involving user defined type Address. Here PersonalAddresses is the relation with
Figure 3: Storing Objects in Relations.
Attribute `uid` of simple type `char array` and attribute `a` - an embedded object of type `Address`. The relation is transformed into a RAID relation containing five attributes of simple type (see Figure 4-a).

However, if the O-Raid relation involves pointers to user defined types then three fields of simple type are created to represent the pointer attributes. Consider the command:

```sql
CREATE TABLE PublicAddresses (char tag[32], Address *pa);
```

The relation is `PublicAddresses` and `pa` is a persistent pointer [10] that points to a shared object of user defined type `Address`. As a persistent pointer in O-Raid is represented by three integers, object identifier (OID), relation identifier (RID), and a offset (OFFSET) [15]. Thus, the relation is transformed with three simple attributes in place of the pointer attribute (see Figure 4).
Replicating Relations  The dbedit program creates a temporary working directory where all the system files and metadata files are built. After the schema specification is completed, the execution of REPLICATE command results in actual creation of database directory on each of the sites forming the distributed database system through replicate.sh shell script. This program determines the sites forming the distributed database system, that is, the sites specified through the CONFIG command. The information such as relation name, attribute types, the method signature to method symbol map, the method code (.o file) has to be fully replicated among all configurable sites. However certain information such as the files which reside at a site is different for each site and is generated differently. The UNIX remote copy command (rcp) is used to replicate system files and relation data files at appropriate sites. The temporary working directory is destroyed upon exit from the dbedit program.

3.3 O-Raid DML

O-Raid DML Features  O-Raid interacts with the underlying RAID system through Read/Write operations and init.transaction, commit.transaction primitives provided by the RAID system. O-Raid DML features are listed as follows:

- Complex types: O-Raid supports static user-defined types. User-defined types allow database designers to create abstractions of data, define new functions that manipulate data according to user requests. An example is to create pattern matching capabilities into the SQL query language, so that users can not only do equality checking between two fields, but also do partial matching or regular expressions in queries.

- Data structures: O-Raid supports both inter-object referencing (or pointer referencing), where the address of an object is stored in another object, and intra-object referencing (or embedded object referencing), where an object is stored within another object. In queries persistent pointers to user defined types in attributes of a relation is allowed. A persistent pointer in O-Raid uniquely identifies an object and its class. It is represented by three integers, object identifier (OID), relation identifier (RID), and a offset (OFFSET) [15]. OID is the Object Identifier. RID uniquely identifies a relation in which the pointed objects are stored. It also
serves as a key to find class information for the object in the CLASSATTRIBUTE meta relation. OFFSET is the tuple offset in that relation for the object.

As for constructing a complex object such as a tree, we do it in two phases. In the first phase each node object is constructed with pointers set to value "NULL" and their addresses stored in temporary variables. On the second round, the pointers are set to point to objects through those temp variables.

- **Functions and variables:** O-Raid supports dynamic execution of user-defined functions. The function code is loaded and linked as needed during query processing. Once it is loaded and linked, later calls to the function will be executed directly (no need to reload and relink). The user-defined functions can be used for construction of objects, filtering of data for selection.

O-Raid supports both temporary and permanent variables that store the results of one query and can be processed by another query. This is equivalent to supporting composition of functions and nested queries. According to Chandra and Harel [16] any query language which can express the existential query and also store the results of queries on the database has the power to compute any first-order query.

Objects, classes, and a predicate-based relational query language are supported. O-Raid objects are compatible with C++ objects and may be read and manipulated by a C++ program without any "impedance mismatch". O-Raid software organization is shown in Figure 1.

O-Raid supports direct store of an object within another object (embedded object) and store of the *address* of an object (shared object). The fundamental components of the O-Raid query processor can be reduced as to support: a) execution of user-defined functions; b) access of attributes through embedded objects and persistent pointers.

**Construct Objects from Tuples** From users' perspective there are two kinds of operations *update* the data and *retrieve* data. Database system designers need to support the corresponding queries. As the data model of O-Raid allows the coexistence of relations and class, data have two different forms. One is relation tuple; the other is class object. Relational form data can exist in
both database servers and clients, while data in an object form only exist in the client part for query processing.

The support query processing involving complex data types, we need to support construction objects from tuples and vice versa; execution of user-defined methods (functions) on the constructed objects. All these require the support of execution of user defined functions in query processor. There are two ways to link the user programs with the query processor program. One is to statically link all the user programs. This is not flexible and results in unnecessary overheads. The other is to dynamically link and load user programs as needed. This allows users to add or drop user functions incrementally.

In O-Raid there are two kinds of special attributes that users can access. One is embedded objects with a relation; The other is to access objects via persistent pointers. For example in Figure 3, suppose that we want to access grad_students’ final exam score with relation grad_students, and the attribute name for the relation grad_students is g, we write:
g.Student->Scores.Exams.final

Accessing Embedded Objects In Relations  During query processing we need to convert the path name of the attribute to the column number within the base relation in which the attribute resides on. For example, to access the class attribute Student.Scores.Exams.final in Figure 3, we need to compute the column number of the base attribute final within class relation Students.

From meta relation CLASSATTRIBUTE we can get the following information: attribute Scores is in column 4 of class relation; attribute Exams is in column 0 of class Scores; attribute Final is in column 1 of class Exams. Therefore, in class Students attribute Final is in column $4 + 0 + 1 = 5$.

Accessing Objects via Pointers  We need to locate the 1) relation; 2) the tuple within the relation which the pointer value indicates. Recall that the pointer consists of three components, namely OID, RID, OFFSET. From RID -- a relation identifier we can access the relation; from OID -- we can access the tuple within the relation. If the OFFSET has values we can directly get to the tuple. For example, to access the class attribute grad,students.Student-> in Figure 3, we need to get the relation identifier RID, and object identifier OID. Based on these two values, we can submit an index read on the indexed attribute OID on relation with RID and get the tuple from the underlying relational database system.

Supporting Dynamic Method Execution  We have two choices for supporting method (function) executions in processing user queries. One is via static linkage of all the precompiled user-defined method code with the query processor program. However, this approach results in a rigid system that does not allow users to later add or drop methods (functions); furthermore, it forces users to pay the overheads for loading the all the methods regardless they are used or not. On the other hand, dynamic execution of user-defined functions allows users to add or drop the precompiled user-defined method code on demand. Users only pay the overheads for loading the methods as needed.

We acquired a library package implemented under UNIX called dlib from public domain of GNU.

\[ ^3 \text{From left to right the first column is column 0} \]
free software. This package was implemented by W. W. Ho and R. A. Olsson in University of California, Davis for C programming language and is available for VAX, Sun 3, and SPARCstation machines. For more details of the did package, refer to [17]. We briefly describe the interface provided by the did package:

- "The link operation is performed by the function dlink(char *filename), where filename specifies either a relocatable object file or an object library."

- "Did provides two functions for unlinking a module: unlink_by_file(char *file, int hard) and unlink_by_symbol(char *symbol, int hard). The first function requires as a parameter the filename corresponding to a module previously linked by dlink, while the second function unlinks the module that defines the specified symbol."

- "The function dld_init(char *filename) performs the required initialization of the did package. It takes as argument the initial executable file of the program and loads the symbol table information of this file into memory."

- "get_func(char *func_name) returns as value an entry point of a dynamically-linked function. This value can later be used as a pointer to the function."

- "The predicate function function_executable_p(char *func_name) tells whether the specified function can be safely executed, i.e., whether the execution of this function might lead to referencing any undefined symbols."

Query Processing After the schema is correctly specified the dbedit creates the distributed database, users can submit SQL++ queries [15] to the user interface to insert and update data in relations. The O-Raid query language is called SQL++ and is an extension to the SQL. It supports queries involving objects [15]. The execution of a query can be divided as follows:

- Parse the query and build syntax tree. Using UNIX Lex and Yacc facility, a syntax tree is constructed for a user query.
• **Get transaction identifier.** Acquire a login session-wise unique transaction identifier from RAID AM server. It is used by RAID servers to distinguish different user queries/transactions.

• **Load the user code for methods if needed.** If a method call is found in the user query, we first consult CLASSATTRIBUTE table to find out the class name. We then consult CLASSRELATION table to find out which object module (.o) file to load.

We use a dynamic linker *did* developed at University of California at Davis to link the method code with the User Interface process. The dynamic linker uses UNIX system calls `setjmp` and `longjmp` switch execution from the current User Interface process to the method code and back to the User Interface process upon its finish.

• **Processing Queries.** Determine what relations are to load, what attributes to project, the predicates that needs to be evaluated for selection of data. For an attribute we need to decide whether it is of a simple type or complex type.

Processing dot and pointer attributes. A complex type is a user-defined type. It can be of two forms in queries. One is in dot attribute notation which refers to embedded objects. This requires to find out the physical column number in a table for a class attribute. This is the column number of a class attribute in a class relation plus the column number of that class attribute in the table. (See Figure ??).

The other is in pointer attribute notation which refers to another object. This requires to load in a class relation and find out the column number of the class attribute in the class relation through ATTRIBUTE table.

• **Reads – Load the relations.** Submit read requests to RAID RC server to load the relations into memory. There are two kinds reads available in RAID. One is to load the whole relation. The other is the index-read, which requires two rounds of read. During the first round it returns a set of tuple identifiers that satisfy index values. On the second round it returns a set of tuples based on the list tuple identifiers.
• **Evaluate predicates and execute methods.** A predicate can be expressions involving attributes or methods (functions) on attributes. For *select* query we need to evaluate query and the predicate may involve a filtering method (function). An *insert* query may call a constructor method for building up the object. Before a method can execute, we need to first construct an in-memory object from a tuple by invoking the constructor defined in the user-defined C++ code. Once that is done, any other C++ method could be invoked on the constructed in-memory object.

• **Writes** - **Commit a transaction and write-set to the database.** In O-Raid, we append all write operations of a transaction to a write-set and defer the actual writes to the database until commit point. At that point we submit the write-set to the RAID RC server.

• **Display the results.** Print out the projection values of a table that satisfy the predicates.

3.3.1 **Dynamic execution of methods**

Class methods can appear in *insert*, *select*, or *invoke* SQL commands. In *insert* command it serves as constructor method of constructing in-memory objects out of user-input parameters. In *select* command it can act as components of the filtering predicate or as computed non-stored attributes. In *invoke* command it allows user to invoke a procedure on an object to change object values.

The control flow of method execution is presented in pseudo code as follows:

• call *dll* library function *dll_init*(prog_name) at start up.

• when evaluate an query expression tree of function type, do the following:
  
  - process nested class attribute from left to right until it reaches the base attribute (e.g., *a.b* -> *c.method*).
  
  - build an in-memory object for the method execution by executing constructor method of the class.

  - call the function *call_method_by_et* with class name, method name, address of the object on which the method executes, the expression tree for parameters. This function in turn does the following:
* build argument list and generate method signatures.
* get the object module (.o file) for the class and load the module into memory.
* get the method symbol name from method signature by consulting the CLASS-METHOD table generated DDL phase.
* execute the method using method symbol name and argument list. The pseudo code steps are: a) call dld_function_executable_p(method_symbol) to check if the function is executable; b) call dld_get_func (method_symbol) to get the pointer to the function. c) based on the number of argument do an explicit call to the function with the arguments, e.g., if the number of argument is three, we write: (*func)(margv[0], margv[1], margv[2]). Note this is the only way in C programming language, because we need to generate exactly the same stack environment as if it were run alone in statically linked version.

3.4 Accessing objects in queries

Accessing objects in queries in general takes the form of path names, Attribute1.Attribute2...Attribute_n. Each component Attribute_i of a path name can refer to an attribute in the same relation as its previous component Attribute_{i-1} or in a different relation. These two types of referencing correspond to embedded object and inter-object referencing.

To be user-friendly we use the same notation for both cases. This is because users should only specify what they want, not how they can get the data. However, the query processor needs to distinguish the two cases to correctly access the data. This is solved by using the meta relation CLASSATTRIBUTE. We add a tag field to the meta relation indicating whether the class attribute is an embedded object or pointer to an object. In a relation there may be more than one attribute of pointer type, we need to distinguish them in meta relation ATTRIBUTE. We postfix the attribute name to OID, RID, and OFFSET field name\(^4\) with a separator character '/' which can not be part of legal attribute name.

\(^4\)Recall that OID, RID, OFFSET together comprise a pointer.
3.5 Basic Queries

For insert query in addition to relational insertion, we have insertion of class relation. Its syntax is:

\[
\text{insert into RELATION : \langle CONSTRUCTOR( ARGUMENTS ) \rangle}
\]

The actions of the command are:

- get the name of the relation to be inserted and see if it exists and what its tuple size is;
- make a tuple of the correct size, number of attribute, and the tag of each attribute;
- set up the system attributes of the new tuple add elements one at a time to the tuple, and type check each element; for each attribute do the following:
  - load the corresponding class modules;
  - create in-memory object of suitable size by consulting classrelation.
  - run constructor and pass appropriate arguments to initialize these objects.
  - convert in-memory objects to corresponding tuple values.
  - return the resulting tuple in a relation similar to the relational case.
- append the tuple to the relation and update the writset.

The select query involving classes is almost the same as the relational counterpart, except that attributes can be path names to objects through embedded objects and pointers, and attributes can be computed through functions; predicates can include method functions.

One issue encountered is how to handle the case of "NULL" pointer in a tuple for select query. There are two choices. One is to treat it as an error and abort the select query; the other is to treat the tuple as not initialized and process further for other tuples. We think that the latter is a better choice than the former, because even though one tuple is unassigned, others may have the data users require.

The query
select g.pStudent->name from grad_students
where g.pAdvisor->interests.contain_key("distributed database");

does the following:

• load the base relation grad_students;

• set up the result relation with correct number of attributes, attribute tags, column numbers, and allocate tuples;

Find out the projected relation Student, attribute name through the path name g.pStudent->name and meta relation CLASS_ATTRIBUTE.

• for each tuple in the base relation do:

  – set up the projected attribute; if the attribute is a pointer, follow it to the pointed tuple of another relation;

  – evaluate the predicate (where clause);

    * follow the path name g.pAdvisor->interests and get the class information;
    * build an in-memory object by invoking constructor method of the class with appropriate data from the tuple (from a start column number to an end column number);
    * invoke the method contain_key("distributed database") on the object.

  – add the tuple to the result relation only if the predicate is evaluated to be true.

4 Overheads in Supporting Objects

4.1 Statement of the Problem

Queries involving objects in a layered approach as in O-Raid incur overheads in method execution, subobject referencing, and format translation. We measure the execution time for insert and select queries involving objects and compare it with the equivalent queries that involve relations only. The goal is to identify the additional overheads incurred in supporting objects.
4.2 Procedure

We defined a relation seminars containing a single attribute of user-defined type Seminar. The execution time for the following insert and select queries was measured:

```sql
insert into seminars:< Seminar(
    "Distributed Composite Objects", 1994, 4, 1,
    "database objects", "Richard Jiang",
    "student", "purdue", "object-oriented composite")>
insert into seminars:<"Distributed Composite Objects",
    1992, 8, 1, ...
/* equivalent insert without object */

select s.title from seminars
    where s.date.get_year()=1990;

select title from seminars where year=1990;
```

We measured the total execution time as well as various sub-components of the query execution. For both experiments, the two queries were executed sequentially and consecutively (each 250 times). They were interleaved to obtain fair comparison under the similar environment (e.g., the load of the machine, context switch of processes, etc.).

The first insert and the first select queries incur the additional overheads of dynamic loading of Seminar class object module (.o file). The second and later queries only incur the overheads of subobject construction, i.e. initialization of the in-memory object through a constructor method execution.

Data was collected for warm starts. The warm start assumes the class object module .o file is already loaded, whereas cold start incurs the overhead of dynamic loading of class module. We repeated the time measurement for different relation sizes (5 tuples to 25 tuples) as the execution depends on the size of the relation.
Figure 9: Translation time for select queries versus the number of tuples submitted query.

- with object case: time to create a in-memory object, initialize it using the constructor, and finally constructing a transaction consisting of a single tuple from the in-memory object.

Since both queries involve format translation, the 4ms difference in translation time can be attributed to the creation and initialization of object through constructor method. We expect that typical constructor methods will involve a series of assignment statement and the observation made here is a representative measure of this cost.

The cost of insert query shows a slight increase as the number of tuples inserted is increased (see Figure 8). One would expect that the cost to be independent of the number of tuples inserted. On examining the subcomponents, we notice that the increase is largely due to the write cost. Moreover, there is a fluctuation in total time, that is mainly contributed by the write time. For writes, the O-Raid AD waits for the RAID servers to commit and return. Thus the fluctuation is likely due to the context switching of RAID servers.

For select query, we observe (Figure 7) that all the components except for the translation time are almost the same for the two select queries (with object and with relation). The translation time
for selection on relation containing 25 tuples is about 82ms \((128 - 46)\) more for the query involving objects. The processing of select query involves extra overhead on each tuple. The work required is in creating date object, invoking the method \texttt{get\_year()}\), and evaluating the predicate \texttt{get\_year()} == 1990. This overhead is approximately 3.3ms per tuple \((82/25)\). Also there is an overhead (once for the entire relation) in determining the projection columns from the subobjects \texttt{s.title}, and \texttt{s.year}.

The observation is further confirmed in the Figure 9. The translation time for both select queries shows a linear increase with the number of tuples. The only difference is that the translation time for query involving object has a larger slope indicating extra processing required for each tuple.

Conclusion The layered approach of supporting objects on top of a relational system is feasible and has low overheads (below 15%, as shown in Figure 7). For the select query the extra overhead of approx 3.3ms per tuple for evaluating predicate method may not be acceptable especially if the the relation being queried is large. For such cases, it would be better to build index on the method by precomputing the values. One such scheme of method precomputation is proposed in [18], where the values are precomputed once and an index is built.

5 Overheads in Supporting Object Replication

5.1 Statement of the Problem

A fully replicated database with Read-One-Write-All (ROWA) replication control algorithm retains local access to data. Therefore, it bears the same retrieval cost as the local retrieval cost. On the other hand, it has \textit{no better} write cost than that of the remote site write.

We examine the retrieval and the update costs for a fully replicated two site system. We measure the time for local and remote access to the objects. The overhead data will be used for analyzing the data in the experiments on composite objects of different configurations.
5.2 Procedure

Figure 10 shows the two-site database. The relation seminars is replicated at both sites and ROWA is chosen as the replica-control algorithm. The relations seminar9 is replicated at raid9 and seminar11 at raid11.

The cost of inserts into the relation is measured. Three configurations are used, namely local-copy-only (seminar9), remote-copy-only (seminar11) and full replication (seminars). The retrieval cost is measured for a relation containing 25 tuples. We vary the number of tuples in the relation queried from 5 to 25 with the increment of 5.

5.3 Data

Figure 11 contains the cost of inserting one tuple on local, remote and fully replicated relations with objects. The size of a tuple is about 800 bytes.

Figure 12 contains the cost of selection out of 25 tuples on local, remote and fully replicated relations with objects.

Figure 13 contains the different costs of select queries based on the size of the relation queried.
5.4 Discussion

From the Figure 11 we observe that every component of the cost is the same except write time. In terms of the order of write time cost, we have local < remote < full. This is because the local write is cheaper than for remote write and for full replication scheme in which we have to write to both local and remote sites.

From Figure 12 we observe that for Read time the order is local ≈ full < remote. This is because we use ROWA replication control algorithm. In the fully replicated case, it could read a local copy. Therefore the cost of Read in the fully replicated case is about the same as the local one. The little extra time is due to the search for the local copy.

We also observe that for the Write time in the local case is about the same as the fully replicated case and both are significantly less than the remote case. The reason is that in commit phase for select query there is zero-write operation. The cost is similar to that in Read.

We observe a difference of 8 ms in the Translation time for remote and full configurations. We expected that they will be approximately the same since they are all local computations. After probing further, we find out that the processing time increases as the size of the metadata files
increases. The metadata files contain replication information. The fully replicated case has the smallest size of metadata file compared to the local and remote cases.

Figure 13 shows that the Write time is independent of the number of tuples as before. There is a linear increase in Read time as the number of tuples increases because no index is set on the attribute involved in the predicate.

Summary and Acknowledgement The current version of O-Raid (1st version) has been designed in such a way that it does not require any change of the underlying relational database system RAID. It adds one layer on top of RAID. From a software engineering point view, the logical definition of O-Raid is clear and clean. It minimizes the work required to provide objects to a relational database system. This work has been done by a group of people including professor Prasun Dewan, professor Bharat Bhargava; students Yin-he Jiang, Jim Mullen, Jagannathan Srinivasan, and Ashish Vikram.

References


