The BUCKY Object-Relational Benchmark *

Michael J. Carey[†] **David J. DeWitt**[†] **Jeffrey F. Naughton**[†]
Mohammad Asgarian[†] **Paul Brown**[‡] **Johannes E. Gehrke**[†] **Dhaval N. Shah**[†]

Abstract

According to various trade journals and corporate marketing machines, we are now on the verge of a revolution—the object-relational database revolution. Since we believe that no one should face a revolution without appropriate armaments, this paper presents BUCKY, a new benchmark for object-relational database systems. BUCKY is a query-oriented benchmark that tests many of the key features offered by object-relational systems, including row types and inheritance, references and path expressions, sets of atomic values and of references, methods and late binding, and user-defined abstract data types and their methods. To test the maturity of object-relational technology relative to relational technology, we provide both an object-relational version of BUCKY and a relational equivalent thereof (i.e., a relational BUCKY simulation). Finally, we briefly discuss the initial performance results and lessons that resulted from applying BUCKY to one of the early object-relational database system products.

1 Introduction

The addition of object-relational features to relational database systems, which currently dominate the database marketplace, is arguably the most striking advance in RDBMS functionality since relational database systems were first introduced approximately 20 years ago [Sto96]. Many of the major players in the RDBMS industry have begun shipping either released products or early beta releases with some degree of object-relational functionality, and the rest are hinting that they will soon do the same. In addition, several companies, including both startups as well as established database vendors, have begun offering full object-relational database systems (or "universal servers," in RDBMS trade journal and advertising-speak). It is clear to most observers of the database industry that the new functionality offered by object-relational database systems will be of considerable benefit to their users, but surprisingly little is known about the performance implications of using these features. As a step towards rectifying this situation, we have defined and implemented the BUCKY[1] Benchmark.

1.1 BUCKY Objectives

In designing BUCKY, our objective has been to test the key features that add the "object" to "object-relational" database systems. There are five areas of object-relational query performance tested by BUCKY:

1. Queries involving row types with inheritance.
2. Queries involving inter-object references.
3. Queries involving set-valued attributes.
4. Queries involving methods of row objects.
5. Queries involving ADT attributes and their methods.

We discuss our rationale for choosing these features in detail in Section 2. Briefly, though, our philosophy was (a) to focus on the essential primitive differences between object-relational and relational database systems, and (b) to avoid testing functionality that is shared between object-relational and traditional relational systems, since that functionality is already tested by existing relational benchmarks (e.g., the Wisconsin Benchmark or TPC C and D)

The BUCKY Benchmark consists of an object-relational schema, a data generation program, and a set of queries over this schema. In this paper, we describe the BUCKY benchmark and discuss the sorts of insights that it can provide into object-relational DBMS performance. We have, in fact, run BUCKY on several existing systems, but the still-fresh memories of our OO7 benchmarking experience [CDN93, CDKN94] have curbed our appetite for trying to publish benchmark results for commercial systems. In addition, in our opinion, while the industry is on the verge of a "flood" of object-relational functionality, it is still somewhat early to try and publish meaningful comparative results.

One of the interesting implications of object-relational database systems is that, as compared to relational systems,

*Work supported by Informix, NCR; also by ARPA through order #017 monitored by the US Army Research Laboratory under Contract DAA 7307-92-6-Q509, by NASA under contracts #USRA-5555-17, #NAG-3895, #NAGW-4229, and by NSF Award IRI-9157357.
Michael Carey's current address: IBM Almaden Research Center. Dhaval Shah's current address: Cisco Systems, Inc.

[†]Computer Sciences Department, University of Wisconsin-Madison, {carey,dewitt,naughton,ma,johannes,dhaval}@cs.wisc.edu

[‡]Informix Corporation, brown@illustra.com

[1] *B*enchmark of *U*niversal or *C*omplex *K*wery *Y*nterfaces

object-relational technology greatly expands the space of alternatives available to a schema designer. For example, one can use an inheritance hierarchy or a set of independent tables; one can use inter-object references or key/foreign-key pairs; one can use set-valued attributes or store the elements of the logically embedded sets in a separate table with a foreign key "linking" them to their parent tuple. Moreover, this relational vs. object-relational design space expansion also applies to methods of ADTs and row objects in some cases. For example, what is logically a method on a row object can sometimes just be directly expressed in all relevant SQL queries rather than having a separate method. Similarly, what is logically an ADT with a set of associated methods can sometimes be expressed by "de-encapsulating" the ADT's internal structure into attributes of the containing SQL row types and converting the ADT's methods into expressions in SQL queries (at least for simple ADTs). Put differently, an application developer has a choice of just how "object-relational" to make the database and application code (i.e., of just how far to go in terms of utilizing the new features offered by these systems).

One of the goals of the BUCKY benchmark is to examine the performance tradeoffs between the different design alternatives offered in the brave new world of object-relational databases. Accordingly, in addition to the object-relational BUCKY schema, load program, and query set, we have also defined a relational BUCKY schema that is semantically equivalent (note Section 5.9) to the object-relational schema, implemented an appropriate load program, and defined a set of relational queries that are semantically equivalent to BUCKY's object-relational queries. By implementing both versions of the benchmark on one system—which is possible since, by definition, an object-relational database system supports full relational DDL and DML—one can compare and contrast the two approaches within a common software framework. This comparison is also important for another reason; the results of this comparison can provide insight into the relative maturity of object-relational (versus relational) query optimizers and runtime systems.

1.2 Preview of Paper

In the remainder of this paper, we will explain the rationale behind the design of the BUCKY benchmark, describe its schema, and present the object-relational and relational versions of the BUCKY query suite. As we go, we will discuss the sorts of lessons that can be learned from running BUCKY – some pertaining to the particular system being tested, with others pertaining to the general state of object-relational DBMS technology today – and we will present some preliminary results obtained by running BUCKY on the Illustra O/R DBMS. It is our hope that the BUCKY benchmark will serve as a tool both for driving and for measuring the improvements that will be needed to make object-relational technology a commercial success.

2 BUCKY Design Rationale

The features tested in BUCKY match the growing consensus in the database field as to what constitutes an object-relational system. One well-known exposition of this consensus appears in Stonebraker's book [Sto96]. Stonebraker lists four features that must be added to a relational DBMS in order for it to be considered an object-relational DBMS:

1. Inheritance.
2. Complex object support.
3. An extensible type system (ADTs).
4. Triggers.

Both the list of features tested by BUCKY and Stonebraker's list contain inheritance as their first item. The second through fourth items in the BUCKY feature list (references, object methods, and set-valued attributes) map to the second and third items in Stonebraker's list, although the mapping is not one-to-one. In more detail, each of references, object methods, and set-valued attributes can be considered part of complex object support; set-valued attributes also contribute to an extensible type system. The last item in the BUCKY list of tested features, ADT support, maps to the third item in Stonebraker's list. Finally, we differ on the last item in his list—the BUCKY benchmark includes no trigger tests. While we agree that advanced trigger support is a very important feature in a DBMS, we think that it is orthogonal to whether or not a system is object-relational (so we leave trigger benchmarking to others).

Some might question the value of defining a benchmark that only tests the object extensions in object-relational systems. After all, there are many aspects to overall application performance; the query performance of object extensions is only a part of the story. We agree. However, our goal for BUCKY was to provide a benchmark that is as specific as possible yet still captures the essence of what is missed by other, existing benchmarks. Object-relational systems provide a superset of relational functionality, so existing relational benchmarks (e.g., the Wisconsin, TPC-A through TPC-D, and Set Query benchmarks [Gra93]) can and should be used to test the relational subset of a given object-relational DBMS. Another important feature that we have also placed outside the scope of BUCKY, often the focus for OODB systems, is client-side "pointer traversal" performance. It is likely that application tools will provide this same kind of functionality on top of object-relational systems (inspired by wrapper tools for relational systems like those from Persistence and Ontos). Given the architecture of a typical object-relational system, the performance of such tools would be more a function of a caching layer on top of the DBMS than of the DBMS itself. Also, this functionality is well-tested by existing OODBMS benchmarks (e.g., OO1 [CS92] and OO7 [CDN93]). Thus, BUCKY does not attempt to test navigational program performance.

Finally, it should also be noted that, due to their extensible nature, object-relational database systems can be "customized" to provide specialized solutions designed for specific application domains or data types. This is perhaps best embodied by Illustra/Informix's "Data Blade" architecture, where for example one can buy a "sequence" data blade, a "GIS" data blade, or a "text" data blade. IBM has a corresponding family of "Database Extenders," and Oracle has a family of "Application Cartridges" with a similar purpose. We regard these data type-specific extensions as targets for their own benchmarks—the Sequoia 2000 benchmark is an example of such a benchmark for GIS data management systems—therefore, domain-specific testing is also outside the scope of the BUCKY benchmark. Results from BUCKY will provide insight into the base performance of an O/R DBMS, shedding light on its resulting ability to do well on a domain-specific benchmark, but domain-specific performance will also depend heavily on the quality of the particular data structures and algorithms employed in a given prepackaged solution.

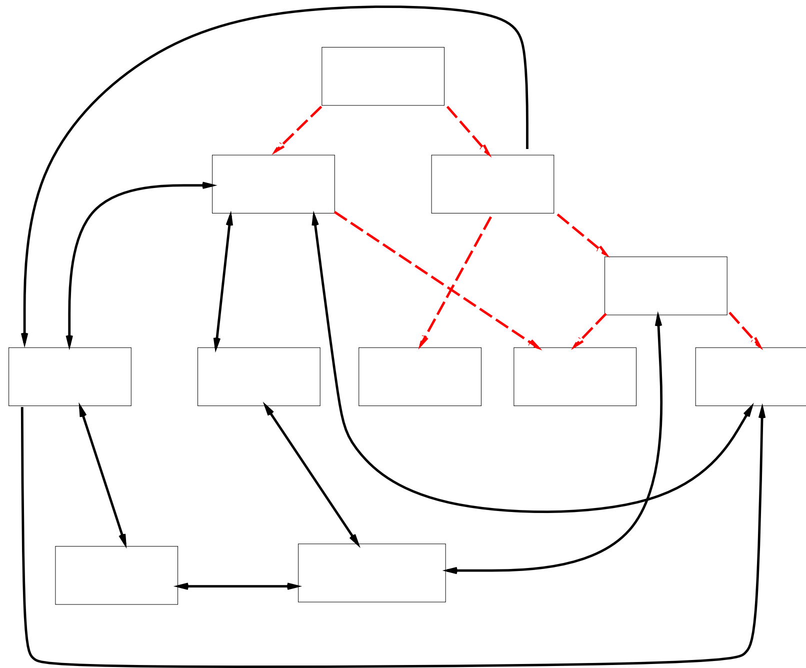

Figure 1: Schema Structure for BUCKY Benchmark.

3 BUCKY Database Description

The database for the BUCKY benchmark is modeled after a university database application. Figure 1 gives a graphical sketch of the schema. The lines in the figure from Person to Student, Person to Employee, Student to TA, Employee to Staff, Employee to Instructor, Instructor to TA, and Instructor to Professor represent inheritance among types. The remaining lines represent relationships between instances of types, and are labeled on each end with the name by which the relationship is known at that end. Though BUCKY is designed to be run on an object-relational system, as we mentioned earlier, it can also be run on a relational system by appropriately mapping its object features onto relational features. In this section we discuss the key features of both versions of the BUCKY schema in order to make sure that their designs are clear; details of each version can be found in the Appendices.

3.1 Inheritance

Using object-relational DDL, the natural way to model the information in BUCKY about university people is by having an inheritance hierarchy rooted at a person type. Thus, the object-relational BUCKY has a root row type called Person_t that contains the attributes common to all university-affiliated people. Person_t has two subtypes, Student_t and Employee_t, that add student- and employee-specific information. Employee_t has two subtypes, Staff_t and Instructor_t, that add information specific to non-instructional staff and instructors, respectively. Finally, there are two subtypes of Instructor_t, namely TA_t and Professor_t, as well; TA_t is also a subtype of Student_t (providing a test case for multiple inheritance). In a BUCKY database, there are no instances of the non-leaf (super) types, so these are like "abstract classes" in C++ parlance; only the leaf types actually have instances. What this means, for example, is that every instructor will either be a teaching assistant or a professor. In addition to these types, each of them has a corresponding table to hold its instances (i.e., Person_t has the Person table, Employee_t has the Employee table, Student_t has the Student table, and so on), and these tables are contained in a table/sub-table hierarchy that mirrors that of the type hierarchy. A complete SQL3-style description of the object-relational BUCKY schema is given in the full paper (tt http://www.cs.wisc.edu/ñaughton/bucky.html).

Since there is no direct way to model inheritance in relational DDL, we created a separate table for each non-abstract type in the hierarchy (Employee, Instructor, Student, and TA), repeating their common fields in each table definition. We felt this was the most natural mapping to use; a complete DDL description for this version of the BUCKY schema is given in the full paper. An alternative would have been to have a single table with the union of all the attributes in the hierarchy plus a type tag, with null values in attributes that do not apply to a particular row. However, we were worried that this approach would end up wasting too much space (depending on how null attributes are represented by the system). Another alternative would have been to use a "vertically decomposed" schema, where each subtype has a corresponding table that contains the key for its least specific supertype(s)'s table (e.g., Person, for Employee) plus only those attributes unique to the type (e.g., the Employee table under this scheme would have just four attributes: `id`, from Person, plus `dateHired`, `status`, and `worksIn`). However, we were concerned that this approach would require too many joins to reassemble objects. (It would be interesting to experiment with these other two mapping alternatives in the future.)

3.2 References

Another salient feature of object-relational DDL is its direct support for inter-object references. Among its attributes,

the Student_t type denotes a student's major using a reference to a row of type Department_t:

```
CREATE ROW TYPE Student_t (
    ...
    major Ref(Department_t),
    ...
)
UNDER Person_t;
```

The department row type has a corresponding "inverse" reference to the set of students majoring in that department:

```
CREATE ROW TYPE Department_t (
    ...
    majors Set(Ref(Student_t)),
    ...
);
```

It should be noted that the presence of such inverse sets is not strictly required, as the student/major relationship is fully captured by the reference contained in the Student_t type. However, we included this set in BUCKY anyway, as it is in the "spirit" of object-relational data modeling, which encourages the representation of (binary) relationships in a bi-directional manner. Unfortunately, unlike some OODB systems, which allow users to tell the system about the inverse nature of relationships of this sort, we are not aware of any current O-R product that has DDL support for making such assertions. SQL3 provides no such support either — in fact, support for collection-valued attributes was very recently moved out of SQL3. We retain them in BUCKY because current O-R products do provide support for them, and we therefore expect them to reappear as an SQL object extension in the not-too-distant future.

In relational DDL, the relationship is modeled as a key/foreign key pair:

```
CREATE TABLE Student (
    ...
    majorDept Integer REFERENCES Department,
    ...
);

CREATE TABLE Department (
    ...
    deptNo Integer NOT NULL PRIMARY KEY,
    ...
);
```

Since the relational model doesn't support set-valued attributes, there is no analogy in the relational case to the `majors` set in the object-relational schema's Department_t type. The relationship is less "directional" in the relational case, as reconstructing it via a query involves writing a join clause (`s.majorDept = d.deptNo`) that has no inherent directionality.

3.3 Sets

Another difference between the object-relational BUCKY schema and the relational version is the availability of set-valued attributes for storing sets of instances of base data types. For example, in object-relational BUCKY, the type definition for Person_t includes an attribute `kidNames` of type `Set(Varchar(10))`; it contains a set of strings, where each one is the name of one of the person's children. In the relational model, since there are no nested sets, this is modeled by adding an additional `Kids` table with an `id` attribute, which is a foreign key referencing the `Person` table, plus a `kidname` attribute, which is the string name of one of the referenced person's children.

3.4 Abstract Data Types

One of the key features of the O-R paradigm is an abstract data type (ADT) facility that enables users to define their own data types for columns of tables. These user-defined types can then be used in SQL commands, just like the built-in (system-defined) types, and users can also define their own functions to operate on ADT instances. To test this facility, the BUCKY schema includes a data type called LocationAdt which is equivalent to the following C++ class definition:

```
class LocationAdt {
  private:
    int lat;
    int lon;
  public:
    int extract_latitude() { return(lat); }
    int extract_longitude() { return(lon); }
    float distance(LocationAdt& loc)
    { return(sqrt((this->lat - loc->lat)**2 +
             (this->lon - loc->lon)**2)); }
}
```

Currently, different object-relational database systems take different approaches to supporting such ADTs. Some provide SQL3-style "value ADTs", where the structural content of an ADT is defined in SQL using a row-definition-like syntax, thus declaring its internal structure to the DBMS. Others provide "black box ADTs" instead, where the DBMS is given nothing more than total size information for each ADT. In the relational BUCKY schema, where no ADT support is assumed, we simply un-encapsulate the LocationAdt type; each of its two data elements becomes a field in each of the relational tables that has a LocationAdt field in its corresponding object-relational table (i.e., in Department, Staff, Professor, Student, and TA).

3.5 Methods

Most object-relational systems allow functions to be written either in SQL (for relatively simple functions) or in an external language like C or C++ (for more complicated functions). To test both flavors, BUCKY includes some functions written each way. The Person row object type and each of its subtypes have a salary function, and these functions are written in SQL. The three LocationAdt functions are written in C (but any external language is acceptable here). For the salary function, BUCKY demands late binding. E.g., for Employees that are Professors, the following function is called to compute their overall salary based on their 9-month academic year salary plus their degree of summer support:

```
CREATE FUNCTION salary(p Professor_t)
RETURNS numeric
RETURN p.AYSalary * (9 + p.monthSummer) / 9.0;
```

The definitions for each of BUCKY's SQL functions are given in the full paper, as are the SQL function signatures for each of the methods of LocationAdt. Since most SQL-based relational DBMSs do not provide an equivalent of ADTs or

Parameter Description	
Parameter	Value
NumStudents	50000
NumDepts	250
TAsPerDept	100
StaffPerDept	100
ProfsPerDept	100
KidsPerPerson	2.5
CoursesPerDept	50
SectionsPerCourse	2
SemestersPerSection	2
StudentsPerSection	20
CoursesPerStudent	2

Table Cardinalities	
Table	Cardin.
Student	50000
Department	250
TA	25000
Staff	25000
Professor	25000
Kids	116759
Course	12500
CourseSection	50000
Enrolled	150000

Figure 2: Parameter setting for populating the BUCKY database.

ADT functions, the relational version of the BUCKY benchmark stores the location data in two columns of the affected tables (as described above) and performs the salary computations directly in the relational versions of BUCKY's ADT test queries (which has the obvious disadvantage of de-encapsulating the details of the salary computations).

4 Experimental Setup

In this section, we explain how a target system should be set up in order to run BUCKY and obtain meaningful numbers.

The queries in the benchmark should be run "cold", that is, with the buffer pool being empty. Moreover, in enviroments where database pages can be cached in the operating system's file buffers, the file system cache should be cold as well. To flush the database buffer pool between queries, a huge table that is not used in the benchmark queries can be scanned. To flush the Unix buffer pool, a huge file that is not a part of the database should be scanned. We found in our experimentation that we were indeed able to generate repeatable query running times this way, so this strategy is effective. (This can be verified by running queries 10 times with flushing; the 10th time should match the first if no significant data caching is occurring between queries.)

The (self-explanatory) parameter settings shown in Figure 2 are to be used for populating the BUCKY database; we show both the parameter values and the resulting table sizes (in terms of the number of rows). While this is a relatively small data set, we have found it to be sufficient for generating interesting ORDBMS performance results and tradeoffs given the current state of the technology.

A few of the attribute value distributions are important to the BUCKY queries, so we mention them here. The kids for each person are generated by, for each person, (1) generating a number x between 0 and 99, then adding kids "girlname$x$" and "boyname$x$", then (2) with probability 1/4 generating another kid with name "girlname$y$", where y is randomly chosen between 100 and 1000, then with probability $(1/4)^2$ choosing another such girlname, etc. This means that everyone has at least one boy and girl, and that the boy and girl share the same numeric suffix on their names; 25% of the people have one additional girl, 12.5% have two additional girls, and so on.

The birth dates are uniformly distributed between 1940 and 1991. Salaries are more complex, since each subclass of Employee represents the salary in a different way (i.e., Staff have an annualSalary, TAs have a monthly salary and a percent time, and Professors have a 9-month salary plus some number of summer months). We generated these numbers so that Query 5, which asks for all employees making over $96000, returns about 5% of the Employees in the database.

Of course, indices should be created on the data in order to speed up the queries as much as possible. The strategy for creating indices should be to look at the benchmark, query by query, to determine (for each query) what indices will potentially improve their performance. It is legal to create the indices *after* the data has been bulk-loaded, so as not to slow down bulk-loading, and to report the bulk-loading time separately from the index creation time.

5 BUCKY Queries and Preliminary Results

This section describes the BUCKY benchmark's query set. As described earlier, we will present two sets of queries that should be run against the system—one set that exercises its object-relational (O-R) capabilities, and another set that uses just the relational subset of the system. As we describe each BUCKY query, we also explain its role in the benchmark.

5.1 SINGLE-EXACT: Exact-Match Over One Table

Find the address of the staff member with id 6966.

This is a simple exact-match lookup. The relational and O-R versions of this query look the same:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Staff e WHERE e.id = 6966;
```

This first test mainly serves to provide a performance baseline that can be helpful when interpreting results of later queries.

5.2 HIER-EXACT: Exact-Match Over Table Hierarchy

Find the address of the employee with id 6966.

In O-R SQL, this query—which must search the Employee table and its subtables—simply looks like:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Employee e WHERE e.id = 6966;
```

In relational SQL, searching all these types requires to explictly union the relational schema's separate tables, yielding:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Staff e WHERE e.id = 6966
    UNION ALL
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Professor e WHERE e.id = 6966
   UNION ALL
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM TA e WHERE e.id = 6966;
```

This tests the efficiency of the O-R system's handling of queries over subtable hierarchies, measuring the impact of the system's approach to scanning and indexing over hierarchies.

5.3 SINGLE-METH: Method Query Over One Table

Find all Professors who make more than 150000 per year.

In O-R SQL, the query involves invoking the salary method (whose body is written in SQL):

```
SELECT p.name, p.street, p.city, p.state, p.zipcode
FROM Professor p WHERE salary(p) >= 150000;
```

In relational SQL, there is no salary method, so the query is instead:

```
SELECT p.name, p.street, p.city, p.state, p.zipcode
FROM Professor p
WHERE (p.AYSalary * (9 + p.MonthSumer) / 9.0)
          >= 150000;
```

This test establishes the efficiency of the O-R system's approach to indexing on function results (as compared to indexing on stored relational attributes).

5.4 HIER-METH: Method Query Over Table Hierarchy

Find all Employees who make more than 96000 per year.

The query returns about 18% of the Staff, TA, and Professor objects (13191 tuples). The salaries of professors are uniformly distributed between 30K and 129K and the salary of tas are uniformly distributed between 10K and 19K (sigh!). In O-R SQL, the query is again a clean-looking call to the salary function; recall that the implementation of this function is different for the various employee subtypes:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Employee e WHERE salary(e) >= 96000;
```

In relational SQL, the method computation must be embedded in the query, which again involves an explicit union:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Staff e
WHERE e.annualSalary >= 96000           UNION ALL
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Professor e
WHERE (e.AYSalary * (9 + e.MonthSummer) / 9.0)
      >= 96000                          UNION ALL
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM TA e
WHERE (apptFraction * (2 * e.semesterSalary))
      >= 96000;
```

This tests the O-R system's handling of indexing on function results in the presence of a table hierarchy.

5.5 SINGLE-JOIN: Relational Join Query

Find all Staff with the same birthdate who live in an area with the same zipcode

This is a fairly traditional relational join. In O-R SQL, this looks as follows (the oid predicate prevents each satisfying staff member pair from appearing twice):

```
SELECT s1.id, s1.name, s1.city,
       s2.id, s2.name, s2.city
FROM Staff s1, Staff s2
WHERE s1.birthDate = s2.birthDate AND
      s1.zipcode = s2.zipcode AND s1.oid < s2.oid;
```

The relational SQL query is almost identical (with ids instead of oids):

```
SELECT s1.id, s1.name, s1.city,
       s2.id, s2.name, s2.city
FROM Staff s1, Staff s2
WHERE s1.birthdate = s2.birthdate AND
      s1.zipcode = s2.zipcode AND s1.id < s2.id;
```

This is the baseline test for join processing, hopefully verifying that the O-R query is just as efficient as the relational query for regular joins.

5.6 HIER-JOIN: Relational Join Over Table Hierarchy

Find all persons with the same birthdate who live in the same zipcode aera.

This is the same query, but over the hierarchy. In O-R SQL, it looks like:

```
SELECT p1.id, p1.name, p1.city,
       p2.id, p2.name, p2.city
FROM Person p1, Person p2
WHERE p1.birthDate = p2.birthDate AND
  p1.oid < p2.oid AND p1.zipcode = p2.zipcode;
```

In relational SQL, Query HIER-JOIN is ten-way union query; each of the ten arms of the union consists of a join between a pair of the tables that hold subtypes of Person (Professors, Students, TAs and Staff) in the relational database. (Due to the length of this query, its statement is not shown.)

This test investigates the efficiency of the O-R system's handling of joins between table hierarchies.

5.7 SET-ELEMENT: Set Membership

Find all Staff who have a child named "girl16."

The kidName values in the database are such that this query returns about 2% percent of the Staff objects (495 people). In O-R SQL, this query is simple; it just tests for membership of 'girl16' in the nested kidName set:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Staff e WHERE 'girl16' IN e.kidNames;
```

In relational SQL, this query involves a join with the table needed to normalize this data in the relational case; the DISTINCT clause in the relational version is needed to force the same semantics as in the O-R query, where each Staff tuple will be output at most once:

```
SELECT DISTINCT e.name, e.street, e.city,
       e.state, e.zipcode   FROM Staff e, Kids k
WHERE e.id = k.id AND k.kidName = 'girl16';
```

This query tests the O-R system's handling of nested sets. As we have mentioned, nested sets have recently been eliminated from SQL3; we are leaving this query in the benchmark because vendors support it and we think users want it. A select/join is required in the relational case, so if the O-R system supports indexing on set-valued attributes, it has an opportunity to win here.

5.8 SET-AND: And'ed Set Membership

Find all Staff who have children named "girl16" and "boy16."

This query also returns about 2% percent of the Staff objects. In O-R SQL, this query is straightforward:

```
SELECT e.name, e.street, e.city, e.state, e.zipcode
FROM Staff e
WHERE 'girl16' IN e.kidNames
  AND 'boy16' IN e.kidNames;
```

In relational SQL, this query again involves joins:

```
SELECT DISTINCT e.name, e.street, e.city,
       e.state, e.zipcode
FROM Staff e, Kids k1, Kids k2
WHERE e.id = k1.id AND e.id = k2.id AND
  k1.kidName = 'girl16' AND k2.kidName = 'boy16';
```

This is a slightly more complex test of the O-R system's handling of queries involving nested set attributes.

5.9 1HOP-NONE: Single-Hop Path, No Selection

Find all student/major pairs

This is the first of BUCKY's path expression test queries. It returns all students and teaching assistants (75000 persons in all).

In O-R SQL, this query is easily written as:

```
SELECT s.id, s.name, s.state, s.major->dno,
       s.major->name, s.major->building
FROM Student s;
```

In relational SQL, it becomes a union of two joins:

```
SELECT s.id, s.name, s.state,
       d.dno, d.name, d.building
FROM Department d, Student s
WHERE s.majorDept = d.deptNo      UNION ALL
SELECT s.id, s.name, s.state,
       d.dno, d.name, d.building
FROM Department d, TA s
WHERE s.majorDept = d.deptNo
```

This tests the efficiency of the O-R system at processing queries that involve path expressions. A well-implemented O-R system should be able to handle the O-R and relational cases with more or less equal efficiency.

We need to point out here that, strictly speaking, O-R path expressions are equivalent to relational systems' left outer joins, not inner joins. Despite this, we explicitly chose to use regular joins in the relational case. The reason for this decision is that, as was mentioned in Section 3, we know that the BUCKY database contains no dangling relationships. Given this knowledge about the database, we have simply written the given query in its most convenient and natural form in each case (i.e., using the most natural O-R and relational formulations).

It is also important to notice that the relational version of this query explicitly encodes more "information" than the object-relational version, as the relational version names both the source and target tables of the relationships involved in this query. In the object-relational case, the information about which tables contain the target objects of references is instead encoded in the schema as reference scope information (as mentioned in Section 3). (In fact, the SQL3 committee recently voted to remove unscoped reference from the standard; we will see one reason for this when we examine the performance results that we obtained by running BUCKY on an object-relational product that pre-dates this decision.)

5.10 1HOP-ONE: Single Hop Path, One-Side Selection

Find the majors of students named "studentName9000".

This query pairs students and department with a selection on student name.

In relational SQL, the query looks like this:

```
SELECT s.id, s.name, d.deptNo, d.name
FROM Student s, Department d
WHERE s.majorDept = d.deptNo AND
      s.name = 'studentName9000'
   UNION ALL
SELECT s.id, s.name, d.deptNo, d.name
FROM TA s, Department d
WHERE s.majorDept = d.deptNo AND
      s.name = 'studentName9000';
```

Note that the union is necessary since a student may either be a TA or a "regular" student.

In O-R SQL, there are two ways to express this. The first, variant A, starts the query from the students and follows the path to their major department, which looks like:

```
SELECT s.id, s.name, s.state,
       s.major->dno, s.major->name,
       s.major->building
FROM Student s WHERE s.name = 'studentName9000';
```

The second, variant B, starts from the departments and follows their (sets of) pointers toward the department's majors. This is a selection on the target of a set-valued reference, and is a bit obtuse due to the SQL3 "everything in the FROM clause is a table" view of the world:

```
SELECT m->id, m->name, m->state,
       d.dno, d.name, d.building
FROM Department d, TABLE(d.majors) t(m)
WHERE m.majors->name = 'studentName9000';
```

Variant A tests the O-R system's handling of short path expressions with predicates on the originating table. Variant B tests the O-R system's efficiency at handling queries involving nested sets of references. (It is also a case where inverse relationships, if supported, could be exploited very effectively due to the nature of the selection predicate.)

5.11 1HOP-MANY: One-Hop Path, Many-Side Selection

Find all students majoring in Department 7.

In relational SQL there is again only one version:

```
SELECT s.id, s.name, d.deptNo, d.name
FROM Student s, Department d
WHERE s.majorDept = d.deptNo
  AND d.name = 'deptname7'
      UNION ALL
SELECT s.id, s.name, d.deptNo, d.name
FROM TA s, Department d
WHERE s.majorDept = d.deptNo
  AND d.name = 'deptname7';
```

Again, there are two ways to express this in O-R SQL. The first, variant A, starts from departments and follows the path to students; this is a selection on the source of a set-valued reference:

```
SELECT m->id, m->name, m->state,
       d.dno, d.name, d.building
FROM Department d, TABLE(d.majors) t(m)
WHERE d.name = 'deptname7'
```

The second, variant B, starts from students and follows the path toward their major departments. This is a selection on the target of a scalar reference, which in O-R SQL looks:

```
SELECT s.id, s.name, s.state, s.major->dno,
       s.major->name, s.major->building
FROM Student s WHERE s.major->name = 'deptname7';
```

Variant B tests the O-R system's handling of queries with path expressions whose target table is restricted by a predicate. With the selection predicate on the path's target table rather than its originating table, an O-R system that handles path queries naively—e.g., to failing to make use of scope information, or failing to reorder path expressions like joins—will likely do poorly on this test. As with the previous test, inverse relationship exploitation is possible (and can be advantageous) on this test.

5.12 2HOP-ONE: Two-Hop Path, One-Side Selection

> *Find the semester, enrollment limit, department number, and department name for all sections of courses taught in room 69.*

In O-R SQL there are many ways to express this query. Like Query 2HOP-NONE, it involves a join of three tables. We can start to follow references either from course sections, courses, or departments. We chose not to start with courses since it seemed unlikely (i.e., awkward) for a user to express the query that way. In variant A, we start from course sections and follows the path through Course to Department. This is a selection on the source of a two-hop chain of scalar valued references. Variant A is thus quite simple-looking:

```
SELECT x.semester, x.noStudents,
   x.course->dept->dno, x.course->dept->name
FROM CourseSection x WHERE x.roomNo = 69;
```

The second O-R variant starts from departments and follows the path through course to course sections. This is a selection on the target of a two-hop chain of set-valued references. Variant B looks like:

```
SELECT x->semester, x->noStudents, d.dno, d.name
FROM Department d, TABLE(d.coursesOffered) t1(c),
     TABLE(c.sections) t2(x)
WHERE x->roomNo = 69;
```

In relational SQL, there is only one variant:

```
SELECT x.semester, x.roomNo, d.deptNo, d.name
FROM CourseSection x, Course c, Department d
WHERE x.deptNo = c.deptNo
  AND x.courseNo = c.courseNo
  AND c.deptNo = d.deptNo AND x.roomNo = 69;
```

This tests the O-R system's handling of path queries with longer paths.

5.13 ADT-SIMPLE: Simple ADT Function

> *Find the latitudes of all staff members.*

We now turn our attention to testing ADT support, starting with the very simple case of a query that has a function invocation in its SELECT list. In object relational SQL, the query looks like

```
SELECT extract_latitude(s2.place)
FROM Staff s2;
```

In relational SQL, where the ADT has been "unencapsulated," we have:

```
SELECT e.latitude
FROM Staff e;
```

This tests the efficiency of the O-R system's function dispatch mechanism (versus the efficiency of retrieving stored data).

5.14 ADT-COMPLEX: Complex ADT Function

> *For each Staff member, find the distance between him and the staff member with id 6966.*

This query applies a more complex ADT function; in object relational SQL, it looks like:

```
SELECT distance(s1.place, s2.place)
FROM Staff s1, Staff s2 WHERE s1.id = 6966 ;
```

In relational SQL, the computation must be spelled out completely in SQL:

```
SELECT SQRT((s1.latitude - s2.latitude)*
            (s1.latitude - s2.latitude)
          + (s1.longitude - s2.longitude)*
            (s1.longitude - s2.longitude))
FROM Staff s1, Staff s2 WHERE s1.id = 6966;
```

This again tests the O-R system's function dispatch mechanism, but this time it does so versus a case where the relational case's expression is quite complex.

5.15 ADT-SIMPLE-EXACT: Exact-Match on an ADT

> *Find the ids of the Staff who live at latitude of 34 and a longitude of 35*

In this query, we are looking for a particular point, which is an exact match. In object relational SQL, the query looks like

```
SELECT s.id
FROM Staff s WHERE s.place = LocationADT(34, 35);
```

In relational SQL, it looks like:

```
SELECT s.id
FROM Staff s
WHERE s.latitude = 34 AND s.longitude = 35;
```

This tests the O-R system's efficiency at handling an exact match query involving an ADT (which requires ADT indexing support).

5.16 ADT-COMPLEX-RANGE: Range on Complex ADT Function

Find the ids and names of Staff whose ids are less then 1500 and are at a distance of 500 units from each other.

We now try a more complex ADT query, which in O-R SQL is:

```
SELECT s1.id, s1.name, s2.id, s2.name
FROM Staff s1, Staff s2
WHERE distance(s1.place, s2.place) < 500
 AND s1.id < 1500 AND s2.id < 1500
 AND s1.id < s2.id;
```

In relational SQL, it looks like:

```
SELECT s1.id, s1.name, s2.id, s2.name
FROM Staff s1, Staff s2
WHERE SQRT((s1.latitude - s2.latitude)*
           (s1.latitude - s2.latitude) +
           (s1.longitude - s2.longitude)*
           (s1.longitude - s2.longitude)) < 500
  AND s2.id < 1500 and s1.id < s2.id
  AND s1.id < 1500;
```

This tests the O-R system's efficiency at handling a range query involving an ADT.

5.17 Other Queries Considered

In addition to the queries described here, we also considered including a number of other test queries. However, these other queries were eliminated because, when running BUCKY against an actual O-R system, we found that their results simply reinforced those that we already presented. The other queries that we tested include: SINGLE-RANGE (Range Query Over Single Table) and HIER-RANGE (Range Query Over Table Hierarchy), whose results were similar to the corresponding exact-match queries; SET-OR (Or'ed Set Membership), the results of which were similar to the other set queries; 1HOP-BOTH (Single-Hop Path, Double-Ended Selection), whose results were essentially predictable based on the corresponding pair of single-ended selections; 2HOP-NONE (Two-Hop Path, No Selection), 2HOP-MANY (Two-Hop Path, Many-Side Selection), and 2HOP-BOTH (Two-Hop Path, Double-Ended Selection), which largely reinforced the corresponding single-hop query results; and, finally, ADT-SIMPLE-RANGE (Range on Simple ADT Function), which produced results similar to those of ADT-SIMPLE-EXACT.

6 Initial BUCKY Results and Lessons

In this section, we briefly describe our preliminary experience in applying the BUCKY benchmark to an actual system – one of the early object-relational products. The system that we tested is Illustra, now owned by Informix.

6.1 Loading the BUCKY Database

A big difference between implementing BUCKY in the relational and object-relational models arose when generating and bulk-loading the input files for the database. Doing this was *much* harder for object-relational data due to the presence of references. Basically, O-R systems make querying simpler by preconnecting objects according to relationships declared in the schema; while queries indeed become more concise in most cases, we quickly discovered that loading becomes both more complex and more time-consuming as a result.

Our approach to loading was to first generate external data files that were then loaded into the database system using its bulk-loading facility. We did this for portability and uniformity reasons: the load files are generated by a stand-alone C++ program, and hence can be used by anyone, ensuring that others will be able to use the exact same input data set. These files can then be bulk-loaded into any DBMS (perhaps after some minor syntactic tweaking to match the field and tuple delimiters used by the particular DBMS's bulk-loading facility).

Our approach to loading was straightforward for the relational BUCKY database, but proved much more difficult in the object-relational case. To see why, consider generating the load file for the Students table. Each student has, among other things, an associated major. For relational systems, the data for a particular student simply includes the department id of the student's major department; as a result, when generating the Department load file, we don't need to know who the department's majors are, as this information is already captured in the Student table and can be recovered later using join queries. In contrast, consider generating the load files for the same two tables (Student and Department) in the object-relational case. First, rather than holding the key of the major department, the student data must now include the OID (object identifier) of the student's major department. Unfortunately, since the major department object has not yet been created, there is no way to know what this OID may eventually be. Current O-R systems address this problem by allowing the use of a *surrogate* for the department object at load time; this surrogate is a temporary, external OID that the system later replaces with the actual OID later during the loading process. Although providing support for such surrogate OIDs makes bulk-loading possible, the problem of generating and managing surrogate OIDs when preparing the data for bulk-loading is far from trivial.

To illustrate the "joys" of loading an O-R database, suppose we are now generating the Department data file. When we come to the 4095th department object, it must include a reference to each student that has this department as a major. We can use the surrogate OIDs of the student objects; the set of surrogates for this particular department might be 56, 157, 3100, the surrogates for the 56th, 157th, and 3100th students. Unfortunately, this means that we must *remember* the association between these students and their departments from the time when we generate the students until the time when we generate their corresponding departments. If we are working with a large database, the number of such associations can be huge, making the data structure needed to store these associations larger than memory. At best, paging of this data structure would make data generation impossibly slow; at worst, its size will exceed the available swap space and the program won't finish running at all. Note that interchanging the generation order of the Department and Student tables won't help, as there is a cyclic dependency between them.

To solve this problem, we used a C++ program that generates relational load files, followed by a series of smaller C++ programs, awk programs, and calls to the Unix sort and join utilities to munge this output into an object-relational load file. As mentioned above, the relational load files include information about which rows are related to which other rows by using key-foreign key pairs; for exam-

Query	R	O-R
SINGLE-EXACT	0.23	0.28
HIER-EXACT	0.25	0.40
SINGLE-METH	3.58	0.67
HIER-METH	11.49	18.73
SINGLE-JOIN	11.25	11.33
HIER-JOIN	140.1	187.2
SET-ELEMENT	5.8	23.7
SET-AND	2.5	24.0
1HOP-NONE	50.9	95.0/39.7
1HOP-ONE	0.30	0.29/0.26
1HOP-MANY	2.32	23.96/6.26
2HOP-ONE	4.95	2.12/1.74
ADT-SIMPLE	5.97	6.43
ADT-COMPLEX	9.43	5.92
ADT-SIMPLE-EXACT	0.20	0.24
ADT-COMPLEX-RANGE	39.6	22.5

Figure 3: Measured times in seconds for BUCKY queries. (Path expression results shown as UNSCOPED/SCOPED pairs of times).

ple, the relationship between a given Student row and its corresponding major Department is represented by storing the department's key in the student tuple. The load munging programs have to replace this representation by putting a reference to the Student tuple in the "majors" set of the Department row and a reference to the Department tuple in the Student row. This can be accomplished by joining the Department and Student class, and then filling in the references with surrogate OIDs instead of writing out joined tuples. This process was implemented as a sort-merge join using the Unix "sort" and "join" utilities. We used a similar approach for the other references in the BUCKY schema as well. It is worth noting that much of this effort would have been unnecessary if object-relational database systems (and SQL3!) supported the notion of bi-directional relationships – we could then have declared the students' major and departments' majors attributes to be inversely related, explicitly linking the data in only one direction (as in the relational case), leaving it to the system to fill in the reverse direction.

Finally, the amount of loading work (I/O and CPU time) that must be done by the system is greater in the O-R case as well, as the O-R system must assign each object a real OID and then replace all uses of that object's surrogate OID with the newly assigned real OID [WN94, WN95]. This extra work was dramatically visible in the loading times that we saw when preparing the BUCKY database; loading took many times longer in the object-relational case (even when the "munging" time required to prepare the O-R input files was excluded).

6.2 Running the BUCKY Queries

In describing the BUCKY queries in Section 5, we indicated briefly what each one was intended to test. Here we present preliminary results that were obtained by running BUCKY on version 3.2 of Illustra, a first-generation O-R database system product. The results reported here were measured at Informix; they took the initial Illustra implementation of BUCKY produced at the University of Wisconsin and improved it in several ways. Some of the improvements that they made over our initial version (in addition to using a faster hardware platform!) were to create proper function indexes and to load all ADT functions statically (rather than dynamically) into the engine prior to running the benchmark queries. Table 3 lists both the relational (R) and object-relational (O-R) BUCKY results.

For queries SINGLE-EXACT and SINGLE-JOIN, which are just relational queries over one table, the R and O-R times are essentially identical, as one would expect. The O-R times for the corresponding queries HIER-EXACT and HIER-JOIN are somewhat worse than the relational times. These differences are due to an O-R query optimizer bug (the optimizer sometimes fails to correctly choose an index-based plan in the presence of a table hierarchy) in the version of Illustra on which we ran the tests.

We now turn to the method queries SINGLE-METH and HIER-METH. Comparing the R and O-R times for query SINGLE-METH shows the large gains that O-R support for indices on functions can provide. The O-R system is able to execute this query by doing an index lookup on the employee salary function, whereas the complexity of the query predicate in the relational case (where the predicate essentially includes an in-query expansion of the O-R function body) forces a query plan that involves a sequential scan. The same performance advantage for O-R should be seen for HIER-METH, but it isn't; instead, the O-R time is actually worse in this case. This is also due to the optimizer bug mentioned above (which causes the plan based on the functional index to be missed).

Next we look at the set queries, SET-ELEMENT and SET-AND. In both cases, the relational version – which involves a join – is significantly faster than the O-R version, showing that the O-R system's handling of nested sets could be improved.

We now come to the path queries. Two O-R times are shown for 1HOP-NONE. The first O-R time (95.0 seconds) is worse than the relational time (50.9 seconds) and resulted from writing the O-R query using a path expression. The reason for the lower O-R performance is that this system does not yet support scoped references[2], as the system was built before the notion of scoped references was added to SQL3. Thus, although the system knows from the schema that the field `s.major` points to an object of type `DepartmentObj`, it has no way of knowing that the target object is in the `Department` table. Consequently, it has to revert to what amounts to a nested-loops join (scanning the `Student` table and following the `major` pointer for each student tuple). Since it is of questionable fairness to compare the performance of an explicitly scoped relational join with an unscoped pointer join, we also include another O-R time (39.7 seconds) in the table. This time was obtained by simulating what an O-R system with support for scoped references would do by explicitly rewriting the path query as an explicit OID-join (i.e., as a join between the `Student` and `Department` tables, just like the relational query but with a join predicate of `s.major = d.oid`). Doing so led to an O-R time that beat the corresponding relational time (due to a better query plan being selected by the optimizer in the second O-R case than in the relational case).

Alternatively, we could have implemented an unscoped join in the relational system. One way to do this would be to replace the "major" attribute with a pair of attributes (tableName, majorDept) then "decode" this pair of attributes in a client application. This would be impossibly slow; we

[2]It supports only unscoped references, which are strictly more powerful but also much more costly in terms of performance in some cases, as we will see here.

didn't test it because we have no notion of a client application anywhere else in the benchmark.

The next path query is 1HOP-ONE. Recall that for the select-join queries, we looked at two ways of expressing each query in O-R SQL; this is because given our schema there are two directions in which to follow each relationship. These two variants are not the source of the pair of numbers in Figure 3; as mentioned previously, the two numbers in the figure are due to the scoped/unscoped option.

In the case of 1HOP-ONE, the O-R times shown are for variant A, where the query is written as a path expression going from students to departments. The two O-R times and the relational time are all more or less identical due to the fact that the three cases all allow the selective student name predicate to be applied first; in this case, the lack of scope information is not a problem. Variant B, which traverses the relationship in the opposite direction using the department's set of majors, is not shown. Illustra's times were slower in this case because it uses nested sets and unscoped references, so there was no point in including these times (though they should be included in tests of systems that support scoped references and nested sets); note that a join-based rewrite of variant B would be the same as that for variant A. Lastly, note that inverse relationship information would allow a system to choose to use forward-traversal plans rather than set-traversal plans when appropriate, which would help here, but neither current O-R systems or SQL3 provide any such support.

For the next query, 1HOP-MANY, the results shown are for variant B, the reference traversal variant; as for query 1HOP-ONE, we omit the times for the variant that traverses through a set of unscoped references, but plan on including it in the benchmark when systems provide scoped references. The relational time for the path variant of 1-HOP-MANY is better than both O-R times here. In the unscoped case, the lack of scope information forces the system to apply the selection last, so the unscoped path query cost is high here. The OID-join version performs much better, though still not as well as the relational version in this case.

The last path query is 2HOP-ONE; again, we show only the forward path traversal results, omitting the traversal in the reverse (set of reference) direction. This query once again involves unscoped references, but O-R performance is quite good despite this due to the highly selective predicate on `roomNo`. In this case, the relational version is slower than both of the O-R versions of the query.

The last group of queries involves use of ADTs and their functions. The ADT-SIMPLE results show a slight overhead for function invocation in the O-R case, as the function body is extremely simple, while the results for ADT-COMPLEX show that ADT function performance beats relational expression evalution for more complex functions. The O-R and R times are essentially identical for query ADT-SIMPLE-EXACT. Finally, the O-R time is better for query ADT-COMPLEX-RANGE; it is clear that the O-R system can take advantage of the ADT in this case.

6.3 Reporting the Bottom Line

In our previous benchmarks, we have avoided the idea of boiling an entire benchmark down to a single number, but it is too much fun not to compress the results somehow. We still believe that a full set of results is by far the best performance profile of a system, but as a challenge to implementors of O-R systems everywhere, we are defining two bottom-line metrics for the BUCKY Benchmark:

1. The BUCKY O-R Efficiency Index.

   This number measures the relative performance of the system's O-R and relational functionality. It is defined to be $G(OR)/G(R)$, where $G(OR)$ is the geometric mean of all object-relational test times and $G(R)$ is the geometric mean of all relational test times.

2. The BUCKY O-R Power Rating.

   This measures the absolute performance of the system's O-R functionality, and is simply $100.0/G(OR)$.

The O-R Power Rating is useful only when comparing two object-relational systems—if system A has a higher power rating than system B, then system A is in some sense "faster" than B. The O-R efficiency index, in contrast, is interesting within a single system. For Illustra, if we omit the set-valued attribute and set-of-reference queries (the ones that have recently been dropped from SQL3), and use the OID-join encoding of the scoped reference queries, the O-R efficiency index is 0.9.

We anxiously await the first O-R system that can get an O-R efficiency rating < 1.0 based on reporting times for all of the queries (without rewrites), indicating that it successfully ran the full O-R version of the BUCKY queries faster than the relational version. For anyone who would like to try, the loading programs and queries are freely available from the database area at the UW CS department web site (`http://www.cs.wisc.edu/ñaughton/bucky.html`).

7 Conclusions

In this paper, we have presented BUCKY, a benchmark for object-relational database systems. BUCKY is a query benchmark that tests the object features offered by object-relational systems, including row types and inheritance, references and path expressions, sets of atomic values and of references, methods and late binding, and user-defined abstract data types and their methods. To help evaluate the current state of the O-R art, we presented both object-relational BUCKY and a relationally mapped simulation thereof, and we strongly advocate running both versions against the same O-R engine. We discussed the lessons that we learned by running BUCKY on an early O-R product; the results highlighted a number of issues related both to current products and to object-relational technology (a la SQL3) in general.

While we expect the BUCKY benchmark to continue to evolve, our initial BUCKY experience, in a nutshell, indicates that object-relational technology is a double-edged sword today. For the most part, the queries are much more naturally and concisely expressible using the power of the object-relational model and SQL extensions. However, at least today, this greater expressive power does not come for free. For example, we found that loading an object-relational database is far more challenging than loading an information-equivalent relational database; inverse relationship support would have helped here. In addition, the new SQL language features that O-R systems offer—such as references, sets, inheritance, methods, and ADTs—provide new implementation challenges for implementors of DBMS engines. We saw that a number of the BUCKY queries currently run faster on the relational version of BUCKY, particularly those involving sets, and we clearly saw one of the reasons that SQL3 advocates the exclusive use of scoped references whenever possible.

Stonebraker refers to object-relational technology as "the next great wave" [Sto96], and it is clear from the activity in the industry that this wave is starting to wash over us today. It is our hope that BUCKY will be useful over the next few years as this wave continues—both for customers of this technology, so they can tell when O-R systems are ready for deployment in their applications, and for its developers, to provide a forcing function for improving the current state of the art. To this end, we have offered two BUCKY performance metrics—the O-R Efficiency Index, for comparing O-R and relational implementations of BUCKY, and the O-R Power Rating, for comparing O-R systems.

References

[CDKN94] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proceedings of the ACM OOPSLA Conference*, pages 414–426, Portland, OR, October 1994.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, Washington D.C., May 1993.

[CS92] R. Cattell and J. Skeen. Object operations benchmark. *ACM Transactions on Database Systems*, 17(1), March 1992.

[Gra93] Jim Gray. *The Benchmark Handbook*. Morgan Kaufmann, San Mateo, CA, 1993.

[Sto96] Michael Stonebraker. *Object-Relational Database Systems: The Next Wave*. Morgan Kaufmann, 1996.

[WN94] Janet L. Wiener and Jeffrey F. Naughton. Bulk loading into an oodb: A performance study. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 120–131, Santiago, Chile, August 1994.

[WN95] Janet L. Wiener and Jeffrey F. Naughton. Bulk loading revisited. In *Proceedings of the 21th International Conference on Very Large Data Bases*, Zurich, Switzerland, August 1995.

A Object-relational BUCKY Schema

```
-- Row types and their tables

CREATE ROW TYPE PersonObj (
  id Integer, name Varchar(20),
  street Varchar(20), city Varchar(10),
  state Varchar(20), zipcode Char(5),
  birthDate Date,
  kidNames Set(Varchar(10)),
  picture Char(100),
  place LocationAdt
);
CREATE TABLE Person OF ROW TYPE PersonObj ...;

CREATE ROW TYPE StudentObj (
  studentId Char(10),
  major Ref(DepartmentObj),
  advisor Ref(ProfessorObj),
  hasTaken Set(Ref(EnrolledObj))
) UNDER PersonObj;
CREATE TABLE Student
  OF ROW TYPE StudentObj UNDER Person ...;

CREATE ROW TYPE EmployeeObj (
  DateHired Date,
  status Integer, salary Real virtual,
  worksIn Ref(DepartmentObj)
) UNDER PersonObj;
CREATE TABLE Employee
  OF ROW TYPE EmployeeObj UNDER Person ...;

CREATE ROW TYPE StaffObj (
  annualSalary Integer
) UNDER EmployeeObj;
CREATE TABLE Staff
  OF ROW TYPE StaffObj UNDER Employee ...;

CREATE ROW TYPE InstructorObj (
  Teaches Set(Ref(CourseSectionObj))
) UNDER EmployeeObj;
CREATE TABLE Instructor
  OF ROW TYPE InstructorObj UNDER Employee ...;

CREATE ROW TYPE ProfessorObj (
  AYSalary Integer, monthSummer Integer,
  advisees Set(Ref(StudentObj))
) UNDER InstructorObj;
CREATE TABLE Professor
  OF ROW TYPE ProfessorObj UNDER Instructor ...;

CREATE ROW TYPE TAObj (
  semesterSalary Integer, apptFraction Real
) UNDER InstructorObj, StudentObj;
CREATE TABLE TA
  OF ROW TYPE TAObj UNDER Instructor, Student ...;

CREATE ROW TYPE CourseObj (
  cno Integer, name Varchar(20),
  dept Ref(DepartmentObj),
  credits Integer,
  sections Set(Ref(CourseSectionObj))
);
CREATE TABLE Course OF ROW TYPE CourseObj ...;

CREATE ROW TYPE CourseSectionObj (
  course Ref(CourseObj),
  semester Integer, textbook Varchar(20),
  noStudents Integer, building Varchar(10),
  roomNo Integer,
  teacher Ref(InstructorObj),
  students Set(Ref(EnrolledObj))
);
CREATE TABLE CourseSection
  OF ROW TYPE CourseSectionObj ...;

CREATE ROW TYPE DepartmentObj (
  dno Integer, name Varchar(20),
  building Varchar(10), budget Integer,
  coursesOffered Set(Ref(CourseObj)),
  chair Ref(ProfessorObj),
  employees Set(Ref(EmployeeObj)),
  majors Set(Ref(StudentObj)),
  place LocationAdt
```

```
);
CREATE TABLE Department
  OF ROW TYPE DepartmentObj ...;

CREATE ROW TYPE EnrolledObj (
  student Ref(StudentObj),
  section Ref(CourseObj),
  grade Char(2)
);
CREATE TABLE Enrolled
  OF ROW TYPE EnrolledObj ...;

-- Salary function for employee types

CREATE FUNCTION salary(e EmployeeObj)
  RETURNS Real  AS RETURN (0);

CREATE FUNCTION salary(i InstructorObj)
  RETURNS Real  AS RETURN (0);

CREATE FUNCTION salary(p ProfessorObj)
  RETURNS Real AS RETURN
  (p.AYSalary * (9 + p.monthSummer) / 9.0);

CREATE FUNCTION salary(s StaffObj)
  RETURNS Real AS RETURN (s.annualSalary);

CREATE FUNCTION salary(t TAObj)
  RETURNS Real AS
  RETURN (t.apptFraction * 2 * t.semesterSalary);

-- ``Black box'' location ADT and functions

CREATE ABSTRACT DATA TYPE LocationAdt (...);

CREATE FUNCTION extract_latitude(l LocationAdt)
  RETURNS Integer AS ...;
CREATE FUNCTION extract_longitude(l LocationAdt)
  RETURNS Integer AS ...;
CREATE FUNCTION distance(l1 LocationAdt,
                         l2 LocationAdt)
  RETURNS Real AS ...;
```

B Relational BUCKY Schema

```
-- Relational tables

CREATE TABLE Staff (
  id Integer NOT NULL PRIMARY KEY,
  name Varchar(20),
  street Varchar(20),
  city Varchar(10),
  state Varchar(20),
  zipcode Char(6),
  birthDate Date,
  picture Char(100),
  latitude Integer,
  longitude Integer,
  dept Integer REFERENCES Department,
  DateHired Date,
  status Integer,
  annualSalary Integer
);

CREATE TABLE Professor (
  id Integer NOT NULL PRIMARY KEY,
  name Varchar(20),
  street Varchar(20), city Varchar(10),
  state Varchar(20), zipcode Char(6),
  birthDate Date, picture Char(100),
  latitude Integer, longitude Integer,
  dept Integer REFERENCES Department,
  DateHired Date, status Integer,
  AYSalary Integer,
  MonthSummer Integer
);

CREATE TABLE Student (
  id Integer NOT NULL PRIMARY KEY,
  name Varchar(20),
  street Varchar(20), city Varchar(10),
  state Varchar(20), zipcode Char(6),
  birthDate Date, picture Char(100),
  latitude Integer, longitude Integer,
  studentNo Integer,
  majorDept Integer REFERENCES Department,
  advisor Integer REFERENCES Professor
);

CREATE TABLE TA (
  id Integer NOT NULL PRIMARY KEY,
  name Varchar(20),
  street Varchar(20), city Varchar(10),
  state Varchar(20), zipcode Char(6),
  birthDate Date, picture Char(100),
  latitude Integer, longitude Integer,
  studentId Integer,
  majorDept Integer REFERENCES Department,
  advisor Integer REFERENCES Professor,
  worksIn Integer REFERENCES Department,
  DateHired Date, status Integer,
  semesterSalary Integer, apptFraction Real
);

CREATE TABLE Department (
  deptNo Integer NOT NULL PRIMARY KEY,
  name Varchar(20),
  building Varchar(10), budget Integer,
  chair Integer REFERENCES Professor,
  latitude Integer, longitude Integer
);

CREATE TABLE Course (
  deptNo Integer NOT NULL REFERENCES Department,
  courseNo Integer NOT NULL,
  name Varchar(20), credits Integer,
  PRIMARY KEY (deptNo, courseNo)
);

CREATE TABLE CourseSection (
  deptNo Integer NOT NULL,
  courseNo Integer NOT NULL,
  sectionNo Integer NOT NULL,
  instructorId Integer, semester Integer,
  textbook Varchar(20), noStudents Integer,
  building Varchar(10), roomNo Integer,
  PRIMARY KEY (deptNo, courseNo, sectionNo),
  FOREIGN KEY (deptNo, courseNo) REFERENCES Course
);
```

```
CREATE TABLE Enrolled (
  studentId Integer NOT NULL REFERENCES Student,
  deptNo Integer NOT NULL,
  courseNo Integer NOT NULL,
  sectionNo Integer NOT NULL,
  semester Integer, grade Char(2),
  FOREIGN KEY (deptNo, courseNo, sectionNo)
        REFERENCES CourseSection
);

CREATE TABLE Kids (
  id Integer NOT NULL, kidName Varchar(10)
);
```