Accessing a Relational Database through an Object-Oriented Database Interface
(extended abstract)

J. A. Orenstein
Object Design, Inc.
jack@odi.com

D. N. Kamber
Credit Suisse
David.Kamber@ska.com

Object-oriented database systems (ODBs) are designed for use in applications characterized by complex data models, clean integration with the host programming language, and a need for extremely fast creation, traversal, and update of networks of objects. These applications are typically written in C or C++, and the problem of how to store the networks of objects, and update them atomically has been difficult in practice. Relational database systems (RDBs) tend to be a poor fit for these applications because they are designed for applications with different performance requirements. ODBs are designed to meet these requirements and have proven more successful in providing persistence for applications such as ECAD and MCAD.

Interest in ODBs has spread beyond the CAD communities, to areas such as finance and telecommunications. These applications have many similarities to CAD applications. For example, in financial applications, the data structures describing a mutual fund's portfolio can be quite complex, applications are written in C or C++, and fast traversal of the data structures is important.

These application areas often have an additional requirement – the need to make use of "legacy" data stored in relational database systems (RDBs). For years, the developers of these applications have worked in non-object-oriented languages, and have had to deal with the problem of turning tuple streams into the complex data structures manipulated by their applications. Now, the developers who have started using object-oriented languages and database systems would like to continue the transition by insulating themselves from the relational model and SQL. Typically, their goal is not to migrate data from relational databases into object-oriented databases. ODBs and RDBs are likely to have different performance characteristics for some time, and it is therefore unlikely that one kind of system can displace the other. Instead, the goal is to provide access to legacy databases through object-oriented interfaces.

For this reason, we designed and developed, in conjunction with the Santa Teresa Labs of IBM, the ObjectStore Gateway, a system which provides access to relational databases through the ObjectStore application programming interface (API). ObjectStore queries, collection and cursor operations are translated into SQL. The tuple streams resulting from execution of the SQL query are turned into objects; these objects may be either transient, or persistent, stored in an ObjectStore database. The main design goal of the Gateway was to insulate application developers from SQL and the relational schema, and to permit them to work entirely in an object-oriented paradigm.

In this abstract we describe how we adapted the ObjectStore API to the purposes of the Gateway, and how we extended the API when there were no equivalent concepts with no equivalent in ObjectStore. We assume the reader is familiar with ObjectStore [LAMB91, OREN92]. The accompanying presentation will discuss the use of the Gateway in an application developed by Credit Suisse to yield enhanced concurrency in both worlds and requiring only a minimal coupling between the object-oriented model and the ER model.

Overview of the Gateway's Design

The main design goal of the Gateway was to support the developers of ObjectStore applications that need access to legacy data, while insulating these developers from all aspects of the RDB: the schema of the RDB being accessed; the programming interface to the RDB; and SQL. Because of the need to provide access to existing databases, we cannot control the relational schema. For example, it would be convenient to dictate the types and values of primary and foreign keys as this would simplify the export of object ids and

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

pointers. However, this approach cannot be used with legacy RDBs.

There is little point in eliminating SQL and the relational schema from view if the alternative is a completely new and unfamiliar interface. For this reason, another important design goal was to allow developers to use the existing ObjectStore data model and API. For example, application developers should be able to use pointers, embedded collections, and relationships as they do normally. It should not be necessary to introduce similar but different constructs, nor should it be necessary to work with a relational style of schema, (i.e. no pointers, no embedded collections), dressed up as C++. We found it necessary to balance this design goal against performance goals. In a number of situations it was necessary to compromise the Gateway's API to avoid introducing serious performance problems.

These design goals motivate the central piece of the Gateway's design, the *schema mapping*. A schema mapping captures the correspondences between a relational schema and an ObjectStore schema. For example, a schema mapping would record the fact that a foreign key corresponds to the two data members in an ObjectStore schema that represent a relationship. A schema mapping is created using a declarative language. Once this has been done, Gateway applications can be built by identifying the schema mapping and linking with Gateway libraries. The Gateway uses the schema mapping to translate ObjectStore operations into SQL, and to materialize objects from the resulting tuple streams and status indicators. (The schema mapping is also used as the foundation for ObjectStore SQL Client – a product which provides access to ObjectStore databases via SQL.)

Schema mapping

There are three inputs to a schema mapping, 1) a relational schema, 2) an ObjectStore schema, and 3) a declarative specification of the connections between the two schemas. All three parts are specified in SML (Schema Mapping Language). The schemas are imported from relational and ObjectStore databases named in the SML specification.

The bulk of an SML specification is in the description of the connections. This part of the language is based on the premise that various modeling constructs are expressed by common "idioms" in each data model. For example, a join between a foreign key and a primary key is often used to represent one-to-one and one-to-many relationships; two such joins through a "junction table" represent a many-many relationship (but can also be used for one-to-one and one-to-many relationships). There are a few ways to represent relationships in ObjectStore[1]. SML supports all combinations of these relational and ObjectStore representations. If a relational schema uses modeling techniques not supported by SML, then the user should create an SQL view. Over time, SML will be enhanced to support more modeling methodologies.

C++ functions can be associated with mappings between tables and classes, and between columns and data members. Examples:

- Run the function `Employee::initialize` whenever an `Employee` object is generated from an `EMP` tuple.
- Run the function `Name::import` whenever three `CHAR` columns are mapped to a data member of type `Name`, (the three `CHAR` columns represent first, middle and last names).
- Run the function `Name::export`, which turns a `Name` into three strings, whenever a `Name` is written back to the relational database.

Connecting to a relational database

In order to connect to an RDB, the user id and password must be supplied. This is done via an object representing the database:

```
os_gw_database& empdb =
    os_gw_gateway::
    get_database("EMPDB");
empdb.set_userid("henry");
empdb.set_password("eraserhead");
```

`EMPDB` is the logical name of the RDB being accessed; this name was given in SML. `empdb` is an object representing this database. In addition to setting the user id and password, various RDB-specific properties can be controlled using a function of `os_gw_database` that takes keywords and values.

Once the user id and password have been supplied, the relational database is accessible from the application.

ObjectStore functions that are translated to SQL

The Gateway translates ObjectStore collection methods (including queries) and cursor methods into SQL. When one of these methods is applied to a

[1] Either with or without the relationship wrappers; the "many" side of a relationship can be a type-safe collection, e.g. os_Set<Employee*>, or type-unsafe, e.g. os_set.

collection representing a table, a semantically equivalent set of SQL commands is generated and submitted to the RDB. In addition to query translation, ObjectStore cursors operations are translated into SQL cursor operations, and insertions to and removals from ObjectStore collections are translated into SQL INSERT and DELETE statements.

There is no function call required by ObjectStore to either express an update to an object, or to note that an update has occurred. For example, the age of a person p can be incremented using this C statement:

```
p->age = p->age + 1;
```

Read and write sets are formed by intercepting read- and write-protect violations [LAMB91]. This mechanism is completely transparent. However, because there is no layer of ObjectStore software involved in the update, (except in case of the first read or write to a page), it is impossible for the Gateway to know when an update to the RDB should be issued. We therefore found it necessary to introduce a function indicating that an object has been modified. When an application invokes this function, a SQL UPDATE statement is issued.

Prefetch paths

Consider how this query might be translated into SQL[2]:

```
employees[: salary > 50000 :]
```

This query yields a set of Employee object ids. What should the target list of the SQL query contain? A literal translation of the ObjectStore query would retrieve only the primary key columns, i.e., enough information to produce object ids.

```
SELECT id
FROM EMP
WHERE salary > 50000
```

Another possibility is to retrieve the entire Employee object. The SQL query would then be this:

```
SELECT id, last_name,
first_name, middle_initial,
    ss_number, salary,
department
FROM EMP
WHERE salary > 50000
```

This isn't really correct. If the application is going to navigate from the Employee to a Department via Employee::department, (a pointer in the ObjectStore schema), then the Department had better be present. This means that the query has to be extended to retrieve matching Departments, or there needs to be a second query to retrieve those Departments. In general, a query might need to retrieve a large fraction of a database in order to support any navigation that the application might perform following the query. This is clearly impractical.

Another possibility is to retrieve just the primary key columns and turn these into object identities or pointers. When the application tries to dereference one of these pointers, another query is generated to retrieve the pointed-to object. As with updates, not all pointer dereferences are visible to ObjectStore, (only the first to a given page is noted), so it is impossible to know when a SQL query implementing the dereference should be generated. Second, suppose that we could intercept all dereferences. What SQL query would be generated? All that is known is the identity and probably the type of the object to be retrieved. This would allow us to construct a simple SQL query which fetches a single tuple given its primary key, (assuming we have a table that maps object ids to keys). This is clearly going to result in very poor performance, compared to the SQL queries given above. Using an RDB as an object server which returns one object with each query is not a good idea. The only advantage of this scheme is transparency – the application can navigate freely, and the Gateway guarantees that the required objects are always available.

We dealt with this issue by requiring the application to indicate what objects to retrieve. A graph of related objects is described using "prefetch paths". A *prefetch path* is simply a description of a path. It starts on an object of one class and navigates through pointer- and set-valued data members. ObjectStore already has an object describing paths, os_index_path, used for naming path indexes. They are reused by Gateway to support prefetch.

Object materialization

The Gateway generates objects to represent data retrieved from an RDB. Object identity is determined by primary keys. If the materialized objects are all transient, then it is clear that object identity should be determined within the context of a single process. E.g., if the Employee with id 419 is retrieved, that can be a distinct object from the Employee with id 419 retrieved by a different execution of the same application, (or by a different application). However, if objects are materialized persistently, into an ObjectStore database, in what context should object identity be determined? The database itself could provide the context. That is, if Employee 419 has been materialized into a particular database, then a second attempt to do so will fail, but if the second retrieval materializes an object in a second database, that will succeed. This isn't a satisfactory solution, because an application may access multiple

[2] "DML" notation is used for clarity. The Gateway currently supports only queries submitted through the function call interface.

ObjectStore databases within the same transaction. It would therefore be possible for the application to see two objects with the same value for Employee::id. Another reason this approach fails is that it might be desirable to have multiple materialization contexts within the same ObjectStore database.

We solved this problem by realizing that the context of materialization needs to be explicitly controlled by the application. For this reason, the concept of a *snapshot* has been added to the Gateway. Object materialization always takes place in the context of a snapshot. It is possible to use different snapshots during the execution of an application, but there is always exactly one snapshot "in effect" at a given time. At the Gateway interface, a snapshot is represented by its name, and a snapshot is selected by specifying the name.

In concrete terms, a snapshot is little more than a mapping from primary key to object identity. The Gateway must deal with the situation in which objects are retrieved multiple times and change between the retrievals. The application controls the Gateways actions by selecting one of four behaviors: 1) Use the value in the object and ignore the tuple; 2) Update the object with non-key values from the tuple; 3) Raise an exception; and 4) ignore the fact that an object has already been materialized, and materialize another one. (4) is potentially dangerous, but in applications where the absence of object identity is known to be safe, this mode leads to a performance improvement since the tables mapping keys to object ids do not have to be maintained.

Transaction management

ObjectStore provides serializable transactions and two-phase commit. An attempt is made to coordinate RDB transaction boundaries with ObjectStore transaction boundaries, but two-phase commit between ObjectStore and RDBs is not yet supported.
By default, a gateway application can execute any number of RDB transactions, and these are run using "cursor stability" semantics. Repeatable read is an option. Inconsistencies due to lack of serializability are dealt with as discussed above.

References

[LAMB91] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D.
"The ObjectStore Database System", *Comm. ACM 34*,10 October 1991.

[OREN92] Orenstein, J., Haradhvala, S., Margulies, B., and Sakahara, D.
"Query Processing in the Objectstore Database System", Proceedings of the ACM SIGMOD Conference, San Diego, California, June 1992.