PROGRESS® DataDirect®

ACCESS AN OBJECT WRAPPER USING ODBC

REQUIREMENTS FOR ACCESSING DATA STORED IN AN OBJECT-ORIENTED DATABASE

This tutorial assumes you have an immediate requirement to access data stored in an object-oriented database using ODBC, OLE DB, and/or JDBC from a PC or a UNIX client. The database is a persistent object-oriented system that has classes and instances.

Following are some of the requirements:

- Access objects of any class as a table - The table view will show the attributes of the specified class and all its parents.
- Support dynamic schema - The classes in the database change from one installation to another and maybe over time (similar to how Oracle database can support different schema for different applications).
- Support one or more of the following standards:
  - ODBC compliant access from Windows or UNIX - Allows standard ODBC compliant applications to access the data.
  - OLE DB compliant access from Windows – Allows standard OLE DB compliant applications to access the data.
  - JDBC compliant access from Windows or UNIX - Allows standard ODBC compliant applications to access the data.
- Client/server - Process database access on the database server platform.
- Use object-oriented database system access mechanisms - Support optimized execution of joins by using the pointer references inherent in OODBMS.

HOW TO QUICKLY PROCESS SQL QUERIES AGAINST AN OBJECT-ORIENTED DATABASE SYSTEM

The Progress® DataDirect® OpenAccess™ SDK allows vendors to provide an ODBC, OLE DB, or JDBC interface to a proprietary non-SQL database by implementing a small set of functions. Here we will explain how an Interface Provider (IP) can be designed to work with the Database Access Manager (DAM) to process SQL queries against an object-oriented database system (also referred to as OODBMS in the following discussions).

The main topics covered are:

- Mapping of object-oriented constructs (classes, instances and references) to SQL constructs
- Exposing the mapped schema to the DAM
- Writing the IP code to implement SELECT type of query on a table EMP
- Writing the IP code to implement an optimized join

HIGHLIGHTS:

- Map object-oriented constructs to SQL constructs
- Expose mapped schema to the Database Access Manager (DAM)
- Implement SELECT queries
- Optimize joins

PROGRESS

MAPPING CLASSES TO TABLES

Mapping of object-oriented entities like classes and instances into SQL entities like tables and relationships is possible. The table below describes how various object oriented-entities map to SQL entities.

Object Oriented Entity	SQL Entity
Class	A table that contains all the attributes of the class and its parents as columns with the same type and name. Attributes referencing other objects are simply mapped as foreign keys to a table containing objects of that class. Attributes that are classes can be expanded out recursively into columns (an attribute of a class type that contains X and Y members is mapped to columns X and Y) or can be treated as foreign key references.
Instance	An instance is mapped as a row in a table of the class it belongs to. Attribute values are mapped to column values. Pointer type attributes take on an undefined value. These columns are only to be used for doing joins.
Pointer Reference	One object can reference another object or objects through pointers. For example, if a class DEPT contains as an attribute a set of pointers to EMP, then we would have a table DEPT and a table EMP. A column empObjId in the DEPT table would point to the associated set of EMP objects and would be marked as a foreign key. The user gets all the members of the set by doing a join between the DEPT and EMP table.

Table 1: *Mapping of OODBMS concepts to relational concepts*

We will clarify these concepts by mapping classes EMP and DEPT.

EXAMPLE CLASSES

First, let's define the classes to be used for the example:

```
Class EMP{
int ID;
char name[32];
int age;
DEPT *dept; // reference to a department
}
Class DEPT{
int ID;
char name[64];
char desc[255];
EMP *emp[]; // set of employees
}
```

Note that the EMP class references one department and the DEPT class references one or more EMP.

For this example we want to execute the following two queries:

1. List all employees in a given department
2. List the department to which the specified employee belongs

An application would list all employees under a department by retrieving an object of type DEPT and then iterating through the emps[] array of pointers. It would find the department to which an employee belongs by retrieving the employee object and then following the department reference.

MAPPING OF THE EXAMPLE OBJECT MODEL TO RELATIONAL

Now that we have defined an object model to relate EMP and DEPT, let's derive a relational model. The rules are as defined in Table 1. First, map every class to a table and all simple attributes to columns. Then, handle any pointer references with foreign keys. Also, a decision must be made as to how classes contained in other classes as attributes are treated. In some cases they can be expanded out to columns of the table (class) in which it is contained. In other cases it may be changed to a foreign key into a table of that class. This decision affects how many tables the end user has to deal with and how complex the queries appear (not necessarily how efficiently they can be executed). Below we define each of the columns in the table by

Table EMP:

Column Name	Data Type	Description
ObjID	NUMBER(38)	Unique identifier of this object
ID	NUMBER(38)	Employee's ID
NAME	VARCHAR(32)	Employee's name
AGE	NUMBER(38)	Employee's age
dept_ObjID	NUMBER(38)	Foreign key to the Employee's department. This column is a place holder for the dept object pointer.

Table DEPT:

Column Name	Data Type	Description
ObjID	NUMBER(38)	Unique identifier of this object
ID	NUMBER(38)	Department's ID
NAME	VARCHAR(64)	Department's name
DESC	VARCHAR(255)	Department's description
emp_ObjID	NUMBER(38)	Foreign key to the Employees in this department. This column is a place holder for the emp[] set of pointers.

Each table exposes a column ObjID to contain the unique object identifier. This column is meant to be used only for performing joins and should not be saved for other use. An object instance (a row) should be selected by specifying a query on one of the other attributes. Not that the DEPT table has only one column for the associated employee id. The value of this column is undefined when a select query is done on the DEPT table. It should only be used to perform joins between the EMP and DEPT tables.

With this model, an application can retrieve all the employees belonging to the engineering department with a query:

```
select * from EMP,DEPT where DEPT.NAME='Engineering' and DEPT.
empObjID=EMP.ObjID;
```

This is a join between the DEPT and EMP table. It can be optimally executed by retrieving the DEPT object with NAME='Engineering' and then iterating through all the emp[] pointers. A less optimal execution is to retrieve rows from both EMP and DEPT tables and then to perform a cartesian product.

An application can retrieve the department to which an employee belongs with a query:

```
select DEPT.NAME from EMP,DEPT where EMP.NAME='Joe Smith' and
EMP.dept _ ObjID=DEPT.ObjID;
```

This is a join between the DEPT and EMP table. It can be optimally executed by retrieving the EMP object with NAME='Joe Smith' and then following the dept reference to retrieve the department name. A less optimal execution is to retrieve rows from both EMP and DEPT tables and then to perform a Cartesian product.

To summarize, the client will view all classes as tables and all instances of those classes as rows within those tables. All is-part-of relationships will be handled through foreign keys. All set attributes will be handled through foreign keys. In the next section we discuss how this table view is exposed to the DAM so that it can perform the parsing and execution of SQL queries.

SCHEMA DATABASE

This section details how a table is exposed to the DAM to allow it to parse a query and execute it.

The DAM supports both static and dynamic schema. For static schema, the developer defines a Schema Database. This database consists of the tables named TABLES and COLUMNS stored in dBase III files. The COLUMNS table lists all columns accessible on this database server. It includes the column name, type, description, and other information. The TABLES table lists all the tables and the interface (IP) responsible for accessing it. The DAM uses this information to associate a table to the IP that will do the data access. There is no limit on the number of tables or columns that can be defined. Other tables contain index, primary key and foreign key information.

Maybe a more suitable schema management scheme is to use our dynamic schema capability in which each IP will have the option of registering all the tables and columns at run-time. Conceptually this will result in the same information as now but in a more dynamic way. An IP will do this by exposing implementation of schema functions. Dynamic schema generation requires the logic to map from native schema to relational schema to be contained in your code. This may not be trivial if you want flexibility in how your database schema is presented.

The DAM uses the schema information to:

- Expose the data dictionary to the client application
- To parse SQL statements, determine which IP to call for each of the referenced table, and to build a structure in memory describing each of the columns referenced in the query in terms of its type and name
- Optimize joins

The data dictionary is used by desktop application development tools to allow users to view and select tables from the database. ODBC specifies a specific format this information is to be returned in. The OpenAccess ODBC driver accesses this information from the Schema Database managed by the DAM.

In order to execute a query, the DAM finds out each table and column referenced by the query. This information is then passed to the IP to allow it to read the required columns. For example, a query to the table EMP of the form:

```
select * from EMP;
```

results in the DAM building a list of columns that contain all the columns in table EMP as defined in the Schema Database. This list of columns is obtained by looking up all columns in the Schema database table COLUMNS where table name is EMP. This list of columns and the table name will then be passed to the function exposed by the IP to support SELECT operation. How this information is used is covered in the next section.

IMPLEMENTING SELECT ON EMP

Once the schema of the database is determined and the Schema Database is set up, the IP can be coded to support the required functions. Only the functions to implement the INIT, CONNECT and EXECUTE operations are required for this simple example (please refer to the OpenAccess SDK Programmer's Guide for more details). Each IP registers the supported operations by placing a pointer to a function in an array of function pointers associated with that IP. To execute a SELECT, the DAM simply executes the function pointed to by the slot for the EXECUTE operation and passes this function the table name and the list of columns to be accessed. This function then performs the reading and processing of rows as described in the OpenAccess SDK Programmer's Guide.

Some key points to note are:

- The IP only exposes one entry point independent of the number of tables it contains. For example, only one function pointer for EXECUTE is registered by the IP. It's up to this IP to use the table name and the operation type information passed in order to perform the necessary processing (the DAM passes in all the required information).
- The functions in IP that do the data processing can use the list of columns passed in by the DAM to build the required row. This means that one routine written to implement the SELECT type of operations should be able to process SELECT requests for all tables and for any sub-set of that table's columns.

The design of the db_oodb_execute() function, which will be registered in the EXECUTE slot for the IP, is intended to support access to any table in the OODBMS database. The diagram and the explanation below assume no optimization to keep it simple (refer to the Details of DAM and IP Processing for more details).

The high-level processing performed by the DAM and the IP to execute a SELECT query from the HP database is shown in Figure 1. The steps are:

1. First the query is received by the DAM and parsed into an expression tree.
2. From the SQL statement, the type of query is determined and the SELECT function associated with the select operation (db_oo_select) is called.
3. The IP looks at the table name and columns list input to determine what class of object to access and what attributes to read.

4. The IP steps through each of the instances of the identified class and evaluates against the expression tree by calling a function in the DAM.
5. If the expression is true for the given row of data, it is added to the result set by calling a function in the DAM.
6. Steps 4-5 are repeated until all rows from the database have been read.
7. The DAM packages up the results set and sends it to the client.

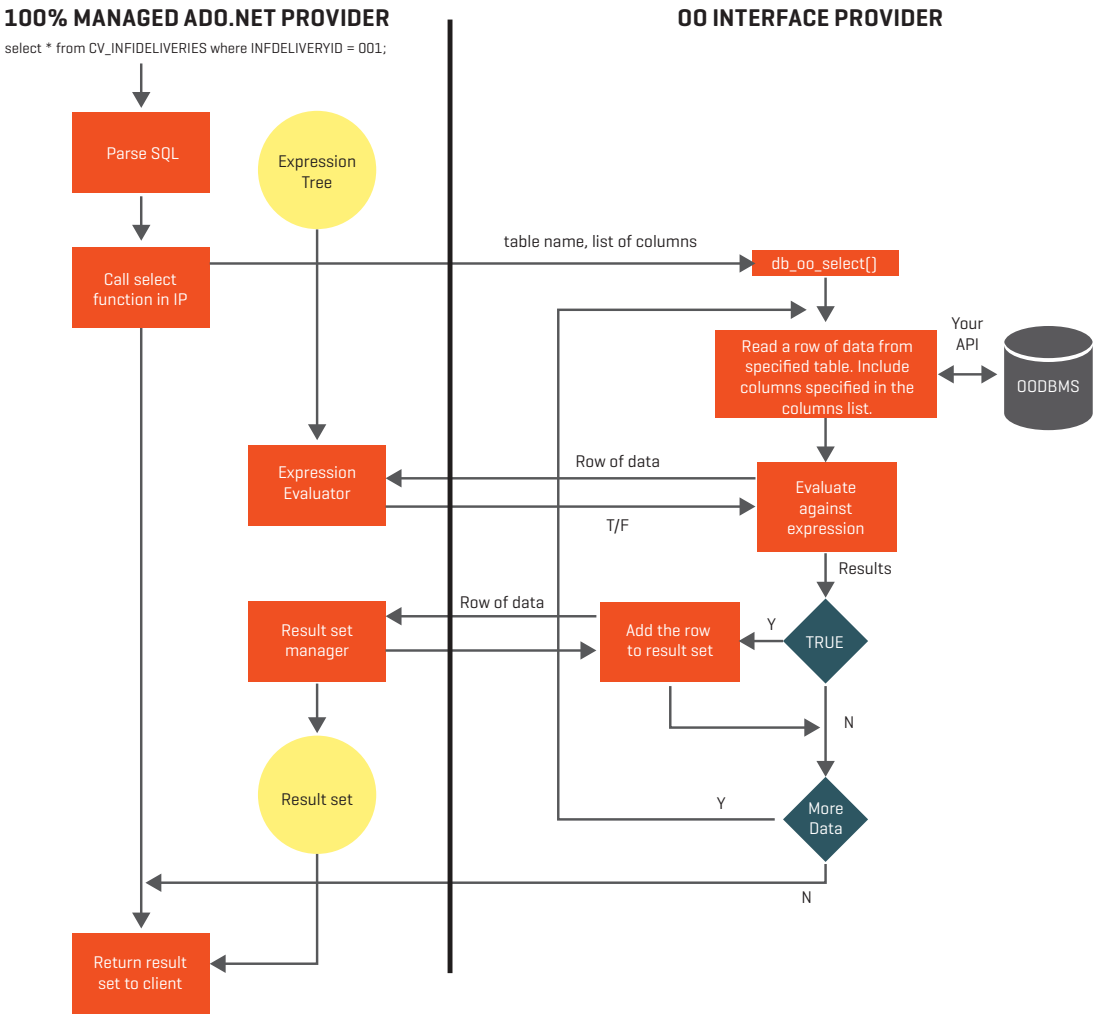

Figure 1: *SELECT Processing*

OPTIMIZING JOINS

The previous section detailed the steps to execute a simple select statement. An IP can implement the joins itself to assist the DAM in processing joins. When processing a table, the IP can find out if it is a parent table of other tables. If this is the case, then it can decide to build the child rows as it builds each of the current table rows. This is where the IP developer can take advantage of pointer references between objects (tables) supported

by OODBMS. As in the example SQL query, to find all employees in a department, a join between the DEPT and EMP table can be handled by the IP more efficiently than by the DAM. The DAM recognizes joins that can be handled by the IP (based on the foreign key and accessibility information returned by the Schema Manager) and passes these joins down to the IP. The IP retrieves the parent object and any referenced objects using pointer access and passes this to the DAM for expression evaluation. As in the normal select processing, the matching set of rows for all referenced tables are added to the result set.

The DAM performs the following operations to use join push-down capability of an IP:

1. Identifies the relations (tables) referenced by the query that are accessible through pointers by the IP.
2. Calls the EXECUTE function within the IP with an optimally ordered set of tables that are referenced by the join query. Each table is tagged with the columns required for the expression processing and the columns required for the result set. The IP starts with the first table and builds up the row with the required columns from all the tables. It then calls the evaluation function in the DAM.
3. Executes on other tables as needed and merges the results.

Let's look at the processing for the sample query we looked at to retrieve all the employees belonging to the engineering department:

```
select * from EMP,DEPT where DEPT.
NAME='Engineering' and DEPT.empObjID=EMP.
ObjID;
```

1. The DAM identifies table DEPT as the primary table and EMP as a table referenced by a pointer from the DEPT table. A linked list of tables starting with DEPT is formed. Each table has associated with it a list of columns that are referenced in the where clause and a list of columns that are to be included in the result set.
2. The DAM assigns expression NAME='Engineering' to the DEPT table.
3. The DAM calls the EXECUTE('SELECT') in your IP with the list of tables and associated expression formulated in steps 1 and 2.
4. The IP checks for equality constraints on any columns of the DEPT table that have an index (and in this case finds a constraint on NAME) and uses this information to retrieve that row. It then retrieves the required columns of the EMP table for expression processing (in this case none). With a row that contains data from the DEPT table, the IP calls the evaluation function in the DAM to decide on whether this row should be accepted or not.
5. If the row is to be accepted, then the IP completes the processing for the selected row. In this example, the IP finds out that the empObjID foreign key is a set and proceeds to build a row for each member of that set. It includes only the columns that are required in the result set. Each row is added to the DAM result set as it is completed.

OTHER OPTIMIZATION

The DAM will pass sort by and other operations to the IP if it is capable of more efficiently processing the order by (sorting) type of operations. This is true for database systems that store data in a certain order. In such a case, the IP can register that the IP can handle such and such type of sorting.

CONCLUSION

OpenAccess SDK allows you to expose Object/Hierarchical databases through a relational model and provide efficient processing of JOINS which are required to access multiple levels.

WORLDWIDE HEADQUARTERS

Progress Software Corporation, 14 Oak Park, Bedford, MA 01730 USA Tel: +1 781 280-4000 Fax: +1 781 280-4095 On the Web at: www.progress.com

Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw

For regional international office locations and contact information, please go to www.progress.com/worldwide

Rev. 9/14