

COMP3062 Advanced Databases

99-07 Laboratory 5: Ontos(4)

1 ONTOS Programming: Object SQL

SQL is a standardized way of querying relational databases. Because of SQL's widespread use, ONTOS DB provides a **simple set** of calls you can use to make SQL queries against an object database. The ONTOS Object SQL(OSQL) interface allows queries to be made over the properties(data members) and procedures(member functions) of persistent objects. An OSQL query consists of a string terminated by a semicolon. Three keywords provide the main functionality: `select`, `from` and `where`. Queries are made within a programmatic query session, which follows a well-defined discipline. Two examples of OSQL query are

```
select name(), age() from Person where age() <= 21;
select name(), age() from Person where eyeColor() = "brown";
select name(), priv_age from Person;
```

You must have “extensions” declared for ONTOS DB types and instances against which you want to perform OSQL queries. The ways to create an extension are given below.

- Use `+c` option when invoking the `classify` utility. More precisely,

```
classify +c<fileName> ...
```

specifies a file containing control statements. For example, in `Person.ct1`¹, there is a statement:

```
class Person has extension
```

Also refer to the `Makefile`.

- Use `+X` option when invoking the `classify` utility for the class.

```
classify +X ....
```

All persistent types created in this run will have extensions.

¹The line of “class Person is persistent” doesn't mean anything.

1.1 Run an Example

```
iwaki% cd ~/introduction
iwaki% make clean-all
iwaki% make chapter8
iwaki% setenv DBNAME ${USER}tutorialDB
```

This example consists of `Person.cxx` and `mainSQL.cxx`.

```
iwaki% ./mainSQL
Creating Person John
Creating Person Jane
Creating Person Alan
Creating Person Mary
Creating Person Bill
Creating Person Jean
Creating Person Mark
Creating Person Lucy
```

```
1: StringSQL  2: ArgLstSQL  3: Average  4: Exit --
```

Choose 1 and RETURN.

```
? select name(), spouse() from Person;
```

Here “?” is a prompt. Also try the OSQL examples given previously. The option 1 and option 2 will return the same results. But the two options actually use different approaches.

1. Option 1 shows how to iterate through the 'rows' satisfying the query, printing the 'row' as one string. This is the easiest method to display the results of the query.
2. Option 2 show how to break down the 'row' into 'columns'. Each 'column' is an instance of the ONTOS class called `OC_Argument`. With the capability to manipulate a single `OC_Argument` we can do lots of other things with our query-results. Here, we just iterate over the 'columns' for each 'row'.

We will discuss these options later in details.

- **Task 1:** How did `classify` specified in the `Makefile` specifies extension for `Person` class?
- **Task 2:** Change the `Makefile` by replacing `+cPerson.ct1` with `+X`, and then run

```
iwaki% cd ~/introduction
iwaki% make clean-db
iwaki% make classify
iwaki% ./mainSQL
```

1.2 Query Structure

An ONTOS Object SQL query is executed by using a `QueryIterator` object. The following steps are always required when using a `QueryIterator`:

- The query session is introduced with `OC_startQuerySession()` and ends with `OC_endQuerySession()`.
- The session should always include an exception handler to handle OSQL errors.
- An instance of an `OC_QueryIterator` is created to query the database.
- The `OC_QueryIterator` object used to make the query must be deleted before ending the session.

An example is given as follows.

```
OC_startQuerySession();
OC_ExceptionHandler sql_handler("OC_SQLProblem");

OC_QueryIterator *an_iterator = (OC_QueryIterator*)OC_null;
if (OC_exceptionDoesNotOccur(sql_handler)){
    // code to make SQL query
} else {
    // code to handle SQL query error
}
delete an_iterator;
OC_endQuerySession();
```

The query iterator, although declared outside of the `OC_ExceptionHandler` scope, is usually instantiated inside the `OC_exceptionDoesNotOccur()` section. This way, any SQL-related errors caused by the instantiation will be caught in the `else` clause of the `OC_ExceptionHandler`.

The actual OSQL query is passed as a C string to the constructor of the `OC_QueryIterator`. OSQL query must be a C string terminated terminated by a semicolon.

```
char the_query[MAXBUFF];
cin.getline(the_query, MAXBUFF);
an_iterator = new OC_QueryIterator(the_query);
```

1.3 Results Returned by Query Iterator: Option 1, `StringSQL()`

The results of an ONTOS OSQL query are returned a row at a time by iteration using an instance of `OC_QueryIterator`. A query iterator is constructed by passing the query string to the `OC_QueryIterator` constructor. The `moreData()` member function is then called on the returned query iterator to determine whether there are more results from the query sent to the constructor. If more results are available to be iterated over, `moreData()` returns `OC_true`; otherwise it returns `OC_false`.

The member function `yieldHeaderString` of `OC_QueryIterator`

```
void yieldHeaderString(char* buffer, int maxLength);
```

copies into the character array `buffer`, a character string representing an explanatory title for each column of the query results. Titles for each column are separated by tab characters. If `maxLength`, the number of characters available in `buffer` is insufficient, an exclamation point(!) is written to the last allocated character position, no further column titles are copied.

The member function `yieldRowString` of `OC_QueryIterator`

```
OC_Boolean yieldRowString(char* row, int maxLength);
```

returns, on each invocation, a character string representing the next successive row satisfying the query. Column entries in the string are separated by tab characters.

■ **Task 3:** Understand `StringSQL()`.

1.4 Argument List Query Results: Option 2, `ArgLstSQL()`

To be able to programmatically access a property returned as the result of an OSQL query, the row of results is returned as an `OC_ArgumentList` object. Each `OC_Argument` object contained in the `OC_ArgumentList` then represents each column item in the resultant row. A separate member function of `OC_QueryIterator` called `yieldRow()` allows it to be accomplished.

The member function `numberOfColumns()` on `OC_QueryIterator` returns the number of columns of output in rows yielded by the query iterator.

Each item in the row returned by `yieldRow()` can be accessed by using the square bracket operator on the `OC_ArgumentList`, then casting the resultant `OC_Argument` to the column's type. This is especially useful when the result contains `OC_Object` objects.

It is also possible with a single function call to convert each `OC_Argument` to a printable form. The free function `OC_localNameGenerate()` produces a generated name for any `OC_Entity`. In the case of `OC_Primitives`, such

as `OC_Integers` and `OC_Strings`, this name consists just of the value. In the case of `OC_Objects`, the name is a unique identifier for the object. `OC_localNameGenerate()` will create temporary names for unnamed objects.

- **Task 4:** Understand `ArgLstSQL()`.
- **Task 5:** Write a function `MyArgLstSQL()` as given on the next page, in order to understand how to retrieve objects. This example illustrates how to retrieve `Person` objects by using `OSQL` and assign the `Person` objects into an array for later use.
- **Thinking 1:** What is the difference between `StringSQL()` and `ArgLstSQL()`?
- **Thinking 2:** What is the similarity between the first assignment and the way `ONTOS` deals with `Sets` and `Extensions`?

```
void MyArgLstSQL()
{
    char buf[BUFLEN], query[BUFLEN];
    Person *persons[64];
    long num = 0;
    long i;

    OC_startQuerySession();
    OC_ExceptionHandler sql_handler("OC_SQLProblem");

    OC_QueryIterator* aQI = (OC_QueryIterator*) OC_null;
    if(OC_exceptionDoesNotOccur(sql_handler)){

        aQI = new OC_QueryIterator("select spouse() from Person;");

        while(aQI->moreData()){
            OC_ArgumentList *anArgL = aQI->yieldRow();
            persons[num++] = (Person *) (OC_Entity *) (*anArgL)[0];
        }

    } else {
        cout << "SQL Error in " << query << "\n";
        cout << sql_handler.handled_exception->format() << "\n";
    }

    delete aQI;
    OC_endQuerySession();

    for (i = 0; i < num; i++)
        if (persons[i] != OC_null)
            cout << persons[i]->name() << endl;
}
```