Object-Oriented Query Language

Zhongling Li
zl2135@columbia.edu

May 6th, 2006

Contents

Chapter 1 Introduction 4
1.1 Background 4
1.2 Design Goals 4
1.2.1 Object-Oriented 4
1.2.2 Database Generic 5
1.2.3 Less Overhead 5
1.2.4 Simple 5
1.2.5 Easy to Integrate 5
1.2.6 Portable 5
1.3 Implementation Overview 6
1.3.1 Entity Definition 6
1.3.2 Query Definition 6
1.4 Related Projects 6
1.5 Roadmap 7
1.6 Summary 7
Chapter 2 Tutorial 8
2.1 Getting Started 8
2.2 About OQL Package 8
2.2.1 Build Script 8
2.2.2 Libraries 8
2.2.3 Java Source Code 9
2.3 Example Script 9
2.3.1 Primitive Data 9
2.3.2 JDBC Connection 9
2.3.4 Entity Type 10
2.3.5 If … Else … 11
2.3.6 Query 11
2.3.7 Array and Loop 12
Chapter 3 Language Reference Manual 13
3.1 Lexical Conventions 13
3.1.1 Comments 13
3.1.2 Identifiers 13
3.1.3 Keywords 13
3.1.4 Strings 13
3.1.5 Other Tokens 13
3.1.6 White Space 13
3.2 Types 14
3.2.1 Primitive Types 14
3.2.2 Reference Types 14
3.2.3 Variables 15
3.3 Expressions 16
3.3.1 Identifier 16
3.3.2 Access array elements 16
3.3.3 Arithmetic expression 16

3.3.4 String expression 16
3.3.5 Relational expression 16
3.3.6 Logical expression........ 17
3.4 Statement........ 17
3.4.1 Assignment statement 17
3.4.2 Conditional statement........ 17
3.4.3 Loop statement 18
3.5 Built-in function 18
3.5.1 print(): Console output 18
3.5.2 connect(): establish JDBC connection 18
Chapter 4 Project Plan........ 19
4.1 Timeline 19
4.2 Development Environment 19
4.2.1 JDK 1.5........ 19
4.2.2 ANTLR 2.7.6........ 19
4.2.3 Subversion........ 19
4.2.4 Intellij IDEA........ 19
4.2.5 Apache ANT 19
4.2.6 HSQL DB........ 19
Chapter 5 Architecture Design........ 20
5.1 Compiler Flow........ 20
5.2 Data Types........ 20
5.2.1 Primitive Types 20
5.2.2 Reference Types:........ 21
5.3 Compiler Classes........ 21
5.3.1 OqlInterpreter........ 21
5.3.2 EntityType........ 22
5.3.3 SessionUtil 22
Chapter 6 Future Improvements........ 23
6.1 Better error reporting........ 23
6.2 SQL aggregation function 23
6.3 Table Joins........ 23
6.4 Transaction support........ 23
Bibliography........ 24
Appendix A Language Syntax 25
A.1 Lexical rules 25
A.2 Syntactic rules 25
Appendix B Parser 27
src/oql/antlr/grammar.g........ 27
Appendix C Walker........ 33
src/oql/antlr/walker.g........ 33

Chapter 1 Introduction

Object-Oriented Query Language (OQL) is designed to provide an object-oriented query interface for traditional relation database systems (RDBMS). The goal is to bridge the gap between object-oriented programming language (specifically Java) and set-oriented Standard Query Language (SQL), and make the persistence layer fit better in an OO system design.

1.1 Background

Though object-oriented programming languages such as Java, Smalltalk, C++ and C# have dominated the server-side business application world for many years, unfortunately object-oriented database did not take off due to numerous reasons. Relational database still is the majority of persistence mechanism.

The standard programming interface to database is SQL, which was originally designed for human interaction purposes. As a set oriented language, with very limited control flow and loose grammar, it slowly becomes a barrier for a pure object-oriented system design. Within the data access layer, too often we find developers throw away the beautiful object model, but start concatenating SQL strings (either directly or indirectly) and create so many messy codes. Even worse, because of the limitable of SQL, many developers use data access layer, or the data model itself as the starting point when designing a system. This ground-up approach tends to make the final system more service-oriented or more procedural.

OQL wraps the basic SQL language into generic object-oriented APIs, and hide the underline complexities such as porting to difference databases. It acts as a mapping tool between objects to rows in database tables, but also has the capability of manipulating data with control flow. Since the output of OQL compiler is Java classes, developers can easily integrate OQL into their applications.

1.2 Design Goals

We want to make OQL a simple language and very easy to learn and use for Java developers. The syntax and grammar is very close to Java instead of SQL. Actually all the logic of generating SQL strings are completely hidden from OQL developer.

1.2.1 Object-Oriented

Everything is an object in OQL; there is no primitive type. All operations are defined at class level, such as Create, Retrieve, Update and Delete (standard CRUD in SQL). Those

very complicated query semantics are nicely wrapped in Criteria, Projection and Join interfaces. From developers' perspective, they are simply dealing with a special set of objects, which happened to be stored in a relational database transparently to their design model.

1.2.2 Database Generic

OQL will be compiled into Java classes, which use ANSI standard SQL and can run against any relational databases without modification. This makes application porting easy, but it also implies a limitation of OQL, we cannot support any database specific features or improvements. In future releases, we will add database flavor option to allow optimization for specific databases.

1.2.3 Less Overhead

The code generated by OQL compiler should not be much slower than hand written native SQL blocks. Since compiler generates all SQL statements (with optimization in mind, such as using prepared statements, or table join ordering) instead of concatenating them at runtime, OQL can actually give better performance in some cases.

1.2.4 Simple

OQL has similar simple syntax and grammar like Java. It has less keyword, and does not support primitive data type. Since it serves a single purpose of presenting SQL query, the API is simple and clean.

1.2.5 Easy to Integrate

Since OQL compiler generates Java classes (or other OO language output in future releases), it is very easy to integrate it into existing systems as data access layer.

1.2.6 Portable

Java is a portable language runs on most operating systems. With its database generic feature, OQL is highly portable to many platforms.

1.3 Implementation Overview

1.3.1 Entity Definition

Entity is defined in a format similar to a Java class. Actually it is a special type of Java class internally which maps to a database table. Each entity contains an "id" field which is the primary key (it is a good design to have surrogate key for every table in the database).

```
entity Address {
        int id,    // implicit primary key
        string street,
        string city,
        int zip,
        boolean valid
}
```

Standard CRUD operations are defined at entity level.

```
// create a new instance of Address
a = Address.create(100, "1234 Main Street", "New York", "100020", true);

a.street = "345 A Street";
Address.update(a);          // update instance a

Address.delete(100);         // delete instance by primary key

b = Address.get(100);    // query instance by primary key
```

1.3.2 Query Definition

Query interface is the most important API of OQL. Criteria and OrderBy are two fundamental objects in the query structure.

```
o = order.asc("zip");          // order by zip, ascending
c1 = Criterion.eq("city", "New York");          // city='New York'
c2 = Criterion.ne("zip", "10020");              // zip != '10020'
list = Address.query(c1, o);                    // query, return Address[]
```

1.4 Related Projects

There are numerous existing projects dealing with Object-Relation mapping issues, such as Hibernate, iBatis, JDO and infamous Entity Bean. OQL actually borrows many ideas from these mature projects. The difference lies in:

- Existing projects are released as libraries instead of its own language.

- Some projects tend to do more than simple things, such as entity state management, caching and complicated transaction management. Hibernate and Entity Bean is the best example.
- Though Hibernate has its own HQL language, the idea still is a mock of SQL string, just replacing the table with Java class, and column with object property.
- iBatis goes to another extreme which simply provides a Object to SQL directly mapping layer.

1.5 Roadmap

There are many features that did not make it to the original design of OQL due to time and resource limitations. In the near future, we plan to support the following list of things:

- Outer joins
- Support operations cross multiple databases
- Declarative transaction support
- Use existing Java classes as entity definition
- Optimization for specific databases, like SQL hints

1.6 Summary

OQL provides a powerful alternative to the Object-Relation access layer. It is pure Object-Oriented, simple to learn, database and platform generic. In addition, since OQL abstracted many good designs from existing OR projects, which makes it a good candidate for any new Java project which requires relational database as persistence layer.

Chapter 2 Tutorial

2.1 Getting Started

- Make sure you have JDK 1.5 installed, OQL source package requires JDK 1.5 to be built and run. Set your JAVA_HOME environment variable properly.
- Unzip the OQL source package oql.zip, say OQL_HOME
- "cd $OQL_HOME"
- "chmod +x build.sh oql.sh" to make them executable
- "./oql.sh etc/Test.oql **-v**" will build and run the test script etc/Test.oql
- "./oql.sh etc/Test.oql" will run it in non-verbose mode to hide debug information (such as SQL script executed)
- "./build.sh clean test" will just build and unit test the whole package (using ANT)
- You can create your own OQL script and use "oql.sh" to compile and execute them
- On Windows system, please use build.bat and oql.bat

2.2 About OQL Package

2.2.1 Build Script

OQL uses standard ANT build script. All the ANT targets are defined under $OQL_HOME/build.xml. Main targets are:

- clean: clean the build and dist directories
- jar: package the release jar file under dist dir
- test: package the release jar and run unit testing

ANT libraries are contained in the package's lib directory. build.sh and oql.sh are scripts used to invoke ANT scripts.

2.2.2 Libraries

All third-party libraries are stored under $OQL_HOME/lib directory, which include:

- ant*.jar: Apache ANT build script with optional tasks for JUnit and ANTLR
- antlr.jar: ANTLR
- log4j.jar: Apache Log4j logging support
- junit.jar: Unit testing framework
- hsqldb.jar: HSQL Java database, used as in-memory database for testing

2.2.3 Java Source Code

$OQL_HOME/src contains all the production Java source code:

- oql/antlr/grammer.g: ANTLR lexer and parser for OQL
- oql/antlr/walker.g: ANTLR AST walker for OQL

- oql/compiler/Main.java: Main class for OQL compiler
- oql/compiler/OqlInterpreter.java: OQL interpreter (control flow and symbol table mapping, reference type function invocation etc.)
- oql/compiler/EntityType.java: EntityType CRUD operations
- oql/compiler/Criteria.java: Criteria
- oql/compiler/Order.java: Order
- oql/compiler/SessionUtil.java: JDBC session utility class
- oql/compiler/CompilerException: OQL compiler exception class
- oql/type/*.java: All OQL internal data structures

$OQL_HOME/src_test contains some JUnit test classes for unit testing.

2.3 Example Script

2.3.1 Primitive Data

Primitive data is handled in a very similar way as how Java does. The difference is that OQL only supports three primitive data types: int, string and boolean. And OQL does not allow (or need) to declare type for each variable.

The following script:

```
a = 8 * (6 + 4);
print("8 * (6 + 4) = " + a);
```

will output:

```
8 * (6 + 4) = 80
```

2.3.2 JDBC Connection

The connect(user, password, url, driver) function should normally be called at the beginning of each script to start a JDBC connection to the specific database. This connection will be used across the lifecycle of the whole script, and only will be committed at the end.

The following script opens a connection to an in-memory HSQL database called “testdb” with the user name “sa” and empty password. OQL package bundles with HSQL library,

for other database, please refer to its documentation for the proper parameters, and please do not forget to include its JDBC driver library in the execution classpath.

```
connect("sa", "", "jdbc:hsqldb:mem:testdb", "org.hsqldb.jdbcDriver");
```

2.3.4 Entity Type

The following script declares an Entity Type called Address, which has five primitive fields.

```
entity Address {
   int id,
   string street,
   string city,
   int zip,
   boolean valid
};
```

After the entity type is registered, a database table with same name will be created, with each entity field maps to a database column. The SQL script will be something like this (you can always pass in "-v" flag to OQL to turn on debugging):

```
CREATE TABLE Address ( id INTEGER  , street VARCHAR(1000) , city VARCHAR(1000)  , zip
INTEGER  , valid BOOLEAN  );
```

Create Entity

A new entity instance can be created by given explicit value for each field. After this call, a new row will be inserted into the database. The data type of variable b is automatically detected:

```
b = Address.create(100, "123 Main Street", "New York", "10020", true);
```

Get Entity By ID

You can get the entity back by its id like this:

```
c = Address.get(100);
print("Address c = " + c);
```

The output will be a dump of the content of Address c:

```
Address c =  { street=123 Main Street, valid=true, zip=10020, city=New York, id=100 }
```

Update Entity

You can update an entity's attribute through assignment, then use Address.update() to persist it back to the database.

```
// update c's street
c.street = "500 Broadway";
Address.update(c);

// get it back
d = Address.get(100);
print("Updated Address = " + d);
```

The output will be:

Updated Address = { street=500 Broadway, valid=true, zip=10020, city=New York, id=100 }

Delete Entity

Entity instance can be deleted by its ID too. If it is deleted from the table, next get by ID will return NULL.

```
// delete by id, should not be able to get it back again
Address.delete(100);
n = Address.get(100);
print("After deletion, n is " + n);    // print NULL
```

2.3.5 If … Else …

If … Else … usage is very similar to Java. Just one note is that keyword "and", "or" and "not" is used in logical expression instead of "&&", "||" and "!" in Java.

The following script should output the first string: *c's id > 50 and is valid*

```
// create an Address
b = Address.create(100, "123 Main Street", "New York", "10020", true);

// get id back by id
c = Address.get(100);
if (c.id > 50 and c.valid == true)  {
   print("c's id > 50 and is valid");
} else {
   print("c's id <= 50");
}
```

2.3.6 Query

Query in OQL is done through two reference data types: criteria and order. Criteria specify which query condition on with field of the entity; order specifies the end results' ordering.

```
c1 = criteria.eq("valid", true);
o1 = order.desc("city");
aList = Address.query(c1, o1);
```

will issue a SQL query like this, and return an array of entity instances:

SELECT id , street , city , zip , valid FROM Address WHERE valid = true ORDER BY city DESC

2.3.7 Array and Loop

Array of entity objects are returned through an entity query. Two major operations on an array are to get its size and a specific indexed element (normally in a loop):

The following script will loop through the list and print out each entity, and it will break when all elements have been accessed:

```
i = 0;
loop {
   if (i == aList.size) {
      print("breaking loop");
      break;
   }
   m = aList[i];
   print(m);
   i = i + 1;
}
```

Chapter 3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

The characters /* introduce a comment, which can expand multiple lines and terminates with the characters */. The characters // starts a single-line comment.

3.1.2 Identifiers

An identifier is a sequence of letters and digits and must start with a letter. Character underscore _ counts as a letter. Identifiers are case sensitive, upper case and lower case are considered different.

3.1.3 Keywords

The following identifiers are reserved for use as keywords, and cannot be used otherwise:

```
entity    criteria       order
int       string         boolean
true      false
and       or             not
loop      break          continue
if        else
```

3.1.4 Strings

A String is a sequence of characters enclosed by double quotes. Two consecutive double quotes represent a double quote.

3.1.5 Other Tokens

```
{    }    (    )    [    ]
+    -    *    /    =
==   >    >=   <    <=   !=
"    ,    ;
```

3.1.6 White Space

White space is defined as space character, horizontal tab, and line terminators.

3.2 Types

3.2.1 Primitive Types

Three primitive data types are supported at current stage. They are static, immutable. Every variable should be declared with a type tag and checked at compilation time.

- int — 32-bit signed integers, default to 0
- string — sequence of characters, default to null
- boolean — only have two possible values: true or false, default to false

3.2.2 Reference Types

There are four kinds of reference types: entity, array, criteria and order.

Entity Type

Entity type maps to a database table. Entity type must be defined before use.

Each entity type should have a unique name and a list of fields of primitive type (int, string or boolean). One of the field must be an "int id" field which is the primary key of the mapping table (normally a surrogate key, like Oracle sequence or SQL Server identity type).

```
entity Address {
      int id,
      string street,
      string city,
      string state,
      boolean valid
}
```

Standard database CRUD (create, retrieve, update and delete) and query operations are provided at entity level, i.e. each entity type has create(), update(), delete(), get() and query() methods.

Objects

An object is an instance of an entity. It maps to a row in the database table. Object can only be created through entity's create() method, or loaded through get()/query() methods. Object does not need explicit type definition when it is first referenced. For example, both of the following two statements defines variable "a" which reference to an object of entity Address type:

```
a = Address.create("123 Main Street", "New York", "NY", "100020");
a = Address.get(100);
```

Entity's field value can only be accessed through its objects. Modification to the field will only be persisted to the database when entity's update() method is explicitly called.

```
print(a.state);   // will print string "NY"
a.state = "CA";   // will set state value to be "CA"
Address.update(a);      // persist the update to database
```

Array

Array is a collection of objects of the same entity type. It is declared with entity name followed by a pair of brackets "[]". Array can only be created at runtime through entity's query() method, which will return a collection of objects which satisfy the query criteria.

Array has index value is an int type start from 0 to the length of array, which is returned by array's length attribute. Object in the array can be referenced by its index value. Null array does not exist in OQL, an empty array (length=0) will be returned from query() if no matching data is found. For example:

```
aList = Address.query(criteria); // query to return array
print(aList.size);      // print the size of array
a = aList[0];     // reference the first object in the array
```

Criteria

Criteria define the basic query conditions. It specifies which entity field to apply the constraint to, and the value and type of the constraint. For example:

```
criteria c1 = criteria.ne("city", "New York");
criteria c2 = criteria.eq("valid", true);
```

Order

Order defines the ordering of the query result. Its reference variable is constructed by specifying which of the entity's field name to be sorted and in what order.

```
order asc = order.asc("city");
order desc = order.desc("state");
```

3.2.3 Variables

A variable is a storage location and has an associated type, which is a primitive type or one of the reference types. A variable of reference type can hold either a null reference or reference to an instance of that reference type.

3.3 Expressions

When an expression is evaluated (executed), the result denotes one of the following: a variable, a value or void.

3.3.1 Identifier

An Identifier is a left-value expression. It will be evaluated to some values bounded to this identifier.

3.3.2 Access array elements

Array expression is left-value. It contains an array identifier, followed by an int index value enclosed by a pair of “[“ and “]”.

3.3.3 Arithmetic expression

Arithmetic expression is applicable only to int type values (an expression or variable). It only supports binary operators like “+”, “-“, “*”, “/” which indicates addition, subtraction, multiplication and division.

Arithmetic expressions are grouped left to right. “*” and “/” are of higher precedence than “+” and “-“.

3.3.4 String expression

Operator “+” is applicable to string values to indicate a string concatenation as expression result.

```
s = "abc" + "123";      // s has value "abc123"
```

3.3.5 Relational expression

Binary relational operators “==”, “<=”, “>=”, “<”, “>” and “!=” evaluates whether the first operand is equal to, less than or equal to, greater than or equal to, less than, greater than or not equal to the second operand. The expression result is a boolean value (true or false).

For primitive type (int, string or boolean), since they are immutable, relation expression compares the value of two operands.

For reference types, only “==” and “!=” applies which compare the reference itself instead of target’s value.

3.3.6 Logical expression

Logical operator “and”, “or”, “not” indicates logical and, or and negation of relational expressions. Operator “or” has the highest precedence, followed by “and” then “or”.

3.4 Statement

A sequence of statements will be executed sequentially unless the flow is altered by flow-control statement. A group of zero or more statements can be surrounded by a pair of “{“ and “}” to be treated as a single statement.

3.4.1 Assignment statement

Assignment statement has the form of two expressions on each side of assignment operator “=”, and terminated by “;”.

```
<left expression> = <right expression>
```

The right expression will be evaluated, and the result value will be assigned to the left expression.

3.4.2 Conditional statement

Conditional statement takes a logical expression and executed one statement based on the evaluation result of the expression.

```
if ( <logical expression> )
      <statement>
```

or

```
if ( <logical expression> )
      <statement>
else
      <statement>
```

If the logical expression is true, first statement will be executed, otherwise the optional second statement is executed.

Each “else” is matched with the closest previous unmatched “if” statement.

3.4.3 Loop statement

A loop statement is in the form of:

```
loop {
       (<statement>)*
       break;
       (<statement>)*
}
```

The statement wrapped inside will be executed continuously until the break statement is executed. A break statement in the form of keyword "break" followed by ";" will break the inner most loop.

3.5 Built-in function

3.5.1 print(): Console output

Function print() takes only one argument and print its value to the console. For primitive type argument, its value will be directly printed. For reference type, all the values of the target object will be printed (all the fields of an entity type, or every entity of an array).

3.5.2 connect(): establish JDBC connection

Function connect() takes four arguments of string type: user, password, JDBC URL and JDBC driver class name. It should appear once as the first statement of a OQL program to prepare connection for execution. The connection will be closed automatically when the program finishes execution.

Chapter 4 Project Plan

4.1 Timeline

Tasks	Finish Date
White paper	2/6/2006
OQL grammar	2/28/2006
Reference manual	3/2/2006
Lexer and parser	3/15/2006
Data types	3/25/2006
AST walker	4/15/2006
Control flow	4/25/2006
Unit Testing	4/30/2006
Documentation	5/6/2006

4.2 Development Environment

4.2.1 JDK 1.5

JDK 1.5 is required for compiling and running OQL package. Generics and auto-boxing features are used in the code for convenience.

4.2.2 ANTLR 2.7.6

Latest stable version of ANTLR is used as the builder of OQL lexer, parser and AST walker.

4.2.3 Subversion

Version control system choice is Subversion since it is free and better than CVS, especially in the area of branching.

4.2.4 Intellij IDEA

The best Java IDE, the only draw back is that it is commercial (but free for open-source projects). It offers much better usability and plug-in integrations (such as version control, or web application server) than Eclipse.

4.2.5 Apache ANT

The standard Java build system.

4.2.6 HSQL DB

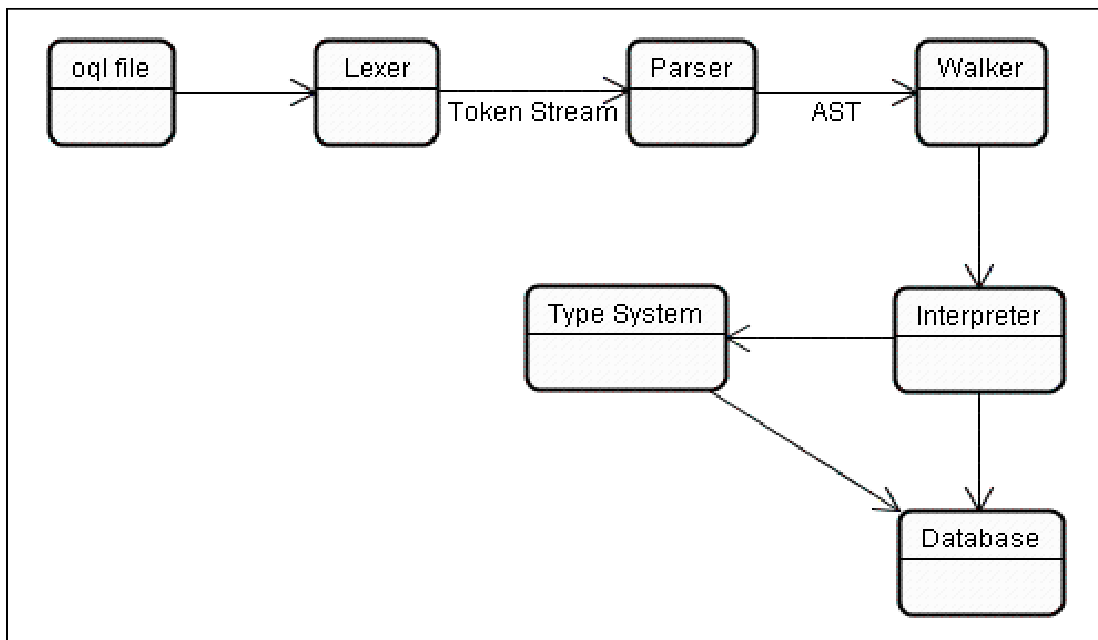
Use its in-memory database ability for testing. Also it is pure Java and very light-weight, perfect for package bundling.

Chapter 5 Architecture Design

5.1 Compiler Flow

The OQL compiler consists the following components:

- Lexer: reads program file into tokens
- Parser: analyze the syntactic structure of the program and converts it to an AST (abstract syntax tree)
- Walker: traverse the AST and calls corresponding functions of interpreter
- Interpreter: maintain the symbol table and divers function calls to the internal type system
- Type system: primitive and reference data types supported by OQL



5.2 Data Types

The core type system contains the following data types. All of them extend the same abstract super class Data, which provides the default implementation of all possible operations on the types.

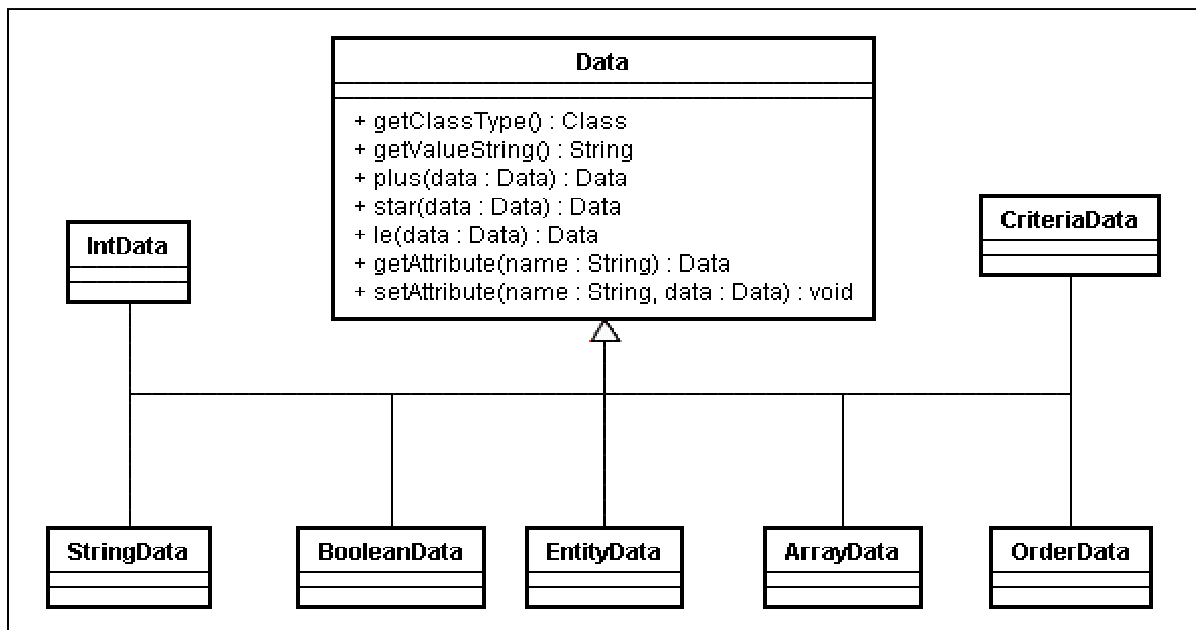
5.2.1 Primitive Types

- IntData: wraps an integer value

- StringData: wraps a string value
- BooleanData: wraps a boolean value

5.2.2 Reference Types:

- EntityData: wraps entity object
- ArrayData: wraps a list of entity objects
- CriteriaData: wraps query condition
- OrderData: wraps query ordering



5.3 Compiler Classes

5.3.1 OqlInterpreter

OqlInterpreter is the core class of the compiler. It is controlled by the AST walker to operate on the internal type system.

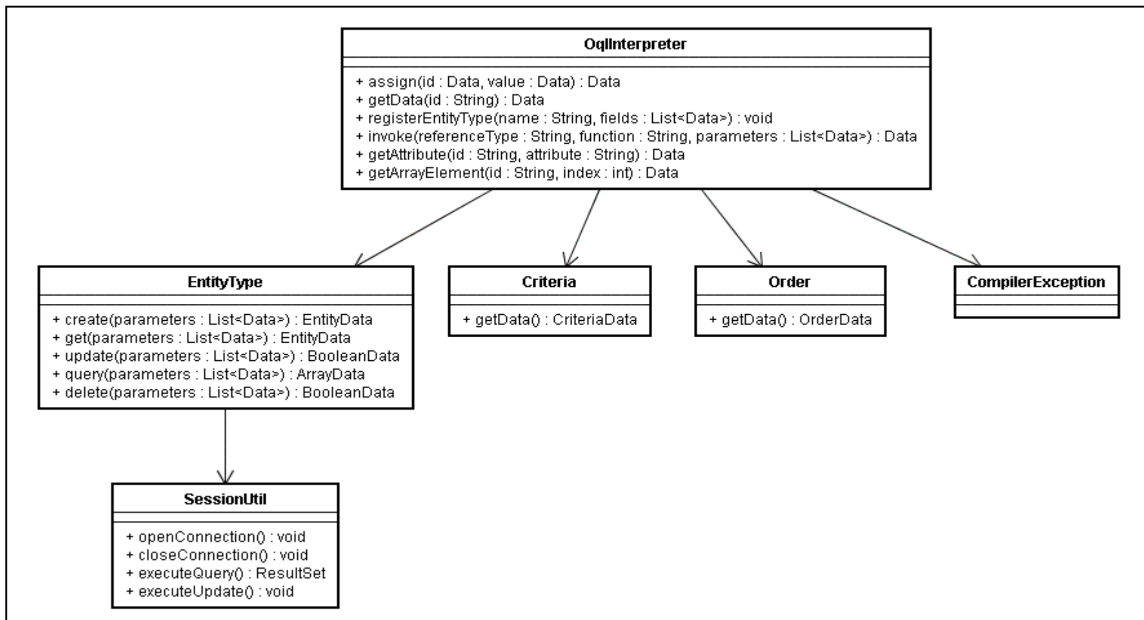
- Maintains global symbol table
- Register entity type
- Invoke function on reference type
- Get entity object's attribute
- Get array element

5.3.2 EntityType

Entity type defines the reference type that maps to a database table. The create, get, update, delete and query operations are defined here. All of them eventually translate into SQL scripts and are executed by SessionUtil.

5.3.3 SessionUtil

SessionUtil wraps all the operations over the underline JDBC connections. The connection is opened through the connect() call, and is closed at the end of the script execution.



Chapter 6 Future Improvements

This is a solo project completed by CVN student. Due to the resource limitation, some areas are left out of this release for future improvements.

6.1 Better error reporting

Currently the engine only supports semantically correct OQL script. A better error reporting system should be in place to guide the programmer to fix the syntax error when it occurs.

6.2 SQL aggregation function

To support sum(), avg(), min() and max() those kind of SQL aggregation functions, which will return arbitrary data type instead of always the entity objects.

6.3 Table Joins

To support multiple tables (entity types) joins, and allow define join conditions. An outer-join mechanism is also desirable.

6.4 Transaction support

Allow transaction control across SQL operations. If possible, even allow cross database entity operations.

Bibliography

[1] Antlr, http://www.antlr.org

[2] Ant, http://ant.apache.org/

[3] Intellij IDEA, http://www.intellij.com

[4] Subversion, http://subversion.tigris.org/

[5] HSQL DB, http://www.hsqldb.org/

[6] Hibernate, http://www.hibernate.org/

[7] iBatis, http://ibatis.apache.org/

[8] JDBC API, http://java.sun.com/products/jdbc/

Appendix A Language Syntax

A.1 Lexical rules

alpha → ‘a’ .. ‘z’ | ‘A’ .. ’Z’ | ‘_’
digit → ‘0’ .. ‘9’
id → alpha (alpha | digit)*
int → (digit)+
string → ‘“’ (~(‘”’) | (‘”’ ‘”’))* ‘”’
entity_type → ‘A’ .. ‘Z’ (alpha | digit)*

Skipped tokens

linebreak → ‘\n’
whitespace → ‘ ‘ | ‘\t’ | ‘\r’ | linebreak
comment → (‘/*’ (~’*/’)* ‘*/’) | ‘//’ (~linebreak)* linebreak

A.2 Syntactic rules

program → (statement)*
statement → entity_stmt | assignment_stmt | if_stmt | loop_stmt | break_stmt | exit_stmt
| print_stmt | connect_stmt | func | ‘{‘ (statement)* ‘}’
entity_stmt → ‘entity’ entity_type { field (‘,’ field)* } ‘;’
field → primitive id ‘;’
primitive → ‘int’ | ‘string’ | ‘boolean’
reference_type → entity_type | ‘criteria’ | ‘order’
func → reference_type ‘.’ id ‘(‘ parameter (‘,’ parameter)* ‘)’
parameter → expression
expression → logic_term (‘or’ logic_term)*
logic_term → logic_factor (‘and’ logic_factor)*
logic_factor → (‘not’)? relational_expr
relation_expr → arithmetic_expr ((‘>=’ | ‘<=’ | ‘>’ | ‘<’ | ‘==’ | ‘!=’) arithmetic_expr)?
arithmetic_expr → arithmetic_term ((‘+’ | ‘-‘) arithmetic_term)*
arithmetic_term → arithmetic_factor ((‘*’ | ‘/’) arithmetic_factor)*
arithmetic_factor → ‘+’ right_value | ‘-‘ right_value | right_value
attribute → id ‘.’ Id
left_value → id | attribute
array → id ‘[‘ expression ‘]’
right_value → left_value | int | string | ‘true’ | ‘false’ | func | array | ‘(‘ expression ‘)’
assignment_stmt → left_value ‘=’ expression ‘;’
if_stmt → ‘if’ ‘(‘ expression ‘)’ statement
loop_stmt → ‘loop’ statement
break_stmt → ‘break’ ‘;’

exit_stmt → ‘exit’ ‘;’
print_stmt → ‘print’ ‘(‘ expression ‘)’ ‘;’
connect_stmt → ‘connect’ ‘(‘ string ‘,’ string ‘,’ string ‘,’ string ‘)’ ‘;’

Appendix B Parser

src/oql/antlr/grammar.g

```
header { package oql.antlr; }

class OqlLexer extends Lexer;

options {
   k = 2; // needed for newline junk
   charVocabulary = '\u0000'..'\u007F'; // allow ascii
   testLiterals = false;
   exportVocab = Oql;
}

protected
ALPHA   : 'a'..'z' | 'A'..'Z' | '_' ;

protected
DIGIT   :   '0'..'9';

/* comment */
COMMENT : ( "/*" (
                options {greedy=false;} :
                (NL)
                | ~( '\n' | '\r' )
              )* "*/"
          | "//" (~( '\n' | '\r' ))* (NL)
        )                 { $setType(Token.SKIP); }
       ;

/* tokens */
LBRACE  : '{';
RBRACE  : '}';
LPAREN  : '(' ;
RPAREN  : ')' ;
LBRK    : '[';
RBRK    : ']';
PLUS    : '+' ;
MINUS   : '-' ;
STAR    : '*' ;
DIV     : '/';
ASSIGN  : '=';
EQ      : "==";
GT      : '>';
GE      : ">=";
```

```
LT      : '<';
LE      : "<=";
NEQ     : "!=";
QUOTE   : '"';
COMMA   : ',';
SEMI    : ';';
DOT     : '.';

/* identifier */
ID  options { testLiterals = true; }
        : 'a'..'z' (ALPHA | DIGIT)*
        ;

ENTITY_TYPE  options { testLiterals = true; }
        : 'A'..'Z' (ALPHA | DIGIT)*
        ;


/* string */
STRING  : '"'!
              (  ~('"' | '\n')
               | ('"'!'"')
              )*
          '"'!
        ;

/* int */
INT     : (DIGIT)+ ;

/* white space */
WS      : (' ' | '\t')+          { $setType(Token.SKIP); }
        ;

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
          { $setType(Token.SKIP); newline(); }
        ;

class OqlParser extends Parser;

options {
   k = 2;
   buildAST = true;
   exportVocab = Oql;
}

tokens {
```

```
   PROGRAM;
   STATEMENT;
   ARRAY;
   FIELD;
   ATTRIBUTE;
   PARAMETER;
   FUNC;
}

/* entity type */
entity_stmt
        : "entity"^ ENTITY_TYPE LBRACE!
            field (COMMA! field)*
            RBRACE! SEMI!
        ;

field
        : primitive ID
            {#field = #([FIELD, "FIELD"], field); }
        ;

primitive
        : "int"
        | "string"
        | "boolean"
        ;

reference_type
        : ENTITY_TYPE^
        | "criteria"^
        | "order"^
        ;

func
        : reference_type DOT! ID LPAREN!
            {#func = #([FUNC, "FUNC"], func); }
            parameter (COMMA! parameter)*
        RPAREN!
        ;

parameter
        : expression
            {#parameter = #([PARAMETER, "PARAMETER"], parameter); }
        ;

/* expression */
```

```
expression
	: logic_term ("or"^ logic_term)*
	;

logic_term
	: logic_factor ("and"^ logic_factor)*
	;

logic_factor
	: ("not"^)? relational_expr;

relational_expr
	: arithmetic_expr ((GE^ | LE^ | GT^ | LT^ | EQ^ | NEQ^) arithmetic_expr )?
	;

arithmetic_expr
	: arithmetic_term ((PLUS^ | MINUS^) arithmetic_term)*
	;

arithmetic_term
	: arith_factor ((STAR^ | DIV^) arith_factor)*
	;

arith_factor
	: PLUS! right_value
	| MINUS! right_value
	| right_value
	;

attribute
	: ID DOT! ID    /* reference type's attribute */
	   {#attribute = #([ATTRIBUTE, "ATTRIBUTE"], attribute); }
	;

left_value
	: ID | attribute
	;

array
	: ID LBRK! expression RBRK!     /* array index */
	   {#array = #([ARRAY,"ARRAY"], array); }
	;

right_value
	: left_value
	| INT
```

```
        | STRING
        | "true"
        | "false"
        | func
        | array
        | LPAREN! expression RPAREN!
        ;

/* statement */
statement
        : entity_stmt
        | assignment_stmt
        | if_stmt
        | loop_stmt
        | break_stmt
        | exit_stmt
        | print_stmt
        | connect_stmt
        | func SEMI!
        | LBRACE! (statement)* RBRACE!
          {#statement = #([STATEMENT, "STATEMENT"], statement); }
        ;

assignment_stmt
        : left_value (ASSIGN^) expression SEMI!
        ;

if_stmt
        : "if"^ LPAREN! expression RPAREN! statement
          (options {greedy = true;}: "else"! statement )?
        ;

loop_stmt
        : "loop"^ statement
        ;

break_stmt
        : "break"^ SEMI!
        ;

exit_stmt
        : "exit" SEMI! { System.exit(0); }
        ;

print_stmt
        : "print"^ LPAREN! expression RPAREN! SEMI!
```

```
        ;

connect_stmt
        : "connect"^ LPAREN!
            STRING COMMA!
            STRING COMMA!
            STRING COMMA!
            STRING
            RPAREN! SEMI!
        ;

/* program */
program
        : (statement)* EOF!
            {#program = #([STATEMENT, "PROGRAM"], program); }
        ;
```

Appendix C Walker

src/oql/antlr/walker.g

```
header { package oql.antlr; }

{
import java.io.*;
import java.util.*;
import java.sql.*;

import oql.type.*;
import oql.compiler.*;
}

class OqlWalker extends TreeParser;
options{
   importVocab = Oql;
}

{
   OqlInterpreter intp = new OqlInterpreter();
}

expr returns [Data data]
{
   data = null;
   Data a, b, c, d;
   String x, y, z;
   List<Data> list = new ArrayList<Data>();
}
   : i:INT         { data = new IntData(i.getText()); }
   | s:STRING      { data = new StringData(s.getText()); }
   | "true"        { data = new BooleanData("true"); }
   | "false"       { data = new BooleanData("false"); }
   | id:ID         { data = intp.getData(id.getText()); }
   | "int"         { data = new IntData(); }
   | "string"      { data = new StringData(); }
   | "boolean"     { data = new BooleanData(); }
   | #(GE a=expr b=expr)        { data = a.ge( b ); }
   | #(LE a=expr b=expr)        { data = a.le( b ); }
   | #(GT a=expr b=expr)        { data = a.gt( b ); }
   | #(LT a=expr b=expr)        { data = a.lt( b ); }
   | #(EQ a=expr b=expr)        { data = a.eq( b ); }
   | #(NEQ a=expr b=expr)       { data = a.neq( b ); }
   | #(PLUS a=expr b=expr)      { data = a.plus( b ); }
```

```
| #(MINUS a=expr b=expr)     { data = a.minus( b ); }
| #(STAR a=expr b=expr)      { data = a.star( b ); }
| #(DIV a=expr b=expr)       { data = a.div( b ); }
| #("and" a=expr b=expr)     { data = a.and(b); }
| #("or" a=expr b=expr)      { data = a.or(b); }
| #("not" a=expr)            { data = a.not(); }
| #(ASSIGN a=expr b=expr)
    { data = intp.assign(a, b); }
| #(FIELD a=expr x=idexpr)
    {
        a.setId(x);
        data = a;
    }
| #("entity" x=refexpr
        (field:. { list.add(expr(#field)); })*)
    {
        intp.registerEntityType(x, list);
    }
| #(FUNC x=refexpr y=idexpr
        (parameter:. { list.add(expr(#parameter)); })*)
    {
        data = intp.invoke(x, y, list);
    }
| #(PARAMETER a=expr)
    { data = a; }
| #(ATTRIBUTE x=idexpr y=idexpr)
    { data = intp.getAttribute(x, y); }
| #(ARRAY x=idexpr b=expr)
    { data = intp.getArrayElement(x, b); }
| #("if" a=expr thenp:. (elsep:.)?)
    {
        if ( !( a instanceof BooleanData ) )
            throw new CompilerException( "if: expression should be boolean" );
        BooleanData bool = (BooleanData) a;
        if ( (Boolean)a.getValue() )
            data = expr(#thenp);
        else if ( null != elsep )
            data = expr(#elsep);
    }
| #("loop" loopbody:.)
    {
        intp.setLooping(true);
        while (intp.isLooping()) {
            data = loopexpr(#loopbody);
        }
        intp.setLooping(false);
```

```
        }
    | "break"    { intp.setLooping(false); }
    | #("connect" a=expr b=expr c=expr d=expr)
        {
            SessionUtil.openConnection(a, b, c, d);
        }
    | #("print" a=expr)
        {
            System.out.println(a.getValueString());
        }
    | #(STATEMENT (stmt:. { data = expr(#stmt); } )*)
    | #(PROGRAM (program:. { data = expr(#program); } )*)
    ;

idexpr returns [String data]
{
    data = "";
}
    : id:ID         { data = id.getText(); }
    ;

refexpr returns [String data]
{
    data = "";
}
    : e:ENTITY_TYPE { data = e.getText(); }
    | "criteria"    { data = "criteria"; }
    | "order"       { data = "order"; }
    ;

loopexpr returns [Data data]
{
    data = null;
}
    : #(STATEMENT (stmt:.
        {
            if (intp.isLooping()) {
                data = expr(#stmt);
            }
        } )*)
    ;
```