EZQL: Simplifying SQL queries

Syed Iqbal Ahmad
Bilal Bhatti

EZQL: Language Overview

1.1 Introduction

EZQL was designed to ease database programming in Java and specifically to automatically map relational data to Java Objects. The language will use syntax and keywords familiar to a database designer or administrator, allow him/her to easily create programs. These programs will generate Objects and List of Objects, as needed which the end programmer can use in his programs. Further the .ezql files will be translated to standard Java.

1.2 Why use Java

These programs will then compile into Java, allowing for ease of use by Java Developers. Java provides a variety of benefits including security, machine portability, network aware, etc. Further, there is corporate commitment to use Java for enterprise application development, especially for database driven applications, and standard Java architecture supports a model of separation of personnel roles and design tiers. We could support other languages, and are not necessarily locked into Java, but for the immediate time Java is the target language.

For more information about Java please see the Java Whitepaper at http://java.sun.com/docs/overviews/java/java-overview-1.html

1.3 Language Overview

As stated above, the EZQL will allow the database developer to use simple syntax to write database logic. He will be able to receive input arguments from a Java application and respond to them as needed. The language will support control flow, such as if statements and for loops. It will also have basic variable declarations. This will allow for logic prior to running the query.

Further writing the query will not contain any complex JDBC logic. Only a query needs to be specified. The Java objects will be automatically generated based on the fields specified in the query. This is ideal for taking relations and making them available to Java applications, and eliminates the need for any expensive O-R tools or overly cumbersome frameworks with descriptor files.

1.4 Main Language Features

The main language features are described below.

1.4.1 Loading a Table

The language will support the retrieving of attributes from a single table. The example below shows the required syntax to return the attributes a,b,c from table role. The Java object MyRole will be generated with getters and setters.

In addition a MyRoleDAO class with a static method getById will be generated to allow for the loading of a MyRole by Id. If more than one row is returned then a Java exception will be thrown. All generated methods within the DAO will throw a standard SQLException with a message explaining what happened.

```
table role as MyRole {
        row getById(int id_p) {
                //No need for quotes to escape. Just keep searching until the
                //semicolon
                return select a, b, c from user where id = :id_p;
        }
}
```

The declaration in the beginning indicates a table role is being accessed and a class MyRole should be generated. The MyRole class will have three elements based on the data.

We plan to support two return data types, object and collection. A collection will be used to indicate the return type contains more than one row of data.

1.4.2 Variables, Control Flow

We want to support standard language features including the use of variables, if statements and for loops.

An if statement can be used to perform conditional checks. In this example it is being used to determine if a database trip should be made. In the generated code, null will be returned.

```
table role as MyRole {
        row getById(int id_p) {
                if (id_p < 0) {
                        return;
                }
                return select a, b, c where id = :id_p;
        }
}
```

The name of the file on disk should be MyRole.esql, otherwise a compiler error will be generated.

In this example, a simple calculation is performed before the query is returned back.

```
table grades as StudentGrades {
        //get classes where the grade is greater than the average of two exams.
        row getClass(int exam1, int exam2) {
```

```
                int avg = (exam1+exam2)/2;
                return select class from grades where grade >= :avg;
        }
}
```

1.4.3 Packaging The Generated Result

```
package com.xyz.admin;

table role as MyRole {
        row getById(int id_p) {
                if (id_p < 0) {
                        return;
                }
                return select a, b, c where id = :id_p;
        }
}
```

A package statement should be specified for each file declaration, so that the generated Java code will have a package. If no package is specified then the default package is assumed.

1.4.4 Writing Arbitrary queries

```
   view MyReport {
        collection doSomething() {
                return select tab_a.a, tab_b.b, tab_b. c from tab_a, tab_b where
                        tab_a.id=tab_b.id;
        }
   }
```

The view structure will allow for writing any arbitrary query, with the same basic language features available, such as passing arguments, using if statements, for loops, etc.

1.4.5 Advanced Features

We plan to support a variety of advanced features, which will allow the database developer to write a small amount of code to load out data.

One example is the relationship:

```
        relationship user_roles {
                        cardinality one user on userid;
                        cardinality many role on idnumber;
        }
```

The relationship can be used in queries easily load one-many or even many-many data

elements.

Another example is the leveraging of primary keys. By having a language keyword, **primarykey** we can allow the simple loading back of a whole table by the primary key without writing any sql.

1.4.6 Some Final Notes

A properties file will have to be available specifying how to obtain a connection to the database. This file will also contain a mapping of Java types to database types. This file will need to be available to compiler in order to proceed with compiling of .ezql files.

1.5 Related work

A variety of frameworks have been created to handle database access, such as EJB, Hibernate, JDO. Our main objective currently is not to replace these frameworks, rather provide a solution that complements them. We intend to focus on being able to retrieve data from a database in an ad hoc manner and not support all of CRUD operations. Actually we will initially only support the SELECT statements.

The impetus for this scripting language is the fact that all of the above-mentioned frameworks are fairly complex. They require multiple artifacts that need to be maintained. In addition they are often overkill or too restrictive, such as when complex queries are involved. In which case almost always it is recommend writing custom SQL for enhancing performance or simply to bypass the restrictions imposed by these frameworks.

In specific our solution will differ from the above in the following areas:

- No need to specify metadata information in the form of XML or other formats
- Be able execute arbitrary SELECT statements with joins between multiple tables
- Retrieve rows from a single table or by a complex query as standard POJOs
- To programmatically support caching of query results
- Provide basic flow control to manipulate the results

We hope to provide a simpler mechanism for data access with basic flow control and error handling. We hope to accomplish this by providing a simpler grammar, which will be translated to Java code, backed by JDBC.