Working with SQL

1. Working with SQL

Note:

As of this writing, the SQL support is in early stages. Some things described here are hardly more than planned and the API will probably change rapidly.

Quite many Java applications working with source generators are also working with SQL. It seems natural to develop a generic SQL generation framework, because it can make the actual generator much more simple and readable.

The basic concepts are comparable to the Java source generation framework: There is a factory, which allows you to create schema, table, column, index, statement objects and the like. And there is also an API which allows to manipulate these objects on a certain level of abstraction.

In what follows we describe the concept of the SQL generator by means of some examples. The generator has the same objects that the actual database will have: A schema, tables, columns, statements, and so on.

2. Creating a schema

The SQL generator assumes that any table belongs to a schema. In other words, before creating a table, you have to create a schema:

```
import org.apache.ws.jaxme.sqls.Schema;
import org.apache.ws.jaxme.sqls.SQLFactory;
import org.apache.ws.jaxme.sqls.impl.SQLFactoryImpl;

SQLFactory factory = new SQLFactoryImpl();
Schema schema = factory.newSchema("myschemaname");
System.out.println(schema.getName().toString()); // Prints myschemaname
```

However, it is not unusual to avoid the schema. Even more, there are SQL databases like MySQL which do not even support a schema. In that case you simply use the so-called default schema. The only difference between the default schema and other schemas is that the former doesn't have a name.

```
Schema schema = factory.getDefaultSchema();
System.out.println(schema.getName().toString()); // Prints null
```

3. Creating a table

Now that we have a schema, we may populate it with tables and the tables with columns. Suggest the following statement:

```
CREATE TABLE mytable (
  id INTEGER NOT NULL PRIMARY KEY,
  name VARCHAR(40) NOT NULL,
  price DOUBLE,
  UNIQUE(name));
```

In the framework this looks like the following:

```
import org.apache.ws.jaxme.sqls.Table;
import org.apache.ws.jaxme.sqls.Column;

Table mytable = schema.newTable("mytable");
Column id = table.newColumn("id", Column.Type.INTEGER);
id.setNullable(false);
mytable.newPrimaryKey().addColumn(id);
Column name = table.newColumn("name", Column.Type.VARCHAR);
name.setLength(40);
name.setNullable(false);
mytable.newKey().addColumn(name);
Column price = table.newColumn("price", Column.Type.DOUBLE);
```

4. Creating an INSERT statement

Now that we have our tables, we probably want to create an INSERT statement like

```
INSERT INTO mytable (id, name, price) VALUES (?, ?, 0.0)
```

Here's how we do this:

```
import org.apache.ws.jaxme.sqls.InsertStatement;
import org.apache.ws.jaxme.sqls.SQLGenerator;
import org.apache.ws.jaxme.sqls.impl.SQLGeneratorImpl;

SQLGenerator generator = new SQLGeneratorImpl();
InsertStatement stmt = table.getInsertStatement();
System.out.println(generator.getQuery(stmt));
```

The `getInsertStatement()` method is a shortcut for the following:

```
InsertStatement stmt = factory.newInsertStatement();
stmt.setTable(mytable);
stmt.addSet(id);
stmt.addSet(name);
stmt.addSet(price, (double) 0.0);
```

5. Creating a SELECT statement

Quite similar to the above is the creation of a SELECT statement:

```
SELECT id, name, price FROM mytable
```

A shortcut for the above would be:

```
import org.apache.ws.jaxme.sqls.SelectStatement;

SelectStatement stmt = table.getSelectStatement();
System.out.println(generator.getQuery(stmt));
```

The longer version goes like this:

```
SelectStatement stmt = factory.newSelectStatement();
stmt.setTable(mytable);
stmt.addResultColumn(id);
stmt.addResultColumn(name);
stmt.addResultColumn(price);
```

However, we possible want to add a WHERE clause like

```
WHERE id=? AND price > 1.0
```

This can be done like the following:

```
import org.apache.ws.jaxme.sqls.CombinedConstraint;

CombinedConstraint whereClause = stmt.getWhere();
BooleanConstraint idClause = whereClause.createEQ();
idClause.addPart(id);
idClause.addPlaceHolder();
BooleanConstraint priceClause = whereClause.createEQ();
priceClause.addPart(price);
priceClause.addPart((double) 1.0);
System.out.println("SELECT statement with WHERE clause: " +
                   generator.getQuery(stmt));
```

6. Creating an UPDATE statement

A typical UPDATE statement for `mytable` would be

```
UPDATE mytable SET name = 'foo', price = ? WHERE id = ?
```

The SQL generator allows to do that in the following manner:

```
import org.apache.ws.jaxme.sqls.UpdateStatement;

UpdateStatement stmt = schema.newUpdateStatement();
stmt.setTable(table);
stmt.addSet("name", "foo");
stmt.addSet("price");
CombinedConstraint whereClause = stmt.getWhere();
BooleanConstraint idClause ) whereClause.createEQ();
idClause.addPart(id);
idClause.addPlaceHolder();
System.out.println("UPDATE statement: " +
                   generator.getQuery(stmt));
```

For this particular `UPDATE` query, there also is a shortcut:

```
UpdateStatement stmt = table.getUpdateStatement();
System.out.println("UPDATE statement: " +
                   generator.getQuery(stmt));
```

7. Creating a DELETE statement

Most probably you already guess how to create the following `DELETE` statement:

```
DELETE FROM mytable WHERE id = ?
```

The shortcut is:

```
import org.apache.ws.jaxme.sqls.DeleteStatement;

DeleteStatement stmt = table.getDeleteStatement();
System.out.println("DELETE statement: " +
                   generator.getQuery(stmt));
```

And here is the longer version:

```
DeleteStatement stmt = schema.newDeleteStatement();
stmt.setTable(table);
```

```
CombinedConstraint whereClause = stmt.getWhere();
BooleanConstraint idClause ) whereClause.createEQ();
idClause.addPart(id);
idClause.addPlaceHolder();
System.out.println("DELETE statement: " +
                   generator.getQuery(stmt));
```

8. Creating one or more CREATE TABLE statements

Is it possible to recreate the above CREATE TABLE statement from the above `table` variable? Of course it is:

```
Collection c = generator.getCreate(table);
for (Iterator iter = c.iterator();  iter.hasNext();  ) {
  String s = (String) iter.next();
  System.out.println("Statement: " + s);
}
```

You probably wonder why the `getCreate(Table)` method returns a collection? And you are mostly right, as the above collection will typically contain a single statement. However, it is not unusual that an index must be created in a separate statement. In this case it might happen, that the collection returns multiple strings.

It is also possible to create a schema:

```
Collection c = generator.getCreate(schema);
for (Iterator iter = c.iterator();  iter.hasNext();  ) {
  String s = (String) iter.next();
  System.out.println("Statement: " + s);
}
```

However, you possibly want to create not only the schema, but the tables as well. Voila:

```
Collection c = generator.getCreate(schema, true);
for (Iterator iter = c.iterator();  iter.hasNext();  ) {
  String s = (String) iter.next();
  System.out.println("Statement: " + s);
}
```

9. Vendor specific extensions

Vendor specific extensions and vendor specific SQL dialects are available through subclasses of `SQLFactoryImpl`. As of this writing, the only available subclass is `DB2SQLFactoryImpl`. To use it, simply replace

```
import org.apache.ws.jaxme.sqls.SQLFactory;
import org.apache.ws.jaxme.sqls.SQLFactoryImpl;

SQLFactory factory = new SQLFactoryImpl();
```

with

```
import org.apache.ws.jaxme.sqls.db2.DB2SQLFactory;
import org.apache.ws.jaxme.sqls.db2.DB2SQLFactoryImpl;

SQLFactory factory = new SQLFactoryImpl();
```

In the special case of DB2 this will soon enable access to features like table spaces or buffer pools.