XML To DDL

"Bringing some sanity to database maintenance."

Table of Contents

Introduction

`XML to DDL` is a set of python programs to convert an XML representation of a database into a set of SQL (or DDL: Data Definition Language) statements.

In addition `XML to DDL` can examine the difference between two XML files and output a sequence of ALTER statements that will update the database to conform to the new schema.

If you install the required python package [1] you can download the XML schema from the database directly (in the SVN repository).

Finallly, `XML to DDL` can generate HTML documentation of your schema.

You can find more information and download files at the Berlios page

And you can find documentation through the many test cases.

You can add comments or help in the wiki pages.

Simple Example

The following is a simple schema XML definition of a database:

```
<schema>
    <table name="students" fullname="List of Students"
        desc="List of students with their full names">
        <columns>
            <column name="id" fullname="Primary Key" type="integer" key="1"
                desc="Primary key for the table"/>
            <column name="student_name" fullname="Student Name" type="varchar" size="80"
                desc="The full name of the student"/>
        </columns>
    </table>
</schema>
```

[1] For PostgreSQL you need `psycopg`, For MySQL you need `MySQLdb` and for Firebird you need `kinterbasdb`.

Here we run the program indicating output for PostgreSQL:

```
python xml2ddl.py --dbms postgres schema1.xml
```

We get the following output:

```
DROP TABLE students;
CREATE TABLE students (
        id integer,
        student_name varchar(80),
        CONSTRAINT pk_students PRIMARY KEY (id));
COMMENT ON TABLE students IS 'List of students with their full names';
COMMENT ON COLUMN students.id IS 'Primary key for the table';
COMMENT ON COLUMN students.student_name IS 'The full name of the student';
```

If we run the program again for Firebird:

```
python xml2ddl.py --dbms firebird schema1.xml
```

we'll get different output:

```
DROP TABLE students;
CREATE TABLE students (
        id integer,
        student_name varchar(80),
        CONSTRAINT pk_students PRIMARY KEY (id));
UPDATE RDB$RELATIONS SET RDB$DESCRIPTION = 'List of students with their full names'
        WHERE RDB$RELATION_NAME = upper('students');
UPDATE RDB$RELATION_FIELDS SET RDB$DESCRIPTION = 'Primary key for the table'
        WHERE RDB$RELATION_NAME = upper('students') AND RDB$FIELD_NAME = upper('id');
UPDATE RDB$RELATION_FIELDS SET RDB$DESCRIPTION = 'The full name of the student'
        WHERE RDB$RELATION_NAME = upper('students') AND RDB$FIELD_NAME = upper('student_name');
```

The example shows a feature of `XML to DDL`, database independence. Currently the program supports the Firebird, PostgreSQL, and MySQL databases, but more will probably become available.

Differencing Example

Another key feature is the ability to examine the changes done to the XML and generate the DDL statements necessary to perform the changes to the database. If this is a new XML schema (schema2.xml):

```
<schema>
    <table name="students" fullname="List of Students"
        desc="List of students">
        <columns>
            <column name="id" fullname="Primary Key" type="integer" key="1"
                desc="Primary key for the table"/>
            <column name="student_name" fullname="Student Name" type="varchar" size="100"
                desc="The full name of the student"/>
            <column name="email" fullname="Electronic mail address" type="varchar" size="100"
                desc="The primary email for the student"/>
        </columns>
    </table>
</schema>
```

Running this program:

```
python diffxml2ddl.py --dbms postgres schema1.xml schema2.xml
```

Produces the following DDL output:

```
ALTER TABLE students ALTER student_name TYPE varchar(80);
ALTER TABLE students DROP email;
COMMENT ON TABLE students IS 'List of students with their full names';
```

However, an older version of PostgreSQL doesn't support altering the column type:

```
python diffxml2ddl.py --dbms postgres7 schema1.xml schema2.xml
```

The a temporary column needs to be created, the data copied over and the old column dropped:

```
ALTER TABLE students ADD tmp_student_name varchar(80);
UPDATE students SET tmp_student_name = student_name;
ALTER TABLE students DROP student_name;
ALTER TABLE students RENAME tmp_student_name TO student_name;
ALTER TABLE students DROP email;
COMMENT ON TABLE students IS 'List of students with their full names';
```

You can find a *complete* list of examples here: http://xml2ddl.berlios.de/testdetails.html

Using a dictionary

If you find yourself repeating the same attributes in your XML schema over and over you can put these in a dictionary:

```
<dictionary name="column">
    <dict class="key" name="id" fullname="Primary Key" type="integer" null="no" key="1"
        desc="Primary key for the table" />
</dictionary>
```

In this example we are telling the parser that the dictionary is for the nodes called `column` and when it sees the class `key`, it should put in the the other attributes listed. So using this dictionary this would be equivalent::

```
...
<columns>
    <column class="key"/>
</columns>
...
```

as:

```
...
<columns>
    <column name="id" fullname="Primary Key" type="integer" null="no" key="1"
        desc="Primary key for the table"/>
</columns>
...
```

In addition you can override any attributes in the dictionary, for example this:

```
...
<columns>
    <column class="key" name="student_id"/>
</columns>
...
```

would then be equivalent to:

```
...
<columns>
    <column name="student_id" fullname="Primary Key" type="integer" null="no" key="1"
        desc="Primary key for the table"/>
</columns>
...
```

The dictionaries can also support multiple inheritance through the `inherits` attribute. Here's a rather contrived example:

```
<dictionary name="column">
        <dict class="index" type="integer" null="no"/>
        <dict class="pk"   key="1"/>
        <dict class="key" inherits="index,pk" name="id" fullname="Primary Key"
                desc="Primary key for the table"/>
</dictionary>
```

Downloading the XML

You can download the XML schema directly from the database. Requires a connection that conforms to the Database API. Without changing the code you can use the following connectivities:

Database	Connectivity
PostgreSQL	`psycopg`
MySQL	`MySQLdb`
Firebird	`kinterbasdb`

Here's how to use the command:

```
python downloadXml.py --dbms <dbms> --database <database> --user <user> --pass <pass> > <filename.xml>
```

`dbms` can be one of `postgres`, `mysql`, or `firebird`. Defaults to "postgres".
`database` the name of the database, defaults to "postgres"
`user` the user name to connect to the database, defaults to "postgres"
`pass` the password to user, defaults to "postgres"
`filename.xml` by default it sends the XML to the console (stdout) you can pipe the output to a file as shown above.

Outputting HTML Documentation

Some of the attributes in the XML are used solely for documentation purposes. For example, `fullname` has no equivalent in most DBMSs. Another, it `deprecated` which indicates that a column or table should no longer be used, but hasn't been deleted yet.

Here's how to output the HTML document:

```
python xml2html.py --file schema.html schema.xml
```

Annotated XML

The following is a list of the tags and attributes that xml2ddl accepts or is planned to be accepted in the future. The attributes enclosed in [square brackets] are optional. Also there are lot of thing not supported yet, and are so indicated. Basically, all the tags below except for <schema> is optional. Note, as all XML files the tag names and attribute name (eg. <table>) is case sensitive (i.e. <Table> will not work!). Attribute, values are case insensitive, (eg. dotschema="Yes" and dotschema="yes" should both work).

```
<schema>
    <include/>
    ...

    <dictionary>
        <dict/>
        ...
    </dictionary>
    ...

    <table>
        <columns>
            <column/>
            ...
        </columns>
        <indexes>
            <index/>
            ...
        </indexes>
        <relations>
            <relation/>
```

```
            ...
        </relations>
        <contraints>
            <contraint/>
            ...
        </contraints>
        <triggers>
            <trigger>
                ..
            </trigger>
            ...
        </triggers>
    </table>
    ...

    <view>
        -- view contents
    </view>
    ...

    <function>
        -- function contents.
    </function>
    ...
</schema>
```

Here are the details of each of the XML tags.

```
<schema [name="1"]
        [dotshema="2"]
        [generated="3"]>
```

Not all databases have schemas, but you still need the tag.

(1) The name of the schema to use.

(2) "yes" or "no". Indicates whether the table names will require the schema name before (i.e. "schema.table") **- Not supported**

(3) If set to "yes" indicates that the XML was generatd from `dowloadXml`.

```
<include href="1"/>
```

You can use includes to break a large XML schema into smaller pieces.

(1) Is the filename of the XML to include.

```
<dictionary name="1">
    <dict class="2" 3="4"/>
</dictionary>
```

The dictionary is a general system for adding attributes.

(1) Here you place the name of the *tag* you want to apply this to. Typically, it should be applied to "column" but it could be "table", "index", etc.

(2) The classname you've given this.

(3) The attribute name to add.

(4) The value of the attribute to add.

```
<table name="1"
      [oldname="2"
      [fullname="3"]
      [desc="4"]
      [inherits="5"]>
```

Create a table definition. The order may be important since xml2ddl isn't too careful about creating contraints before the table exists.

(1) The name of the database table

(2) You must enter oldname if you want to rename a table.

(3) The full name of the table, usually just the table name with spaces instead of underscores, for example. This is purely for documentation purposes.

(4) A long description of the table. The description is stored in the database if possible.

(5) The idea is to specify another table which this table will inherit columns from. It would possibly inherit the indexes, triggers, and constraints too. For databases that don't support the features it will repeat the values. **- Not supported**

```
<columns>
    <column name="1"
           [oldname="2"]
           [fullname="3"]
           [desc="4"]
            type="5"
           [size="6"]
           [precision="7"]
           [null="8"]
           [unique="9"]
           [key="10"]
           [default="11"]
           [autoincrement="12"]
           [deprecated="13"]/>
    </column>
</columns>
```

The <columns> tag gives an order list of attributes. Currently, xml2ddl doesn't reorder the columns if you move things around.

(1) Name of the column (aka attribute, aka field). Note I chose the name *column* instead of *attribute* because I felt it would be easier for beginners to grasp.

(2) You need to enter the oldname if you want to rename a column.

(3) Fullname used only for documentation. Typically, it the same as *name* but with spaces and any hungarian notation removed.

(4) Long description of the attribute.

(5) The type of the column, should probably stick with the SQL99 types, if possible.

(6) The size of the column, used for varchar() and the like.

(7) The precision of the numeric value, must be used in conjuction with *size*. `type="numeric" size="10" precision="2"` would produce `decimal(10, 2)`.

(8) "yes" or "no" or "not". `no` or `not` expands to `NOT NULL`. The default is `NULL` if nothing is specified.

(9) If "yes" then the column will have a unique constraint added to it. The name of the constraint will be `unique_<colname>`. **- Not supported**

(10) A number from 1 to *N*. Indicates that this column will participate in being a primary key. Every table *should* have a primary key, but it isn't enforced.

(11) Default value, if any. If none used, it typically defaults to NULL.

(12) If set to "yes" will try and make this column autoincrement if NULL is passed to in in an insert. On some systems I'll create a sequence and a trigger or default value. Typically, you will need to put in `null="no"` and `key="1"` as well.

(13) Value "yes" if used. Means that the column is deprecated and shouldn't be used (but it still exists in the database). This is used purely for documentation purposes.

```
<colums>
    <column ....>
        <enumeration [name="1"] [fullname="2"] [desc="3"] [constraint="4"]>
            <enum val="5" [display="6"] [desc="7"]/>
            ...
        </enumeration>
    </column>
</columns>
```

- Not supported Enumerations is a limited list of values that a column can contain. One purpose of enumerations is to aid in coding, to automaticaly create an enum in code, *forcing* the developer to use one of the enumerated types.

(1) Name to use for the enumeration constraint, and/or the enumeration in code.

(2) Full name of the constraint, for documentation purposes.

(3) Description of the enumeration.

(4) "yes" if a constraint should be created if possible for the DBMS.

(5) The actual value stored in the database. Must be provided.

(6) What to typically display to the user, if omitted, assumed to be `val`.

(7) A long description of the value, to put in help, perhaps.

```
<relations>
    <relation [name="1"]
              [oldname="2"]
               column="3"
               table="4"
              [fk="5"]
              [ondelete="6"]
              [onupdate="7"]/>
</relations>
```

Relations is an unordered list of foreign key contraints to other tables and columns. For DBMS that don't support this, the relations would be used only for documentation purposes.

(1) The name of the constraint, defaults to `fk_<column>` if not provided.

(2) If you rename the relation need to put the original name here. **- Not supported**.

(3) The list of columns of this table that forms part of the relation separated by commas. Note I may either change the name to `columns` or just support both `column` and `columns`.

(4) The name of the other table that forms part of the relation.

(5) The name of the other columns that form part of the relation, separated by commas. If no name is given it defaults to the same name(s) as given in column.

(6) If used should pass `cascade` or `setnull`.

(7) If used should pass `cascade` or `setnull`.

```
<indexes>
    <index [name="1"]
           [oldname="2"]
            columns="3"
           [unique="4"]
           [using="5"]
           [where="6"]/>
</indexes>
```

Index are an unorder list of indexes on a table (i.e. the order of the <index/> tags does not matter).

(1) The name of the index. Defaults to `idx_<table><columns>` where the columns are separeted by underscores.

(2) Must provide the old name if you want to rename the index **- Not supported**

(3) List of columns that form part of the index separated by commas.

(4) If set to "yes" then it creates a unique index. **- Not supported**

(5) Type of index to create. **- Not supported**

(6) Where clause. **- Not supported**

```
<constraints>
    <constraint [name="1"]
                [oldname="2"]
                [longname="3"]
                [desc="4"]
                 columns="5"
                [unique="6"]
                [check="7"]/>
</constraints>
```

The <constraints> tag lists an unorder list of contraint rules, if the database supports it. **- Not supported**

(1) The name of the constraint. Defaults to `con_<table><columns>` where the columns are separeted by underscores. **- Not supported**

(2) If you rename the constraint you must put the old name here. **- Not supported**

(3) The long name of the constraint if any (for documentation only)

(4) The description of the constraint, for documentation.

(5) List of columns that form part of the constaint separated by commas **- Not supported**

(6) If `yes` means it's a unique constraint **- Not supported**

(7) If set, it's a check constraint **- Not supported**

```
<triggers>
    <trigger  name="1"
              [oldname="2"]
              [longname="3"]
              [desc="4"]
               timing="5"
               events="6"
              [fire="7"]
              [function="8"]>
         (9)
    </trigger>
</triggers>
```

- Not supported The <triggers> tag lists an unorder list of triggers for the table, if the database supports it.

(1) The name of the trigger, required.

(2) If you rename the trigger must put the old name here.

(3) The long name of the trigger, if any (for documentation only)

(4) The description of the trigger for documentation purposes.

(5) The timing of the trigger, one of 'before' | 'after'

(6) The events that causes the trigger. One of 'insert', 'update', or 'delete'. Multiple events can be specified by separating with commas.

(7) Specifies whether the trigger fires 'once' or 'per-row'.

(8) The name of an existing function to call on the trigger event, if the database supports this.

(9) The body of the trigger. Can't have used `function` as well, it's one or the other.

```
<dataset [only="1"]>
    <val 2="3"/>
</dataset>
```

A dataset is a set of data that should be in the table. Useful, when you need to store a small set of values in the table.

(1) If set to true, the program will clean out the table before inserting the values.

(2) The left hand side of the equals is the name of the column to store this data value

(3) The right hand side of the equals is the value to store in this data cell. For example, `<val id="1" name="Bob"/>` creates `INSERT INTO table (id, name) VALUES (1, 'Bob')`

```
<view name="1"
     [fullname="2"]
     [desc="3"]
     [columns="4"]>
    (5)
</view>
```

- Not supported Create a view to the table.

(1) Name of the view to be stored in the database.

(2) Typically, this is the name with spaces added.

(3) A full description of the view.

(4) You can optionally specify the column names, but most DBMS can infer them from the select statement.

(5) The contents of the view.

```
<function name="1"
         [oldname="2"]
         [fullname="3"]
         [desc="4"]
         [arguments="5"]
         [returns="6"]
         [language="7"]
         [dbms="8"]
         [volatile="9"]>
    (10)
</function>
```

- Not supported You can specify the body of a stored procedure or function.

(1) Name of the function or procedure to be stored in the database.

(2) If you rename the function you must place the old name here. **- Not supported**

(3) Typically, this is the name with spaces added.

(4) A full description of the function.

(5) Comma separeted list of arguments. If no arguments, void is assumed.

(6) If `returns` is not there or empty it's considered a procedure.

(7) Language is assumed "SQL" or "PL/SQL" if not specified.

(8) Because the code is likely to change depending on the database system used you could specify the same function multiple times, one for each type of DBMS. If not ther all dbms systems are assumed.

(9) Can be "yes", "no", or "stable". This is an execution hint for PostgreSQL.

(10) The contents of the function or procedure.

Advantages

Storing the schema in this form has some advantages:

1. All the information about a table is stored together in one place. Finding linked tables, sequence tables etc. should be simplified.
2. Being text it can easily be stored in a VCS Repository, like Subversion or CVS.
3. Also because it is text you can compare differences between older and newer versions. In fact this is one of the main goals of this project.
4. Since the description of the schema is abstract, it isn't tied to a specific database.
5. Documentation can easily be generated from the XML schema.
6. A pretty schema diagram can be drawn from the XML see Dia and Dot (note, this functionality hasn't been implemented yet).
7. A history of changes made to the table (by whom, when and why) can all be contained in the repository. Normally, metadata changes made to a database never stored anywhere.
8. Migration scripts can be stored in the meta-data for certain changes that require the data to be modified. For example, if a column is split into two columns the procedure to make this modification can be stored into the repository (not implemented yet).
9. Destructive changes can have backed ups made as part of its process. For example, if a column is to be deleted that column along with its primary key(s) can be stored into a file. This way if they do undo the changes they can do so without needing to go to a full backup. (to do)
10. Additional useful information can be stored in the XML. Columns can be flagged as deprecated or obsolete, for example.
11. Scripts can be generated to automatically check that the data fits the domain. For example, that status is 1, 2, 3, or 4 or that telephone numbers are in the format (999) 9999-99999. (to do)
12. Code can use the XML to it's own purposes. One example is to write code that figures out the best joins to use between two tables. Another example is to change a status code (ex. 1, 2, or 3) into an enumeration (ex. READY, PROCESSING, DONE).

To do

Here are the major directions I see `XML to DDL` going:

- Support for more databases (currently I've written code only for PostgreSQL, Firebird, and MySQL). Note for Firebird users, there is a chance I'll temporarily drop support for Firebird and fill out the feature set for MySQL and PostgreSQL first. For MySQL users I'll probably drop support for versions before 5.0. I'm really hoping that others will step up and implement the support for their favorite DBMS once I have good support done for these two DBMSs.
- Build the XML schema from an existing database. Basic implementations for Postgres, MySQL, and Firebird is already done.
- Support comparing differences from the database as well as another XML file. This is a bit different since the database may be more up-to-date, but the XML probably has more information (like fullname).

- Support for some database specific features.
- Hooks for developers to put in their own code on certain events.
- Filling out the missing funcionality listed above as listed as '**- Not supported**'.

Similar Work

I've been pointed to another project which looks similar calle ERW A quick look shows that it tries to work at a higher level than my XML does (i.e. more abstract). It also generates code for PHP and produces nicer documentation.