3D City Database for CityGML

3D City Database Version 2.0.6-postgis
Importer/Exporter Version 1.5.0-postgis

Release Version

Port documentation: PL/SQL to PL/pgSQL

21 January 2013

Geoinformation Research Group
Department of Geography
University of Potsdam

Felix Kunde
Hartmut Asche

Institute for Geodesy and
Geoinformation Science
Technische Universität Berlin

Thomas H. Kolbe
Claus Nagel
Javier Rerreruela
Gerhard König
Content:

1 Introduction................................................................................................................... 4

2 General differences........................................................................................................ 5
   2.1 Basics................................................................................................................... 5
   2.2 Procedures and functions..................................................................................... 5
   2.3 Messages............................................................................................................. 6
   2.4 Dynamic SQL...................................................................................................... 7
   2.5 Cursors............................................................................................................... 7
   2.6 Recursive SQL..................................................................................................... 8
   2.7 Global Temporary Tables.................................................................................... 9

3 Explicit differences....................................................................................................... 10
   3.1 Packages and user-defined types....................................................................... 10
   3.2 Working with user-defined types....................................................................... 12
   3.3 Differences in system tables............................................................................. 14
   3.4 Non-translated parts.......................................................................................... 17
   3.5 Additional functions.......................................................................................... 18

4 References.................................................................................................................. 18
1. Introduction

Welcome to the documentation about ported PL/SQL scripts for the PostGIS version of the 3D City Database (abbreviated as 3DCityDB in the following). The 3DCityDB contains PL/SQL stored procedures which are used by the Importer/Exporter tool. They help to reduce the number of JDBC connections by letting the database undertake a group of tasks. Fortunately PostgreSQL’s procedural language of SQL PL/pgSQL comes close to the PL/SQL grammar which facilitated the porting of scripts. This documentation will present some general translation examples that appeared when porting the 3DCityDB to PostGIS (chapter 2). Parts that could not be translated directly will appear in the third chapter.

For the Oracle version the procedures and functions were grouped into packages. In Oracle packages are used to structure stored procedures and also to hide helper functions that do not fulfill a purpose by itself from a public user interface. Their architecture is very much object oriented (details in chapter 3). However, regarding PostgreSQL the package concept only exists in the commercial Plus Advance Server by EnterpriseDB. Another alternative that is suggested by the PostgreSQL documentation [www1] and which was implemented in the end, is the usage of schemas. A schema is a separate namespace with own tables, views, sequences, functions etc. Packages from the Oracle release are represented in one PostgreSQL schema called “geodb_pkg”. For a better overview the functions were given name prefixes:

<table>
<thead>
<tr>
<th>former package name</th>
<th>Prefix</th>
<th>Source (PL_pgSQL/GEODB_PKG/)</th>
</tr>
</thead>
<tbody>
<tr>
<td>geodb_delete_by_lineage</td>
<td>del_by_lin_</td>
<td>DELETE/DELETE_BY_LINEAGE.sql</td>
</tr>
<tr>
<td>geodb_delete</td>
<td>del_</td>
<td>DELETE/DELETE.sql</td>
</tr>
<tr>
<td>geodb_idx</td>
<td>idx_</td>
<td>INDEX/IDX.sql</td>
</tr>
<tr>
<td>geodb_match</td>
<td>match_</td>
<td>MATCHING/MATCH.sql</td>
</tr>
<tr>
<td>geodb_merge</td>
<td>merge_</td>
<td>MATCHING/MERGE.sql</td>
</tr>
<tr>
<td>geodb_stat</td>
<td>stat_</td>
<td>STATISTICS/STAT.sql</td>
</tr>
<tr>
<td>geodb_util</td>
<td>util_</td>
<td>UTIL/UTIL.sql</td>
</tr>
</tbody>
</table>

For each example a small info box will signalize its occurrence in the functional groups (gray if not occurred or not needed to be translated).
2. General differences

2.1 Basics

The block structure of a function in PL/SQL and PL/pgSQL is very similar. Just look at the example to spot the differences. In PL/pgSQL the function body has to be quoted with ‘…’ or $$ ... $$ or $BODY$ ... $BODY$. In the function specification of PL/pgSQL the RETURN definition is slightly different. RETURN datatype IS becomes RETURNS datatype AS.

```sql
FUNCTION exp_func(params) RETURN datatype IS
    DECLARE
    BEGIN
    END;
/
```

```
CREATE FUNCTION exp_func(params) RETURNS datatype AS
$$
DECLARE
    BEGIN
    END;
$$
LANGUAGE plpgsql;
```

2.2 Procedures and functions

Procedures do not provide a returning value, but functions do. PL/pgSQL only knows functions. But they can still act like procedures by returning the empty void data type. They do not even need a RETURN block in the function body. The keyword SETOF was used to receive a 0 row result set. In the Oracle examples the CREATE keyword is always missing because of the use of packages (see next chapter).

```sql
PROCEDURE exp_proc(params)

CREATE FUNCTION exp_proc(params) RETURNS SETOF void AS
```

If no parameters are assigned to a function PL/pgSQL still needs an empty block of brackets, PL/SQL does not.
Sometimes it is necessary to assign default values to function parameters. This is done with the `DEFAULT` keyword or its abbreviation “:=”. PL/pgSQL cannot compile the short form when it stands inside the function specification.

The same applies to row-type variables (:%ROWTYPE). It is not possible to pass a record data type as a function argument. This case appeared in the delete package but could be substituted by handing over just the ID value of a record type as it was mostly the only parameter needed for the function.

If a function or procedure is calling another procedure PL/pgSQL needs the keyword `PERFORM`.

```sql
CREATE FUNCTION exp_func() RETURNS datatype AS
BEGIN
    procedure(params);
END;
```

```sql
CREATE OR REPLACE FUNCTION geodb_pkg.del_pre_delete_citymodel(citymodel_rec_id NUMERIC) RETURNS SETOF void AS
BEGIN
    PERFORM procedure(params);
END;
```
2.3 Messages

For writing messages on the output prompt the `dbms_output` package is used in *Oracle*. For PL/pgSQL `RAISE NOTICE` is equivalent to this. It can use placeholders instead of concatenating a string.

```sql
dbms_output.put_line('message for id %: %', exp_id, SQLERRM);
RAISE NOTICE 'message for id %: %', exp_id, SQLERRM;
```

2.4 Dynamic SQL

When using dynamic SQL queries are created and changed during runtime which is very useful for database applications. These queries are executed as a string which the database can reuse without parsing and checking its grammar again. It is possible to exchange variables in the statement. They are substituted by placeholders and bind to the query with the `USING` keyword (bind variables). The differences between PL/SQL and PL/pgSQL are marginal as seen in the following example.

```sql
EXECUTE IMMEDIATE 'SELECT column FROM table WHERE column=:1'
INTO var USING bind_var;

EXECUTE 'SELECT column FROM table WHERE column=$1'
INTO var USING bind_var;
```

2.5 Cursors

The handling of cursors could directly be ported to PL/pgSQL as they are only used in `FOR` loops. The declaration has to be changed.

```sql
CURSOR exp_cur IS SQL-Statement
exp_cur CURSOR FOR SQL-Statement
```
2.6 Recursive SQL

In a hierarchical and also recursive model like CityGML the performance of queries can greatly benefit from recursive SQL. It guarantees that the child elements are deleted first before the corresponding parent object is removed. For the 3DCityDB this is used for deleting rows of the tables `texturparam` and `surface_geometry` by `id` and `parent_id` values. Oracle offers a specific clause that differs from the ANSI SQL standard that is implemented in PostgreSQL.

```sql
EXECUTE IMMEDIATE 'DELETE FROM textureparam WHERE surface_geometry_id IN (SELECT id FROM (SELECT id FROM surface_geometry START WITH id=:1 CONNECT PRIOR BY id=parent_id ORDER BY level DESC))' USING pid;
EXECUTE IMMEDIATE 'DELETE FROM surface_geometry WHERE id IN (SELECT id FROM (SELECT id FROM surface_geometry START WITH id=:1 CONNECT PRIOR BY id=parent_id ORDER BY level DESC))' USING pid;
EXECUTE IMMEDIATE 'DELETE FROM textureparam WHERE surface_geometry_id IN (WITH RECURSIVE recursive_query(id, parent_id, level) AS (SELECT id, parent_id, 1 AS level FROM surface_geometry WHERE id=$1 UNION ALL SELECT sg.id, sg.parent_id, rq.level + 1 AS level FROM surface_geometry sg, recursive_query rq WHERE sg.parent_id = rq.id ) SELECT id FROM recursive_query ORDER BY level DESC)' USING pid;
EXECUTE IMMEDIATE 'DELETE FROM surface_geometry WHERE id IN (WITH RECURSIVE recursive_query(id, parent_id, level) AS (SELECT id, parent_id, 1 AS level FROM surface_geometry WHERE id=$1 UNION ALL SELECT sg.id, sg.parent_id, rq.level + 1 AS level FROM surface_geometry sg, recursive_query rq WHERE sg.parent_id = rq.id ) SELECT id FROM recursive_query ORDER BY level DESC)' USING pid;
```
2.7 Global Temporary Tables

Temporary tables are defined for the match and merge scripts and used by the Matching/Merging plugin of the Importer/Exporter. As temporary tables only exist during a session PostgreSQL would not find them if initially defined during the creation of the 3DCityDB. That is why their definition was put inside of the functions that are called first in the Matching/Merging process.

```sql
PROCEDURE collect_cand_building(
    lod    NUMBER,
    lineage    cityobject.lineage%TYPE)
IS
BEGIN
  -- truncate tmp table
  EXECUTE IMMEDIATE 'TRUNCATE TABLE match_tmp_building';

  -- retrieve . . .

CREATE OR REPLACE FUNCTION geodb_pkg.match_collect_cand_building(
    lod    INTEGER,
    lineage    cityobject.lineage%TYPE)
RETURNS SETOF void AS $$
BEGIN
  -- creates the temporary table match_tmp_building
  EXECUTE 'CREATE GLOBAL TEMPORARY TABLE match_tmp_building(
    id INTEGER,
    parent_id INTEGER,
    root_id INTEGER,
    geometry_id INTEGER
  ) ON COMMIT PRESERVE ROWS';

  -- retrieve
```
3. Explicit differences

3.1 Packages and user-defined types

To understand the differences between the package structure of the PL/SQL files and the rather flat PL/pgSQL files please take a close look on the following example from the INDEX package, which also contains other features that are unknown to the PostgreSQL world.

```
--create user-defined type
CREATE OR REPLACE TYPE INDEX_OBJ AS OBJECT (  
    index_name VARCHAR2(100),  
    table_name VARCHAR2(100),  
    attribute_name VARCHAR2(100),  
    type NUMBER(1),  
    srid NUMBER,  
    is_3d NUMBER(1, 0),

--specification of member functions of user-defined type (constructors)  
STATIC FUNCTION construct_spatial_3d  
    (index_name VARCHAR2, table_name VARCHAR2, attribute_name VARCHAR2,  
    srid NUMBER := 0) RETURN INDEX_OBJ;

STATIC function construct_spatial_2d
    ...
);  
/

--bodies of member functions
CREATE OR REPLACE TYPE BODY INDEX_OBJ IS

STATIC FUNCTION construct_spatial_3d(  
    index_name VARCHAR2,  
    table_name VARCHAR2,  
    attribute_name VARCHAR2,  
    srid NUMBER := 0) RETURN INDEX_OBJ IS
    BEGIN  
    RETURN INDEX_OBJ(upper(index_name), upper(table_name),  
    upper(attribute_name), 1, srid, 1);  
    END;

STATIC FUNCTION construct_spatial_2d(  
    ...
);  
/

--CREATE PACKAGE
--create specification for package geodb_idx
CREATE OR REPLACE PACKAGE geodb_idx AS

--index_table is a nested table for INDEX_OBJ
TYPE index_table IS TABLE OF INDEX_OBJ;

FUNCTION index_status(idx INDEX_OBJ) RETURN VARCHAR2;

FUNCTION ...
END geodb_idx;
/
```
--package body
CREATE OR REPLACE PACKAGE BODY geodb_idx
AS
--package-variables which can be used by functions
NORMAL CONSTANT NUMBER(1) := 0;
SPATIAL CONSTANT NUMBER(1) := 1;
INDICES CONSTANT index_table := index_table(
INDEX_OBJ.construct.spatial_3d('CITYOBJECT_SPX', 'CITYOBJECT', 'ENVELOPE'),
INDEX_OBJ.construct.spatial_3d('SURFACE_GEOM_SPX', 'SURFACE_GEOMETRY',
'GEOMETRY'),
INDEX_OBJ.construct.normal('CITYOBJECT_INX', 'CITYOBJECT', 'GMLID,
GMLID_CODESPACE'),
INDEX_OBJ.construct.normal('SURFACE_GEOMETRY_INX', 'SURFACE_GEOMETRY',
'GMLID, GMLID_CODESPACE'),
INDEX_OBJ.construct.normal('APPEARANCE_INX', 'APPEARANCE', 'GMLID,
GMLID_CODESPACE'),
INDEX_OBJ.construct.normal('SURFACE_DATA_INX', 'SURFACE_DATA', 'GMLID,
GMLID_CODESPACE'));
--function-bodies
FUNCTION index_status(idx INDEX_OBJ) RETURN VARCHAR2
END;
END geodb_idx;
/

--create user-defined type
DROP TYPE IF EXISTS geodb_pkg.INDEX_OBJ CASCADE;
CREATE TYPE geodb_pkg.INDEX_OBJ AS (
  index_name VARCHAR(100),
table_name VARCHAR(100),
attribute_name VARCHAR(100),
type NUMERIC(1),
srid INTEGER,
is_3d NUMERIC(1, 0)
);
--no member-functions in PostgreSQL
--create constructor functions as normal functions
CREATE OR REPLACE FUNCTION geodb_pkg.idx_construct.spatial_3d(index_name VARCHAR,
table_name VARCHAR,
attribute_name VARCHAR,
srid INTEGER DEFAULT 0) RETURNS geodb_pkg.INDEX_OBJ AS $$
DECLARE
  idx geodb_pkg.INDEX_OBJ;
BEGIN
  idx.index_name := index_name;
  idx.table_name := table_name;
  idx.attribute_name := attribute_name;
  idx.type := 1;
  idx.srid := srid;
  idx.is_3d := 1;

  RETURN idx;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE STRICT;
3.2 Working with user-defined types

As seen in the previous example a constant INDICES was created. It is of the type INDEX_TABLE which is a nested table filled with 6 INDEX_OBJs. This constant is used for performing one command on all the 6 INDEX_OBJs in a FOR loop. Their single attributes are accessed via dot notation. For PL/pgSQL this loop was organized in another way as the INDEX_OBJs were stored in a normal table. The FOR loop is looping through a query result of this table. The access on the attributes of INDEX_OBJ is also done with dot notation but needs extra brackets. Note: The data type STRARRAY is a nested table of VARCHAR2 and also user-defined. It was replaced by an array of PostgreSQL’s TEXT data type.
In the UTIL package the user-defined data type DB_INFO_OBJ and the according nested table DB_INFO_TABLE were not ported. As they were only used by one function it was sufficient to let this function return a table with columns for each attribute of the former DB_INFO_OBJ. A code example follows in 3.3 on page 15.
3.3 Differences in system tables

Some functions in the INDEX and UTIL package are querying system tables of Oracle to receive certain information. Usually this information can also be found in the PostgreSQL system tables, but sometimes this works only indirectly as columns are called differently or simply do not exist.

Table with coordinate reference systems

The PostGIS pendant to Oracle’s SDO_COORD_REF_SYS table is the spatial_ref_sys table. A first look on the number of columns reveals that the retrieval of some attributes can be a bit complicated.

<table>
<thead>
<tr>
<th>SDO_COORD_REF_SYS</th>
<th>spatial_ref_sys</th>
</tr>
</thead>
<tbody>
<tr>
<td>srid</td>
<td>srid</td>
</tr>
<tr>
<td>coord_ref_sys_name</td>
<td>auth_name</td>
</tr>
<tr>
<td>coord_ref_sys_kind</td>
<td>auth_srid</td>
</tr>
<tr>
<td>coord_sys_id</td>
<td>srtext</td>
</tr>
<tr>
<td>datum_id</td>
<td>proj4text</td>
</tr>
<tr>
<td>geog_crs_datum_id</td>
<td></td>
</tr>
<tr>
<td>source_geog_srid</td>
<td></td>
</tr>
<tr>
<td>projection_conv_id</td>
<td></td>
</tr>
<tr>
<td>cmpd_horiz_sri</td>
<td></td>
</tr>
<tr>
<td>cmpd_vert_srid</td>
<td></td>
</tr>
<tr>
<td>information_source</td>
<td></td>
</tr>
<tr>
<td>data_source</td>
<td></td>
</tr>
<tr>
<td>is_legacy</td>
<td></td>
</tr>
<tr>
<td>legacy_code</td>
<td></td>
</tr>
<tr>
<td>legacy_wktext</td>
<td></td>
</tr>
<tr>
<td>legacy_cs_bounds</td>
<td></td>
</tr>
<tr>
<td>is_valid</td>
<td></td>
</tr>
<tr>
<td>supports_sdo_geometry</td>
<td></td>
</tr>
</tbody>
</table>

Fortunately all the information needed is covered by the text value in the srtext column. The relevant content is extracted with string functions which is a kind of ugly way though. Hopefully this will change in future releases of PostGIS.
Until now PostGIS does not offer 3D spatial reference systems by default. INSERT examples for PostGIS can be found at spatialreference.org [www2]. As seen before there is no column which detects the dimension of the reference system. There are also no separate views for reference systems like in Oracle (SDO_CRS_GEOGRAPHIC3D, SDO_CRS_COMPOUND). The solution can again be found inside the entries of the srtext column. Only 3D SRIDs have got an “UP” Axis.

```
EXECUTE 'SELECT count(*) FROM spatial_ref_sys WHERE auth_srid=$1 AND srtext LIKE ''%UP%'' INTO is_3d USING srid;
```
Index status

In **Oracle** the system table **USER_INDEXES** provides information on the status of an index. If errors occurred while building the index the status will be 'INVALID' and if dropped the status will also be 'DROPPED', which means that the metadata entry for the dropped index still exists. Spatial indexes are detected by the column **domidx_opstatus**. In **PostgreSQL** information on indexes is a bit more branched. A status field can be found in the **pg_index** table called **indisvalid**. Unfortunately **pg_index** does not contain a column which specifies the indexed column. Two joins are needed to be able to query by the column name. If an index is dropped it is also deleted from the system tables. So the status 'DROPPED' will not appear in a result set.

```
FUNCTION index_status(table_name VARCHAR2, column_name VARCHAR2)
RETURN VARCHAR2
IS
  internal_table_name VARCHAR2(100);
  index_type VARCHAR2(35);
  index_name VARCHAR2(35);
  status VARCHAR2(20);
BEGIN
  internal_table_name := table_name;

  IF geodb_util.versioning_table(table_name) = 'ON' THEN
    internal_table_name := table_name || '_LT';
  END IF;

  execute immediate 'SELECT UPPER(INDEX_TYPE), INDEX_NAME FROM
    USER_INDEXES WHERE INDEX_NAME=(SELECT UPPER(INDEX_NAME)
      FROM USER_IND_COLUMNS WHERE TABLE_NAME=UPPER(:1)
      and COLUMN_NAME=UPPER(:2))'
    into index_type, index_name using internal_table_name, column_name;

  IF index_type = 'DOMAIN' THEN
    execute immediate 'SELECT UPPER(DOMIDX_OPSTATUS) FROM USER_INDEXES
    WHERE INDEX_NAME=:1' into status using index_name;
  ELSE
    execute immediate 'SELECT UPPER(STATUS) FROM USER_INDEXES WHERE
    INDEX_NAME=:1' into status using index_name;
  END IF;

  RETURN status;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RETURN 'DROPPED';
  WHEN others THEN
    RETURN 'INVALID';
END;
```
All functions or part of functions that deal with history management were dropped from the files. This affected the INDEX, UTIL and STAT package. Scripts for the PLANNING MANAGER were dropped as well.

PL/SQL functions for supporting the management of raster data (formerly grouped in MOSAIC.sql) were attempted to port but dropped in the end as their functionalities only fit to the tables of the Oracle version of the 3DCityDB e.g. RDT and IMP tables.

In the UTIL package the to_2d function is substituted by the PostGIS function ST_Force_2D.
3.5 Additional functions

During the development of the port some helper functions were programmed for test cases. Some of them are now part of the release. They are not mandatory for the Importer/Exporter but might be helpful when working with the 3DCityDB.

- **geodb_pkg.util_change_db_srid**
  - defines a new reference system for the 3DCityDB
  - drops indexes and spatial columns and creates new ones
  - should only be executed on an empty database

- **geodb_pkg.util_on_delete_action**
  - helper function for geodb_pkg.util_update_constraints
  - drops a foreign key constraint and adds it again but with a different setting for delete cases e.g. ON DELETE CASCADE
  - **Attention!**: with ON DELETE CASCADE the deletion of a value will also delete values from referential columns

- **geodb_pkg.util_update_constraints**
  - default behavior: uses the function geodb_pkg.util_on_delete_action for updating all foreign keys of the 3DCityDB to ON DELETE CASCADE. If any other char parameter is passed to the function the foreign keys are set to RESTRICT, which is the default behavior of the 3DCityDB

4. References

**Links:**

www1  http://www.postgresql.org/docs/9.1/static/plpgsql-porting.html
www2  http://spatialreference.org

**List of tables:**

Table 1:  Function grouping in Oracle and PostgreSQL.................................4