3D City Database for CityGML

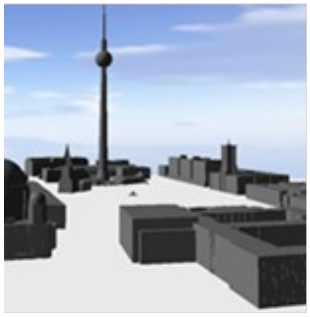
3D City Database Version 2.0.6-postgis

Importer/Exporter Version 1.5.0-postgis

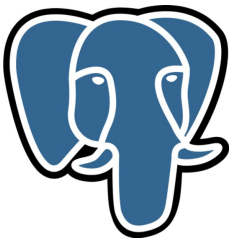
Release Version

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 Herreruela
Gerhard König

(Page intentionally left blank)

Content:

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:

Tab. 1: Function grouping in Oracle and PostgreSQL

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

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`.

del	del_by_lin	idx	match	merge	stat	util

```
FUNCTION exp_func(params) RETURN datatype
IS
  --Declaration
BEGIN
  --Body
END;
/
```

```
CREATE FUNCTION exp_func(params) RETURNS datatype AS
$$
DECLARE
  --Declaration
BEGIN
  --Body
END;
$$
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).

del	del_by_lin	idx	match	merge	stat	util

```
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.

del	del_by_lin	idx	match	merge	stat	util

```
FUNCTION exp_func RETURN datatype
```

```
CREATE FUNCTION exp_func() RETURNS datatype AS
```

Sometimes it is necessary to assign default values to function parameters. This is done with the `DEFAULT` keyword or its abbreviation "`:=`". PL/pgSQL can not compile the short form when it stands inside the function specification.

del	del_by_lin	idx	match	merge	stat	util

```
FUNCTION exp_func(param := '0') RETURN datatype
```

```
CREATE FUNCTION exp_func(param DEFAULT '0') RETURNS datatype AS
```

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.

del	del_by_lin	idx	match	merge	stat	util

```
PROCEDURE pre_delete_citymodel(citymodel_rec citymodel%ROWTYPE)
```

```
CREATE OR REPLACE FUNCTION geodb_pkg.del_pre_delete_citymodel
      (citymodel_rec_id NUMERIC) RETURNS SETOF void AS
```

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

del	del_by_lin	idx	match	merge	stat	util

```
BEGIN
  procedure(params);
END;
```

```
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.


```
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.


```
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.


```
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*.

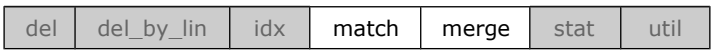
del	del_by_lin	idx	match	merge	stat	util

```
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 '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 '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.




```
PROCEDURE collect_cand_building(
      lod           NUMBER,
      lineage       cityobject.lineage%TYPE)
IS
BEGIN
    -- truncate tmp table
    EXECUTE IMMEDTIATE '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.

del	del_by_lin	idx	match	merge	stat	util

```
--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(
  . . .
END;
/

--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;
```

```
CREATE OR REPLACE FUNCTION geodb_pkg.idx_construct_spatial_2d(
. . .

--no nested tables in PostgreSQL
--create normal table with a column for INDEX_OBJs
DROP TABLE IF EXISTS geodb_pkg.INDEX_TABLE;
CREATE TABLE geodb_pkg.INDEX_TABLE (
       ID                   SERIAL NOT NULL,
       idx_obj              geodb_pkg.INDEX_OBJ
);

--fill index_table by using constructor functions
INSERT INTO geodb_pkg.index_table VALUES(
       1, geodb_pkg.idx_construct_spatial_3d(
              'cityobject_spx', 'cityobject', 'envelope'));
INSERT INTO geodb_pkg.index_table VALUES (
       2, geodb_pkg.idx_construct_spatial_3d(
              'surface_geom_spx', 'surface_geometry', 'geometry'));
INSERT INTO geodb_pkg.index_table VALUES (
       3, geodb_pkg.idx_construct_normal('cityobject_inx',
              'cityobject', 'gmlid, gmlid_codespace'));
INSERT INTO geodb_pkg.index_table VALUES (
       4, geodb_pkg.idx_construct_normal('surface_geometry_inx',
              'surface_geometry', 'gmlid, gmlid_codespace'));
INSERT INTO geodb_pkg.index_table VALUES (
       5, geodb_pkg.idx_construct_normal('appearance_inx', 'appearance',
              'gmlid, gmlid_codespace'));
INSERT INTO geodb_pkg.index_table VALUES (
       6, geodb_pkg.idx_construct_normal('surface_data_inx', 'surface_data',
              'gmlid, gmlid_codespace'));


--no packages in PostgreSQL and thus no global variables for functions
--create package functions as normal functions
CREATE OR REPLACE FUNCTION geodb_pkg.idx_index_status(
       idx geodb_pkg.INDEX_OBJ) RETURNS VARCHAR AS $$
. . .
END;
$$
LANGUAGE plpgsql;
. . .
```

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.

```
FUNCTION create_indexes(type SMALLINT) RETURN STRARRAY
IS
   log STRARRAY;
   sql_error_code VARCHAR2(20);
BEGIN
   log := STRARRAY();

   FOR i IN INDICES.FIRST .. INDICES.LAST LOOP
      IF INDICES(i).type = type THEN
         sql_error_code := create_index(INDICES(i),
        geodb_util.versioning_table(INDICES(i).table_name) = 'ON');
         log.extend;
         log(log.count) := index_status(INDICES(i)) || ':' ||
       INDICES(i).index_name || ':' ||
             INDICES(i).table_name || ':' || INDICES(i).attribute_name || ':' ||
                   sql_error_code;
      END IF;
   END LOOP;

   RETURN log;
END;
```

```
CREATE OR REPLACE FUNCTION geodb_pkg.idx_create_indexes(type INTEGER)
RETURNS text[] AS
$$
DECLARE
    log text[] := '{}';
    sql_error_code VARCHAR(20);
    rec RECORD;
BEGIN
    FOR rec IN select * from geodb_pkg.index_table LOOP
        IF (rec.idx_obj).type = type THEN
            sql_error_code := geodb_pkg.idx_create_index(rec.idx_obj);
            log := array_append(log, geodb_pkg.idx_index_status(rec.idx_obj) ||
                    ':' || (rec.idx_obj).index_name || ':' ||
                    (rec.idx_obj).table_name || ':' ||
                    (rec.idx_obj).attribute_name || ':' || sql_error_code);
        END IF;
    END LOOP;

    RETURN log;
  END;
$$
LANGUAGE plpgsql;
```

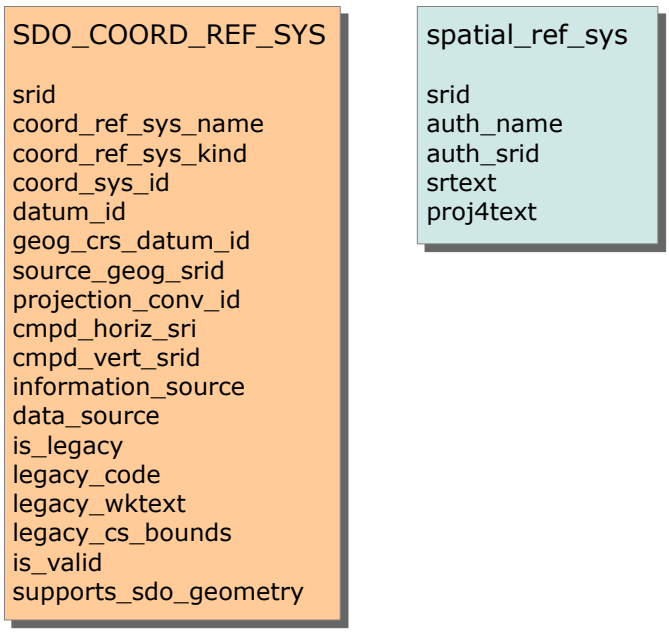
In the UTIL package the user-defined data type DB_INFO_OBJ and the according nested table DB_INFO_TABLE were not ported. As they were only used by one function it was sufficient to let this function return a table with columns for each attribute of the former DB_INFO_OBJ. A code example follows in 3.3 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.



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*.

```
FUNCTION db_metadata RETURN DB_INFO_TABLE
IS
    info_ret DB_INFO_TABLE;
    info_tmp DB_INFO_OBJ;
BEGIN
    info_ret := DB_INFO_TABLE();
    info_ret.extend;

    info_tmp := DB_INFO_OBJ(0, NULL, NULL, 0, NULL);

    execute immediate 'SELECT SRID, GML_SRS_NAME from DATABASE_SRS' into
       info_tmp.srid, info_tmp.gml_srs_name;
    execute immediate 'SELECT COORD_REF_SYS_NAME, COORD_REF_SYS_KIND from
       SDO_COORD_REF_SYS where SRID=:1' into info_tmp.coord_ref_sys_name,
             info_tmp.coord_ref_sys_kind using info_tmp.srid;

    info_tmp.versioning := versioning_db;
    info_ret(info_ret.count) := info_tmp;
    return info_ret;
END;
```

```
CREATE OR REPLACE FUNCTION geodb_pkg.util_db_metadata() RETURNS TABLE(
       srid INTEGER,
       gml_srs_name VARCHAR(1000),
       coord_ref_sys_name VARCHAR(2048),
       coord_ref_sys_kind VARCHAR(2048)) AS
$$
BEGIN
    EXECUTE 'SELECT SRID, GML_SRS_NAME FROM DATABASE_SRS' INTO srid,
       gml_srs_name;
    EXECUTE 'SELECT srtext, srtext FROM spatial_ref_sys WHERE SRID=' || srid || ''
       INTO coord_ref_sys_name, coord_ref_sys_kind;
    coord_ref_sys_name := split_part(coord_ref_sys_name, '"', 2);
    coord_ref_sys_kind := split_part(coord_ref_sys_kind, '[', 1);
    RETURN NEXT;
END;
$$
LANGUAGE plpgsql;
```

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;
```

```
CREATE OR REPLACE FUNCTION geodb_pkg.idx_index_status(
    table_name VARCHAR,
    column_name VARCHAR)
RETURNS VARCHAR AS $$
DECLARE
    is_valid BOOLEAN;
    status VARCHAR(20);
BEGIN
    EXECUTE 'SELECT DISTINCT pgi.indisvalid FROM pg_index pgi
        JOIN pg_stat_user_indexes pgsui ON pgsui.relid=pgi.indrelid
        JOIN pg_attribute pga ON pga.attrelid=pgi.indexrelid
        WHERE pgsui.relname=$1 AND pga.attname=$2' INTO is_valid USING
             lower(table_name), lower(column_name);

    IF is_valid is null THEN
        status := 'DROPPED';
    ELSIF is_valid = true THEN
        status := 'VALID';
    ELSE
        status := 'INVALID';
    END IF;

    RETURN status;

EXCEPTION
    WHEN OTHERS THEN
        RETURN 'FAILED';
END;
$$
LANGUAGE plpgsql;
```

3.4 Non-translated parts

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: