Implementing large objects

Things are well remembered when they are listed and this is how we will remember PostgreSQL large objects implementation in our memory:

In PostgreSQL there exist several ways to manage Binary Large Objects (LOB, BLOB):

1. Basic binary data type BYTEA
2. Basic character data type TEXT
3. Large object (LO) facility 'pg_largeobject'
4. Data type DATALINK (that is just spec., no implementation)

The **BYTEA** data type allows storage of binary strings.

- It stores a LOB within the table, respectively using TOAST.
- It is thus limited to 1 GB
- The storage is octal and allows non printable characters (in contrast to character strings which don't).
- The input/output format is HEX (as of PostgreSQL 9.0).

Notes:

- BYTEA comes close to the SQL standard binary string type 'BLOB'. The functions and operators provided by BYTEA are mostly the same, while HEX input format of BYTEA is different.
- BYTEA is slower for lengths >20 MB than the LO facility (it has no random accees).

TEXT

Basic data type *text* it's just here for completeness. This is a variable character type with unlimited length (up to 1 GB). character types allow locale settings.

It's not a byte string but one could still use it when the binary string is preprocessed and encoded into printable form (e.g. base64 or hex).

Large object (LO) facility

Large objects (LO) can also placed in a single system table called 'pg_largeobject' which has to be accessed via identifiers of data type OID.

- There is a read/write API which offers client (= modules written in C) and server-side (= SQL) functions.

- The main related SQL functions are: lo_creat(), lo_create(), lo_unlink(), lo_import(), and lo_export(). lo_import() and lo_export() need permissions of the database's owning user (i.e. superuser).
- LO are broken into "chunks" and stored in btree-indexed rows.
- LO allows values up to 2 GB in size, whereas TOASTed fields (like BYTEA) can be at most 1 GB.
- LO entries can be randomly modified using a read/write API that is more efficient than performing such operations using TOAST (and e.g BYTEA).

Note:

- When PostgreSQL doc. mentions 'lo' (LO = Large Object) it typically refers to this facility.
- In contrast to e.g. BYTEA - LO is not a data type on its own but a table, a 'facility'.

IMPORTANT NOTE when using JDBC BLOB (or @Lob annotation in Hibernate)

Since PostgreSQL considers a LO entry as an object on it's own, deleting or upating rows in the user table does'nt delete or delete entries in pg_largeobjects. pg_largeobjects therefore grows infinitely unless a separate cleaning is made (see this error report in Hibernate forum).

To prevent this, typically a trigger needs to be added which deletes entries in pg_largeobject as described in module 'lo'.

DATALINK

NOTE: There is currently no implementation in PostgreSQL for that. It's just a specification defined in Standard SQL 'SQL/MED'.

The DATALINK type stores file URLs in database columns and applies constraints on it.

- Maintains a link to a specific file in external storage.
- Database system takes over control over external files (rename, delete, permissions are done with SQL) if defined so.
- File size is unlimited, respectively limited by external storage. No need to store file contents in database system.

DATALINK parameters:

- NO LINK CONTROL Datalink value need not reference an existing file/URL.
- FILE LINK CONTROL Datalink value must reference an existing file/URL.
- INTEGRITY ALL Referenced files can only be renamed or deleted through SQL.
- INTEGRITY SELECTIVE Referenced files can be renamed or deleted through SQL or directly.
- INTEGRITY NONE (implied for NO LINK CONTROL)
- ON UNLINK DELETE File is deleted from file system when deleted from database.
- ON UNLINK RESTORE File’s original permissions are restored when deleted from database.
- ON UNLINK NONE No change in file permissions when file reference is deleted from database.
- RECOVERY YES PITR applies to referenced files.
- RECOVERY NO PITR does not apply to referenced files.

Status and Installation:

- unclear.

```
CREATE TABLE image (
   id integer,
   name text,
   picture DATALINK [NO LINK CONTROL]
 );

 INSERT INTO persons VALUES (
   1,
   'Jon Doe',
   DLVALUE('file://some/where/1.jpg')
 );
```

• Large objects, unlike BYTEA, are not a data type but an entry in a system table.

• All large objects are stored in the pg_largeobject system table.

• Each large object also has a corresponding entry in the pg_largeobject_ metadata system table.

• Large objects are broken up into chunks of default size and further stored as rows in the database.

• These chunks in rows are B-tree indexed; hence, this ensures fast searches during read/write operations.

• From PostgreSQL 9.3 onwards, the maximum size of a large object in a table can be 4 TB.

• Large objects are not stored in user tables; rather, a value of the **Object Identifier** (**OID**) type is stored. You will use this OID value to access the large object. So, when you have to access a large object, you will reference the OID value that points to a large object present on the pg_largeobject system table.

• PostgreSQL provides the read/write **Application Program Interface** (**API**) that offers client- and server-side functions. Using this API, you can perform operations such as create, modify, and delete on large objects. OIDs are used in this function as a reference to access large objects, for example, to transfer the contents of any file to the database or to extract an object from the database into a file.

• From PostgreSQL 9.0 onwards, large objects now have an associated owner

and a set of access permissions. Retrieving data using these functions gives

you the same binary data you added. Examples of the functions are lo_

create(), lo_unlink(), lo_import(), and lo_export().

• PostgreSQL provides the ALTER LARGE TABLE feature to change the definition

of a large object. Remember that its only functionality is to assign a new owner.

Steps to use large objects :

1: create table test_large_objects(pic_id int,name varchar(30),picture oid,constraint pk1 primary key(pic_id));

2: insert into test_large_objects values(1,'kuldeep kumar',lo_import('/stage/kuldeep@koenig.jpg'));

3: select * from pg_largeobject;

4: select * from test_large_objects;

More Examples :

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);       -- returns OID of new, empty large object

SELECT lo_create(43213);   -- attempts to create large object with OID 43213

SELECT lo_unlink(173454);  -- deletes large object with OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- same as above, but specify OID to use
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

You can store images in postgresql using the following function :

```
create or replace function bytea_import(p_path text, p_result out bytea)
                language plpgsql as $$
declare
  l_oid oid;
  r record;
begin
  p_result := '';
  select lo_import(p_path) into l_oid;
  for r in ( select data
             from pg_largeobject
             where loid = l_oid
             order by pageno ) loop
    p_result = p_result || r.data;
  end loop;
  perform lo_unlink(l_oid);
end;$$;
```

```
insert into my_table(bytea_data) select bytea_import('/my/file.name');
update "Employees" set "Photo" = ( select bytea_import('/usr/local/pgsql/
backups/nwindpics/emp/1.jpg')) where "EmployeeID"=1;
```