CIRAD
BIOS Department, DAP Unit, ID Team,
GnpAnnot Project

Valentin GUIGNON, 17/06/2009
Version 2.0

CHADO CONTROLLER DESIGN

Summary

1. Introduction

The Chado Controller is a communication layer between a Chado database and Chado clients like GBrowse, Apollo or Artemis. It controls Chado database access to manage data access rights and validate and audit data edition. Therefore, it can be divided in three parts: the access rights management part will be called the "**access restriction system**"; the data validation part will be called the "**annotation inspector**"; the data auditing will be handled by a customized version of the original Chado "**audit module**".

2. Architecture

Without the controller, Chado clients are directly connected to Chado data using a PostgreSQL connection as shown in Figure 1.

Figure 1: Classic Chado access.

The Chado controller layer is a middleware (handled by PostgreSQL) between Chado and its clients. Any query to fetch or edit data is processed by the controller as shown in Figure 2. The controller validates the content of a query ("annotation inspector" part), limits the result set of a query according to the user access rights ("access restriction system") and also store data modification ("audit module").

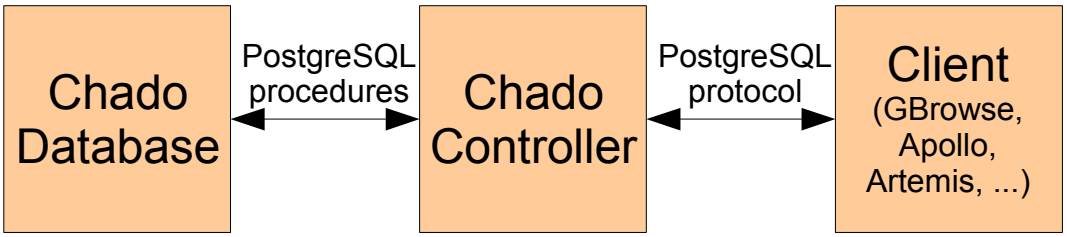

Figure 2: Controlled Chado access.

2.1. Access control

To manage users and groups access right, a database is used to store user authentication data and access level to each Chado feature. Figure 3 shows the structure of that database. The table `feature_access` is used to associate a feature access level with a user (or a group). If a feature is not referred in this table for a specified user (or his/her groups) then the feature won't be accessible to that user. By default, newly created features are given the access right of their parent (using srcfeature_id once an associated featureloc is set). Of course, the user who created the feature also has full access to that feature. In order to access or create features, users must also have a PostgreSQL account sharing the same login as the one used in the user database.

The `annotator` table could be seen as redundant since users must also have a PostgreSQL

account but its purpose is to provide another way than the PostgreSQL server way to manage users. Therefore, some administrative informations about users and groups can be stored (roles, administrator comments) and accounts could be disabled or automatically locked by a custom login interface to block password attacks.

Feature access levels are:

- 0: no access. The user can't see the feature;
- 1: read only. The user can only view the feature but can not modify it;
- 2: read and edit the feature. The user can view and modify the feature;
- 3: read, edit and remove the feature. The user has complete control over the feature.

Note: to be able to edit a feature, the user must also have the flag 0x000001000 of his annotator account or group set (cf. "`annotator`" table of 4.2.). That flag also allows the user to add new features to the Chado database.

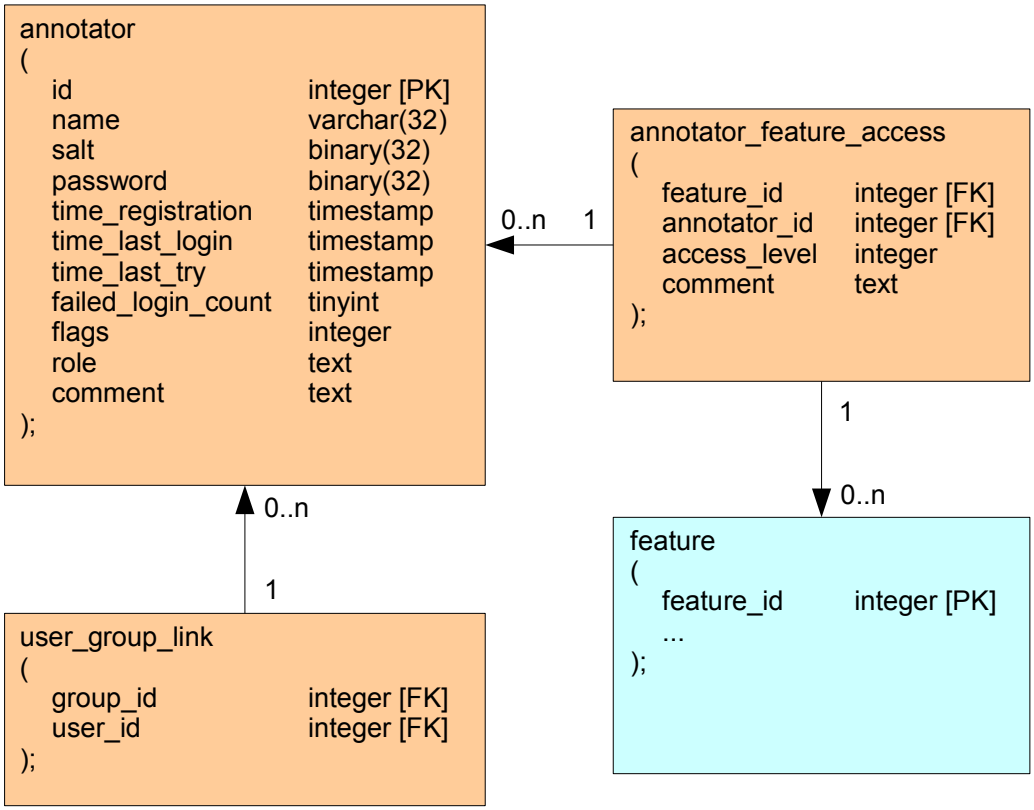

Figure 3: User access right database (feature table shown in blue belongs to the Chado schema).

Some technical details of the user access database are provided in appendix 4.2. and some example of access are shown in 4.1. .

2.2. Auditing

The auditing process is based on GMOD audit module recommendations (http://gmod.org/wiki/Chado_Audit_Module). Basically, each audited table has a clone table named with an "`_audit`" suffix and containing three additional columns "`transaction_date`", "`transaction_type`" and "`transaction_user`". Each transaction is recorded by the Chado controller in the corresponding "`_audit`" table that holds the new values, the transaction date and type and the name of the user who made the change.

There are four kinds of transaction types. When installing the Chado Controller Audit Module on a non-empty database, previous records are stored in the audit table with the transaction type "i" (lower case) for "install". When new rows are added to a table, the transaction type used is 'I' (upper case) for "INSERT"; for UPDATE transactions, it is "U"; and for "DELETE" transactions it is "D".

The Chado Controller Audit Module differs from the recommendations of GMOD in 2 ways: first, each audit table has three more columns; then, for each transaction, the result of the transaction is recorded instead of the previous value of the row being modified. That last point implies that the database size (on disk) is doubled when the module is installed and any current row of a table has a clone in its associated audit table telling when that row has been added and who did it.

2.3. Data Validation

At the time this design documentation is being written, only the owner of a feature is automatically managed by the annotation inspector. We plan to add more features to the annotation inspector as soon as possible.

3. Implementation

The Chado Controller is handled by the PostgreSQL server engine. To manage access rights, auditing and validate data, the original table "feature" has been renamed into "feature_data" and replaced by a view with associated rules allowing insert, update and delete transactions (note: the "copy" transaction is not supported, see "install_annotators_management.sql" script notes for details). Functions and triggers are written in "`plpgsql`" language so no major change on existing Chado servers are required.

Since managing access to many features of a user that belongs to several groups can become memory and time consuming, a temporary table is created to optimize access checks during a user session (ie. while a user is connected to the database). This table is automatically created when the user issues his/her first query on the feature view which will slow down that query. To avoid such a delay on the first regular query, the function "`init_access()`" can be called manually (`SELECT init_access();`).

This solution works well with applications like Apollo and Artemis which have a permanent connection (session) to the Chado database. However, for web applications like GBrowse, each time a web page is requested, a new connection to the database is issued and the delay of creation of the temporary table is added! In order to avoid that delay which would turn genome browsing into a real pain, another solution has been developed.

GBrowse (v1) access are made using an account which has full access to the database (admin flag set) and restrictions are handled inside GBrowse scripts. A login module has also been created that allows users to login/logout and change their password (on both the annotator table and their PostgreSQL account). That module can be enabled or disabled using the data source configuration file. So far, this optimization does not seem to significantly slow down genome browsing.

4. Appendix

4.1. Access example

The following section describes the idea of how access rights are managed but it does not show how the Chado Controller handles them through views (and temporary tables).

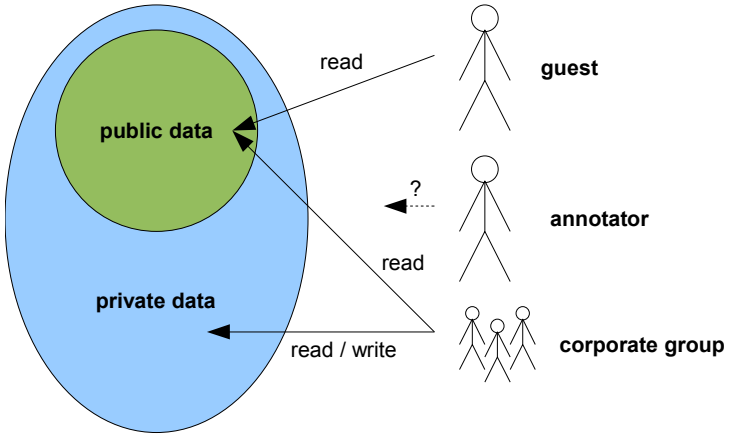

Figure 4: Example starting access schema.

In this schema, a guest user can only view public data. The corporate group can view both public and private data and can also edit private data. When created (step 1), the annotator user gets the same access as the guest user. The annotator user can then be added to the corporate group (step 2) to be allowed to see and edit private data. Then, it can also receive the right to edit public data (step 3). However, other users of the corporate group won't get that right. After that, if the annotator is removed from the corporate group (step 4), he/she could only edit public data.

In the database point of view, let's say we have a feature 1 which is public and a feature 2 which is private. We have the guest user which has user identifier 1, the corporate group which has user identifier 2 and the annotator which has user identifier 3.

feature_id	annotator_id	access_level	comment
1 (public)	1 (guest)	1 (read)	
1 (public)	2 (corporate)	1 (read)	
1 (public)	3 (annotator)	1 (read)	default right
2 (private)	2 (corporate)	3 (full edition)	

Table 1: Initial feature access schema (**step 1**).

The following SQL query will retrieve all the features the guest user can see:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(1) AND fa.access_level > 0);
```

It will return feature 1. The query used to retrieve the features that the corporate group can edit is:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(2) AND fa.access_level >= 2);
```

It will return feature 2. What the annotator can see is given by:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(3) AND fa.access_level > 0);
```

It will return feature 1 but it would have returned no feature if we didn't have the “default right” line (see comments column in Table 1). Now, if we add the annotator to the corporate group (**step 2**),

we will add the line "2, 3" to the "`user_group_link`" table as shown in Table 2:

group_id	user_id
2 (corporate)	3 (annotator)

Table 2: Annotator user is part of the corporate group.

Then, the query used to get what the annotator user will see, will be modified to:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(2, 3) AND fa.access_level > 0);
```

and will return features 1 and 2. The annotator user can now edit feature 2 but not feature 1:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(2, 3) AND fa.access_level >= 2);
```

Now, to grant annotator user the right to edit public data (**step 3**), we will add a new line to the "`feature_access`" table as shown in Table 3:

feature_id	annotator_id	access_level	comment
1 (public)	3 (annotator)	3 (full edition)	

Table 3: Annotator user can edit public data (**step 3**).

If we remove (**step 4**) the annotator user from the corporate group (delete line of Table 2), he/she will loose the full edition right on feature 2 and the query:

```
SELECT * FROM feature f WHERE f.feature_id IN (SELECT fa.feature_id FROM
feature_access fa WHERE fa.annotator_id IN(3) AND fa.access_level >= 2);
```

will only return feature 1 as the only feature the annotator user can edit.

4.2. Access Restriction System PostgreSQL installation script

```
-- sequence for annotator IDs
DROP SEQUENCE IF EXISTS annotator_seq CASCADE;
CREATE SEQUENCE annotator_seq;

-- annotator table: contains both users and groups (see flag comments for details)
DROP TABLE IF EXISTS annotator;
CREATE TABLE annotator
(
    id                INTEGER NOT NULL DEFAULT NEXTVAL('annotator_seq'), -- user or group
identifier
    name              NAME,                                              -- User login (could be
an e-mail address) or group name
    salt              BYTEA,                                             -- Random binary array
used to pre-crypt (XOR) password before hashing (NULL for groups)
    password          CHARACTER(32),                                     -- Hash of the (pre-
crypted) user password (NULL for groups)
    time_registration  TIMESTAMP DEFAULT NOW(),                          -- Time when the
account was created
    time_last_login    TIMESTAMP DEFAULT NULL,                           -- Last time the user
logged in (NULL for groups)
    time_last_try      TIMESTAMP DEFAULT NULL,                           -- Last time an
unsuccessful login was tried in the user account (NULL for groups)
    failed_login_count SMALLINT DEFAULT 0,                               -- Number of
consecutive (less than a few minutes between tries) unsuccessful login tries (NULL for groups)
    flags              INTEGER NOT NULL DEFAULT 0,                       -- User/group account
flags
        -- b'0000000000000001': if set to 0 the account belongs to a user, 1 the account belongs
to a group
        -- b'0000000000000010': if set, the account is disabled and the user will not be allowed
to login or the group access rights will be ignored
        -- b'0000000000000100': if set, the user will not be allowed to login before a time delay
added to "time_last_try" (useless for groups)
        -- b'0000000000001000': if set, the user/group can write in Chado
        -- b'0000000000010000': if set, the user is not allowed to change the password (useless
for groups)
        -- b'0000000000100000': if set, the user must change the password (useless for groups)
        -- b'0000000001000000': if set, the password is not required for login (useless for
groups)
        -- b'0000000010000000': if set, the user has admin access
        -- Note: use CAST(b'0000000010000000' AS INTEGER) to work with binary synthax
    role               TEXT,                                             -- A description of the
tasks that belongs to the user/group
    comment            TEXT,                                             -- Any administrative
information about the user/group account
    UNIQUE (name),
    PRIMARY KEY (id)
);

COMMENT ON TABLE annotator IS 'Contains user and group accounts';

-- Creates the 2 default (requiered) users with specific IDs
-- Note: if anonymous user is modified, don't forget to also update get_access_list() function!
-- Note: admin account has no access restriction
INSERT INTO annotator VALUES (0, 'anonymous', NULL, NULL, NOW(), NULL, NULL, 0, 81, 'anonymous
account', 'public access'); -- flags: group, no password, password locked
INSERT INTO annotator VALUES (1, 'gnpannot',
E'\x0a\xd7\xf3\xc7\x27\xf3\xe9\x1c\x3f\x44\x95\x09\x6e\xa0\x7a\x53\xe7\x4f\x91\x36\x14\x68\x32\x4
5\xeb\x47\xec\x72\x27\xb6\x14\x25', 'a1016362e106122d103bf843c6310eaa', NOW(), NULL, NULL, 0,
168, 'administrator account', 'admin'); -- password: toto1234; flags: write access, must change
password, admin
ALTER SEQUENCE annotator_seq RESTART WITH 2;

-- user_group_link table: enables users<->groups association
DROP TABLE IF EXISTS user_group_link;
CREATE TABLE user_group_link
(
    group_id INTEGER NOT NULL,
    user_id  INTEGER NOT NULL,
    FOREIGN KEY (group_id) REFERENCES annotator (id) ON DELETE CASCADE,
    FOREIGN KEY (user_id)  REFERENCES annotator (id) ON DELETE CASCADE,
    PRIMARY KEY (group_id, user_id)
```

```
);

COMMENT ON TABLE user_group_link IS 'Links a user to a group (or another user) to let the user
inherits the group (or the other user) permissions';

-- feature_access table: assiociate to a feature, a user and his/her access level
DROP TABLE IF EXISTS feature_access;
CREATE TABLE feature_access
(
    feature_id   INTEGER NOT NULL, -- The feature identifier associated to the access rule
    annotator_id INTEGER,          -- A user or group identifier
    access_level INTEGER NOT NULL, -- The access level: 0=no access, 1=read only, 2=1+can add and
modify, 3=2+can delete
    comment      TEXT,             -- Any administrative information about the access right
    FOREIGN KEY (feature_id)   REFERENCES feature (feature_id) ON DELETE CASCADE,
    FOREIGN KEY (annotator_id) REFERENCES annotator (id) ON DELETE CASCADE,
    PRIMARY KEY (feature_id, annotator_id)
);

COMMENT ON TABLE feature_access IS 'Gives a user or a group an access level to a specific
feature';

-- insert full rights on all feature for admin
INSERT INTO feature_access SELECT feature_id, 1, 3 FROM feature;

-- get_access_list: returns the list of annotator ID current user belongs to (it always includes
at least anonymous ID)
CREATE OR REPLACE FUNCTION get_access_list() RETURNS SETOF INTEGER AS
$$
    DECLARE
        rid integer;

    BEGIN
        RETURN NEXT 0; -- uses anonymous identifier as any user must at least have anonymous
rights
        SELECT INTO rid id FROM annotator WHERE name = session_user;
        RETURN NEXT rid;
        FOR rid IN SELECT ug.group_id FROM user_group_link ug, annotator a WHERE ug.user_id =
a.id AND a.name = session_user LOOP
            RETURN NEXT rid;
        END LOOP;
        RETURN;
    END;
$$
LANGUAGE plpgsql SECURITY DEFINER;

-- create and initialize feature_access_max_temp temporary table to improve queries performance
on feature view
CREATE OR REPLACE FUNCTION init_access() RETURNS BOOLEAN AS
$$
    DECLARE
        is_admin BOOLEAN;

    BEGIN
        BEGIN
            EXECUTE 'SELECT feature_id FROM feature_access_max_temp LIMIT 1';
        EXCEPTION
            WHEN UNDEFINED_TABLE THEN
                EXECUTE 'CREATE TEMPORARY TABLE feature_access_max_temp
                (
                    feature_id   INTEGER NOT NULL,
                    access_level INTEGER NOT NULL
                ) ON COMMIT PRESERVE ROWS;';
                EXECUTE 'CREATE INDEX feature_access_max_temp_pkey ON feature_access_max_temp
(feature_id);';
                SELECT INTO is_admin (id = 1) FROM annotator WHERE name = session_user;
                IF is_admin THEN
                    -- if the current user is gnpannot user (aka user id 1, aka admin), then
bypass access restriction
                    EXECUTE 'CREATE OR REPLACE TEMPORARY VIEW feature AS SELECT * FROM
feature_data';
                    -- since the feature view as been overridden by a temporary view, rules have
to be recreated
                    EXECUTE 'CREATE OR REPLACE RULE feature_insert AS ON INSERT TO feature
    DO INSTEAD
```

```
        INSERT INTO feature_data (feature_id, dbxref_id, organism_id, name, uniquename, residues,
seqlen, md5checksum, type_id, is_analysis, is_obsolete, timeaccessioned, timelastmodified)
            SELECT
                COALESCE(NEW.feature_id, NEXTVAL(''feature_feature_id_seq'')),
                NEW.dbxref_id,
                NEW.organism_id,
                NEW.name,
                NEW.uniquename,
                NEW.residues,
                NEW.seqlen,
                NEW.md5checksum,
                NEW.type_id,
                COALESCE(NEW.is_analysis, FALSE),
                COALESCE(NEW.is_obsolete, FALSE),
                COALESCE(NEW.timeaccessioned, NOW()),
                COALESCE(NEW.timelastmodified, NOW())
            RETURNING feature_data.*';
                    EXECUTE 'CREATE OR REPLACE RULE feature_update AS ON UPDATE TO feature
    DO INSTEAD
        UPDATE feature_data
            SET feature_id = NEW.feature_id,
                dbxref_id = NEW.dbxref_id,
                organism_id = NEW.organism_id,
                name = NEW.name,
                uniquename = NEW.uniquename,
                residues = NEW.residues,
                seqlen = NEW.seqlen,
                md5checksum = NEW.md5checksum,
                type_id = NEW.type_id,
                is_analysis = NEW.is_analysis,
                is_obsolete = NEW.is_obsolete,
                timeaccessioned = NEW.timeaccessioned,
                timelastmodified = NEW.timelastmodified
            WHERE feature_id = OLD.feature_id';
                    EXECUTE 'CREATE OR REPLACE RULE feature_delete AS ON DELETE TO feature
    DO INSTEAD
        DELETE FROM feature_data
            WHERE feature_id = OLD.feature_id';
                ELSE
                    -- for other users, fill feature_access_max_temp with the best user access
rights for performance improvements
                    EXECUTE 'INSERT INTO feature_access_max_temp SELECT feature_id,
MAX(access_level) AS access_level FROM feature_access WHERE annotator_id IN (SELECT * FROM
get_access_list()) GROUP BY feature_id';
                    EXECUTE 'GRANT SELECT ON feature_access_max_temp TO gnpannot_users';
                    EXECUTE 'CREATE OR REPLACE TEMPORARY VIEW feature_access_max AS SELECT * FROM
feature_access_max_temp;';
                    EXECUTE 'CREATE OR REPLACE TEMPORARY VIEW feature AS SELECT f.* FROM
feature_data f INNER JOIN feature_access_max fa USING (feature_id) WHERE fa.access_level >= 1';
                    -- since the feature view as been overridden by a temporary view, rules have
to be recreated
                    EXECUTE 'CREATE OR REPLACE RULE feature_insert AS ON INSERT TO feature
    DO INSTEAD
        INSERT INTO feature_data (feature_id, dbxref_id, organism_id, name, uniquename, residues,
seqlen, md5checksum, type_id, is_analysis, is_obsolete, timeaccessioned, timelastmodified)
            SELECT
                COALESCE(NEW.feature_id, NEXTVAL(''feature_feature_id_seq'')),
                NEW.dbxref_id,
                NEW.organism_id,
                NEW.name,
                NEW.uniquename,
                NEW.residues,
                NEW.seqlen,
                NEW.md5checksum,
                NEW.type_id,
                COALESCE(NEW.is_analysis, FALSE),
                COALESCE(NEW.is_obsolete, FALSE),
                COALESCE(NEW.timeaccessioned, NOW()),
                COALESCE(NEW.timelastmodified, NOW())
                WHERE has_write_access()
            RETURNING feature_data.*';
                    EXECUTE 'CREATE OR REPLACE RULE feature_update AS ON UPDATE TO feature
    DO INSTEAD
        UPDATE feature_data
            SET feature_id = NEW.feature_id,
```

```
                dbxref_id = NEW.dbxref_id,
                organism_id = NEW.organism_id,
                name = NEW.name,
                uniquename = NEW.uniquename,
                residues = NEW.residues,
                seqlen = NEW.seqlen,
                md5checksum = NEW.md5checksum,
                type_id = NEW.type_id,
                is_analysis = NEW.is_analysis,
                is_obsolete = NEW.is_obsolete,
                timeaccessioned = NEW.timeaccessioned,
                timelastmodified = NEW.timelastmodified
            WHERE feature_id = OLD.feature_id  AND has_update_access(OLD.feature_id)';
                    EXECUTE 'CREATE OR REPLACE RULE feature_delete AS ON DELETE TO feature
    DO INSTEAD
        DELETE FROM feature_data
            WHERE feature_id = OLD.feature_id AND has_delete_access(OLD.feature_id)';
                    -- give gnpannot_users (wich should include session_user) the right to use
restricted parts of feature table
                    EXECUTE 'GRANT ALL ON feature TO gnpannot_users';
                END IF;
            END;
        RETURN TRUE;
    END;
$$
LANGUAGE plpgsql SECURITY DEFINER;

-- feature_access_max view: used to only get the highest right of current user (including rights
from his/her associated groups) on each feature
-- Note: the first time during a session the view is used, it calls init_access() and is replaced
by a temporary faster view
CREATE OR REPLACE VIEW feature_access_max AS SELECT feature_id, MAX(access_level) AS access_level
FROM feature_access WHERE annotator_id IN (SELECT * FROM get_access_list()) GROUP BY feature_id;


-- Trigger to assign default rights extracted from the scaffold to new features
-- This is performed using insert event on featureloc table as it links a feature to its scaffold
CREATE OR REPLACE FUNCTION assign_default_rights() RETURNS TRIGGER AS
$$
    BEGIN
        INSERT INTO feature_access (feature_id, annotator_id, access_level)
            SELECT NEW.feature_id, fa.annotator_id, fa.access_level
                FROM feature_access fa
                WHERE (NEW.srcfeature_id = fa.feature_id)
                          AND (NOT EXISTS (SELECT 1
                          FROM feature_access fa2
                          WHERE (fa2.feature_id = NEW.feature_id)
                                AND (fa2.annotator_id = fa.annotator_id)));
        RETURN NEW;
    END
$$
LANGUAGE plpgsql SECURITY DEFINER;

DROP TRIGGER IF EXISTS assign_default_rights_i ON featureloc;
CREATE TRIGGER assign_default_rights_i
    AFTER INSERT ON featureloc
    FOR EACH ROW
    EXECUTE PROCEDURE assign_default_rights();

-- give full access rights to current user (the one who just created a new feature)
CREATE OR REPLACE FUNCTION assign_user_rights() RETURNS TRIGGER AS
$$
    BEGIN
        IF (NOT EXISTS (SELECT name FROM annotator WHERE (name = session_user))) THEN
            RAISE EXCEPTION 'Current user is not part of the annotator table! Unable to assign
feature access rights!';
        END IF;
        INSERT INTO feature_access (feature_id, annotator_id, access_level)
            SELECT COALESCE(NEW.feature_id, CURRVAL('feature_feature_id_seq')), a.id, 3 -- access
level 3: SELECT/INSERT/UPDATE/DELETE access
                FROM annotator a
                WHERE (a.name = session_user);
        BEGIN
            INSERT INTO feature_access_max_temp (feature_id, access_level)
                SELECT COALESCE(NEW.feature_id, CURRVAL('feature_feature_id_seq')), 3;
```

```
        EXCEPTION
            WHEN UNDEFINED_TABLE THEN
        END;
        RETURN NEW;
    END
$$
LANGUAGE plpgsql SECURITY DEFINER;

DROP TRIGGER IF EXISTS assign_user_rights_i ON feature_data;
CREATE TRIGGER assign_user_rights_i
    AFTER INSERT ON feature
    FOR EACH ROW
    EXECUTE PROCEDURE assign_user_rights();


-- turn feature table into a view to manage access control and associate rules for insert, update
and delete
ALTER TABLE feature RENAME TO feature_data;

-- create feature view that includes access restriction
CREATE OR REPLACE VIEW feature AS SELECT f.* FROM feature_data f INNER JOIN feature_access_max fa
USING (feature_id) WHERE init_access() AND fa.access_level >= 1;

-- has_write_access: returns TRUE if current user has write access, otherwise throws an exception
CREATE OR REPLACE FUNCTION has_write_access() RETURNS BOOLEAN AS
$$
    DECLARE
        no_write_access BOOLEAN;

    BEGIN
        SELECT INTO no_write_access (0 = (a.flags & CAST(b'0000000000001000' AS INTEGER))) FROM
annotator a WHERE a.name = session_user;
        IF (no_write_access) THEN
            RAISE EXCEPTION 'You are not allowed to create new features!';
        END IF;
        RETURN TRUE;
    END;
$$
LANGUAGE plpgsql SECURITY DEFINER;


-- has_update_access: returns TRUE if current user has update access to a specified, otherwise
throws an exception
CREATE OR REPLACE FUNCTION has_update_access(INTEGER) RETURNS BOOLEAN AS
$$
    DECLARE
        has_update_access BOOLEAN;

    BEGIN
        SELECT INTO has_update_access EXISTS (SELECT fa.access_level FROM feature_access_max fa
WHERE fa.feature_id = $1 AND fa.access_level >= 2);
        IF (NOT has_update_access) THEN
            RAISE EXCEPTION 'You are not allowed to modify this feature (%)!', $1;
        END IF;
        RETURN has_update_access;
    END;
$$
LANGUAGE plpgsql SECURITY DEFINER;


-- has_delete_access: returns TRUE if current user has update access to a specified, FALSE
otherwise
CREATE OR REPLACE FUNCTION has_delete_access(INTEGER) RETURNS BOOLEAN AS
$$
    DECLARE
        has_delete_access BOOLEAN;

    BEGIN
        SELECT INTO has_delete_access EXISTS (SELECT fa.access_level FROM feature_access_max fa
WHERE fa.feature_id = $1 AND fa.access_level >= 3);
        IF (NOT has_delete_access) THEN
            RAISE EXCEPTION 'You are not allowed to remove this feature (%)!', $1;
        END IF;
        RETURN has_delete_access;
    END;
```

```
$$
LANGUAGE plpgsql SECURITY DEFINER;


CREATE OR REPLACE RULE feature_insert AS ON INSERT TO feature
    DO INSTEAD
        INSERT INTO feature_data (feature_id, dbxref_id, organism_id, name, uniquename, residues,
seqlen, md5checksum, type_id, is_analysis, is_obsolete, timeaccessioned, timelastmodified)
            SELECT
                COALESCE(NEW.feature_id, NEXTVAL('feature_feature_id_seq')),
                NEW.dbxref_id,
                NEW.organism_id,
                NEW.name,
                NEW.uniquename,
                NEW.residues,
                NEW.seqlen,
                NEW.md5checksum,
                NEW.type_id,
                COALESCE(NEW.is_analysis, FALSE),
                COALESCE(NEW.is_obsolete, FALSE),
                COALESCE(NEW.timeaccessioned, NOW()),
                COALESCE(NEW.timelastmodified, NOW())
                WHERE has_write_access()
        RETURNING feature_data.*;


CREATE OR REPLACE RULE feature_update AS ON UPDATE TO feature
    DO INSTEAD
    UPDATE feature_data
       SET feature_id = NEW.feature_id,
           dbxref_id = NEW.dbxref_id,
           organism_id = NEW.organism_id,
           name = NEW.name,
           uniquename = NEW.uniquename,
           residues = NEW.residues,
           seqlen = NEW.seqlen,
           md5checksum = NEW.md5checksum,
           type_id = NEW.type_id,
           is_analysis = NEW.is_analysis,
           is_obsolete = NEW.is_obsolete,
           timeaccessioned = NEW.timeaccessioned,
           timelastmodified = NEW.timelastmodified
     WHERE feature_id = OLD.feature_id  AND has_update_access(OLD.feature_id);


CREATE OR REPLACE RULE feature_delete AS ON DELETE TO feature
    DO INSTEAD
    DELETE FROM feature_data
     WHERE feature_id = OLD.feature_id AND has_delete_access(OLD.feature_id);


-- Assign rights...
CREATE ROLE gnpannot_users; -- this query may rise an error that can be ignored if the group
already exists
GRANT ALL ON              feature           TO GROUP   gnpannot_users;
GRANT EXECUTE ON FUNCTION init_access()     TO GROUP   gnpannot_users;
GRANT EXECUTE ON FUNCTION get_access_list() TO GROUP   gnpannot_users;
GRANT SELECT ON TABLE     annotator         TO GROUP   gnpannot_users;
GRANT SELECT ON TABLE     user_group_link   TO GROUP   gnpannot_users;
GRANT SELECT ON TABLE     feature_access    TO GROUP   gnpannot_users;
REVOKE ALL ON TABLE       feature_access    FROM GROUP gnpannot_users;
REVOKE ALL ON TABLE       annotator         FROM GROUP gnpannot_users;
REVOKE ALL ON TABLE       user_group_link   FROM GROUP gnpannot_users;
REVOKE ALL ON TABLE       feature_data      FROM GROUP gnpannot_users;
REVOKE ALL ON TABLE       feature_audit     FROM GROUP gnpannot_users; -- prevent access to
history (audit)
```