Chado Controller Technical Documentation

V1.0.0
2011-11-30

The Chado Controller package and all associated files are copyright (c) 2008 CIRAD, Montpellier, France.

The Chado Controller package is distributed under the Artistic License 2.0.

Preface

This technical documentation describes how are implemented the different parts of the Chado Controller. To understand the documentation, you need the specific skills:

- SQL and SQLTemplate language;
- Notion of PostgreSQL;
- PERL;
- HTML with notions of sessions and cookies.

Table of content

Chado Controller Technical Documentation 1
- Preface 3
- Table of content 4
- Basics 5
- Access Restriction Module 5
  - Chado Schema Modifications 5
  - Account Management Considerations 7
  - Optimizations and Behaviour 8
  - Compatibility Mode 11
  - Access level 12
  - Authentication 13
- Annotation Inspector 13
  - Modularity 13
  - Validation procedures 13
- Annotation History 15
- Contacts 17

Basics

The Chado Controller (CC) is made of a set of SQL functions, triggers, views and rules, all embedded in a Chado database. This set is a layer between Chado data and client software (Figure 1). Each of the 3 modules of the CC has its own specificities. The Access Restriction module uses views and rules. The Annotation Inspector is based on triggers and functions. Finally, the Annotation History relies on mirror tables and triggers. The embedded part in the Chado database could be enough for the CC to work on its own but in order to take advantage of its features and optimize database access, client applications may have to call some SQL functions of the CC. In the following parts, we will describe how each module works, what their specificities are and how they are inter-connected with client software.

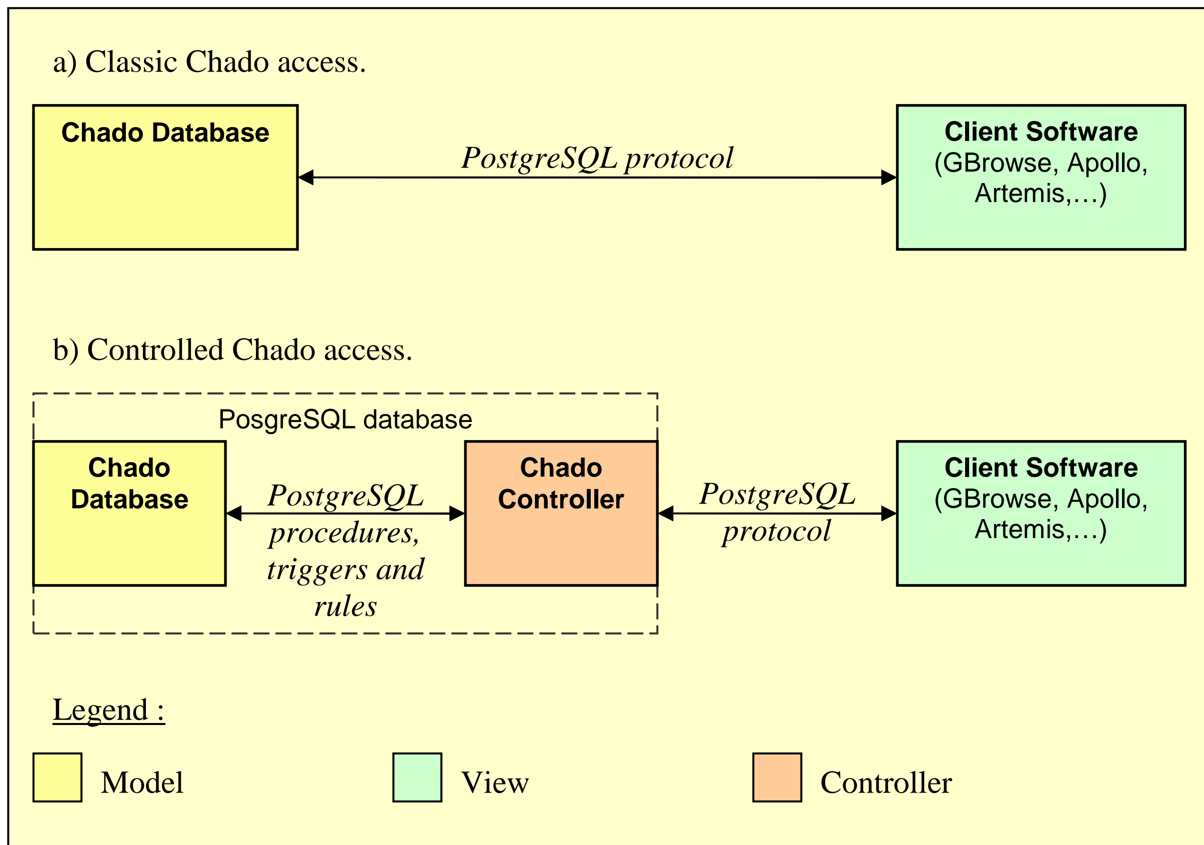

Figure 1. Model-View-Controller Architecture.

Access Restriction Module

Chado Schema Modifications

To manage users' and groups' access right, some modifications in Chado schema and additional tables are required (Figure 2). The Access Restriction module protects the "feature" table by renaming it into "feature_data" and removing access rights to that table from non-admin users. In order to let client software access feature data, a "feature" view is created with associated rules allowing SELECT, INSERT, UPDATE and DELETE SQL queries. This view lets the user only see the features he/she is allowed to see. The "annotator" table contains user account and group data. This table can also be used to store password but usually, as people use more than one Chado database instance, another separate database is

employed to store passwords. The table “user_group_link” is used to store the relationship between users and groups (*i.e.* which users belong to which groups). “feature_access” is the table that stores all access rights by associating a user account or a group, an access level to a feature. “feature_access_max_temp” is a temporary table that is created and valid during a user session only and removed when the user disconnects. It only contains the highest access rights of current user on each available feature. “feature_access_max” is a view used to retrieve the highest access right of a user on a feature by crossing “feature_access” table with user’s groups. This view is overridden by a temporary view at runtime for optimization purpose. The persistent version of the view can not use the temporary table “feature_access_max_temp” while the overriding temporary optimized version can and does. Finally, the “version” table is a table added by the CC to keep track of versions of installed modules (in case of update or compatibility checking).

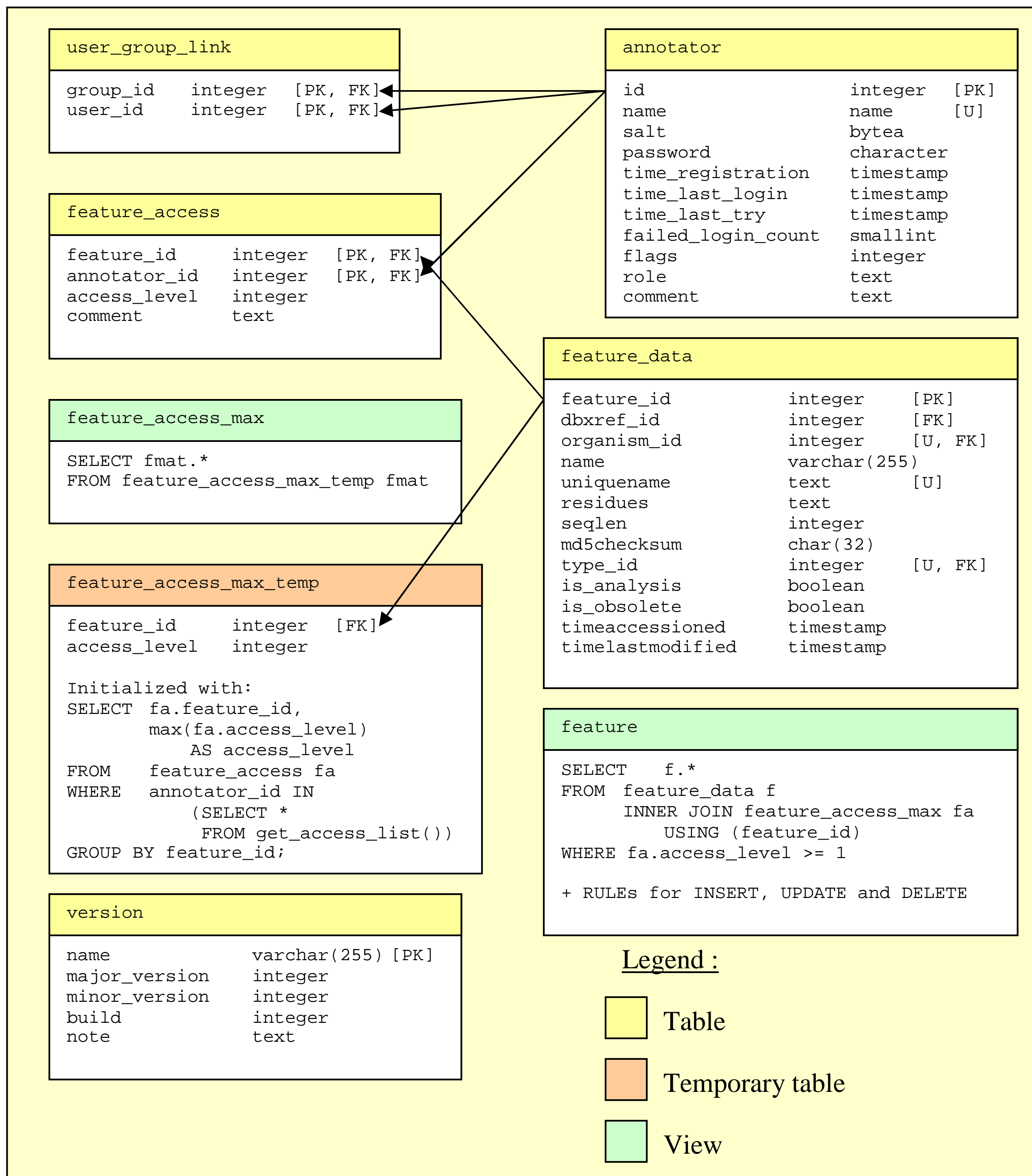

Figure 2. Chado schema modifications for access restriction.

Account Management Considerations

When a user fetches features, the “feature” view will only return allowed features of “feature_data” table. For each feature of the table “feature_data”, only the highest access right coming either from the user account or his/her group will be taken in account. To identify the user, the Access Restriction module uses PostgreSQL global variable “session_user”. That’s the reason why, in order to access or create features, **users must also have a PostgreSQL account sharing the same login as the one used in the “annotator” table.**

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 information about users and groups can be stored (roles, administrator comments) and accounts can be disabled or locked per database (for instance by the login interface after several failures to prevent password attacks).

Optimizations and Behaviour

As the "feature_data" table can contain a large number of rows that can be multiplied by the number of user and/or group-specific access right, the "feature_access" table can rapidly become huge and slow down access to features. To optimize feature access, a temporary table "feature_access_max_temp" that only contains the highest access right of current user is created dynamically for each session. To perform this task, the procedure "init_access" should be called at the beginning of each PostgreSQL session (before any transaction). As some client software may not perform that task, it is automatically performed by the Access Restriction module during the first query interacting with the "feature" view. Therefore, this first single composite query may be really slower than a call to "init_access" and then, performing the same query. See Figure 3 for details.

To call "init_access()", the source code of Artemis has to be modified. By default, Artemis uses "com.ibatis.common.jdbc.SimpleDataSource" class which does not provide a way to initialize each connection of its pool and gets really slow with the CC. Instead of using "com.ibatis.common.jdbc.SimpleDataSource" class in DatabaseDocument.java, "org.apache.commons.dbcp.BasicDataSource" class is used. Then, in config file "chado_iBatis_config.xml", "init_access()" can be called as the validation procedure:

```
<property name="validationQuery" value="SELECT init_access();"/>
```

For Apollo, a new data adapter called "PostgresChadoControllerAdapter.java" is provided and should be included when compiling Apollo. This adapter should be used in "chado-adapter.xml" config file:

```
<chado-adapter>
…
    <chadodb>
…
        <adapter>apollo.dataadapter.chado.jdbc.PostgresChadoAdapter</adapter>
…
    </chadodb>
</chado-adapter>
```

Please note that Apollo was not the annotation editor we used for functional annotation and it has not been exhaustively tested with the CC.

In the case of Web applications such as GBrowse, each page loaded on user side means a new database connection is opened. Running "init_access()" each time a page is loaded can become quite annoying. Moreover, GBrowse uses only one database login account and does not allow user to connect to the database using their account. Therefore, in order to have a more comfortable browsing experience, another optimisation way has been employed. GBrowse is configured to use a database account with full read access to all features but each query to the feature table includes an access restriction sub-query. This has been achieved by patching the Chado adapter Perl library (Bio::DB:DAS::Chado). "Chado.pm" and "Segment.pm" has been modified in order to include a restriction sub-query in each query made on feature_data table if access restriction is enabled. Otherwise, the Chado adapter just

behaves like usual. The restriction sub-query crosses “feature_access” and “feature_data” tables using an administrator account and involves user id and user group id found when the user logged in using the login interface added to GBrowse (kept in session object). This approach slows down each query but the Web page will load faster than calling “init_access()” each time a page is loaded.

The case of “gmod_bulk_load” script raises another issue. This script uses “COPY” SQL queries which can not be used on views, even with rules. Therefore, the feature view prevents the script from loading data into the database. To avoid this issue, the Access Restriction compatibility mode (c.f. “Compatibility Mode” below) can be used but when activated, users can not access to the “feature” table (which remains protected). That’s why a modified version of “gmod_bulk_load” script which works on the “feature_data” table (instead of “feature” view) is provided.

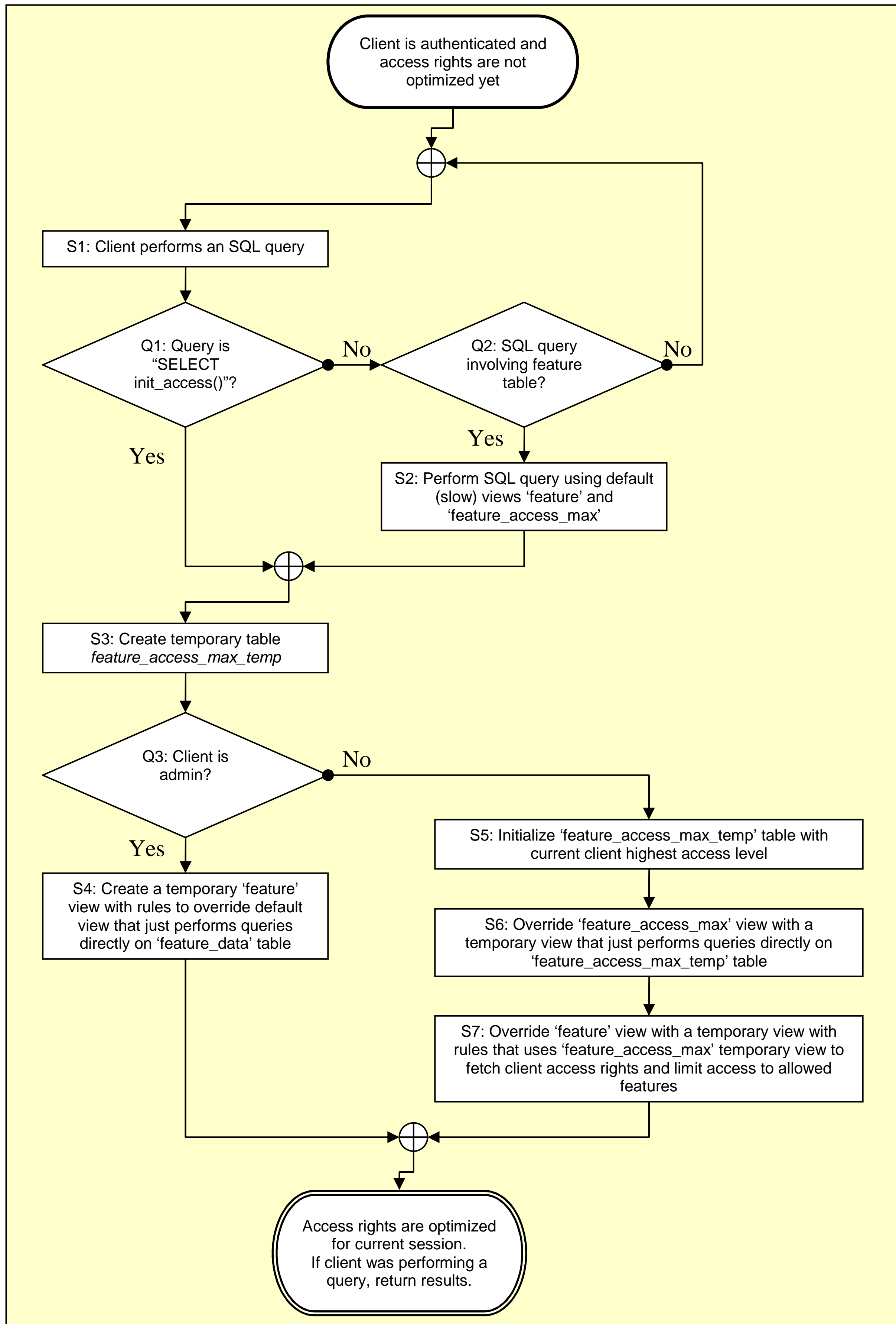

Figure 3: State diagram of session access rights optimization.

At state S1, if the client software does not support the Access Restriction module, it may execute queries on

feature view. This will lead to state S2 through Q1 (no) and Q2 (yes). Otherwise, if the client software is "Access Restriction module aware", it will call init_access() first and go to state S2 through Q1 (yes).
In the case of an administrator account which has full access to feature_data table, no restriction is needed and it leads from state S3 to state S4 through Q3 (yes) directly. Otherwise, several optimization states (S5, S6 and S7) are done.

Compatibility Mode

As the Access Restriction module replace "feature" table with a view, some scripts or programs may not work properly because they expect "feature" relation to be a table and not a view. In order to address that issue, a "Compatibility Mode" can be enabled and disabled. The "Compatibility Mode" renames the "feature" view into "feature_view", the "feature_data" table into "feature" table and creates a "feature_data" view that transfers queries to "feature" table (Figure 4). In this way, the scripts or programs requiring the relation "feature" to be a table can work again. The counter-part of this is that only the administrator account (which owns the "feature" table) can access to this table and use those programs. While the compatibility mode is turned on, regular annotators can not access to the feature table; they must use the "feature_view" view instead. The compatibility Mode has not been designed to be turned on all the time but just for temporary administration tasks such as tracks loading using the "COPY" SQL query for instance.
To enable the compatibility mode, the following SQL query should be used:

```
SELECT set_ar_compatibility(TRUE);
```

To disable the compatibility mode and put back the Access Restriction module in its initial state, the following SQL query should be used:

```
SELECT set_ar_compatibility(FALSE);
```

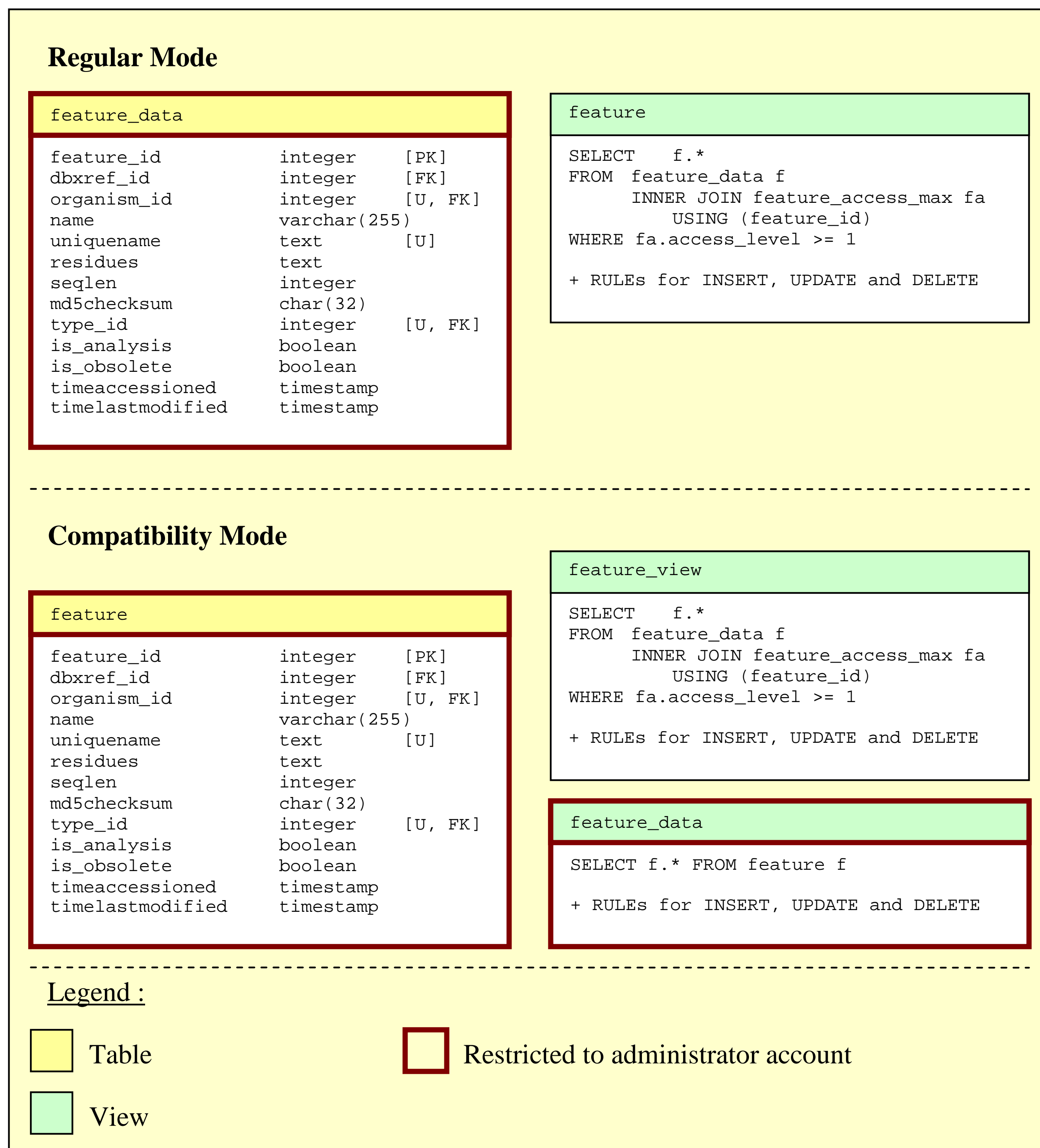

Figure 4. Chado Controller schema changes between regular mode and Access Restriction Compatibility Mode.

Access level

By default, newly created features inherits access right from their parent (found using "srcfeature_id" of "featureloc" table, see installed "assign_default_rights()" PostgreSQL procedure in your Chado database for details). Of course, the originator of the feature also has full access to that feature. If a feature has no access set for a specified user (or his/her groups) then the feature won't be accessible to that user. The policy of the Access Restriction module is to forbid access to features by default until an access right has been explicitly given.

Four Feature access levels exist:
0 or no level set: 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 a complete control over the feature.

Note: to be able to edit a feature, the user must also have the binary flag 0b000001000 (i.e. ANNOTATOR_FLAG_WRITE_ACCESS) of his annotator account or group set (c.f. "annotator" table of Figure 2). That flag also allows the user to add new features to the Chado database.

Authentication

As mentioned earlier, Chado annotator accounts must have a corresponding PostgreSQL account (cf. Account Management Considerations). However, the Access Restriction module cannot use PostgreSQL to authenticate the annotators of a Chado instance from GBrowse. Therefore, passwords must be stored in a database. The Access Restriction module can either store passwords in each Chado instance or use a shared password database to help synchronizing password changes. For obvious security reasons, passwords are not stored in clear text: password MD5 hashes (with random salt) are stored instead.

GBrowse 1.x does not include login facilities. In order to enable users to login on GBrowse 1.7x interface, additional modules and patches for GBrowse code are provided. The module to use to authenticate annotators can be specified in GBrowse configuration file allowing custom authentication modules to be written. As GBrowse 2.3x includes built-in login facilities, an authentication plug-in has been written.

Annotation Inspector

Modularity

The Annotation Inspector is a set of SQL procedures that are either triggered by events or called by user programs. The Annotation Inspector has been written in a modular way: each part of the Inspector can be installed or not and enabled or disabled at runtime (cf. "config_annotation_inspector.tmpl" file generated during the installation process in the installation directory). For instance, if you do not wish to install the transposable element management part, add "INSTALL_AUTO_TE_RELATIONSHIP=0" to the Chado Controller installation command line. At runtime, each trigger or validation procedure can be disabled manually or all triggers can be disabled using the Annotation Inspector Compatibility Mode made for this purpose.

Validation procedures

Annotation Inspector validation procedures perform various annotation checks and automate some tasks to build a feature annotation consistency report. These procedures are called by (modified) annotation software like Apollo or Artemis when the user wants to commit the changes but they can also be called manually using the SQL query:

```
SELECT validate_annotations(<transaction group identifier>, <fore commit status>);
```

The transaction group identifier is the numeric value that regroups modifications made by the annotator in "*_audit" tables (cf. Annotation History section). When the annotation work starts, the procedure "start_new_transaction_group()" is called and returns the transaction

group identifier that can be used later on to validate the annotation work done. If "start_new_transaction_group()" is not called, current PostgreSQL session identifier is used. Each transaction group identifier is unique and strictly positive (session identifier) or negative (identifier from start_new_transaction_group() procedure). If 0-value is used as the transaction group identifier, the validation process is performed on the entire database features.

The "force commit status" parameter is a boolean value. When set to false, only a report is provided and no data changes are performed. If set to true, some feature properties may be automatically added to annotated features to remind the annotator some problems remain to be fixed. For instance, if a stop codon is found inside frame, the feature property "stop_in_frame" is added. If several stop codons are found, then "multiple_stop_in_frame" feature property is added. If no stop codon is found inside frame, none of these properties are added.

Two columns are returned by the "validate_annotations(…)" procedure. The first one is the validation report (human readable text) and the second one is an integer reflecting validated features. If that number is positive or null, no error was encountered and if that number is negative, the absolute value is the number of errors encountered.

Validation procedures can also be enabled or disabled using the column "enabled" in the "annotation_inspector_procedures" table (cf. Figure 5). This table can also be used to add new custom validation procedures. Writing new validation procedure requires knowledge in PostgreSQL procedure language and a good understanding of how Chado data are stored and how the Annotation History module works. A good way to start writing a new validation procedure would be to copy and modify an existing one.

Basically, a validation procedure retrieves the features to check using "*_audit" tables. Then, it performs its validation process and chooses either to just report errors or also add properties to record encountered errors. To be called by the "validate_annotations(…)" procedure, a validation procedure must appear in "annotation_inspector_procedures" table (cf. Figure 5) with enabled column set to true. Annotation procedures are called in priority order (priority column), the highest priority value being called first.

annotation_inspector_procedures	
name	varchar(255) [PK]
priority	integer
enabled	boolean
is_trigger	boolean
description	text

version	
name	varchar(255) [PK]
major_version	integer
minor_version	integer
build	integer
note	text

Figure 5. Tables added to Chado by the Annotation Inspector module.

Beside validation procedures and triggers, the Annotation Inspector also comes with various helper functions wich complement Chado API. The documentation of each function is available before the code of each function in "install_annotation_inspector.tmpl". Provided functions are:

- `insert_or_update_feature_property;`

- `set_feature_cvterm;`
- `retrieve_polypeptide;`
- `retrieve_repeat_region;`
- `retrieve_annotated_feature;`
- `retrieve_te;`
- `retrieve_gene_related_features;`
- `retrieve_repeat_region_related_features;`
- `retrieve_related_features.`

The feature annotation consistency report is displayed in a Java dialog box in Artemis (c.f. DatabaseDocument.java) or Apollo (c.f. PostgresChadoControllerAdapter.java) at commit time.

Annotation History

The Annotation History module derivates from Chado audit module (c.f. http://gmod.org/wiki/Chado_Audit_Module). Like the Chado audit module, the Annotation History module creates an "audit" clone of each existing table but it brings several differences. First, Chado Audit module adds a row to the corresponding audit table when the original row is being changed. Therefore, the corresponding audit table only contains previous versions of modified rows. This behaviour saves space but was not easy to query to retrieve the history and current version of a feature, especially when several tables need to be joined (some identifier may be missing in audit tables). That's one good reason why the Annotation History module stores duplicates of current data in each audit table. It's also a way to know when and in which order rows are inserted.
Then, the original audit module did not include several useful information such as the user who did the changes and in which order changes where made. Indeed, the transaction date was recorded but when using PostgreSQL transaction blocks, several transactions could share the exact same date (*i.e.* the commit date)! With the Annotation History module, this information is added. The added "transaction_user" column records the effective PostgreSQL user account that did the transaction (c.f.
http://www.postgresql.org/docs/8.4/interactive/datatype-character.html#DATATYPE-CHARACTER-SPECIAL-TABLE for column type details). The "transaction_group"column tells which transactions were done together and the "transaction_id" column gives the order of transactions. The transaction type codes were changed (existing code were upper cased and lower case "i" was added):

- Type 'i' corresponds to rows already there when the Annotation History module was installed;
- Type 'I' means the audit row contains the data which has been inserted using "INSERT" query;
- Type 'U' means the audit row contains the data which has been updated using "UPDATE" query;
- Type 'D' means the audit row contains the data which has been deleted using "DELETE" query.

*_audit	
*	*
transaction_date	timestamp
transaction_type	char(1)
transaction_user	name
transaction_group	integer
transaction_id	integer

version	
name	varchar(255) [PK]
major_version	integer
minor_version	integer
build	integer
note	text

Figure 6. Tables added to Chado by the Annotation History module.

The annotation history of a feature can be displayed with GBrowse 1.7x using the CGI script "gbrowse_history" provided in the CC installation package. This script derivates from "gbrowse_details" script and can be customized either by modifying the source code or changing the content of the arrays "@gene_properties_to_display" (for genes) and "@default_properties_to_display" (other types of feature).

Contacts

valentin.guignon@cirad.fr
stephanie.sidibe-bocs@cirad.fr