Chado Documentation

Chado

Sep 08, 2019

Contents:

Chado is a relational database schema that underlies many GMOD installations. It is capable of representing many of the general classes of data frequently encountered in modern biology such as sequence, sequence comparisons, phenotypes, genotypes, ontologies, publications, and phylogeny. It has been designed to handle complex representations of biological knowledge and should be considered one of the most sophisticated relational schemas currently available in molecular biology. The price of this capability is that the new user must spend some time becoming familiar with its fundamentals.

CHAPTER 1

Installation

1.1 Installing Chado

First you will need database software, or Relational Database Management System (RDBMS). The recommended RDBMS for Chado currently is Postgres. Postgres is free software, usually used on a Unix operating system such as Linux or Mac OS X. You can also install Postgres, and Chado, on Windows but most Chado installations are found on some version of Unix - you'll probably get the best support by choosing Unix. (See Databases and GMOD for more discussion.) Once you've installed your RDBMS you can install Chado.

1.1.1 Download a Stable Release of Chado

See Downloads

1.1.2 Chado From SVN

You can get the most up-to-date, not even released yet, version of Chado from Subversion. To get a copy of the latest Chado source, enter this at the command line:

```
svn co https://svn.code.sf.net/p/gmod/svn/schema/trunk
```

Once the package has been downloaded cd to the `trunk/chado` directory.

Follow the instructions in the `INSTALL.Chado` file, including the installation of the prerequisites. Or, read the file online.

1.2 Loading Data

After completing these steps, you can load your Chado schema with data in a number of ways:

Load RefSeq into Chado HOWTO http://gmod.org/wiki/Load_RefSeq_Into_Chado Load GFF into Chado HOWTO http://gmod.org/wiki/Load_GFF_Into_Chado Using XORT http://gmod.org/wiki/XORT

You can also use the application Apollo to curate data in Chado.

CHAPTER 2

Using Flyway

2.1 What is Flyway?

Flyway is a database migration tool. As of Chado 1.4, the schema is distributed and updated via Flyway.

Flyway creates a `flyway_schema_history` table in your Chado database. Once Flyway is configured, it will look for migrations (versioned SQL scripts) in your specified migration folder(s). These migrations are tracked in the `flyway_schema_history` table, and will be run in order when you run the migrate command. As the Chado schema is updated with new migrations, they can be tracked and updated with Flyway.

Note: Please read the Flyway getting started guide for more details before continuing.

2.2 Adding Flyway integration to an existing site

First, follow the instructions for installing Flyway using your distribution method of choice. You'll need to provide a `flyway.conf` file which describes the connection details to your Chado database. For quick testing, place one in your home directory

Note: Flyway will search for and automatically load config files from the following paths if present:

- `<install-dir>/conf/flyway.conf`
- `<user-home>/flyway.conf`
- `<current-dir>/flyway.conf`

For managing multiple development Chado databases on a single machine, we recommend placing your application-specific `flyway.conf` in the application root directory and then running Flyway commands from there.

Here is an example configuration file where the PostgreSQL server is on the same machine as Flyway (i.e. localhost) and Chado is installed inside a database named `drupal` and a schema named `chado` (a typical arrangement for a Tripal-installed instance of Chado):

```
flyway.url=jdbc:postgresql://localhost:5432/drupal
flyway.user=user
flyway.password=secret`
flyway.schemas=chado
flyway.locations=filesystem:/path/to/Chado/chado/migrations
flyway.validateOnMigrate=false
```

For your setup, replace `drupal` in the `flyway.url` setting with the name of the database where Chado is installed, and change the `flyway.schemas` setting to the schema where Chado is installed. If Chado was installed using the Perl-based installer this will be `public`.

Once Flyway is configured properly, running `flyway info` should report a connected database. The first step is to run `flyway baseline`. This tells Flyway what version of Chado to start at. Chado switched to Flyway prior to `1.4`, so specifying `flyway baselineVersion=1.3 baseline`, or just `flyway baseline` (defaults to `1.0`) should be appropriate for your pre-existing database. Now, running `flyway info` should list all of the migrations available to your database with their state as **Pending**.

```
flyway info
Flyway Community Edition 5.2.4 by Boxfuse
Database: jdbc:postgresql://localhost:5432/drupal (PostgreSQL 10.5)
Schema version: 1

+-----------+---------+-----------------------+----------+---------------------+------
↪----+
| Category  | Version | Description           | Type     | Installed On        |␣
↪State    |
+-----------+---------+-----------------------+----------+---------------------+------
↪----+
|           | 1       |  Flyway Baseline      | BASELINE | 2018-12-20 14:31:32 |␣
↪Baseline |
| Versioned | 1.3.3.001 | add stock biomat table | SQL      |                     |␣
↪Pending  |
+-----------+---------+-----------------------+----------+---------------------+------
↪----+
```

2.3 Updating a Chado database with Flyway

Once your migrations appear in Flyway, you can run `flyway migrate` to execute all new migrations. By default, Flyway will run all pending Migrations greater than your current version. Afterwards, when new updates are added to Chado, you can update the Chado codebase on your server to an official tagged release or to the development branch and run `flyway migrate`. Once a migration is completed, `flyway info` will update the **State** of the migration from **Pending** to **Success**.

2.4 Writing new migrations

To create a new migration, create a new `.sql` file in the migration folder. The file should contain plain SQL that performs the proposed changes. However, the file name is important: the first migration, for example, is `V1.3.3.001__add_stock_biomat_table.sql`. Subsequent migrations should follow the schema: *V{version}{increment}__{description}*‘ where: - {version}: is the current version of chado - {increment}: is a number

that should be incremented for each new migration - {desrciption}: a very brief description of what the migration changes.

Please note that proposed changes to the Chado schema should follow the project contribution guidelines. You can have multiple migration folders to allow for site-specific migrations.

CHAPTER 3

Modules

3.17 Stock

No guide exists.

3.18 WWW

No guide exists for this module.

CHAPTER 4

Chado Best Practices

Chado is a generic schema, which means anyone writing software to query or write to Chado (either middleware or applications) should be aware of the different ways in which data can be stored. We want to strike a nice balance between exibility and extensibility on the one hand, and strong typing and rigor on the other. We want to avoid the situation we have with GenBank entries where there are a dozen ways of representing a gene model, but we need to be able to cope with the constant surprises biology throws at us in an attempt to confound our nice computable models. This page on Best Practices represents the collective wisdom of those who use Chado on a daily basis and are also familiar with its theoretical underpinnings.

See also:

- Chado Sequence Module - description of many of the terms used here
- Introduction to Chado - useful visualizations of some of the models described here
- IGS Data Representation - further discussion on these conventions and how they were implemented at IGS (for comparison)

Contents

4.1 Canonical Gene Models

The central dogma model states that "gene makes mRNA makes polypeptide" - for many people using Chado this may be the only data model that's relevant. This typical protein-coding gene model consists of one gene, one or more mRNAs, one or more exons, and at least one polypeptide.

Alternately spliced genes have a 1-to-many relation between gene and mRNA. Exons can be part_of more than one mRNA. No two distinct exon rows should have exact same featureloc coordinates (this would indicate they are the same exon).

Every localized feature must have a featureloc with rank=0 and locgroup=0 (see below for info on features with localization). The value of the srcfeature_id column should be identical (i.e. all features are located relative to the same feature), except in rare circumstances such as when a feature crosses two contigs (software is not guaranteed to support this). The srcfeature_id can point to a contig, a chromosome, chromosome_arm or other appropriate assembly unit.

This scenario involves rows in the following tables:

Table	type_id	Number	Comments
feature	SO:Gene	1	The gene must always be provided
feature	SO:mRNA	1 or more	One or more transcripts are required, and these are always of type SO:mRNA for protein-coding genes.
feature	SO:exon	1 or more	Exons are always required, even if the genome under consideration has no introns.
feature	polypeptide	at least 1	A protein-coding gene always produces a polypeptide, by definition. The polypeptide is located relative to the same genomic feature as the exons, mRNAs and gene. A single featureloc is used, with fmin and fmax indicating the start and stop codon positions (location is inclusive of stop codon). The polypeptide sequence should be specified as an amino acid sequence.

Some feature types such as introns are not normally manifested as rows in chado. They are normally derived on-the-fly from the gaps between consecutive exons. See below for a discussion of adding these implicit features.

4.1.1 Querying for Canonical Genes

Sample query: retrieve a gene, "Dredd", along with its transcripts, proteins and exons. Since this is a "canonical gene" we can assume that its feature graph has 3 levels. If we follow this assumption:

```
SELECT * FROM feature AS gene
    INNER JOIN
  feature_relationship AS feat0 ON (gene.feature_id = feat0.object_id)
    INNER JOIN
  feature AS subfeat1 ON (subfeat1.feature_id = feat0.subject_id)
    INNER JOIN
  feature_relationship AS feat1 ON (subfeat1.feature_id = feat1.object_id)
    INNER JOIN
  feature AS subfeat2 ON (subfeat2.feature_id = feat1.subject_id)
WHERE
  gene.name = 'Dredd';
```

This query should fetch a 3-deep graph rooted at "Dredd".

Application support for canonical genes

- Supported by Apollo
- Supported by GBrowse

4.2 Feature Localization

All features with sequence annotation should be localized using featureloc.

Localized features must have a featureloc with rank=0 and locgroup=0. This is the primary location of the feature. The location always indicates the boundaries of the feature. If the feature is composed of distinct subfeatures (e.g. a transcript composes of exons), then it is **not** permitted to use multiple featurelocs to indicate this. Instead, there must be rows for the subfeatures, each with their own featureloc.

In a feature graph (i.e. a group of features connected via feature_relationship rows), all features will typically be localized relative to the same source feature (i.e. they will all have the same value for featureloc.srcfeature_id).

Features are typically localized to some kind of genomic or assembly feature, but chado does not constrain you to using only this. For example, localizing features relative to a transcript or polypeptide or even exon is permitted, but unusual practices will most likely not be recognized by most software.

4.2.1 Feature Localization to Contigs in Assembly

In an assembled genome, it is common to locate relative to the top-level assembly units (e.g. chromosomes). However, it is also permissible to locate to smaller units such as contigs or golden_path_units.

If a genome assembly is not stable, it is common to locate relative to assembly units such as contigs. These contigs may then be localized relative to the top-level assembly units. This is known in chado terms as a location graph.

We discuss here location graphs of depth 2. See also N-level assemblies. This scenario is often invisible to software interoperating with Chado. The software is free to only look at the main features and the contig-level feature and ignore the top-level assembly feature. It may sometimes be desirable to have software that can perform location transformations, mapping features from contigs to top-level units and back.

Application support for localization to contigs

- Apollo: Status unclear
- GBrowse: Status unclear

Apollo should be happy to treat contigs just as if they were top-level units as chromosome arms. However, the user may have to explicitly provide contigs if location queries are desired. For example, Apollo may retrieve nothing if the user asks for a certain range on "chromosome 4", and the features are located relative to contigs which are themselves on "chromosome 4".

GBrowse may expect features to be located relative to top-level units such as chromosomes.

4.2.2 Redundant Localizations to Different Assembly Levels

Features can be located relative to both contigs and top-level assembly units.

Chado allows redundant feature localization using featureloc.locgroup > 0. This allows a database to have primary locations for features relative to contigs, and secondary locations relative to top-level units such as chromosomes. The converse is also allowed.

However this scenario is discouraged unless the chado db admin knows what they are doing. They must implement solutions to ensure that featurelocs with varying locgroup do not get out of sync. These solutions are not part of the standard Chado software suite. Nevertheless, this scenario may be useful for advanced users in certain circumstances

Application support for localization to different assembly levels

- Apollo: Status unclear
- GBrowse: Status partial

It is not clear if GBrowse uses locgroup in querying. If it constrains by locgroup, then this is essentially the same as feature localization to contigs in assembly.

Not clear if Apollo uses locgroup in querying. If it constrains by locgroup, then this is essentially the same as feature localization to contigs in assembly. Apollo will not preserve redundant featurelocs when writing back to the database. This could lead to the database getting out of sync.

4.2.3 N-level Assemblies

In theory it is possible (but rare) to have assemblies with variable depths, or with depths > 2. This scenario is rare. If required, then Chado can deal with this - there is no theoretical limit to the depth of a location graph. One can have annotated features located relative to minicontigs which are located relative to supercontigs which are located relative to chromosomes. Most software that interoperates with Chado will not be able to deal with this, so this scenario is discouraged except by advanced users who have no other option.

4.2.4 Unlocalized Features

A gene without sequence based localization.

Many chado instances are purely concerned with genome annotation - in these cases it would be strange to have genes or other features such as transcripts with no localization (i.e. no featurelocs). However, this scenario is actually common when Chado is used in a wider context. We may learn of the existence of genes through non-sequence evidence such as genetics. When we have no sequence-based localization it is perfectly valid to have gene features with no featurelocs. When the time comes to create genome annotations for these, we just 'fill out' the gene feature by adding transcript and exon features.

4.3 Other Types of Gene Models

This section describes how one describes other commonly encountered gene models in Chado.

4.3.1 Noncoding Genes

Similar to canonical model (see above), except with noncoding RNA. Not all genes are protein-coding - for example, genes can code for tRNA, miRNA, snoRNA, etc. A noncoding gene model is identical to a canonical model, with the following exceptions:

- There is no polypeptide feature
- Instead of an mRNA feature, there is a feature that is some other sub-type of RNA

Application support for noncoding genes

- Supported by Apollo
- Supported by GBrowse

4.3.2 Pseudogene

A pseudogene is a non-functional relic of a gene. A pseudogene may look like an ordinary gene, and may even have discernible parts such as exons. It may sometimes be desirable to annotate the exon structure of a pseudogene - this can in principle be done using SO types such as decayed_exon. In practice no one is using Chado to do this. There are currently two practices for psuedogenes:

- Pseudogenes are treated analogously to Noncoding Genes (see above). That is, there are normal "gene" and "exon" features. However, in place of a subtype of RNA, there is a feature of type pseudogene. This practice is **strongly discouraged** (it is not compliant with the relations in the Sequence Ontology, as it gives false counts to the number of real genes in the database). Note that this is the current default for FlyBase.

- Pseudogenes are normal singleton features (see below). There is no annotation of exon structure. This practice is encouraged. If at a later date it becomes desirable to annotated the exon structure of a pseudogene, it will be compatible with this.

Application support for pseudogenes

- Apollo: status is unclear

Apollo by default treats pseudogenes using the first method, above. It may also be possible to configure it to the second, singleton, method. Annotating the exon structure of pseudogenes the correct way has not yet been attempted to our knowledge.

4.3.3 Singleton Feature

Many types of features are singletons - that is they are not related to other features through the feature_relationship table. Storage of these is basic and as one may expect. Singleton features present no major problems. Unlike genes, which typically have parts (with the parts having subparts), singletons do not form feature graphs (or rather, they form feature graphs consisting of single nodes). Singleton features are located relative to other features (usually the genome, but once can have singletons that are located relative to other features - this may not be supported by all applications).

Application support for singletons

- Supported by Apollo
- Supported by GBrowse

Apollo supports singletons provided they are located relative to the genome (singletons located relative to other features will be ignored). It may be necessary to configure apollo to make the feature type "1-level".

4.3.4 Trans-spliced Gene

A trans-spliced gene has one or more transcripts in which that transcript may be spliced together from different parts of the genome.

A trans-spliced transcript is spliced from exons coming from different parts of the genome. The distance between each trans-spliced part may be large, or it may be in the same location on the opposite strand.

Most *C. elegans* genes have a trans-spliced leader sequence. This is different from the trans-splicing involved in *Drosophila*, where we observe what appears to be two transcripts on separate strands (both containing coding sequence) joining together in a single functional transcript.

There are two proposals for dealing with this. One treats the trans-spliced transcript as a single transcripts, with exons coming from different locations. The other treats the trans-spliced transcript as a mature transcript created from two distinct primary transcripts. Note that these proposals focus on the *Drosophila* example. A solution for the *C. elegans* example has not been proposed.

We treat this as an ordinary gene model, but relax our rules for exon locations in a transcript. For example, for the canonical *Drosophila* trans-spliced gene, we would allow transcripts to have exons on different strands. Note that in Chado, exon ordering comes from *feature_relationship.rank* (between exon and transcript), not from the featureloc of the exon. Chado has no problem with this. However, some software may make assumptions that all exons are on the same strand, or may try to order exons by their location to get a transcript sequence. This software will have unintended consequences with trans-spliced genes modeled using this proposal.

We would introduce extra transcripts, and have relations between the transcripts. Only the mature, spliced, transcript would have a relation to the polypeptide. This may model the biology better. However, it introduces a major departure from the canonical gene model. For this reason this proposal is unlikely to be adopted.

Application support for Trans-spliced Genes

- Apollo: status unclear
- GBrowse: status unclear

4.3.5 Gene with Implicit Features Manifested

Some feature types such as introns are not normally manifested as rows in chado. They are normally derived on-the-fly from the gaps between consecutive exons. See for an example. Occasionally it may be desirable to store the introns as actual rows in the feature table - for example in a report database.

4.3.6 Immature or Primary RNA

Generally we do not explicitly represent primary RNA transcripts unless there is something useful to say about them. If one wants to instantiate these they would be represented as features, and the mature message would be related to the primary message with derived_from as type_id in the feature_relationship table.

Application support for unlocalized genes

- Supported by Apollo
- Supported by GBrowse

GBrowse supports this scenario in that unlocalized features will be ignored from the genome viewer, which is appropriate.

Apollo supports this scenario in that unlocalized features will be ignored, which is appropriate behaviour for a genome annotation tool.

4.3.7 Gene model types that still need best practices

- Operons
- Dicistronic genes (similar to operons) - See Intro to Chado Feature Graphs for a proposed solution for storing dicistronic genes.
- Gene with Regulatory Elements - Regulatory elements may be implicitly or explicitly associated with a gene.

4.4 Transposons

Transposons can be annotated as singleton features or as complex annotations. You would create a feature of type transposon insertion, with a loc of type 0 for insertion sites when the insertion is absent, 1 if present, and -1 (?) to link to the "template" – generic representation of the transposon?

A transposon may consist of various parts such as long_terminal_repeat and gene models coding for genes like gag, pol, and env. These parts may have all decayed over time. Transposon annotation typically ignores these subtleties as all that is usually required is a singleton-feature of type transposable_element. In this case, there is no difficulty.

If one requires detailed transposon annotation then one is entering uncharted water as far as both Chado and annotation tools are concerned (this scenario still needs best practices). One option would be to treat each transposon part as distinct singletons, but this may be unsatisfactory as one may desire to have the appropriate part_of relations between the parts.

4.5 Genomic Rearrangements

4.5.1 Inversions

Create a feature of type inversion with location spanning the inverted region with rank 0. If there is a version of the sequence containing the inversion create a featureloc to the inverted region with rank 1. The ranks serve to distinguish the two versions in case several sequences carry one or the other, but the choice of which is 0 and which is 1 is arbitrary, unless 0 is used for "wild type".

For example, for a rearrangement that exchanges the ends of two chromosomes A and B, create two features of Afrag and Bfrag of type rearranged segment, "locate" on A and B, then create features for A' and B' (post-rearrangement) and locate on B' and A', respectively. How to capture the fact that Afrag and Bfrag and A' and B' are part of same rearrangement? Use feature relations?

4.5.2 Translocations

Create a feature of type translocation with a location spanning the translocated region, rank 0. Rank=1 is used for insertion locations: whether latent (i.e. the site of the insertion on a contig that lacks the insertion) or explicit (the site of the insertion on a contig that carries it). The coordinates are adjusted accordingly.

4.5.3 Still Needs Best Practices

- Deletions
- Copy Number Variations

4.6 Single Nucleotide Polymorphisms (SNPs)

This outlines one way of modeling SNPs in chado. It also illustrates use of the featureloc table.

Most of this applies to other variation features, but we will illustrate using SNPs for now to keep it simple.

A SNP is represented as a single feature in chado.

Let's take a basic example - a SNP that changes an A to a G on the genome.

Here we would have one feature and two featurelocs.

```
(feature
  (name "SNP_01")
  (featureloc
    (srcfeature "Chromosome_arm_2L") ;;; dna feature identifier
    (nbeg 1000000)
    (nend 1000001)
    (strand 1)
    (residue_info "A")
    (rank 0)
```

(continues on next page)

(continued from previous page)

```
    (locgroup 0))
  (featureloc
    (residue_info "G")
    (rank 1)
    (locgroup 0)))
```

The first location is on the chromosome arm (presumably wild type). The second location has no srcfeature value (i.e. it is set to null). However, it is effectively paired with the first location. if we later wished to instantiate the mutant chromosome arm feature, we would fill in the second locgroup's srcfeature.

Let's take another example - a SNP that has only been characterised at the protein level. This SNP changes an I to a V.

```
(feature
 (name "SNP_02")
 (featureloc
   (srcfeature "dpp-P1")    ;;; protein feature identifier
   (nbeg 23)
   (nend 24)
   (strand 1)
   (residue_info "I")
   (rank 0)
   (locgroup 0))
 (featureloc
   (residue_info "V")
   (rank 1)
   (locgroup 0)))
```

Again, the second featureloc has no srcfeature. The mutant protein is implicit. The mutant protein sequence can be infered by taking the sequence of dpp-P1 and substituting the 24th residue with a V.

To do a query for all SNPs that switch I to V or vice versa:

```
SELECT snp.*
 FROM
  featureloc AS wildloc,
  featureloc AS mutloc,
  feature AS snp,
  cvterm AS ftype
 WHERE
  snp.type_id = ftype.cvterm_id        AND
  ftype.termname = 'snp'               AND
  wildloc.feature_id = snp.feature_id  AND
  mutloc.feature_id = snp.feature_id   AND
  wildloc.locgroup = mutloc.locgroup   AND
  wildloc.residue_info = 'I'           AND
  mutloc.residue_info = 'I';
```

Note that this query remains the same even if mutant protein features are instantiated as opposed to left implicit.

Let's look at a more complex example. If we have a SNP that has been localised to the genome, and the SNP has an effect on a protein (Isoleucine to Threonine), and we want to redundantly store the SNP effect on the genome, transcript and translation.

Note that in this example, the transcript is on the reverse strand, so the residue is reverse complemented.

```
(feature
 (name "SNP_03")
```

```
;; position on genome
 (featureloc
  (srcfeature "chrom_arm_3R")
  (nbeg 2000000)
  (nend 2000001)
  (strand 1)
  (residue_info "A")
  (rank 0)                          ;; wild
  (locgroup 0))
 (featureloc
  (residue_info "G")
  (rank 1)                          ;; mutant
  (locgroup 0))
```

```
;; position on transcript
(featureloc
 (srcfeature "blah-transcript001")     ;; processed transcript ID
 (nbeg 1000)
 (nend 1001)
 (strand 1)
 (residue_info "T")
 (rank 0)                          ;; wild
 (locgroup 1))
(featureloc
 (residue_info "C")
 (rank 1)                          ;; mutant
 (locgroup 1))
```

```
;; position on protein
(featureloc
 (srcfeature "blah-protein001")    ;;; protein feature identifier
 (nbeg 23)
 (nend 24)
 (strand 1)
 (residue_info "I")
 (rank 0)                          ;; wild
 (locgroup 2))
(featureloc
 (residue_info "T")
 (rank 1)                          ;; mutant
 (locgroup 2)))
```

Here we have 6 locations for one SNP. The 6 locations can be imagined to be in a 2-D matrix. The purpose of rank and locgroup is to specify the column and row in the matrix.

Allele	Genome	Transcript	Protein
Wild-type	A	T	I
Mutant	G	C	T

rank is used to group the strain and locgroup is used for the grouping within that strain. rank=0 should be used for the wildtype, but this is not always possible; locgroup=0 should be used for primary (as opposed to derived) location, this is not always possible. The important thing is consistency within a SNP to preserve the matrix.

One can imagine rare (but entirely possible) cases where by a single SNP causes different protein level changes in two proteins (for instance, HIV carries a doubly encoded gene - i.e. the ORFs overlap but have different frames).

Here we would want to add another locgroup, for the second protein.

Allele	Genome	Transcript	Protein1	Protein2
Wild-type	A	T	I	Y
Mutant	G	C	T	H

Again, if we don't need to instantiate the 2 mutant proteins, but their sequence can be reconstructed from the wild type proteins plus the corresponding mutation.

Remember chado uses interbase coordinates, and postgresql substring counts from 1.

The following query dynamically constructs mutant feature residues based on the wild type feature and the mutant residue changes. this should work for a variety of variation features, not just SNPs. Note that we need to use locgroup to properly group wild type/mutant pairs of locations, otherwise this query will give bad data.

```
SELECT
snp.name,
wildfeat.name,
substr(wildfeat.residues,
      1,
      wildloc.nbeg) ||
mutloc.residue_info  ||
substr(wildfeat.residues,
      wildloc.nend+1)
FROM
 featureloc AS wildloc,
 feature AS wildfeat,
 featureloc AS mutloc,
 feature AS snp,
 cvterm AS ftype
WHERE
 snp.type_id = ftype.cvterm_id         AND
 ftype.termname = 'snp'                AND
 wildloc.feature_id = snp.feature_id   AND
 mutloc.feature_id = snp.feature_id    AND
 wildloc.locgroup = mutloc.locgroup    AND
 wildloc.srcfeature = wildfeat
```

4.6.1 Extensions

The above will also work if we have a polymorphic site with a number of different possibilities across multiple strains. We just extend the number of rows in the location matrix (i.e. we have rank > 1).

We could also instantiate multiple SNPs, one per strain, and keep the locations pairwise.

4.6.2 Similarities to Alignments

You should hopefully notice the parallels between modeling SNPs and modeling pairwise (e.g. BLAST) and multiple alignments. The difference is, alignments would always have locgroup=0, with the rank distinguishing query from subject. Also, with an HSP feature, the residue_info is used to store the alignment string.

4.6.3 Redundant Storage of Coordinates on Different Assembly Levels

Some groups may find it advantageous to redundantly store features relative to both BACs and chromosomes (or to mini-contigs and scaffolds... choose your favourite assembly units). The approach outlined above works perfectly

well with this, we would simple add another column in the location matrix (i.e. another wild type/mutant pair with a distinct locgroup). All queries should work the same.

4.7 Sequence Alignments

4.7.1 Results from BLAST

These steps will add a BLAST analysis to a Chado database. Load_BLAST_Into_Chado provides a worked example of this using BioPerl and GMOD scripts, bp_search2gff.pl and gmod_bulk_load_gff3.pl

- Create a record for the BLAST search itself is an entry in the analysis table.
- Create a feature for both the query and target sequences. If these have database accessions or identifiers then records in the dbxref table should be created for each. * Optionally include the residues of the features.
- Store the hits in both the feature and analysisfeature tables, as well as in the featureloc table, where the featureloc entry for the query sequence has a featureloc.rank of 0 and the featureloc.rank of the target sequence is 1. The SO term is match. * Note: featureloc.locgroup is not used.
- Store the HSPs in both the feature and analysisfeature tables, as well as in the featureloc table, where the featureloc entry for the query sequence has a featureloc.rank of 0 and the featureloc.rank of the target sequence is 1. The SO term is match_part. * Note: featureloc.locgroup is not used.
- Map the hits and the HSPs to each other via entries in the feature_relationship tables.
- Standard scores (rawscore, normscore, significance, identity) are stored in the analysisfeature table. For BLAST searches those would correspond to bits, score, e-value and frac_identical.

4.7.2 Still Needs Best practices

- Multiple Sequence alignments

4.8 GO annotations

The details on GO annotation can be found on the Gene Ontology Consortium website. GO annotations can be captured in the Chado schema using the CV and the SEQUENCE modules. The CV module can be used to store the GO Ontology. Details of the CV module can be found at CV module documentation. The actual GO annotation which is a association between Gene Product/Gene is stored in the Feature_cvterm table. It is recommended that the GO term should be associated with a Gene Product feature. But, it could be associated with Gene feature. The Evidence code and qualifier information are stored in the Feature_cvtermprop table. Feature_cvterm_Dbxref table should be used to store the external ids associated with the evidence code, and Feature_cvterm_pub should link publications to annotations. For example evidence IEA with dictyBase:DDB0185051

4.9 More Needed Best Practices

- Posttranslational Modifications
- Genotypes
- Phenotypes
- Cleavage

- Protein Complexes
- Genome Versions

CHAPTER 5

Schema

You can browse the full schema here. This documentation includes table definitions, comments, constraints, and indices.

CHAPTER 6

Contribution Guidelines

The following guidelines are meant to encourage contribution to the GMOD Chado schema by making the process open, transparent and collaborative. **These guidelines apply to everyone contributing to Chado whether it's your first time (Welcome!) or you're a project management committee member.**

6.1 Pull Requests (PRs)

The goal of this document is to make it easy for A) contributors to make pull requests that will be accepted, and B) Chado committers to determine if a pull request should be accepted.

6.1.1 Requirements for Changes

In order for a change to be appropriate for the full Chado community, **you should be able to answer yes to the following questions**:

- Is the proposed schema/change generic (not site specific)?
- **Is it generally useful?**
  - If the answer isn't obviously yes, please provide a clear use case.
- **Is your solution Performative?**
  - Have you tested it on real data?
  - Have you tested it on large-scale datasets?
- Does your change maintain database integrity?
- Is your change Commented/Documented?
- Does your PR pass continuous integration (CI)?

6.1.2 Guidelines

- **All PRs MUST be linked to an issue.**
  - This provides a means for the Chado community to weigh in on any changes.
- **Issues must be open for 2 weeks before a PR is made.**
  - Any objection will freeze this time period until it is resolved or retracted.
  - If an objection is made and clarification requested, it should be provided in a timely manner.
  - If the objection is unable to be resolved, the Chado Project Management Committee (PMC) will evaluate the concern. The PMC can override objections if it unanimously agrees.
  - Issues with no feedback after 10 days should be bumped letting community members know it needs discussion promptly.
- **PRs require a minimum number of reviews before they are merged (number depends on the size of change).**
  - Small Changes: 2+ reviews (1+ PMC review).
  - Medium Changes: 4+ reviews (2+ PMC reviews; 1+ Non-Tripal reviews).
  - Large Changes: 6+ reviews (5+ PMC reviews; 1+ Non-Tripal reviews).
  - Not all reviews need to be from PMC members, the community is encouraged to chime in!
  - Definitions for magnitude of changes are below.

6.1.3 Change Magnitude Definitions

The following definitions attempt to provide guidance as to the magnitude of the change to the Chado database through providing examples. However, they are not yet complete or exhaustive and may change as Chado governance evolves. If your change does not align with any of these examples, please justify where you feel it fits in the PR and the Chado PMC will vote based on that information.

Small Changes

- New index.
- Update to documentation.

Medium Changes

- Adding a column to an existing table.
- Addition of a small "chado-esque" table (follows an existing template such as a prop or _cvterm table).
- New linking table.

Large Changes

- Any non-backwards compatible changes.
- A new module (i.e. base table with associated linking tables).
- A large infrastructure change.

CHAPTER 7

Mailing List

Table 1: Chado Mailing List

Mailing List Link	Description	Archive(s)
GMOD-schema	All issues regarding Chado, Chado::AutoDBI, and Bio::Chado::Schema	Gmane, Nabble (2010/05+), Sourceforge
GMOD commits	Chado code updates	Sourceforge Archive

CHAPTER 8

Pronunciation

Chado is usually pronounced like this.