The Chado Schema Manual

Chris Mungall

December 21, 2006

Contents

1 Introduction **1**

1.1 Schema 1

1.1.1 Module System 1

1.2 View layers 1

1.2.1 Inter-schema bridges 2

1.3 DBMS Functions 2

1.4 Software 3

2 The general Module: Identifiers **4**

2.1 Purpose 4

2.2 Design patterns 4

2.3 Tables 5

2.4 URLs and URIs 5

2.5 Identifiers and interoperability between Chado instances 6

2.6 Table Definitions 6

3 The cv Module: Ontologies **11**

3.1 Introduction 11

3.1.1 Transitive Closure 12

3.2 Table Definitions 14

4 The Sequence Module: Features **25**

4.1 The role of features in Chado 25

4.2 Best Practices 34

4.2.1 Chado Compliance Layers 34

4.2.2 Examples: Current implementations 34
4.3 Table definitions 37

Appendices **90**

A Chado Naming Conventions **90**
A.1 Case sensitivity 90
A.2 Table names 90
A.3 Column names 90
A.3.1 Primary and foreign key names 91

Chapter 1

Introduction

general intro here

outlines - overview then module descriptions

See also GMOD

and FlyBase

1.1 Schema

naming convention

design patterns

1.1.1 Module System

Module Metadata

1.2 View layers

Views can be thought of as *virtual tables*. They provide a powerful abstraction layer over the database. All views should be portable across all DBMSs

Views in chado are defined on a per module basis. View definitions are maintained in the `chado/modules/MODULE-NAME/views` directory.

Included in the view directory are *report* views. These can usually be found in a file called `chado/modules/MODULE-NAME/views/MODULE-NAME-report.sql`

Collections of view definitions are bundled into packages, each package is a `.sql` file.

1.2.1 Inter-schema bridges

GODB Bridge

BioSQL Bridge

1.3 DBMS Functions

DBMS Functions in Chado are entirely optional

Functions in chado are defined on a per module basis. Function definitions are maintained in the `chado/modules/MODULE-NAME/functions` directory.

Collections of function definitions are bundled into packages. Each package comes with an *interface descriptions* and one or more *implementations*.

Function Interface Definitions

The interface descriptions are stored in a `.sqlapi` file. The syntax used is a variant of SQL and is intended primarily as a consistent way of providing information for human, although it should be parseable by software.

Here is an example, taken from the top of the `chado/modules/sequence/functions/subsequence.sqlapi` package. This package provides basic subsequencing functions. It has dependencies on two other function packages, declared at the top of the file. The package declares multiple functions, only the first of which is show here, a function for extracting subsequences from the sequence of a feature.

```
IMPORT reverse_complement(TEXT) FROM 'sequtil';
IMPORT get_feature_relationship_type_id(TEXT) FROM 'sequence-cv-helper';

-----------------------------------
-- basic subsequencing functions --
-----------------------------------

DECLARE FUNCTION subsequence(
   srcfeature_id        INT REFERENCES feature(feature_id),
   fmin                 INT,
   fmax                 INT,
   strand               INT
)
 RETURNS TEXT;

COMMENT ON FUNCTION subsequence(INT,INT,INT,INT) IS 'extracts a
subsequence from a feature referenced by srcfeature_id, within the
interbase boundaries determined by fmin and fmax, reverse
complementing if strand = -1. The sequence can be DNA or AA. Strand
```

```
must always by >0 for AA sequences';
```

Function Implementations

The goal is to provide implementations for different dialects of procedural SQL. Currently only PostgreSQL dialect is supported. The psql implementations are stored in `.plpgsql` files.

1.4 Software

Chapter 2

The general Module: Identifiers

2.1 Purpose

General purpose tables are housed in the module general. The primary purpose of this module is to provide a means of providing data entities with stable, unique identifiers. In Chado, all identifiable data entities have bipartite identifiers, consisting of a dbname plus an accession, together with an optional version suffix.

By convention, these are normally presented using a `':'` separator. An example of an identifier in this notation would be `GO:0008045` or `FlyBase:FBgn00000001`. In the Chado schema the atomic units are the dbname and the accession, the separator is introduced only in the presentation layer. Each dbname uniquely identifies the authority responsible for a particular ID-space (so there cannot be two `GO` in any single Chado instance). The accession must be unique within the ID-space. Thus there can be two accessions `0008045`, but there can only be one data artefact identified as `GO:0008045`.

These uniqueness constraints are encoded in the schema, so it is impossible for any Chado relational database instance to violate them.

Each identifier is stored as a row in the dbxref table, with the dbname stored in the db table. Keeping the dbname in a separate db table ensures that the Chado schema retains its commitment to normalization. Entries in other tables can refer to entries in the dbxref table by means of foreign keys.

Note that all stable identifiers are stored in the dbxref table, whether or not they refer to 'external' data entities. Chado does not have an explicit notion of a data entity being external. Some dbxrefs have further information fully fleshed out in other tables in the database, and others are 'dangling' dbxrefs.

2.2 Design patterns

Primary identifiers: ENTITY.dbxref_id REFERENCES dbxref(dbxref_id)

Secondary identifiers: ENTITY_DBXREF.dbxref_id

2.3 Tables

The two main tables are `dbxref` (for the identifier itself) and `db` (for the name of the DB or ID-granting authority). By separating the db into its own table rather than duplicating the name in the dbxref we retain *normalization*

A dbxref identifier has two key parts: a `db_id` column that refers to an entry in the `db` table, and an `accession` column, that must be a locally unique identifier within the `db` referred to by the `db_id` column. An optional third column is the version column. Taken together, these 3 columns constitute a unique key.

The db is a database authority. Typical dbs in bioinformatics are FlyBase, GO, UniProt, NCBI, MGI, etc. The authority is generally known by this sortened form (the `db.name`, which is unique within the bioinformatics and biomedical realm. See below for more on uniqueness. This name is typically in short mnemonic (but human-friendly) form, and uniquely identifies the DB/authority (enforced by uniqueness constraint). Examples include FlyBase, GO, MGI. Short human-friendly names are encouraged, although longer names (such as full LSID prefixes) may also be used. The name should be a valid XML NMTOKEN (see XML specification for details) - for example, it should not start with a number. This constraint is to help syntactic interoperability with other identifier schemes. To ensure interoperability with other Chado databases, the same db.names should be used (e.g. `FlyBase` should be used consistently instead of `FB`). This will prevent duplicate dbxref rows being created if and when databases are merged. At the same time, uniqueness must be preserved: there must not be two `GO`s. See below for more information.

2.4 URLs and URIs

See the following for background:

http://en.wikipedia.org/wiki/Uniform_Resource_Identifier

http://en.wikipedia.org/wiki/URL

Basically, a URI is an addressing scheme. The form of URI most people are familiar with are URLs; but not all URIs are URLs. Another URI addressing scheme is the URN; for example, LSIDs use URNs.

People often expect URLs to be resolvable using standard technology (eg a web browser) to a resource intended for humans, but this isn't always the case. URNs may require other software to resolve them; eg LSID resolver.

The `db` table has columns for both URL and URI. The former is intended just to go to a website like the FlyBase or GO home page. The latter is intended as a globally unique addressing scheme that should be interoperable with other schemes. For example `GO` may be a unique identifier for the Gene Ontology ID space *by fiat* within the bioinformatics community, but not outside. Although Chado only cares about the former, it may have to interoperate with schemes that care about truly

global uniqueness, hence URIs.

This column is nullable, so it is possible to defer decision on what the unique URI for a particular authority is if this information is not known up-front. See below for mechanisms for assigning URIs to DBs and ensuring uniqueness.

2.5 Identifiers and interoperability between Chado instances

2.6 Table Definitions

tableinfo

NULL

Table 2.1: tableinfo

Column	Datatype	Description
tableinfo_id	integer	
name	varchar	
primary_key_column	varchar	
is_view	integer	
view_on_table_id	integer	
superclass_table_id	integer	
is_updateable	integer	
modification_date	date	

db

A database authority. Typical dbs in bioinformatics are FlyBase, GO, UniProt, NCBI, MGI, etc. The authority is generally known by this sortened form, which is unique within the bioinformatics and biomedical realm.

Table 2.2: db

Column	Datatype	Description
db_id	integer	
name	varchar	A (typically short, mnemonic) name of the ID-space, database or ID-granting authority. The db.name uniquely identifies the DB/authority. Examples include FlyBase, GO, MGI. Short human-friendly names are encouraged, although longer names (such as full LSID prefixes) may also be used. The name should be a valid XML NMTOKEN (see XML specification for details) - for example, it should not start with a number. This constraint is to help syntactic interoperability with other identifier schemes. To ensure interoperability with other Chado databases, the same db.names should be used (e.g. FlyBase should be used consistently instead of FB). This will prevent duplicate dbxref rows being created if and when databases are merged. At the same time, uniqueness must be preserved: there must not be two "GO"s. See supporting docs for more info
description	varchar	contact_id int,
urlprefix	varchar	
url	varchar	A W3C compliant URL with the address of a website containing information about the DB/authority. For example, http://www.flybase.org, http://www.geneontology.org. The URL is intended for humans rather than software agents
uri	varchar	A W3C compliant URI that contains a unique namespace for the DB/authority. Some ID schemes (eg LSID) require this. The URI is intended for software agents rather than humans. It does not need to be a resolvable URL. However, certain DBs may prefer the URI to be a resolvable URL that has human-readable information on the other end. Other DBs may provide URNs (eg LSID URNs) that require software agents to be resolved. Note that it is perfectly acceptable for the db.name column to be the same as the URI column (provided it is a valid URI). However, it is encouraged that a short form is used as the db.name. See supporting docs for more information

dbxref

A unique, global, public, stable identifier. Not necessarily an eXternal reference - can reference data items inside the particular chado instance being used. Typically a row in a table can be uniquely identified with a primary identifier (called dbxref_id); a table may also have secondary identifiers (in a linking table ¡T¿_dbxref). A dbxref is generally written as ¡DB¿:¡ACCESSION¿ or as ¡DB¿:¡ACCESSION¿.the ID-spa¡VERSION¿.

Table 2.3: dbxref

Column	Datatype	Description
dbxref_id	integer	
db_id	integer	
accession	varchar	The local part of the identifier. Guaranteed by the db authority to be unique for that db.
version	varchar	
description	text	

project

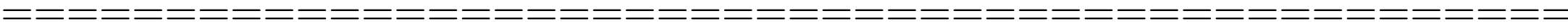


Table 2.4: project

Column	Datatype	Description
project_id	integer	
name	varchar	
description	varchar	

Chapter 3

The cv Module: Ontologies

3.1 Introduction

We have seen how the sequence module makes extensive use of terms taken from various ontologies such as SO and the OBO Relations Ontology, using the type_id foreign key column. In addition, features can be annotated using ontologies such as GO using the feature_cvterm linking table. These terms are modelled using the cv module, the core of which is the cvterm table.

An ontology, terminology or cv (controlled vocabulary) , is a collection of terms (here equivalent to what are more typically called classes, types, categories or kinds in the ontology literature[REF]) in a particular domain of interest. Examples include "gene" (from SO), "transcription factor activity" (from GO molecular function) and "lymphocyte" (from OBO-Cell). The chado cv module is based on the GO Database schema, described here[14]. Terms are stored in the cvterm table, and relationships between terms are stored in the cvterm_relationship table. This table follows an analogous structure to the feature_relationship table, in that it has columns subject_id, object_id and type_id. Here, all three of these foreign keys refer to rows in the cvterm table.

A detailed treatment of relationship types in biological ontologies can be found here[13]. Of particular interest to Chado is the is_a relation, which specifies a sub- typing relationship between two terms or classes. Recall that tables in the sequence module frequently (such as the feature table) defined a type_id foreign key column to indicate the specific type or class of entity for each row in that table. The combination of the type_id column and the is_a relationship gives Chado a data sub- classing system, beyond what is possible with traditional SQL database semantics.

This is discussed further in a later section The collection of cvterms and cvterm_relationships can be considered to constitute vertices and edges in a graph. This graph is typically acyclic (a DAG), though it is not guaranteed to be as certain relationship types are allowed to form cycles.

3.1.1 Transitive Closure

Rules

The cvtermpath is for calculating the reflexive transitive closure of a relationship, and any derived relationships

Normal (direct) relationships are stored in the cvterm_relationship table. A entry in this table represents a cvterm_relationship S over some relation R.

```
S = Subj R Obj
```

For example:

`S = "cardioblast" develops_from "mesodermal cell"`

The relation `is_a` represents a special kind of relation - subsumption, or inheritance.

If X `is_a` Y, then it follows that all of Y's cvterm_relationship statements are inherited by X

```
[Rule 1]
If   X is_a Y
and  Y R Z
then X R(inh) Z
\begin{verbatim}

For example

\begin{verbatim}
     "cilium axoneme"  is_a    "axoneme"
     "axoneme"         part_of "cell projection"
THEREFORE:
     "cilium axoneme"  part_of(inh) "cell projection"
```

Here we use T(inh) to represent an inherited relationship.

Populating cvterm_path

The cvtermpath table stores the reflexive transitive closure of a relationship, taking into account subsumption/inheritance. The number of intermediate relationships is represented in the 'distance' column of the table.

Here we use T(path) to represent the 'path' or closure of a relationship. Every T(path) is stored in cvtermpath. We use the same cvterm for T, the fact that it is a path is implicit.

We use these rules:

Reflexive relationships:

for all relations T, X T(path) X

In this case the distance=0

Direct relationships:

these are also included in the cvtermpath table, distance=1

If X T Y Then X T(path) Y

Transitive relationships:

these have distance ¿ 1; these also make use of inheritance rule, [Rule1], which gives us T(inh)

If X T(inh) Y and Y T(path) Z Then X T(path) Z

Note that this rule is recursive.

These rules should be used for populating cvtermpath. Attempting to calculate a more general closure where all relations are treated equally or ignored will produce combinatorial explosions over certain ontologies (eg flybase anatomy ontology)

What does this mean in practice?

For a typical database, which may only have relations `is_a` , part_of and develops_from, we will end up with 3 sets of paths.

The `is_a` closure, `is_a` (path) will include paths over cvterm_relationships that look like this:

```
a is_a b is_a c is_a d is_a e

The "part_of" closure, part_of(path) will include paths over
cvterm_relationships that look like this:

a is_a b part_of c part_of d is_a e part_of f

The "develops_from" closure, develops_from(path) will include paths over
cvterm_relationships that look like this:

a develops_from b develops_from c is_a d is_a e develops_from f
```

It may be tempting to mix different non `is_a` relationships in the same path, but this should NEVER be done - there will be an unacceptable combinatorial explosion in many cases. Besides, there is no use for such a cvtermpath; it is meaningless.

Note that for amigolike query behaviour, it is necessary only to query cvtermpath ignoring cvtermpath.type_id (these are obtained by querying cvterm_relationship)

3.2 Table Definitions

cv

A controlled vocabulary or ontology. A cv is composed of cvterms (aka terms, classes, types, universals - relations and properties are also stored in cvterm)) and the relationships between them

Table 3.1: cv

Column	Datatype	Description
cv_id	integer	
name	varchar	The name of the ontology. This corresponds to the obo-format -namespace-. cv names uniquely identify the cv. In obo file format, the cv.name is known as the namespace
definition	text	A text description of the criteria for membership of this ontology

cvterm

A term, class, universal or type within an ontology or controlled vocabulary. This table is also used for relations and properties. cvterms constitute nodes in the graph defined by the collection of cvterms and cvterm_relationships

Table 3.2: cvterm

Column	Datatype	Description
cvterm_id	integer	
cv_id	integer	The cv/ontology/namespace to which this cvterm belongs
name	varchar	A concise human-readable name or label for the cvterm. uniquely identifies a cvterm within a cv
definition	text	A human-readable text definition
dbxref_id	integer	Primary identifier dbxref - The unique global OBO identifier for this cvterm. Note that a cvterm may have multiple secondary dbxrefs - see also table: cvterm_dbxref
is_obsolete	integer	Boolean 0=false,1=true; see GO documentation for details of obsoletion. note that two terms with different primary dbxrefs may exist if one is obsolete
is_relationshiptype	integer	Boolean 0=false,1=true relations or relationship types (also known as Typedefs in OBO format, or as properties or slots) form a cv/ontology in themselves. We use this flag to indicate whether this cvterm is an actual term/class/universal or a relation. Relations may be drawn from the OBO Relations ontology, but are not exclusively drawn from there

cvterm_relationship

a name can mean different things in different contexts; for example "chromosome" in SO and GO. A name should be unique within an ontology/cv. A name may exist twice in a cv, in both obsolete and non-obsolete forms - these will be for different cvterms with different OBO identifiers; so GO documentation for more details on obsoletion. Note that occasionally multiple obsolete terms with the same name will exist in the same cv. If this is a possibility for the ontology under consideration (eg GO) then the ID should be appended to the name to ensure uniqueness

Table 3.3: cvterm_relationship

Column	Datatype	Description
cvterm_relationship_id	integer	
type_id	integer	The nature of the relationship between subject and object. Note that relations are also housed in the cvterm table, typically from the OBO relationship ontology, although other relationship types are allowed
subject_id	integer	the subject of the subj-predicate-obj sentence. The cvterm_relationship is about the subject. In a graph, this typically corresponds to the child node
object_id	integer	the object of the subj-predicate-obj sentence. The cvterm_relationship refers to the object. In a graph, this typically corresponds to the parent node

cvtermpath

The reflexive transitive closure of the cvterm_relationship relation. For a full discussion, see the file populating-cvtermpath.txt in this directory

Table 3.4: cvtermpath

Column	Datatype	Description
cvtermpath_id	integer	
type_id	integer	The relationship type that this is a closure over. If null, then this is a closure over ALL relationship types. If non-null, then this references a relationship cvterm - note that the closure will apply to both this relationship AND the OBO_REL:is_a (subclass) relationship
subject_id	integer	
object_id	integer	
cv_id	integer	Closures will mostly be within one cv. If the closure of a relationship traverses a cv, then this refers to the cv of the object_id cvterm
pathdistance	integer	The number of steps required to get from the subject cvterm to the object cvterm, counting from zero (reflexive relationship)

cvtermsynonym

A cvterm actually represents a distinct class or concept. A concept can be refered to by different phrases or names. In addition to the primary name (cvterm.name) there can be a number of alternative aliases or synonyms. For example, -T cell- as a synonym for -T lymphocyte-

Table 3.5: cvtermsynonym

Column	Datatype	Description
cvtermsynonym_id	integer	
cvterm_id	integer	
synonym	varchar	
type_id	integer	A synonym can be exact, narrow or borader than

cvterm_dbxref

In addition to the primary identifier (cvterm.dbxref_id) a cvterm can have zero or more secondary identifiers/dbxrefs, which may refer to records in external databases. The exact semantics of cvterm_dbxref are not fixed. For example: the dbxref could be a pubmed ID that is pertinent to the cvterm, or it could be an equivalent or similar term in another ontology. For example, GO cvterms are typically linked to InterPro IDs, even though the nature of the relationship between them is largely one of statistical association. The dbxref may be have data records attached in the same database instance, or it could be a "hanging" dbxref pointing to some external database. NOTE: If the desired objective is to link two cvterms together, and the nature of the relation is known and holds for all instances of the subject cvterm then consider instead using cvterm_relationship together with a well-defined relation.

Table 3.6: cvterm_dbxref

Column	Datatype	Description
cvterm_dbxref_id	integer	
cvterm_id	integer	
dbxref_id	integer	
is_for_definition	integer	A cvterm.definition should be supported by one or more references. If this column is true, the dbxref is not for a term in an external db - it is a dbxref for provenance information for the definition

cvtermprop

Additional extensible properties can be attached to a cvterm using this table. Corresponds to -AnnotationProperty- in W3C OWL format

Table 3.7: cvtermprop

Column	Datatype	Description
cvtermprop_id	integer	
cvterm_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm
value	text	The value of the property, represented as text. Numeric values are converted to their text representation
rank	integer	Property-Value ordering. Any cvterm can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

dbxrefprop

Metadata about a dbxref. Note that this is not defined in the dbxref module, as it depends on the cvterm table. This table has a structure analagous to cvtermprop

Table 3.8: dbxrefprop

Column	Datatype	Description
dbxrefprop_id	integer	
dbxref_id	integer	
type_id	integer	
value	text	
rank	integer	

organism

The organismal taxonomic classification. Note that phylogenies are represented using the phylogeny module, and taxonomies can be represented using the cvterm module or the phylogeny module

Table 3.9: organism

Column	Datatype	Description
organism_id	integer	
abbreviation	varchar	
genus	varchar	
species	varchar	A type of organism is always uniquely identified by genus+species. When mapping from the NCBI taxonomy names.dmp file, the unique-name column must be used where it is present, as the name column is not always unique (eg environmental samples). If a particular strain or subspecies is to be represented, this is appended onto the species name. Follows standard NCBI taxonomy pattern
common_name	varchar	
comment	text	

organism_dbxref

Table 3.10: organism_dbxref

Column	Datatype	Description
organism_dbxref_id	integer	
organism_id	integer	
dbxref_id	integer	

organismprop

tag-value properties - follows standard chado model

Table 3.11: organismprop

Column	Datatype	Description
organismprop_id	integer	
organism_id	integer	
type_id	integer	
value	text	
rank	integer	

Chapter 4

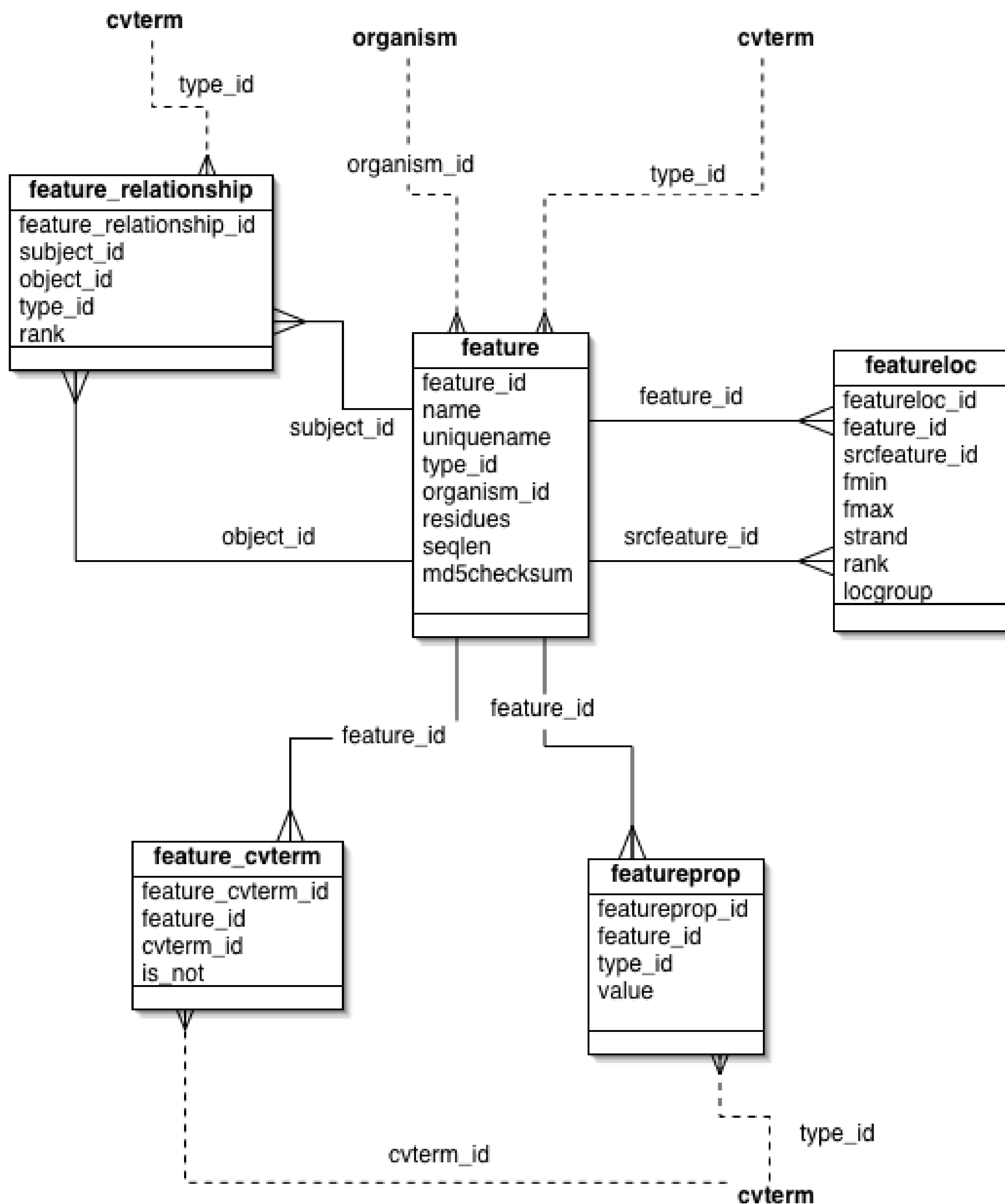
The Sequence Module: Features

4.1 The role of features in Chado

The central module in Chado is the sequence module. The fundamental table within this module is the `feature` table, for describing biological sequence features. Chado defines a feature to be a region of a biological polymer (typically a DNA, RNA, or a polypeptide molecule) or an aggregate of regions on this polymer. As the term is used here, region can be the entire extent of the molecule, or a junction between two bases. Features can be typed according to a classification scheme[6], they can be localized relative to other features, and they can form part-whole and other relationships with other features.

There are many different types of features. Examples include gene, exon, transcript, regulatory region, chromosome, sequence variation, polypeptide, protein domain and cross-genome match regions. Chado does not have a different table for each kind of feature; all features are stored in the feature table. Types of feature are differentiated using a type_id column, which is a foreign key to the cvterm table in the cv (ontology) module, described later. This allows us to type features according to the Sequence Ontology. The use of ontologies to type tables gives Chado a subtyping mechanism, which is absent from the standard relational model. For example, SO tells us that mRNA and snRNA are different kinds of transcript. This is discussed in more in the next section. For the purposes of discussion in this document, it can be assumed that any reference to genes, exons, polypeptides, SNPs, chromosomes, transcripts and various kinds of RNAs and so on refers to features of that sequence ontology type.

The Chado feature table has a text-valued column named residues for storing the sequence of the feature. The value of this column is string of IUPAC[REF] symbols corresponding to the sequence of biochemical residues encoded by the feature. This column is optional, because the sequence of the feature may not be known. Even if the sequence of a feature is known, it may not be desirable to store it in the feature table, as it may be possible to infer the sequence from the sequence of other features in the database. For example, exon sequences are generally not stored, as these can trivially be inferred from the sequence of the genomic feature on which the exon is located. In contrast, mRNA and other processed transcript sequences are stored as it is less trivial and more computationally expensive to dynamically splice together the mRNA sequence.



cvterm
type_id
organism
organism_id
cvterm
type_id
feature_relationship
feature_relationship_id
subject_id
object_id
type_id
rank
subject_id
object_id
feature
feature_id
name
uniquename
type_id
organism_id
residues
seqlen
md5checksum
feature_id
srcfeature_id
featureloc
featureloc_id
feature_id
srcfeature_id
fmin
fmax
strand
rank
locgroup
feature_id
feature_id
feature_cvterm
feature_cvterm_id
feature_id
cvterm_id
is_not
featureprop
featureprop_id
feature_id
type_id
value
type_id
cvterm_id
cvterm

It is important to realize that the existence of a row in the feature table does not necessarily imply that the feature has been characterized as a result of genome annotation. It is possible to have features of SO type gene for genes that have only been characterized through genetic studies [REF], and for which neither sequence nor sequence location is known. This is in contrast to other feature schemas (such as GFF) in which it is not possible to represent features without representing a location in sequence coordinates. This design decision is crucial for the use of Chado as a database for integrating information about the same entity from multiple perspectives.

Because the sequence is stored as a column in the feature table rather than as an independent table, sequences cannot exist in the absence of a row in the feature table; sequences are dependent upon features. This is in contrast with almost all other genomics schemas that allow independent treatment of sequences and features. This design decision follows for both philosophical and pragmatic reasons. The feature table also contains columns seqlen and md5checksum, for storing the length of the sequence and the 32-character checksum computed using the MD5 [RL Rivest. RFC 1321: The md5 message-digest algorithm. Technical report, Internet Activities Board, April 1992.] algorithm. The length and checksum can be stored even when the residues column is null valued. The checksum is useful for checking if two or more features share the same sequence, without comparing the entire sequence string.

The existence of these columns means that this table is no longer in third normal form (3NF)[REF], which is usually a desirable formal property of relational database. On balance, the utility of these columns outweighs the disadvantages of violating 3NF [updates]. In practical terms, it means that the values of the residues, seqlen and md5checksum columns are interdependent and cannot be updated independently of one another.

The feature table has a Boolean valued column, is_analysis, indicating whether this is an annotation or a computed feature from a computational analysis. Annotations are features that are generated or blessed by a human curator, or in some cases by an integrated genome pipeline[7-9] capable of synthesising gene models and other annotations from in-silico analyses. They constitute the definitive version of a particular feature, in contrast to the features generated by gene prediction programs and sequence similarity searches such as BLAST.

The feature table has a dbxref_id column that refers to a global, stable public identifier for the feature. This column is optional, because not all classes of features have such identifiers for example, features resulting from gene predictions and blast HSP features may be less stable and thus lack public identifiers. It is recommended that most annotated features have dbxref_ids. The organism_id column refers to a row in the organism table (defined in the organism module). This column is mandatoryall features derive from a single organism.

The name and uniquename columns allow features to be labelled. The name column is optional, but it is recommended that all annotated features (as opposed to those that arise from purely computational methods) have names. The name should be a simple, concise, human-friendly display label (such as a gene or gene product symbol, as defined by the nomenclature rules of governing the organism). User interface software (such as GBrowse[10] and Apollo[11]) can use the name column for labelling feature glyphs in user displays. Uniqueness of name within any particular organism or genome project is a desirable characteristic, but is not enforced in the schema, since there are occasions where name clashes are unavoidable. In contrast, the uniquename column is required, and guaranteed to be unique when taken in combination with organism_id and type_id this is

enforced by a constraint in the relational schema. The uniquename may be human-friendly (for example, it can be the same as the name); however, it is not guaranteed to be so, and in general should not be displayed to the end user. Its use is mainly as an alternate unique key on the table .

The uniquename normally conforms to some naming rule these rules may vary across chado instances, but they should all guarantee the uniqueness of the uniquename, organism_id, type_id triple.

Feature synonyms

In addition to having a name or symbol, it is common for features such as genes to have multiple synonyms or aliases. These synonyms may exist due to different publications referring to the same gene with different symbols, or because one gene was once believed to be two or more separate genes. A common curation operation on genes[REF] is splitting and merging, which results in the creation of synonyms.

This is modelled in Chado with a synonym table and a feature_synonym linking table; thus multiple features can potentially share the same, and a single feature can be have multiple synonyms. Use of a synonym in the literature is indicated with a pub_id foreign key referencing the pub table (described later in the section on publications module), indicating historical provenance for the use of a synonym.

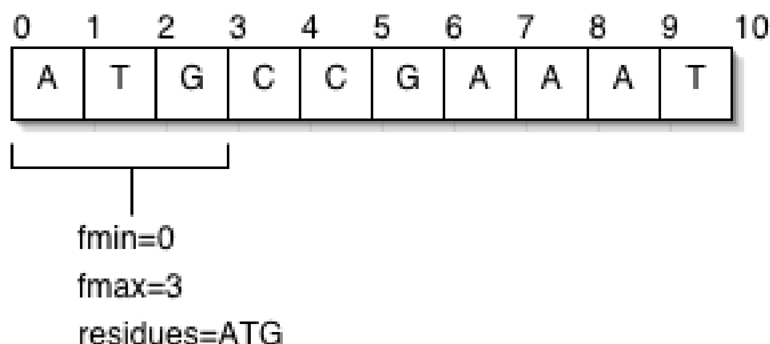
Feature locations

Features can potentially be localized using a sequence coordinate system. A relative localization model is used, so all feature localizations must be relative to another feature. Some features such as those of type *chromosome* are not localized in sequence coordinates. Locations are stored in the featureloc table, also part of the sequence module. Other non-sequence oriented kinds of localization (such as physical localization from in situ experiments, or genetic localizations from linkage studies) are modelled outside the sequence module (for example, in the expression or map module).

A feature can have zero or more featurelocs, although it will typically have either one (for localized features for which the location is known) or zero (for unlocalized features such as chromosomes, or for features for which the location is not yet known, such as a gene discovered using classical genetics techniques). Features with multiple featurelocs will be explained later.

A featureloc is an interval in interbase sequence coordinates (see figure), bounded by the fmin and fmax columns, each representing the lower and upper linear position of the boundary between bases or base pairs, with directionality indicated by the strand column. Interbase coordinates were chosen over the more commonly used base-oriented coordinate system because they are more naturally amenable to the standard arithmetic operations that are typically performed upon sequence coordinates. This leads to cleaner and more efficient database coding logic that is arguably less prone to errors. Of course, interbase coordinates are typically transformed into the more common base-oriented system used by BLAST reports and so forth prior to presentation to the end-user.

The relational schema includes a constraint which ensures that fmin ¡= fmax is always true any attempt to set the database in a state which violates this will flag an error .

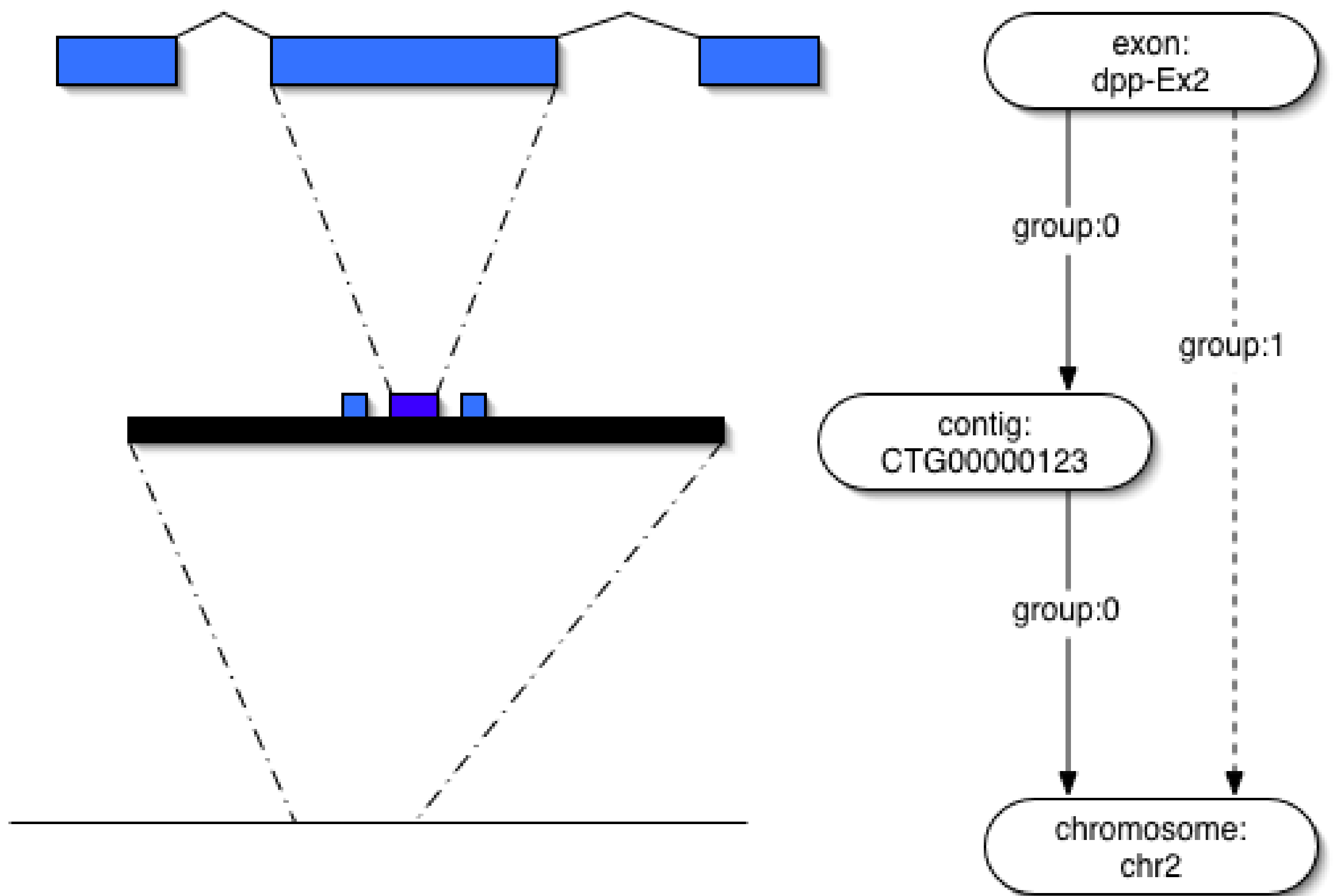


As mentioned previously, a featureloc must be localized relative to another feature, indicated using the srcfeature_id foreign key column, referencing the feature table. There is nothing in the schema prohibiting localization chains; for example, locating an exon relative to a contig that is itself localized relative to a chromosome (see figure). The majority of Chado database instances will not require this flexibility; features are typically located relative to chromosomes or chromosomes arms. Nevertheless, the ability to store such localization networks or location graphs can be useful for unfinished genomes or parts of genomes such as heterochromatin [REF], in which it is desirable to locate features relative to stable contigs or scaffolds, which are themselves localized in an unstable assembly to chromosomes or chromosome arms. Localization chains do not necessarily only span assemblies protein domains may be localized relative to polypeptide features, themselves localized to a transcript (or to the genome, as is more common). Chains may also span sequence alignments.

We will now present a short formal treatment of the properties of these hierarchies of localization using graph theory. This treatment can be ignored for the purposes of understanding the basics of the Chado schema; the end-user of the database will be entirely unaware of such technicalities. However, for the purposes of software engineering and ensuring interoperability between different Chado database instances and different applications, formal treatments such as these are an essential requirement for software specifications.

We can define a featureloc graph (LG) as being a set of vertices and edges, with each feature constituting a vertex, and each featureloc constituting an edge going from the parent feature_id vertex to the srcfeature_id vertex. The node is labelled with column values from the feature table, and the edge is labelled with column values from the featureloc table. The LG is not allowed to contain cycles it is a directed acyclic graph (DAG). This includes self-cycles - no feature may be localized relative to itself.

The roots of the LG are the features that do not have featureloc row typically chromosomes or chromosome arms, although LG roots may also be unassembled contigs, scaffolds or features for which sequence localization is not get known (such as genes discovered through classical genetics techniques). The leaves of the LG are any features that are not present as a srcfeature_id in any featurelocs row typically the bulk of features, such as genes, exons, matches and so on. The depth of a particular LG g, denoted D(g), is the maximum number of edges between any leaf- root pair. As has been previously noted, many Chados will have LGs with a uniform depth of 1. Such LGs are said to be simple and the features within them are said to be singletons. The maximum depth of all LGs in a particular database instance i is denoted LGDmax(i).


exon:
dpp-Ex2
group:0
group:1
contig:
CTG00000123
group:0
chromosome:
chr2

The schema does not constrain the maximum depth of the LG. This flexibility proves useful when applying Chado to the highly variable needs of multiple different genome projects; however, it can lead to efficiency problems when querying the database. It can also make it more difficult to write software to interoperate with the database, as the software must take into account different contingencies. We can solve this problem by collapsing the LG, in which a graph of arbitrary depth is flattened to a depth of 1, transforming or projecting featurelocs onto the root features (typically chromosomes or chromosome arms). The original featurelocs are left unaltered in the database, and additional redundant featurelocs between leaf and root features are added to the database. These new featurelocs are known as inferred featurelocs. In the schema inferred featurelocs are differentiated from direct featurelocs using the locgroup column. Direct (non-inferred) localizations are indicated by the locgroup column taking value 0, and transitive localizations are indicated by this column having value ¿0.

The terminology used above can be used to define specifications for applications intended to interoperate with the database. Feature location pairs Certain kinds of features have paired locations. These include hits and high-scoring- pairs (HSPs) coming from sequence search programs such as BLAST, and syntenic chromosomal regions. These kinds of features have two featurelocs (in contrast to the usual 1) one on the query feature and one on the subject (hit) feature. We differentiate the two featurelocs with the rank column. A rank of 0 indicates a location relative to the query (as is the default for most features), and a rank of 1 indicates a location relative to the subject (hit) feature.

For multiple alignments (e.g. CLUSTALW [REF] results), this scheme is extended to unbounded ranks [0..n], with arbitrary ordering. Alignments are stored in the residue_info column. CIGAR format[REF] is used for pairwise alignments.

Multiple featurelocs may also be required for features of type sequence_variant (SO:0000109), indicating points or extents which vary between reference and non- reference sequences. From a modelling standpoint, variants are conceptually similar to alignments; with variants we are noting a difference as opposed to a similarity. Here a rank of zero indicates the wild-type (or reference) feature and a rank of one or more indicates the variant (or non-reference) feature, with the residue_info column representing the sequence on wild-type and variant. [?figure] A featureloc is uniquely identified by the [feature_id, rank, locgroup] triple. This means that no feature can have more than one featureloc with the same rank and locgroup. In other words, rank and locgroup uniquely identify a featureloc for any particular feature.

Difference between the chado location model and other schemas

There is a crucial difference between the Chado location model and the sequence location model used in other schemas, such as GFF, GenBank, BioSQL, BioPerl, etc.

First, Chado is the only model to use the concept of rank and locgroup. Second, and perhaps more important, all these other models allow discontiguous locations (also known as split locations). These will be familiar to anyone who has inspected GenBank annotated DNA records for an organism that has introns within the transcripts; the transcript location is modelled as a sequence of non-contiguous intervals on the genome. The interval represents the location of an exon.

Although Chado allows a feature to have multiple locations, this is only with variable rank and locgroup this is enforced by a uniqueness constraint in the relational schema. We made a conscious decision to avoid discontiguous locations, because the extra degree of freedom this affords results in either redundancies or ambiguities. Redundancies arise when exons are stored in addition to a discontiguous transcript, and ambiguities arise by virtue of the fact that explicit representation of the exons may be seen as optional. Ambiguities are undesirable as it makes it harder for databases to interoperate. The omission of discontiguous locations does not restrict the expressive capacity of Chado in any way, because any discontiguous location can be modelled as a collection of features with contiguous locations. For example, a transcript with a discontiguous location can be modelled as a collection of exons with contiguous featurelocs, and a transcript with a single contiguous featureloc representing the outer boundaries defined by the outermost exons.

Extensible feature properties

The feature table has a fairly limited set of columns for recording feature data. For example, there is no anticodon column for recording the RNA triplet for the adapter in a tRNA feature (all feature types, including tRNAs, are recorded as rows in the feature table). If we were to add columns such as anticodon then the number of columns in the table would become very large and difficult to manage; most would end up being nullable (for example, anticodon does not apply to non-tRNA features). This is because different organisms, different types of feature and different projects have differing needs regarding what extra data should be attached to any one feature. How then are we to attach both biologically relevant and project specific data to features? Chado solves this by using an extensible mechanism for attaching attribute- value pairs to features via the featureprop table. The featureprop.type_id foreign key column references a property in the Sequence Feature Property Ontology (SFPO)[url], distributed as part of Chado. The value text column stores the value filler for that property. Sets or lists of values for any property can be stored in the featureprop table, differentiated by the value of the rank column. Provenance for the featureprop assignment is stored using the featureprop_pub table in the publications module, described later, allowing multiple publications to be associated with any one assignment.

Because featureprop values can be of an arbitrary size, they are modelled using a SQL TEXT type. This has some disadvantages from a query efficiency perspective.

Numeric values cannot be indexed correctly, and sorting the results of a query can only be done via a SQL casting operation, or in software outside of the database management system, either of which may result in poorer performance. This is one of several areas in Chado where performance has been traded in favour of a simpler, more abstract and generic model. Later on we will look at strategies for offsetting some of these performance penalties.

[example table]

Feature annotations

Detailed annotations, such as associations to Gene Ontology[5] (GO) terms or Cell Ontology[12] terms, can be attached to features using the feature_cvterm linking table. This allows multiple ontology terms to be associated with each feature.

Provenance data can be attached with the feature_cvtermprop and feature_cvterm_dbxref higher-order linking tables. It is up to the curation policy of each individual Chado database instance to decide which kinds of features will be linked using feature_cvterm. Some may link terms to gene features, others to the distinct gene products (processed RNAs and polypeptides) linked to the gene features (see next section)

Relationships between features

Biological features are inter-related; exons are part of transcripts, transcripts are part of genes, and polypeptides are derived from messenger RNAs. Relationships between individual features are stored in the feature_relationship table, which connects two features via the subject_id and object_id columns (foreign keys referring to the feature table) and a type_id (a foreign key referring to a relationship type in an ontology, either SO[6], or the OBO relationship ontology, OBO-REL[13]) indicating the nature of the relationship between subject and object features. The core relationships between features are part-whole (part_of) or temporal (derives_from). ”Subject” and ”Object” describes the linguistic role the two features play in a sentence describing the feature_relationship. In English, many sentences follow a subject, predicate, object word order. To say that ”exons are part_of transcripts” is the correct way to describe a typical biological relationship. To say ”transcripts are part_of exons” is either grammatically or biologically incorrect.

We use this same terminology (which comes from RDF[REF]) again in the cv module. The collection of features and feature_relationships can be considered as vertices and edges in a graph, known as the Feature Graph (FG). Some example feature graphs are shown [figure FEATURE-GRAPH]. The FG is independent of the LG in general the FG and the LG should have no edges in common if there is a featureloc connecting two features, then the addition of a feature_relationship between these same two features is redundant.

The FG is required in order to query the database for such things as alternately spliced genes, exons shared between transcripts, etc.

Although the chado schema admits any FG, certain configurations are biologically meaningless, and should not be used. The FG can be constrained by the Sequence Ontology. Standardized FG structures are required for complex applications to be interoperable - this is discussed later on.

Unlike the LG, the FG may be cyclic, although cycles in the FG are not common. The subset of the FG corresponding to certain kinds of relationship may be acyclic for example, the subset of the FG connecting parts with wholes via part_of must be acyclic.

Canonical gene models

Regulatory regions

Sequence variants

Feature example

[Diagram showing an example that puts this all together]

4.2 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 flexibility 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.

Chado uses a layered model - this is tried and tested in software engineering. Some generic software can be targeted at the lower layers and be guaranteed to work no matter what. Other more specific software needs a more tightly defined rigorous model and should be targeted at the upper layers.

We require validation software and more formal/computable descriptions of these layers and policies - for now natural language descriptions will have to suffice.

4.2.1 Chado Compliance Layers

Layer 0: Relational Schema

Level 0 conformance basically means the schema is adhered to. Obviously, this is enforced by the DBMS.

Layer 1: Ontologies

Level 1 conformance is minimal conformance to SO - all feature.types must be SO terms, and all feature_relationship.types must be SO relationship types.

Layer 2: Graph

Level 2 conformance is graph conformance to SO - all feature_relationships between a feature of type X and Y must correspond to relationship of that type in SO; for example, mRNA can be part_of gene, but mRNA can not be part_of golden_path_region. [more detailed/formal explanation to come]. In practice Level 2 conformance may be undesirable, we may need to make modifications to SO.

Orthogonal to these layers are various additional policy decisions. Some of these are more tolerant of non-conformance than others. (there is also some overlaps with levels 1/2).

4.2.2 Examples: Current implementations

I have listed how FB implements each policy choice - other chado instances feel free to add....

TIGR: Currently at level 0 conformance, though most (if not all) of the terms being used have an obvious counterpart in SO. Therefore these "TIGR Ontology" terms are used in the answers to the SO-related questions that appear below. We plan on updating our terms with SO terms very soon.

SO terms used for standard central-dogma gene model

FB: gene mRNA exon protein [other types are derivable]

TIGR: gene transcript CDS exon protein [though the strict answer is for any of these SO questions is "none" since we do not yet meet level 1 conformance]

NOTE: we should be using 'polypeptide' instead of 'protein'. For now, software should be tolerant of both these uses.

SO terms used for storing alignments

FB: match

TIGR: match

NOTE: we want to use the new more specific SO types for match_set, match_part, for hits and hsps respectively. For now, software should be tolerant of either usage.

TIGR: We've also extended the model for storing pairwise alignments to store multiple alignments. Each member of the alignment is featureloced to the 'match' feature. We've used this representation to store paralogous/orthologous gene families.

feature_relationship.types

FB: partof (for mRNA to gene and exon to mRNA) producedby (for protein to mRNA)

TIGR: part_of (gene-assembly, exon-transcript, assembly-supercontig) produced_by (protein-CDS, CDS-transcript, transcript-gene)

NOTE: this should be "part_of" and "derived_from" to conform to SO. Most read-only software should be able to safely ignore feature_relationship.type anyway. Protein should be polypeptide - see note above

NOTE: the main difference between FB and TIGR here is that TIGR introduce an intermediate CDS feature between mRNA and protein

featureloc policy

FB: all constituent parts of a central dogma gene model are located relative to the same srcfeature (the chromosome arm). No redundant locations (ie featureloc.group ¿ 0) are used

TIGR: Redundant locations are used and indicated with featureloc.group ¿ 0.

NOTE: we want to allow some flexibility with this policy. I believe that the constituent parts linked located relative to the feature should always be followed. This can be stated more formally as:

```
IF   X is linked to Y via feature\_relationship
AND  X is located relative to Z via featureloc.srcfeature\_id
THEN Y must also be located relative to Z via featureloc.srcfeature\_id
```

TIGR: We've followed this policy in adding a featureloc between the protein and genomic contig in our databases (such a featureloc does not appear in the Chado usage documents). This additional featureloc simplifies many queries, especially when looking at the genomic context of 'match' features associated with proteins.

We should also expect that the fmin/fmax boundaries of a feature be defined the the outermost boundaries of the outermost constituent part features (this rule may require refinement when we have promoters, enhancers and so on - but for now we don't).

As to what the srcfeature should be, it could be a contig, and assembly or a top-level locatable feature such as chromosome or chromosome arm. Software should be tolerant of different choices here. Whilst it is generally always best to locate relative to the topmost feature (ie the arm/chromosome), sometimes this is not possible or desirable (eg low coverage, heterochromatin).

non-central dogma gene models

FB: we store a lot of non-central dogma gene models; noncoding gene models and pseudogenes [need to fill in more details here]

TIGR: not many of these stored yet, save for a few pseudogenes and the occasional non-coding ORF

other features

FB: the FlyBase implementation includes many other feature types, including polyA_site and sequence_variant [need to fill in details]

TIGR: using 'SNP' in some databases

derivable features types

FB: derivable features (introns, UTRs, intergenic_region) are not included. Feature typing is always done to the most specific, non-derivale level. For example, we never use types "5_prime_exon", "dicistronic_gene", "coding_exon" as these are always inferrable. We always use type "gene" - the specific type of gene is inferred from the child type (mRNA, tRNA, snRNA, etc).

TIGR: derivable features are not included. currently not storing any tRNAs or snRNAs.

NOTE: whilst it is perfectly permissable to include redundant derivable features (useful for warehouse-style querying), you should not write software that expects to find these if you want the software to work on different chado db instances.

sequence_variants

FB: these are included in chado, but they are lacking full detail

TIGR: only SNPs so far. the SNPs currently being stored are computed from pairwise alignments of sequences already loaded into Chado, so each SNP feature is featureloc'ed to the appropriate place on each of the two sequences (rather than having one of the featurelocs "dangling", as indicated in some of the Chado usage documents.) featureloc.residue_info is used to redundantly store the base referenced in each of the two sequences.

NOTE: variation features should specify the edit that makes one feature (such as the reference/wild-type) from another (the variant/mutant/non-reference). There were perhaps 2 proposals for this [more details required...]

4.3 Table definitions

feature

A feature is a biological sequence or a section of a biological sequence, or a collection of such sections. Examples include genes, exons, transcripts, regulatory regions, polypeptides, protein domains, chromosome sequences, sequence variations, cross-genome match regions such as hits and HSPs and so on; see the Sequence Ontology for more

Table 4.1: feature

Column	Datatype	Description
feature_id	integer	
dbxref_id	integer	An optional primary public stable identifier for this feature. Secondary identifiers and external dbxrefs go in table:feature_dbxref
organism_id	integer	The organism to which this feature belongs. This column is mandatory
name	varchar	The optional human-readable common name for a feature, for display purposes
uniquename	text	The unique name for a feature; may not be necessarily be particularly human-readable, although this is prefered. This name must be unique for this type of feature within this organism
residues	text	A sequence of alphabetic characters representing biological residues (nucleic acids, amino acids). This column does not need to be manifested for all features; it is optional for features such as exons where the residues can be derived from the featureloc. It is recommended that the value for this column be manifested for features which may may non-contiguous sublocations (eg transcripts), since derivation at query time is non-trivial. For expressed sequence, the DNA sequence should be used rather than the RNA sequence
seqlen	integer	The length of the residue feature. See column:residues. This column is partially redundant with the residues column, and also with featureloc. This column is required because the location may be unknown and the residue sequence may not be manifested, yet it may be desirable to store and query the length of the feature. The seqlen should always be manifested where the length of the sequence is known
md5checksum	char	The 32-character checksum of the sequence, calculated using the MD5 algorithm. This is practically guaranteed to be unique for any feature. This column thus acts as a unique identifier on the mathematical sequence
type_id	integer	A required reference to a table:cvterm giving the feature type. This will typically be a Sequence Ontology identifier. This column is thus used to subclass the feature table
is_analysis	boolean	Boolean indicating whether this feature is annotated or the result of an automated analysis. Analysis results also use the companalysis module. Note that the dividing line between analysis/annotation may be fuzzy, this should be determined on a per-project basis in a consistent manner. One requirement is that there should only be one non-analysis version of each wild-type gene feature in a genome, whereas the same gene feature can be predicted multiple times in different analyses
is_obsolete	boolean	Boolean indicating whether this feature has been obsoleted. Some chado instances may choose to simply remove the feature altogether, others may choose to keep an obsolete row in the table

featureloc

The location of a feature relative to another feature. IMPORTANT: INTERBASE COORDINATES ARE USED.(This is vital as it allows us to represent zero-length features eg splice sites, insertion points without an awkward fuzzy system). Features typically have exactly ONE location, but this need not be the case. Some features may not be localized (eg a gene that has been characterized genetically but no sequence/molecular info is available). NOTE ON MULTIPLE LOCATIONS: Each feature can have 0 or more locations. Multiple locations do NOT indicate non-contiguous locations (if a feature such as a transcript has a non-contiguous location, then the subfeatures such as exons should always be manifested). Instead, multiple featurelocs for a feature designate alternate locations or grouped locations; for instance, a feature designating a blast hit or hsp will have two locations, one on the query feature, one on the subject feature. features representing sequence variation could have alternate locations instantiated on a feature on the mutant strain. the column:rank is used to differentiate these different locations. Reflexive locations should never be stored - this is for -proper- (ie non-self) locations only; i.e. nothing should be located relative to itself

Table 4.2: featureloc

Column	Datatype	Description
featureloc_id	integer	
feature_id	integer	The feature that is being located. Any feature can have zero or more featurelocs
srcfeature_id	integer	The source feature which this location is relative to. Every location is relative to another feature (however, this column is nullable, because the srcfeature may not be known). All locations are -proper- that is, nothing should be located relative to itself. No cycles are allowed in the featureloc graph
fmin	integer	The leftmost/minimal boundary in the linear range represented by the featureloc. Sometimes (eg in bioperl) this is called -start- although this is confusing because it does not necessarily represent the 5-prime coordinate. IMPORTANT: This is space-based (INTERBASE) coordinates, counting from zero. To convert this to the leftmost position in a base-oriented system (eg GFF, bioperl), add 1 to fmin
is_fmin_partial	boolean	This is typically false, but may be true if the value for column:fmin is inaccurate or the leftmost part of the range is unknown/unbounded
fmax	integer	The rightmost/maximal boundary in the linear range represented by the featureloc. Sometimes (eg in bioperl) this is called -end- although this is confusing because it does not necessarily represent the 3-prime coordinate. IMPORTANT: This is space-based (INTERBASE) coordinates, counting from zero. No conversion is required to go from fmax to the rightmost coordinate in a base-oriented system that counts from 1 (eg GFF, bioperl)
is_fmax_partial	boolean	This is typically false, but may be true if the value for column:fmax is inaccurate or the rightmost part of the range is unknown/unbounded
strand	integer	The orientation/directionality of the location. Should be 0,-1 or +1
phase	integer	phase of translation wrt srcfeature_id. Values are 0,1,2. It may not be possible to manifest this column for some features such as exons, because the phase is dependant on the spliceform (the same exon can appear in multiple spliceforms). This column is mostly useful for predicted exons and CDSs
residue_info	text	Alternative residues, when these differ from feature.residues. for instance, a SNP feature located on a wild and mutant protein would have different alresidues. for alignment/similarity features, the altresidues is used to represent the alignment string (CIGAR format). Note on variation features; even if we dont want to instantiate a mutant chromosome/contig feature, we can still represent a SNP etc with 2 locations, one (rank 0) on the genome, the other (rank 1) would have most fields null, except for altresidues
locgroup	integer	This is used to manifest redundant, derivable extra locations for a feature. The default locgroup=0 is used for the DIRECT location of a feature. !!

featureloc_pub

COMMENT ON INDEX featureloc_c1 IS 'locgroup and rank serve to uniquely

Table 4.3: featureloc_pub

Column	Datatype	Description
featureloc_pub_id	integer	
featureloc_id	integer	
pub_id	integer	

feature_pub

Provenance. Linking table between features and publications that mention them

Table 4.4: feature_pub

Column	Datatype	Description
feature_pub_id	integer	
feature_id	integer	
pub_id	integer	

featureprop

A feature can have any number of slot-value property tags attached to it. This is an alternative to hardcoding a list of columns in the relational schema, and is completely extensible

Table 4.5: featureprop

Column	Datatype	Description
featureprop_id	integer	
feature_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. Certain property types will only apply to certain feature types (e.g. the anticodon property will only apply to tRNA features) ; the types here come from the sequence feature property ontology
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

featureprop_pub

for any one feature, multivalued property-value pairs must be differentiated by rank

Table 4.6: featureprop_pub

Column	Datatype	Description
featureprop_pub_id	integer	
featureprop_id	integer	
pub_id	integer	

feature_dbxref

links a feature to dbxrefs. This is for secondary identifiers; primary identifiers should use feature.dbxref_id

Table 4.7: feature_dbxref

Column	Datatype	Description
feature_dbxref_id	integer	
feature_id	integer	
dbxref_id	integer	
is_current	boolean	the is_current boolean indicates whether the linked dbxref is the current -official- dbxref for the linked feature

feature_relationship

features can be arranged in graphs, eg exon part_of transcript part_of gene; translation madeby transcript if type is thought of as a verb, each arc makes a statement [SUBJECT VERB OBJECT] object can also be thought of as parent (containing feature), and subject as child (contained feature or subfeature) – we include the relationship rank/order, because even though most of the time we can order things implicitly by sequence coordinates, we cant always do this - eg transpliced genes. its also useful for quickly getting implicit introns

Table 4.8: feature_relationship

Column	Datatype	Description
feature_relationship_id	integer	
subject_id	integer	the subject of the subj-predicate-obj sentence. This is typically the subfeature
object_id	integer	the object of the subj-predicate-obj sentence. This is typically the container feature
type_id	integer	relationship type between subject and object. This is a cvterm, typically from the OBO relationship ontology, although other relationship types are allowed. The most common relationship type is OBO_REL:part_of. Valid relationship types are constrained by the Sequence Ontology
value	text	Additional notes/comments
rank	integer	The ordering of subject features with respect to the object feature may be important (for example, exon ordering on a transcript - not always derivable if you take trans spliced genes into consideration). rank is used to order these; starts from zero

feature_relationship_pub

Provenance. Attach optional evidence to a feature_relationship in the form of a publication

Table 4.9: feature_relationship_pub

Column	Datatype	Description
feature_relationship_pub_id	integer	
feature_relationship_id	integer	
pub_id	integer	

feature_relationshipprop

Extensible properties for feature_relationships. Analagous structure to featureprop. This table is largely optional and not used with a high frequency. Typical scenarios may be if one wishes to attach additional data to a feature_relationship - for example to say that the feature_relationship is only true in certain contexts

Table 4.10: feature_relationshipprop

Column	Datatype	Description
feature_relationshipprop_id	integer	
feature_relationship_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. Currently there is no standard ontology for feature_relationship property types
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature_relationship can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

feature_relationshipprop_pub

Provenance for feature_relationshipprop

Table 4.11: feature_relationshipprop_pub

Column	Datatype	Description
feature_relationshipprop_pub_id	integer	
feature_relationshipprop_id	integer	
pub_id	integer	

feature_cvterm

Associate a term from a cv with a feature, for example, GO annotation

Table 4.12: feature_cvterm

Column	Datatype	Description
feature_cvterm_id	integer	
feature_id	integer	
cvterm_id	integer	
pub_id	integer	Provenance for the annotation. Each annotation should have a single primary publication (which may be of the appropriate type for computational analyses) where more details can be found. Additional provenance dbxrefs can be attached using feature_cvterm_dbxref
is_not	boolean	if this is set to true, then this annotation is interpreted as a NEGATIVE annotation - ie the feature does NOT have the specified function, process, component, part, etc. See GO docs for more details

feature_cvtermprop

Extensible properties for feature to cvterm associations. Examples: GO evidence codes; qualifiers; metadata such as the date on which the entry was curated and the source of the association. See the featureprop table for meanings of type_id, value and rank

Table 4.13: feature_cvtermprop

Column	Datatype	Description
feature_cvtermprop_id	integer	
feature_cvterm_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. cvterms may come from the OBO evidence code cv
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature_cvterm can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

feature_cvterm_dbxref

Additional dbxrefs for an association. Rows in the feature_cvterm table may be backed up by dbxrefs. For example, a feature_cvterm association that was inferred via a protein-protein interaction may be backed by by refering to the dbxref for the alternate protein. Corresponds to the WITH column in a GO gene association file (but can also be used for other analagous associations). See http://www.geneontology.org/doc/GO.annotation.shtml#file for more details

Table 4.14: feature_cvterm_dbxref

Column	Datatype	Description
feature_cvterm_dbxref_id	integer	
feature_cvterm_id	integer	
dbxref_id	integer	

feature_cvterm_pub

Secondary pubs for an association. Each feature_cvterm association is supported by a single primary publication. Additional secondary pubs can be added using this linking table (in a GO gene association file, these corresponding to any IDs after the pipe symbol in the publications column

Table 4.15: feature_cvterm_pub

Column	Datatype	Description
feature_cvterm_pub_id	integer	
feature_cvterm_id	integer	
pub_id	integer	

synonym

A synonym for a feature. One feature can have multiple synonyms, and the same synonym can apply to multiple features

Table 4.16: synonym

Column	Datatype	Description
synonym_id	integer	
name	varchar	The synonym itself. Should be human-readable machine-searchable ascii text
type_id	integer	types would be symbol and fullname for now
synonym_sgml	varchar	The fully specified synonym, with any non-ascii characters encoded in SGML

feature_synonym

Linking table between feature and synonym

Table 4.17: feature_synonym

Column	Datatype	Description
feature_synonym_id	integer	
synonym_id	integer	
feature_id	integer	
pub_id	integer	the pub_id link is for relating the usage of a given synonym to the publication in which it was used
is_current	boolean	the is_current boolean indicates whether the linked synonym is the current -official- symbol for the linked feature
is_internal	boolean	typically a synonym exists so that somebody querying the db with an obsolete name can find the object theyre looking for (under its current name. If the synonym has been used publicly & deliberately (eg in a paper), it my also be listed in reports as a synonym. If the synonym was not used deliberately (eg, there was a typo which went public), then the is_internal boolean may be set to -true- so that it is known that the synonym is -internal- and should be queryable but should not be listed in reports as a valid synonym

feature

A feature is a biological sequence or a section of a biological sequence, or a collection of such sections. Examples include genes, exons, transcripts, regulatory regions, polypeptides, protein domains, chromosome sequences, sequence variations, cross-genome match regions such as hits and HSPs and so on; see the Sequence Ontology for more

Table 4.18: feature

Column	Datatype	Description
feature_id	integer	
dbxref_id	integer	An optional primary public stable identifier for this feature. Secondary identifiers and external dbxrefs go in table:feature_dbxref
organism_id	integer	The organism to which this feature belongs. This column is mandatory
name	varchar	The optional human-readable common name for a feature, for display purposes
uniquename	text	The unique name for a feature; may not be necessarily be particularly human-readable, although this is prefered. This name must be unique for this type of feature within this organism
residues	text	A sequence of alphabetic characters representing biological residues (nucleic acids, amino acids). This column does not need to be manifested for all features; it is optional for features such as exons where the residues can be derived from the featureloc. It is recommended that the value for this column be manifested for features which may may non-contiguous sublocations (eg transcripts), since derivation at query time is non-trivial. For expressed sequence, the DNA sequence should be used rather than the RNA sequence
seqlen	integer	The length of the residue feature. See column:residues. This column is partially redundant with the residues column, and also with featureloc. This column is required because the location may be unknown and the residue sequence may not be manifested, yet it may be desirable to store and query the length of the feature. The seqlen should always be manifested where the length of the sequence is known
md5checksum	char	The 32-character checksum of the sequence, calculated using the MD5 algorithm. This is practically guaranteed to be unique for any feature. This column thus acts as a unique identifier on the mathematical sequence
type_id	integer	A required reference to a table:cvterm giving the feature type. This will typically be a Sequence Ontology identifier. This column is thus used to subclass the feature table
is_analysis	boolean	Boolean indicating whether this feature is annotated or the result of an automated analysis. Analysis results also use the companalysis module. Note that the dividing line between analysis/annotation may be fuzzy, this should be determined on a per-project basis in a consistent manner. One requirement is that there should only be one non-analysis version of each wild-type gene feature in a genome, whereas the same gene feature can be predicted multiple times in different analyses
is_obsolete	boolean	Boolean indicating whether this feature has been obsoleted. Some chado instances may choose to simply remove the feature altogether, others may choose to keep an obsolete row in the table

featureloc

The location of a feature relative to another feature. IMPORTANT: INTERBASE COORDINATES ARE USED.(This is vital as it allows us to represent zero-length features eg splice sites, insertion points without an awkward fuzzy system). Features typically have exactly ONE location, but this need not be the case. Some features may not be localized (eg a gene that has been characterized genetically but no sequence/molecular info is available). NOTE ON MULTIPLE LOCATIONS: Each feature can have 0 or more locations. Multiple locations do NOT indicate non-contiguous locations (if a feature such as a transcript has a non-contiguous location, then the subfeatures such as exons should always be manifested). Instead, multiple featurelocs for a feature designate alternate locations or grouped locations; for instance, a feature designating a blast hit or hsp will have two locations, one on the query feature, one on the subject feature. features representing sequence variation could have alternate locations instantiated on a feature on the mutant strain. the column:rank is used to differentiate these different locations. Reflexive locations should never be stored - this is for -proper- (ie non-self) locations only; i.e. nothing should be located relative to itself

Table 4.19: featureloc

Column	Datatype	Description
featureloc_id	integer	
feature_id	integer	The feature that is being located. Any feature can have zero or more featurelocs
srcfeature_id	integer	The source feature which this location is relative to. Every location is relative to another feature (however, this column is nullable, because the srcfeature may not be known). All locations are -proper- that is, nothing should be located relative to itself. No cycles are allowed in the featureloc graph
fmin	integer	The leftmost/minimal boundary in the linear range represented by the featureloc. Sometimes (eg in bioperl) this is called -start- although this is confusing because it does not necessarily represent the 5-prime coordinate. IMPORTANT: This is space-based (INTERBASE) coordinates, counting from zero. To convert this to the leftmost position in a base-oriented system (eg GFF, bioperl), add 1 to fmin
is_fmin_partial	boolean	This is typically false, but may be true if the value for column:fmin is inaccurate or the leftmost part of the range is unknown/unbounded
fmax	integer	The rightmost/maximal boundary in the linear range represented by the featureloc. Sometimes (eg in bioperl) this is called -end- although this is confusing because it does not necessarily represent the 3-prime coordinate. IMPORTANT: This is space-based (INTERBASE) coordinates, counting from zero. No conversion is required to go from fmax to the rightmost coordinate in a base-oriented system that counts from 1 (eg GFF, bioperl)
is_fmax_partial	boolean	This is typically false, but may be true if the value for column:fmax is inaccurate or the rightmost part of the range is unknown/unbounded
strand	integer	The orientation/directionality of the location. Should be 0,-1 or +1
phase	integer	phase of translation wrt srcfeature_id. Values are 0,1,2. It may not be possible to manifest this column for some features such as exons, because the phase is dependant on the spliceform (the same exon can appear in multiple spliceforms). This column is mostly useful for predicted exons and CDSs
residue_info	text	Alternative residues, when these differ from feature.residues. for instance, a SNP feature located on a wild and mutant protein would have different alresidues. for alignment/similarity features, the altresidues is used to represent the alignment string (CIGAR format). Note on variation features; even if we dont want to instantiate a mutant chromosome/contig feature, we can still represent a SNP etc with 2 locations, one (rank 0) on the genome, the other (rank 1) would have most fields null, except for altresidues
locgroup	integer	This is used to manifest redundant, derivable extra locations for a feature. The default locgroup=0 is used for the DIRECT location of a feature. !!

featureloc_pub

COMMENT ON INDEX featureloc_c1 IS 'locgroup and rank serve to uniquely

Table 4.20: featureloc_pub

Column	Datatype	Description
featureloc_pub_id	integer	
featureloc_id	integer	
pub_id	integer	

feature_pub

Provenance. Linking table between features and publications that mention them

Table 4.21: feature_pub

Column	Datatype	Description
feature_pub_id	integer	
feature_id	integer	
pub_id	integer	

featureprop

A feature can have any number of slot-value property tags attached to it. This is an alternative to hardcoding a list of columns in the relational schema, and is completely extensible

Table 4.22: featureprop

Column	Datatype	Description
featureprop_id	integer	
feature_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. Certain property types will only apply to certain feature types (e.g. the anticodon property will only apply to tRNA features) ; the types here come from the sequence feature property ontology
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

featureprop_pub

for any one feature, multivalued property-value pairs must be differentiated by rank

Table 4.23: featureprop_pub

Column	Datatype	Description
featureprop_pub_id	integer	
featureprop_id	integer	
pub_id	integer	

feature_dbxref

links a feature to dbxrefs. This is for secondary identifiers; primary identifiers should use feature.dbxref_id

Table 4.24: feature_dbxref

Column	Datatype	Description
feature_dbxref_id	integer	
feature_id	integer	
dbxref_id	integer	
is_current	boolean	the is_current boolean indicates whether the linked dbxref is the current -official- dbxref for the linked feature

feature_relationship

features can be arranged in graphs, eg exon part_of transcript part_of gene; translation madeby transcript if type is thought of as a verb, each arc makes a statement [SUBJECT VERB OBJECT] object can also be thought of as parent (containing feature), and subject as child (contained feature or subfeature) – we include the relationship rank/order, because even though most of the time we can order things implicitly by sequence coordinates, we cant always do this - eg transpliced genes. its also useful for quickly getting implicit introns

Table 4.25: feature_relationship

Column	Datatype	Description
feature_relationship_id	integer	
subject_id	integer	the subject of the subj-predicate-obj sentence. This is typically the subfeature
object_id	integer	the object of the subj-predicate-obj sentence. This is typically the container feature
type_id	integer	relationship type between subject and object. This is a cvterm, typically from the OBO relationship ontology, although other relationship types are allowed. The most common relationship type is OBO_REL:part_of. Valid relationship types are constrained by the Sequence Ontology
value	text	Additional notes/comments
rank	integer	The ordering of subject features with respect to the object feature may be important (for example, exon ordering on a transcript - not always derivable if you take trans spliced genes into consideration). rank is used to order these; starts from zero

feature_relationship_pub

Provenance. Attach optional evidence to a feature_relationship in the form of a publication

Table 4.26: feature_relationship_pub

Column	Datatype	Description
feature_relationship_pub_id	integer	
feature_relationship_id	integer	
pub_id	integer	

feature_relationshipprop

Extensible properties for feature_relationships. Analagous structure to featureprop. This table is largely optional and not used with a high frequency. Typical scenarios may be if one wishes to attach additional data to a feature_relationship - for example to say that the feature_relationship is only true in certain contexts

Table 4.27: feature_relationshipprop

Column	Datatype	Description
feature_relationshipprop_id	integer	
feature_relationship_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. Currently there is no standard ontology for feature_relationship property types
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature_relationship can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

feature_relationshipprop_pub

Provenance for feature_relationshipprop

Table 4.28: feature_relationshipprop_pub

Column	Datatype	Description
feature_relationshipprop_pub_id	integer	
feature_relationshipprop_id	integer	
pub_id	integer	

feature_cvterm

Associate a term from a cv with a feature, for example, GO annotation

Table 4.29: feature_cvterm

Column	Datatype	Description
feature_cvterm_id	integer	
feature_id	integer	
cvterm_id	integer	
pub_id	integer	Provenance for the annotation. Each annotation should have a single primary publication (which may be of the appropriate type for computational analyses) where more details can be found. Additional provenance dbxrefs can be attached using feature_cvterm_dbxref
is_not	boolean	if this is set to true, then this annotation is interpreted as a NEGATIVE annotation - ie the feature does NOT have the specified function, process, component, part, etc. See GO docs for more details

feature_cvtermprop

Extensible properties for feature to cvterm associations. Examples: GO evidence codes; qualifiers; metadata such as the date on which the entry was curated and the source of the association. See the featureprop table for meanings of type_id, value and rank

Table 4.30: feature_cvtermprop

Column	Datatype	Description
feature_cvtermprop_id	integer	
feature_cvterm_id	integer	
type_id	integer	The name of the property/slot is a cvterm. The meaning of the property is defined in that cvterm. cvterms may come from the OBO evidence code cv
value	text	The value of the property, represented as text. Numeric values are converted to their text representation. This is less efficient than using native database types, but is easier to query.
rank	integer	Property-Value ordering. Any feature_cvterm can have multiple values for any particular property type - these are ordered in a list using rank, counting from zero. For properties that are single-valued rather than multi-valued, the default 0 value should be used

feature_cvterm_dbxref

Additional dbxrefs for an association. Rows in the feature_cvterm table may be backed up by dbxrefs. For example, a feature_cvterm association that was inferred via a protein-protein interaction may be backed by by refering to the dbxref for the alternate protein. Corresponds to the WITH column in a GO gene association file (but can also be used for other analagous associations). See http://www.geneontology.org/doc/GO.annotation.shtml#file for more details

Table 4.31: feature_cvterm_dbxref

Column	Datatype	Description
feature_cvterm_dbxref_id	integer	
feature_cvterm_id	integer	
dbxref_id	integer	

feature_cvterm_pub

Secondary pubs for an association. Each feature_cvterm association is supported by a single primary publication. Additional secondary pubs can be added using this linking table (in a GO gene association file, these corresponding to any IDs after the pipe symbol in the publications column

Table 4.32: feature_cvterm_pub

Column	Datatype	Description
feature_cvterm_pub_id	integer	
feature_cvterm_id	integer	
pub_id	integer	

synonym

A synonym for a feature. One feature can have multiple synonyms, and the same synonym can apply to multiple features

Table 4.33: synonym

Column	Datatype	Description
synonym_id	integer	
name	varchar	The synonym itself. Should be human-readable machine-searchable ascii text
type_id	integer	types would be symbol and fullname for now
synonym_sgml	varchar	The fully specified synonym, with any non-ascii characters encoded in SGML

feature_synonym

Linking table between feature and synonym

Table 4.34: feature_synonym

Column	Datatype	Description
feature_synonym_id	integer	
synonym_id	integer	
feature_id	integer	
pub_id	integer	the pub_id link is for relating the usage of a given synonym to the publication in which it was used
is_current	boolean	the is_current boolean indicates whether the linked synonym is the current -official- symbol for the linked feature
is_internal	boolean	typically a synonym exists so that somebody querying the db with an obsolete name can find the object theyre looking for (under its current name. If the synonym has been used publicly & deliberately (eg in a paper), it my also be listed in reports as a synonym. If the synonym was not used deliberately (eg, there was a typo which went public), then the is_internal boolean may be set to -true- so that it is known that the synonym is -internal- and should be queryable but should not be listed in reports as a valid synonym

genotype

===

Table 4.35: genotype

Column	Datatype	Description
genotype_id	integer	
uniquename	text	
description	varchar	

feature_genotype

==

Table 4.36: feature_genotype

Column	Datatype	Description
feature_genotype_id	integer	
feature_id	integer	
genotype_id	integer	
chromosome_id	integer	
rank	integer	
cgroup	integer	
cvterm_id	integer	

environment

===

Table 4.37: environment

Column	Datatype	Description
environment_id	integer	
uniquename	text	
description	text	

environment_cvterm

===

Table 4.38: environment_cvterm

Column	Datatype	Description
environment_cvterm_id	integer	
environment_id	integer	
cvterm_id	integer	

phenstatement

Phenotypes are things like "larval lethal". Phenstatements are things like "dpp[1] is recessive larval lethal". So essentially phenstatement is a linking table expressing the relationship between genotype, environment, and phenotype.

Table 4.39: phenstatement

Column	Datatype	Description
phenstatement_id	integer	
genotype_id	integer	
environment_id	integer	
phenotype_id	integer	
type_id	integer	
pub_id	integer	

phendesc

a summary of a _set_ of phenotypic statements for any one gcontext made in any one publication

Table 4.40: phendesc

Column	Datatype	Description
phendesc_id	integer	
genotype_id	integer	
environment_id	integer	
description	text	
pub_id	integer	

phenotype_comparison

comparison of phenotypes eg, genotype1/environment1/phenotype1 ”non-suppressible” wrt genotype2/environment2/phenotype2

Table 4.41: phenotype_comparison

Column	Datatype	Description
phenotype_comparison_id	integer	
genotype1_id	integer	
environment1_id	integer	
genotype2_id	integer	
environment2_id	integer	
phenotype1_id	integer	
phenotype2_id	integer	
type_id	integer	
pub_id	integer	

phenotype

a phenotypic statement, or a single atomic phenotypic observation a controlled sentence describing observable effect of non-wt function – e.g. Obs=eye, attribute=color, cvalue=red

Table 4.42: phenotype

Column	Datatype	Description
phenotype_id	integer	
uniquename	text	
observable_id	integer	The entity: e.g. anatomy_part, biological_process
attr_id	integer	Phenotypic attribute (quality, property, attribute, character) - drawn from PATO
value	text	value of attribute - unconstrained free text. Used only if cvalue_id is not appropriate
cvalue_id	integer	Phenotype attribute value (state)
assay_id	integer	evidence type

phenotype_cvterm

NULL

Table 4.43: phenotype_cvterm

Column	Datatype	Description
phenotype_cvterm_id	integer	
phenotype_id	integer	
cvterm_id	integer	

feature_phenotype

NULL

Table 4.44: feature_phenotype

Column	Datatype	Description
feature_phenotype_id	integer	
feature_id	integer	
phenotype_id	integer	

featuremap

NOTE: this module is all due for revision...

Table 4.45: featuremap

Column	Datatype	Description
featuremap_id	integer	
name	varchar	
description	text	
unittype_id	integer	

featurerange

==

Table 4.46: featurerange

Column	Datatype	Description
featurerange_id	integer	
featuremap_id	integer	
feature_id	integer	
leftstartf_id	integer	
leftendf_id	integer	
rightstartf_id	integer	
rightendf_id	integer	
rangestr	varchar	

featurepos

==

Table 4.47: featurepos

Column	Datatype	Description
featurepos_id	integer	
featuremap_id	integer	
feature_id	integer	
map_feature_id	integer	
mappos	float	

featuremap_pub

map_feature_id links to the feature (map) upon which the feature is

Table 4.48: featuremap_pub

Column	Datatype	Description
featuremap_pub_id	integer	
featuremap_id	integer	
pub_id	integer	

Appendix A

Chado Naming Conventions

A.1 Case sensitivity

We use lowercase in all tables and column names - DBMSs differ in how they treat case sensitivity. oracle will auto caps everything. so it's best to be neutral and use lowercase.

A.2 Table names

In table names, we use underscores for linking tables; eg `feature_dbxref` is a linking table between `feature` and `dbxref`

where a table name is a noun phrase rather than a single noun, we concatenate the words together. for instance the table for describing feature properties is called `featureprop` . it could be argued this is harder to read, but it does allow consistent usage of underscores as above. FeatureProp could be used where it is known the DBMS is case insensitive.

A.3 Column names

in column names, we also use concatenated noun phrases, except in the case of primary / foreign keys, eg `dbxref_id` .

we try to keep column names unique where appropriate, which is useful for large join statements / views, in avoiding column name clash between different tables. the convention is to use an abbreviated form of the table name plus a noun describing the column, for instance `fmin` in the `feature` table. by consistently using abbreviated forms we stop column names getting too big [many DBMSs will barf on long column names]

A.3.1 Primary and foreign key names

we use the same column name for primary and foreign key columns - very useful for NATURAL JOIN statements