IMPLEMENTATION OF VECTORBASE COMPONENTS:
ENSEMBL GENOME BROWSER, SEARCH, AND DATABASE TOOLS

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

by

E.O. Stinson, B.S.

__________________________________________
Frank Collins, Director

Graduate Program in Biology

Notre Dame, Indiana

September 2007
CONTENTS

Table of Figures.......................................................................................................................iv

Chapter 1: Introduction............................................................................................................1
  1.1. Background..................................................................................................................1

Chapter 2: Genome Browser................................................................................................3
  2.1. Justification................................................................................................................3
  2.2. Overview.....................................................................................................................4
  2.3. Adaptation..................................................................................................................5
  2.4. Configuration.............................................................................................................10
  2.5. Summary....................................................................................................................11

Chapter 3: Databases..........................................................................................................12
  3.1. Introduction...............................................................................................................12
  3.2. Schemas...................................................................................................................12
  3.3. Data stored...............................................................................................................15
  3.4. Loading Data............................................................................................................16
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5. Retrieving data</td>
<td>18</td>
</tr>
<tr>
<td>3.6. Summary</td>
<td>22</td>
</tr>
<tr>
<td>Chapter 4: Search</td>
<td>23</td>
</tr>
<tr>
<td>4.1. Indexing</td>
<td>23</td>
</tr>
<tr>
<td>4.2. Searching</td>
<td>25</td>
</tr>
<tr>
<td>4.3. User Interface</td>
<td>26</td>
</tr>
<tr>
<td>4.4. SOAP and Axis</td>
<td>28</td>
</tr>
<tr>
<td>4.5. Summary</td>
<td>29</td>
</tr>
<tr>
<td>Chapter 5: Future Work</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 6: Conclusions</td>
<td>34</td>
</tr>
<tr>
<td>6.1. Usage</td>
<td>34</td>
</tr>
<tr>
<td>6.2. Summary</td>
<td>34</td>
</tr>
<tr>
<td>Appendix A: Changes to the default EnsEMBL install</td>
<td>36</td>
</tr>
<tr>
<td>Appendix B: Search Engine Documentation</td>
<td>43</td>
</tr>
<tr>
<td>Appendix C: Chapollo Documentation</td>
<td>72</td>
</tr>
<tr>
<td>Appendix D: modularLoader Documentation</td>
<td>91</td>
</tr>
<tr>
<td>Bibliography</td>
<td>126</td>
</tr>
</tbody>
</table>
## TABLE OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>The official EnsEMBL Genome Browser (from <a href="http://www.ensembl.org">http://www.ensembl.org</a>)</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The VectorBase adaptation of the EnsEMBL Genome Browser</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The Apollo genome annotation tool</td>
<td>19</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The databases and utilities used to move data between them and the VectorBase web site</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5</td>
<td>An example ChadoSearchResult from the VectorBase web site</td>
<td>27</td>
</tr>
<tr>
<td>Figure 6</td>
<td>An example ChadoArchiveIdSearchResult from the VectorBase web site</td>
<td>27</td>
</tr>
<tr>
<td>Figure 7</td>
<td>An example ChadoBACImageSearchResult from the VectorBase web site</td>
<td>27</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Search Engine Data Flow</td>
<td>29</td>
</tr>
</tbody>
</table>
CHAPTER 1:

INTRODUCTION

1.1. Background

VectorBase is a web-based bioinformatics repository for insect vectors of human pathogens funded by NIAID (Collins 2007a). It was created in response to the need for a centralized source of insect genomes, publications, and other data. These data are provided not only for download but also in various views throughout the web site, including through the EnsEMBL genome browser and many custom views. In addition to acting as a data repository, VectorBase offers analysis tools and a search engine.

Data at VectorBase are kept in a set of relational databases designed for both genomic and other data. The size of the data sets in question required the generation of bespoke utilities both for transferring data between databases and transforming data for use by other applications. This automated system of data movement decreases the amount of time that must be spent massaging data and streamlines the process for adding new data to VectorBase.
Searching at VectorBase is enabled through the Lucene search package, which has been adapted to index data from the VectorBase databases. On the query side an interface has been developed to make the search engine accessible via web services which are then presented to the user through an interface that has been designed to speed access to the most frequently used information.
2.1. Justification

A genome browser is a necessary component of any genomics repository that wishes to allow easy graphical access to arbitrary points on the genome. At VectorBase, this role is filled by the Ensembl genome browser, developed jointly by EBI-EMBL and the Sanger Centre (EBI-EMBL 2007a). Although it is not within the scope of this document to describe the genome browser, suffice it to say that it is extremely full-featured. It provides several different views of the genome, including a graphical overview of the genome (with genomic features drawn in their relative positions), sequence views, feature views (for detailed information on genes, transcripts, and so forth), and movement tools for the various graphical views including zooming and scrolling. Although it is probably the most complete and stable web-based genome browser available, it is also still under heavy development with new features added on a regular basis.

During the initial stages of development, we considered both the Ensembl genome browser and the GBrowse, the GMOD browser for Chado. Despite our use of the Chado as our primary database schema, we eventually decided on the the Ensembl
browser. It offered a more user-friendly interface, more features, and the added benefit of being the original browser for the _Anopheles gambiae_ mosquito before the advent of VectorBase. Furthermore, VectorBase has a close relationship with the Ensembl project; some members responsible for providing data are also members of Ensembl, so were already familiar with the datasets involved.

2.2. Overview

The Ensembl installation consists of two components: the databases and the Perl code. The Ensembl databases are supplied by VectorBase members at the EMBL-EBI in Hinxton, UK. The Perl code is developed by the Ensembl web team in Hinxton, who are not part of VectorBase.

A set of databases ("core," "est," and "variation," or some subset of the three) exists for each organism with an annotated genome in VectorBase, and a comparative database (ensembl_compara) exists that contains relationship information between the organisms. Ensembl databases are "versioned," and a new version is released approximately every 3-6 months, although this may change as VectorBase's Ensembl data generation pipeline becomes more independent from the rest of Ensembl. There are often small changes in the schema between versions that require the Perl code to be updated at the same time as the databases in order to maintain compatibility. Data loading is done using standard MySQL tools, mysqlimport, in particular.

Of note is that Ensembl runs as a memory-resident process inside the Apache2 web server; the mod_perl Apache module is used to launch Ensembl at Apache startup. This makes configuration of Apache somewhat nonstandard. First, the configuration file for
Apache (httpd.conf) is partially written in Perl, and automatically detects locations of certain EnsEMBL components. It also includes some EnsEMBL initialization code that may make Apache fail to start if EnsEMBL databases are not properly installed or if syntax errors exists in core components of the EnsEMBL code. Several configuration files are also included in the vectorbase/ensembl/conf directory and subdirectories. Organism-specific configuration directives are also found in vectorbase/ensembl/public-plugins/ensembl/conf.

2.3. Adaptation

The EnsEMBL genome browser is designed to be a stand-alone web application, complete with full HTML generation, mod_perl integration, and a consistent theme (Figure 1). Unfortunately, this conflicts with one of our primary goals at VectorBase, which is to provide a consistent and highly integrated interface across all components of VectorBase. This extends beyond look-and-feel to include things like links from VectorBase search engine results to EnsEMBL or from EnsEMBL to VectorBase help pages (Figure 2).

None of the changes to the interface and behavior of the EnsEMBL genome browser are particularly complex, but there is a caveat. The EnsEMBL project is well-staffed, and the official genome browser software is updated frequently and extensively, requiring corresponding updates at VectorBase. To decrease the time spent on upgrades, we at VectorBase have tried to keep our changes as minimal as possible to allow easy upgrades. We do this partially by closely examining candidate changes for usefulness (so as to limit the total number of changes) and partially by implementing most of the adaptation in “wrappers” that run the original EnsEMBL code and then modify or filter the output.
Other changes have been implemented as direct changes to the EnsEMBL code. There are currently nineteen files (not counting configuration files) modified from the official EnsEMBL release code, out of approximately 1600 Perl modules. Most of these changes involve changing hard-coded URLs to point to VectorBase pages instead of EnsEMBL pages. There are also some modifications to the Distributed Annotation System (DAS) architecture to allow better integration of VectorBase data. The last set of major changes adds the ability for EnsEMBL to work within our “wrapper” architecture, and includes changes to the way EnsEMBL pages are initialized as well as URL redirection to allow EnsEMBL-style URLs to be mapped to VectorBase URLs and vice-versa.

The changes to hard-coded URLs deal almost exclusively with links to the documentation pages. Whereas the official EnsEMBL release relies on a static set of documentation that changes only when new versions are released, VectorBase uses a wiki that allows changes and additions to be made by any of a set of moderators. This allows both quick updates in response to users’ questions and easy delegation of documentation writing to those responsible for the data or tool in question.

The Distributed Annotation System is a protocol and tool set that allows a client to pull data from multiple sources both remote and local. The EnsEMBL genome browser can use DAS servers as a source of data for genomic features (and feature data for existing features). At VectorBase, DAS is used internally to integrate manually annotated genes, microarray expression data, and gene comments into the EnsEMBL genome browser. To better serve these purposes, the basic DAS functionality of EnsEMBL has been extended to
include a new option for DAS tracks, “VectorBase DAS,” which has several effects on the display of DAS information.

Setting the “VectorBase DAS” option to “true” modifies the display of DAS features in the graphical views to more closely resemble genes provided in the EnsEMBL databases. The parsing of textual DAS information has also been modified to allow the inclusion of raw HTML; this gives us the ability to format text for display on the individual feature information pages. (For example, community comments are displayed in tabular format on the genes information page.) A final small modification to the DAS system changes text referring to microarrays from the default for better integration with the BASE microarray system at base.vectorbase.org.

The final major section of EnsEMBL code modified for VectorBase is the Apache URL handling code. While the official EnsEMBL website does not strongly differentiate views of different species, VectorBase provides different subdomains with different color schemes for each species. Further, some of the information stored at VectorBase applies to all (or several) of the organisms. This difference is reflected in the web addresses used at VectorBase. Whereas the EnsEMBL website uses URLs of the form “www.ensembl.org/Anopheles_gambiae/contigview,” VectorBase uses URLs of the form “agambiae.vectorbase.org/Genome/ContigView.” While the capitalization is purely aesthetic, the use of subdomains for different species makes it easy us to provide tailored start pages for each the contributing genome projects.

1Web/vectorbase/ensembl/modules/EnsEMBL/Web/Apache/Handlers.pm
2See agambiae.vectorbase.org vs. aaegypti.vectorbase.org vs. www.vectorbase.org.
The creation of a new level in the path portion of the URL ("Genome" for the EnsEMBL genome browser) makes room for other sections such as “Tools,” “Documents,” and so forth, where each can be tailored on a per-organism basis. VectorBase can still handle EnsEMBL URLs; if the EnsEMBL code directs users to “www.vectorbase.org/Anopheles_gambiae/contigview,” they are automatically redirected to “agambiae.vectorbase.org/Genome/ContigView.” This removes the need to change the various URLs hard-coded into EnsEMBL code while still providing a consistent user experience.

These redirections do not actually forward users to the original EnsEMBL Perl scripts for each page. Rather, users are redirected to one of several PHP wrapper scripts3. These scripts are responsible for including the VectorBase “container” formatting – the header and footer. In turn, they make a request to the web server for a more generic PHP wrapper script4. This script sets up environment variables that our adapted EnsEMBL expects (species, page name, and page sections to fetch), and makes a third internal request to a Perl script5. This final script acts as an amalgamation of the original EnsEMBL page generation scripts; it initializes the necessary EnsEMBL components and renders the HTML for the page. The interaction of these three scripts enables the use of our relatively unmodified EnsEMBL. Although some delay is introduced by the series of nested requests, it is not significant in practical terms; the generation of the actual EnsEMBL content in the third stage takes by far the longest.

3Web/vectorbase/sections/Genome/*View.php
4Web/vectorbase/ensembl/htdocs/ensembl_wrapper.php
5Web/vectorbase/ensembl/perl/default/testoutside
There exist other small changes to the Ensembl code at VectorBase, but these are minor: non-critical bug fixes, removing warnings that clog log files, and changes to some text content to better fit the VectorBase data. (The full list of changes is given in .)

Figure 1: The official Ensembl Genome Browser (from http://www.ensembl.org).
2.4. Configuration

The EnsEMBL code is distributed with several configuration files. These include Perl modules, INI files, and the Apache configuration files. The Apache configuration file\(^6\) and Perl modules\(^7\) have been modified to allow the use of PHP (as well as Perl), and to allow access to the VectorBase content that is not part of EnsEMBL. The VectorBase installation of Apache is also configured to run a secure server (SSL) for logins and other sensitive data. Logging is modified for statistics tracking.

\(^6\)Web/ensembl/conf/httpd.conf
\(^7\)Web/ensembl/conf/SiteDefs.pm, Web/ensembl/conf/perl.startup
The rest of the configuration files deal with various simple configuration locations (such as the location of the various Ensembl databases), and with species-specific information (such as the set of DAS tracks visible for an organism, or the assembly version). (For the full list of changes, see .)

2.5. Summary

The Ensembl genome browser has been integrated into VectorBase with a minimum of changes to the official build. From the user's perspective, the genome browser appears consistent and tightly integrated, while from the developer's perspective, there is sufficient separation to make updates easy. Updating the system takes less than a day of developer time, although slightly longer than a day if downloading the new databases and loading them into MySQL is considered. Data integration is mediated by the internal VectorBase DAS server, allowing the both Chado and Ensembl databases to provide data to the website. By utilizing the Ensembl genome browser, the VectorBase project has benefited from the addition of a large feature set without an unacceptable increase in development time.
CHAPTER 3:
DATABASES

3.1. Introduction
At the core of the VectorBase system lie the data. With several sources of incoming data and two internal database schemas, keeping data up-to-date and consistent present some challenges. Genomes and their annotations are large enough that the time taken to do some types of loading operations is on the order of days, and multiple asynchronous sources of data both large and small make it infeasible to reload the database from scratch with each new input. Database engine speed limitations also necessitate the creation of multiple database instances with the same relational structure and schema but different data.

3.2. Schemas
VectorBase uses two primary open-source database schemas for genomic information as well as ancillary schemas for web site management that will not be discussed here. The Chado schema, developed by the GMOD consortium (GMOD 2007), is the primary one for VectorBase and is used by all components with the exception of the
integrated EnsEMBL genome browser. The Chado schema is being actively promoted as a model-genome database by GMOD, and is currently used at BeetleBase, FlyBase, and TIGR, among others. The second genomic database schema used at VectorBase is the EnsEMBL schema, used both for driving the EnsEMBL genome browser and as the primary source of genomic information used to populate the various Chado schemas. For technical reasons and the preference of the original developers, the EnsEMBL schema works with MySQL, and the Chado database is designed to work with PostgreSQL.

Although it is not usually a good practice to maintain two schemas with overlapping data, especially with different database engines, there are factors that we believe make it the best choice for our situation. Primary among these is the desire to use the EnsEMBL browser, as it is the most full-featured solution we have discovered. Initial efforts were made to modify the genome browser to work with Chado, but the high-speed development of EnsEMBL made the merging of the Chado code with new EnsEMBL code untenable given size of our development team. Additionally, the members of VectorBase doing initial gene builds and merges are also members of EnsEMBL, and would be generating the EnsEMBL databases as their initial output, so there was no overhead in terms of getting the data into the EnsEMBL schema format.

The argument could also be made that the EnsEMBL database be used as the sole genomic data schema for VectorBase, but it would have required major extension to hold the additional data types that the Chado schema already accommodates. As a comprehensive resource for invertebrate vectors, VectorBase is also committed to providing documents, images, expression and population data, and controlled
vocabularies/ontologies. Additionally, the Chado schema is well-normalized, meaning that information is infrequently repeated, whereas the EnsEMBL schema is optimized for speed and thus includes repeated data. Attempting to load data from other sources into an EnsEMBL database would incur a significantly higher overhead in terms of the development time for loading scripts and tools.

VectorBase currently provides information for five major organisms. Many more are planned, including genomic and accompanying information for closely related organisms, as in the case of the M and S forms of *Anopheles gambiae*. As sequencing and assembly become more accessible, the number of sequences for various forms of the VectorBase organisms is likely to grow quickly.

At the heart of every genomics database lie the sequence and genomic features of organisms, which are in large part a reflection of the core dogma of molecular biology: DNA begets RNA begets proteins. Chado keeps track of this concept by defining features and relationships using “feature relationships.” In Chado parlance, a “feature” is anything that can be aligned to a genome, be it a gene, transcript, protein, repeat region, or even BLAST hit. These features are stored in the *feature* table and then related through the *feature_relationship* table. Features are located on the genome through the *feature_location* table, and so forth (GMOD 2006).

The EnsEMBL schema stores features somewhat differently, with separate tables for different sorts of features. An API written in Perl provides a way to access features in a similarly layered way (EBI-EMBL 2007b), (Curwen 2004). The API adds another benefit
by offering a common interface to the EnsEMBL schema despite changes and updates by the EnsEMBL project team.

3.3. Data stored

The primary form taken by data coming into VectorBase is a set of populated EnsEMBL databases provided by members of VectorBase involved in the initial genome annotation process. These databases provide sequence, genes, transcripts, proteins, repeat regions, transposons, microarray probes, and various sorts of analysis features (for instance BLAST hits to other genomes). When a new or updated EnsEMBL database is provided to the VectorBase web team, the first task is to load the data into two instances of the Chado schema.

One of these databases is populated with a subset of the EnsEMBL data including all of the basic feature types but lacking the majority of analysis features. This database is then used to drive the search engine and to provide access to genomic features for other parts of the website. (For instance, generating some of the content on the interactive BLAST results page.) Analysis features are unused by the web code and so are left out. They would massively increase the size of the database and slow down web response times. For situations where analysis features are needed, a second Chado database is created containing all of the EnsEMBL feature data including these analysis features. This database is used in cases where completeness is more critical than speed. Currently, this includes the generation of ChadoXML formatted data for use with the Apollo annotation tool, and the generation of the BioMart databases for tabular data retrieval.
There is a third instance of the Chado database used for manually annotated genes. These data are kept separately from the other Chado data to ease tracking of manually annotated genes. These genes are provided by the community either through a web interface or as uploaded ChadoXML. The are eventually rolled into the EnsEMBL gene builds and databases so that they do not have to be separately reloaded into the Chado database. There are plans, as yet unimplemented, for merging these data into the core Chado database.

3.4. Loading Data

In order to streamline the loading of data into the Chado database, we have developed a Perl application called modularLoader that, as implied by the name, consists of a set of input modules and a set of organism modules. These modules be run in sequence or independently. Currently, four modules exist for loading EnsEMBL data: one for basic features (genes, transcripts, and proteins), one for microarrays, one for analysis features, and one for archived gene identifiers. A fifth module, currently in development, will be used to load ChadoXML generated by Apollo. The organism modules consist of settings defining available data sources for those organisms.

The modularLoader tool consists of three basic components. The first of these is the Chado interface that does the work of inserting data into the Chado database. Along with encapsulation of SQL statements, this component also provides insertion forgiveness. Insertion forgiveness allows the software to attempt to insert data and then safely handle cases where that data violates constraints preserving uniqueness.
The database module attempts to create a feature first (for speed), but if a matching feature already exists, it returns the database identifier for that feature. This allows the application to behave redundantly for common features. For example, each of the three EnsEMBL modules attempts to create all of the base features (chromosomes or supercontigs, depending on the organism), but if they have already been loaded, the module utilizes the existing features. Since the default mode is to attempt feature creation before looking for an existing one, a second identical run of a module will show a significant slowdown, but the data will remain consistent. If data is simply being added (and not changed or removed) the modules can be run repeatedly.

The second component is the set of organism modules. A Perl module exists for each organism that contains basic information such as genus and species as well as configuration data for the rest of the application. These configuration directives are passed to the data loading modules to determine database connections to sources in EnsEMBL, including the base genome units (chromosome or scaffold). The organism module can also specify broad limits on the sorts of data that will be loaded. For instance, no EST data are currently provided by EnsEMBL for *Aedes aegypti* ESTs, and this is reflected in the organism module.

The final component of the modularLoader tool is the data input component. The four EnsEMBL modules are EnsEMBLBasicFeatures, EnsEMBLAnalysisFeatures, EnsEMBLArchiveIDs, and EnsEMBLMicroarrays. The latter three illustrate the extensibility of modules; each inherits most of its functionality from the EnsEMBLBasicFeatures module, replacing only the code to pull features from the various
EnsEMBL databases. Each module can be run independently, or chained together in any order. Likewise, they can be run for different organisms independently providing the option to one set of features for one organism and only the basic features for another.

Allowing mixing and matching of components and extension with new modules makes the modularLoader tool useful for anyone doing repetitive data loading into Chado database, especially using the EnsEMBL databases as a starting point. With the continuing expansion of the GMOD project, more tools are appearing that work primarily with the Chado database, and existing users of EnsEMBL databases should consider the modularLoader tool as an automated way to get the benefits of both EnsEMBL and GMOD solutions with very little extra effort. Projects with other sources of data may also be interested in adapting the forgiving and fast framework for other purposes.

3.5. Retrieving data

Although most data are presented through the VectorBase web site, there are cases where third-party tools are required. In particular, the Apollo annotation tool, (initially designed for small genomes and regions of genomes) works well with ChadoXML files that have been generated for a particular region of interest (Lewis 2002). We have developed Chapollo, a tool that provides a web interface for dumping specific regions and feature types of a genome from a Chado database into Apollo-compatible ChadoXML\(^8\) (see Figure 3). Although a Chado-to-ChadoXML tool exists\(^9\), it is not very flexible and will always convert a whole database to XML, which can often exceed the capabilities of

\(^8\)http://www.vectorbase.org/User/UserTools
\(^9\)XML-XORT
Apollo. *Chapollo* is currently used by VectorBase annotators as part of our community annotation process, although it will work as is with any VectorBase/FlyBase compatible Chado database.

![Figure 3: The Apollo genome annotation tool.](image)

*Chapollo* consists of two main components. The workhorse component is written in Perl and relies on an extension to the Bio::SeqIO::chado class (part of BioPerl) that adds exporting of genomic features. The original class only provides a way to export sequence features, with little associated metadata. Our extension, Bio::SeqIO::chadoxml_apollo, adds methods for writing Apollo-specific "appdata" tags as well as generic features (including feature relationships) passed in as Perl hashes that mirror the eventual XML formatting. Additionally, tags are reordered to conform to the Apollo-specific expectations of ChadoXML formatting. This quirk-tolerant code is in part responsible for our decision to include Apollo in the name of our new module.
The second component of *Chapollo* is the web input form. Two factors encouraged us to differentiate the web code and Perl code. Depending on the amount of data requested, the export process could take some time, so it was decided to offload the work onto the VectorBase application servers. Second, the Perl component requires a complex input, including organism, genomic regions to export, analysis and feature types to export, and filters on those features and analyses. It would take significantly more effort to maintain Perl CGI code with integrated HTML, especially as it needs to conform to the VectorBase look-and-feel. The solution implemented was to generate a block of Perl code defining the input variables, and then pass it to the Perl code over our existing application distribution framework. In cases where standalone installations are desired, any method can be used to generate the input code; further, multiple front ends can be used with the same Perl application.

*Chapollo* works by first requesting a region (or set of regions on the same chromosome/supercontig) from the user and the types of data they wish to retrieve. Gene features are displayed in the feature track of Apollo, allowing the user to modify them. This can be used at VectorBase by curators of the community annotation pipeline. Analysis features are displayed in the evidence tracks of Apollo, and are generated from BLAST results and other computational comparisons. There are three filters that can be applied to these features: score, noise, and compression. The score filter removes features that fall below a specified score, identity, or significance. The noise filter removes partial hits that fall far away from other matches. The compression filter combines overlapping hits into single features to reduce the total number of features. While these filters can be useful from
an analysis perspective, they are primarily useful in reducing the amount of data Apollo must load (and thus hardware requirements).

By providing the capability to select data by region and filter by various attributes, *Chapollo* makes it possible to use Apollo with any FlyBase-compatible Chado database, regardless of size. This is of great benefit to projects such as VectorBase that host large data sets. It is also a more accessible system than XML-XORT, allowing users to fetch Apollo-compatible data from a Chado database without an underlying knowledge of Chado or ChadoXML.

Figure 4: The databases and utilities used to move data between them and the VectorBase web site.
3.6. Summary

We have described solutions for two interaction methods with the GMOD Chado schema. The first, *modularLoader*, tackles integration of outside data (especially EnsEMBL databases) into a Chado-based system, allowing increased flexibility and extensibility of existing systems. The second, *Chapollo*, addresses the current lack of robust tools for working with Chado data in Apollo, despite the ostensible compatibility of the two projects. Both *Chapollo* and *modularLoader* have been created as generalized solutions, allowing easy adaptation by future projects. *Chapollo* requires only an installation of BioPerl and a populated Chado database; *modularLoader* requires an incompletely populated Chado database and a data source (such as a populated EnsEMBL database and accompanying API).
CHAPTER 4:
SEARCH

As with any web site providing a wide array of information, VectorBase is designed to balance ease of use with comprehensiveness of data. It is desirable to reduce the number of actions necessary to get to any particular page, but it is similarly important to avoid presenting excessively complicated choices to the user. Fortunately, adding a search engine solves this conflict by removing intermediate pages without decreasing the accessibility – searching is a familiar interface for most users.

4.1. Indexing

Although it is a common solution to enable searching by running queries directly on a database, the sheer volume of data at VectorBase makes this sort of querying very slow. Fortunately, an open source solution exists in Lucene (Apache 2007a). Developed by the Apache Software Foundation, Lucene is a search engine framework written in Java that can be adapted to index data from any source accessible through a Java interface. It runs under the Apache Tomcat servlet container, also used to provide DAS sources and part of the community annotation pipeline at VectorBase (Apache 2007b). Queries and responses are transferred between PHP and Tomcat using the Simple Object Access Protocol (SOAP),
which allows the transfer of simple objects between different languages over HTTP (W3C 2007).

The Lucene package consists an indexer and a search engine. The indexer is responsible for generating an optimized index file. The search engine uses the index file to perform very fast searches given a boolean query. As such, the search results are only as up-to-date as the index. Indexing VectorBase's data using Lucene was a matter of creating a Java utility that connects to the Chado database and selects the features of interest. Features are retrieved from the database using ordinary SQL queries. For each feature, a Lucene “Document” object is created with the name of the feature and other associated metadata. These document objects are then passed to the Lucene indexer which generates the index file used to run the search engine.

The feature types currently indexed are basic genomic features both known and predicted (genes, transcripts, and proteins), genomic regions (chromosomes, supercontigs, contigs, and golden path regions), and evidence features (BAC clone locations and microarray spots). During the indexing process text fields are tokenized. Text is split into tokens at spaces and punctuation to improve searching. For example, the phrase “gene description” will be split so that it matches queries for both “gene” and “description.” Some modifications to the default Lucene tokenizer have been made to increase the number of punctuation characters. In addition, feature names that match the Ensembl stable identifier format are split into the prefix and identifier number.$^{10}$

---

$^{10}$ PREFIX000#####; for example “AGAP005164” is split into “AGAP” and “5164”
Each feature's index entry also includes the genomic location (chromosome/supercontig, start, end, and length), feature type, organism, controlled vocabulary terms, and the feature's description field, if available. For transcripts and proteins, the parent gene name is also stored so that searches for the gene turn up the derived transcript and peptide features. Two fields are added that are not indexed for search but are used instead only in the results display on the website. One is a reference to any image that may be associated with a feature (as with BAC hybridization images); the other is the link to the feature report page in the genome browser.

4.2. Searching

Once the index file is generated it is moved to the VectorBase application server where the search engine resides. The search engine, running as a servlet within Tomcat, is induced (via a SOAP call) to load the newly uploaded index file at which point the indexed data is available for querying.

Queries in Lucene exist as Java objects. Simple one-term queries can be a single object, while more complex queries are built by nesting objects. For example, a `BooleanQuery` is built from a set of clauses, where each `BooleanClause` consists of a query and a boolean operator. VectorBase uses two basic query types in addition to boolean queries. The first type is a simple term query (for example, a feature name); the second is a range query (as between a start and end coordinate on a chromosome). Term queries can include a wildcard component, providing a way to find partial matches to gene names, descriptions, and other text fields where the full term is unknown.
Of course, the complexity of Lucene is hidden from users of the VectorBase website. A “quick search” box on the VectorBase home page and the organism home pages presents a single search field for fast feature lookup. Entering an exact gene name will even redirect immediately to the gene report page. An advanced search page is also available\textsuperscript{11}, and allows the user to specify the organism and chromosomal location as well as multiple search terms. There is also a specific search page for BAC clone hybridization images\textsuperscript{12}.

4.3. User Interface

Submissions from the search are executed by PHP using a remote SOAP call to the search engine. The response from the server is therefore a SOAP object, which is then cast\textsuperscript{13} into one of three PHP classes. A \textit{ChadoSearchResult} is the catch-all result type for ordinary genomic features, and links to the feature's detail page (the gene report page, for example). A \textit{ChadoBACImageSearchResult} is an image result, linking to the image display for the aforementioned BAC clone hybridization image. A \textit{ChadoArchiveIdSearchResult} is created for feature identifiers that are retired gene names; they link to the identifier history page of the genome browser\textsuperscript{14}, showing other names for the same gene.

From the array of returned objects, a result page is generated using PHP. Each feature type has its own template for display. A \textit{ChadoSearchResult}'s display (Figure 5) includes a link to the feature report page for the feature and to the genome browser view

\textsuperscript{11}http://www.vectorbase.org/Search/
\textsuperscript{12}http://www.vectorbase.org/Images/BACs/
\textsuperscript{13}In this context, “cast” means assigned the properties of. A generic type can be cast to a more specific type to add functionality that does not apply in the general case.
\textsuperscript{14}For example: http://agambiae.vectorbase.org/Genome/IdhistoryView/?db=core;gene=ENSSANGG00000023529
showing the surrounding region on the genome. It also shows the feature name, type, location on the genome (both as text and as an image), and description, if any. A ChadoArchiveIdSearchResult's display (Figure 6) includes only a link to the feature history page, which shows the new identifier for the gene (if any), and the status of the gene model. It also displays the type (“Deprecated gene” as in Figure 6) and a short description of what archiving means. A ChadoBACImageSearchResult's display (Figure 7) provides a link to the genome browser view showing the region surrounding the BAC clone. Also included are the type (always “BAC_clone”), location on the genome (as text and image), description (if any), and a link to view hybridization images.

Figure 5: An example ChadoSearchResult from the VectorBase web site.

Figure 6: An example ChadoArchiveIdSearchResult from the VectorBase web site.

Figure 7: An example ChadoBACImageSearchResult from the VectorBase web site.
4.4. SOAP and Axis

SOAP is a protocol for running methods and sending complex data types across an HTTP connection. It has been implemented in many programming languages, including the four used most frequently at VectorBase (PHP, Perl, Java, and Javascript). In the case of the VectorBase search system, the standard PHP SOAP functionality is used on the client (in this case, the web server). On the application server, Java methods and objects are translated to SOAP by the Apache Axis web services engine (Apache 2007c). Axis also generates the Web Service Definition Language (WSDL) file that defines the offered functions and services for the search engine.

There are four methods of the org.vectorbase.www.search.lookup.Search class that allow execution of queries. The findInChado function is the most generic, and searches for features given some subset of genomic location, feature type, and name. The other three functions, findMicroArrayInChado, findArchiveIdInChado, and findBACImageInChado, are very similar except that they automatically filter by type and are thus simpler to access from the client. They also return objects of a specific type. Whereas findInChado returns an all-inclusive ChadoSearchResult type, findArchiveIdInChado returns a ChadoArchiveIdSearchResult. These types are included when the results are returned over SOAP, and determine which PHP class they will be cast as.
Two other remote methods are implemented for administration tasks. The `reloadIndices` method causes the running search engine to re-read the index file so a new index may be loaded without taking the search engine offline. Similarly, `reloadConfig` causes the search engine to reread its configuration file, which specifies the location of the index file.

Thanks to Axis, very little extra work was required to offer these functions as remote SOAP methods. As long as the objects being passed over SOAP conform to the JavaBean specification (Hamilton 1997), Axis will automatically convert them to the XML necessary for transmission via SOAP (Apache 2007d).

4.5. Summary

The Lucene search engine framework has been adapted to work with genomic features stored in the Chado format. Additionally, the Lucene engine has been integrated with Tomcat and Axis, allowing the search engine to be accessed on the VectorBase
internal network by any client that can generate SOAP requests. Currently the only client is the PHP driving the search pages for the web site, but others can easily be added.
CHAPTER 5:  
FUTURE WORK

In its current form, VectorBase meets many of the needs of researchers interested in genomic data of insect vectors. However, there are still some features that are unpolished, and a few planned sections that have yet to be implemented. For instance, integration of population data has yet to be addressed, and although there is a controlled vocabulary browser, links between anatomical terms and genes have yet to be created. There are also updates to the genome browser, database tools, and search engine mentioned in this document that should be implemented.

As one of the most heavily used parts of VectorBase (and as the display for the genomes themselves), the genome browser is one of the most complete sections of the site. All of the basic EnsEMBL functionality is active, including the genome, feature, and comparative views. What is missing is a good local implementation of the BioMart data retrieval tool (EBI, CSHL 2007). Currently, users must visit the generic BioMart page to fetch the most recent data. Furthermore, VectorBase provides a BioMart database for microarray data, but this cannot be fully integrated with genomic data in BioMart without
hosting a local copy. Some work has been done with BioMart, but more is necessary for a fully functioning system.

The _Chapollo_ and _modularLoader_ database tools have only recently reached a usable state, and are still missing some features. In particular, _Chapollo_ is lacking a command line interface, and _modularLoader_ is lacking a way to load ChadoXML data into a Chado database. A command line interface is not necessary for the current web-based use of _Chapollo_, but would prove useful for testing and if _Chapollo_ is distributed as a generic GMOD component. More urgent is the addition of a new module to _modularLoader_ for parsing ChadoXML back into a Chado database as part of the community annotation pipeline (Bruggner 2007). Without this ability, community representatives cannot easily make corrections to gene models submitted by the community, hampering the curation process.

As noted often by members of the VectorBase user community, the search engine is very useful for finding genomic features, but it has no capacity for finding anything else. The search index should be grown to include general VectorBase web pages (such as organism home pages), downloadable documentation (such as genome project PDFs), and downloadable data (such as FASTA files). A search for “aedes” should optimally return results that include the _Aedes aegypti_ genome home page, the white-paper for the _Aedes_ genome project, and the FASTA files containing _Aedes_ transcript and protein sequences.

None of these tasks are particularly difficult, and now that the core of VectorBase is functional, more development time has opened up for non-critical tasks. As it stands, the web site meets many of the needs of researchers, and we have managed to create the
consistent interface and tight integration outlined in our original design plans. With the
exception of displaying population data (an unexplored field that will require considerable
research), further work at this point is mostly polish and the addition of ancillary
functionality to existing features.
CHAPTER 6:

CONCLUSIONS

6.1. Usage

Among the possible resources for insect vector genomics, VectorBase has already become the closest thing to a one-stop resource to date. Usage reflects this, even in the early stages of outreach. There are approximately 110 daily unique visitors to the VectorBase web site on average, with an average of more than 5000 page views per day. The search engine is used 70 times per day on average, and there are on average 430 different views of the genome browser daily. The bioinformatics tools (BLAST, etc.) are also heavily utilized to analyze sequence against the data sets at VectorBase\textsuperscript{15}.

6.2. Summary

VectorBase first became an alternative to other insect genome resources with the addition of the genome browser. As new pieces were added, including search, integrated BLAST, annotation tools, and new organisms, VectorBase surpassed the individual functionality of any of these sites. Not only have we created a useful repository for insect vector information, we have gone beyond solving VectorBase-specific problems by

\textsuperscript{15}See the six-month report (Collins 2007b) for overview statistics.
building tools that will work with many GMOD/Chado databases. The database management tools developed not only increase the amount of time VectorBase developers have for development (as opposed to maintenance), but are also generic enough to be used by other GMOD projects. Likewise, the search engine developed at VectorBase not only improves the user experience, but it is trivial to adapt to Chado databases outside of VectorBase.

Furthermore, the VectorBase web site exhibits a high level of consistency in interface, with close integration between tools that are often left unlinked at other web sites. BLAST results are displayed in the genome browser, side-by-side with genes submitted by the community for curation. All of the pages exhibit the same theme, going beyond simple color changes to include consistent menus, a shared help system. Organisms are branded for individual projects, but are still consistent in terms of interface, so that tracking an ortholog from one organism to another is not confusing for users.

This integration and consistency is used to present an advanced set of tools and resources for researchers in the insect vector community. VectorBase is now the most thorough resource for the organisms it hosts, and will only continue to become more useful as further features are added.
APPENDIX A:

CHANGES TO THE DEFAULT ENSEMBL INSTALL

Following is a list of EnsEMBL files that have been changed from those that come with the default install, along with a list of changes made to each.

- Web/vectorbase/ensembl/ensembl-draw/modules/Bio/EnsEMBL/GlyphSet/das.pm
  - Added a link to the VectorBase Help wiki.
  - Added `vbgmenu` subroutine, a mirror of the `gmenu` subroutine except for output formatting changes.
  - Added several `RENDER_vb*` subroutines, mirrors of the `RENDER_*` subroutines except for formatting.
    - Graphical features are rendered to look more like EnsEMBL gene and transcript features
    - HTML is not escaped (or is specifically unescaped) in the DAS “note” field to allow complex formatting on the GeneView, TranscriptView, and ProteinView pages.

- Web/vectorbase/ensembl/ensembl-draw/modules/Bio/EnsEMBL/GlyphSet/generic_match.pm
  - Commented out a link to the “pfetch” utility; we don't run any “pfetch” sequence servers at VectorBase, so it is better to hide this that display an error message when clicked.

- Web/vectorbase/ensembl/ensembl-draw/modules/Bio/EnsEMBL/GlyphSet/generic_microarray.pm
  - Changed naming system for microarrays to print "Oligo feature" or "Microarray feature" prefixes.
• Added a link in microarray feature floating menus ("zmenus") to base.vectorbase.org.

• Web/vectorbase/ensembl/ensembl-draw/modules/Sanger/Graphics/JSTools.pm

• Unescape angle brackets: translate “&gt;” and “&lt;” with “>” and “<”.

• Web/vectorbase/ensembl/ensembl-external/modules/Bio/EnsEMBL/ExternalData/DAS/DAS.pm

• Reformatted “note” field of DAS sources for VectorBase DAS tracks

• Web/vectorbase/ensembl/ensembl-external/modules/Bio/EnsEMBL/ExternalData/DAS/DASAdaptor.pm

• Added vbchadodas subroutine (setter/getter) and VBCHADODAS argument option

• Web/vectorbase/ensembl/modules/Bio/Das/Request/Dsn.pm

• There exist two files in the Request directory: Dsn.pm and dsn.pm (case sensitive). Only one is necessary: Dsn.pm. There are no changes to this file, but we do ignore the presence of dsn.pm.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/ExternalDAS.pm

• Added VBCHADODAS flag for DAS tracks.

• Commented out unnecessary warning message.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Object.pm

• Update links to help system to point at the VectorBase Help wiki.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/UserConfig.pm

• Added VBCHADODAS option.

• Changed caption for microarray features.

• Turned on VectorBase microarray features by default.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Apache/Handlers.pm

• scriptview->ScriptView redirection (recapitalization).

• Organism host name redirection (Anopheles_gambiae to agambiae).

• Removed Apache2::SizeLimit (does not work on Mac OS X).

• Added Apache2::Request and Apache2::ServerUtil to get more detailed info about HTTP request.

• Added URI::Escape for escaping of redirection URLs.
• Changed HTTP_TEMPORARY_REDIRECT to REDIRECT to improve URL redirection in some web browsers.
• Changed script location from /perl to /ensembl/perl.
• Commented out all of the CleanupHandlers that use Apache2::SizeLimit.
• Changed default index file from .html to .php.
• Changed to create the EnsEMBL session cookie manually since we're inside the PHP script wrappers.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Apache/SendDecPage.pm
  • Removed unnecessary warning message.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Component/DASCollection.pm
  • Added configuration keys for VBCHADODAS.
  • Add selection box to DAS configuration screen for making VBCHADODAS tracks.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Configuration/DASCollection.pm
  • Added configuration keys for VBCHADODAS.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Configuration/Location.pm
  • Bug fix - Added check to make sure SNP or VEGA EnsEMBL databases exist before adding link to BioMart in floating menus (“zmenu”).

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Document/Panel/Fragment.pm
  • Got rid of multiple unnecessary warning messages.
  • Bug fix - Added a check to make sure that HTML is generated by the renderer before attempting to unescape it.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Factory/DASCollection.pm
  • Added configuration keys for VBCHADODAS.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Factory/Feature.pm
  • Added search link to “Feature Not Found” messages.

• Web/vectorbase/ensembl/modules/EnsEMBL/Web/Factory/Location.pm
  • Added informational error message for case when no EnsEMBL coordinate system is specified in a species INI file.
• Web/vectorbase/ensembl/perl/default/*
  • Changed the path of Perl executable to Mac OS X default.
  • Inserted altered contents of scripts into
    Web/vectorbase/ensembl/perl/default/testoutside for use by PHP wrappers.
• Web/vectorbase/ensembl/htdocs/ensembl_wrapper.php
  • Added this file; it is the second stage of PHP wrapper scripts.
• Web/vectorbase/ensembl/htdocs/img/dd-menus/min-box.gif
  • Altered the minimization image to better fit the VectorBase layout.
• Web/vectorbase/ensembl/htdocs/img/dd-menus/plus-box.gif
  • Altered the size restore image to better fit the VectorBase layout.
• Web/vectorbase/ensembl/htdocs/js/help.js
  • Changed help links to point to the VectorBase Help wiki.
• Web/vectorbase/ensembl/htdocs/js/new_old_zmenu_42.js
  • Changed help links to point to the VectorBase Help wiki.
• Web/vectorbase/ensembl/htdocs/js/new_support.js
  • Changed to not set the Javascript LOADED variable to false; this variable indicates
    whether a page is fully loaded, and is now set in Web/vectorbase/index.php.
• Web/vectorbase/ensembl/htdocs/js/zmenu.js
  • Changed help links to point to the VectorBase Help wiki.
• Web/vectorbase/ensembl/htdocs/js/zmenu_42.js
  • Changed help links to point to the VectorBase Help wiki.
• Web/vectorbase/ensembl/public_plugins/ensembl/conf/SiteDefs.pm
  • Commented out organism configurations for species not hosted at VectorBase.
  • Changed the default organism to A. gambiae.
• Web/vectorbase/ensembl/public_plugins/ensembl/conf/ini-files/Aedes_aegypti.ini
  • Added assembly information for current gene build of A. aegypti.
  • You do want to update assembly versions and such if they're available. (for instance
    SPECIES_RELEASE_VERSION or ASSEMBLY_ID)
• You also want to change the database names to point to the right place.
  • (aedes_aegypti_core_43_1a becomes aedes_aegypti_core_45_1a or whatever)
• Added DAS tracks.

- Web/vectorbase/ensembl/public_plugins/ensembl/conf/ini-files/Anopheles_gambiae.pm
  • Added assembly metadata.
  • Changed database names to point to installed EnsEMBL databases.
  • Added VectorBase-specific DAS tracks.

  • Added assembly metadata.
  • Changed database names to point to installed EnsEMBL databases.

- Web/vectorbase/ensembl/public_plugins/conf/ini-files/MULTI.ini
  • Changed database names to point to installed EnsEMBL databases.

- Web/vectorbase/ensembl/conf/SiteDefs.pm
  • Added port 443 to ENSEMBL_PORT to let Apache run on SSL as well as regular HTTP.
  • Changed default administrator email addresses to webmaster@vectorbase.org.
  • Changed mail server to point to internal VectorBase mail server.
  • Changed default web server address to www.vectorbase.org.
  • Added Web/vectorbase as the default document directory
  • Removed references to the biomart-perl document directory (BioMart is installed elsewhere).
  • Set default database names for web_user_db.
  • Changed help links to point to the VectorBase Help wiki.

- Web/vectorbase/ensembl/conf/httpd.conf
  • Added PHP module to handle PHP pages.
  • Changed the Listen option to accept multiple ports
  • Changed the DocumentRoot directive to point to Web/vectorbase.
• Changed paths to /ensembl/<previous path> in many places (ScriptAlias, among others)
• Changed the default directory index file from index.html to index.php.
• Added new root directory (vectorbase instead of vectorbase/ensembl).
• Changed locations of error pages to /Error/###.
• Enabled ExecCGI directive for the ensembl/perl directory.
• Moved BioMart warnings inside a conditional check to see if BioMart is being loaded.
• Changed LogFormat to be compatible with AWStats
• Added .pl as a valid CGI extension
• Added PHP and SSL certificate type handlers
• Removed /das location since VectorBase does not use EnsEMBL to serve DAS.
• Added Deny directives to keep prevent browsing of the EnsEMBL source code.
• Added SSL certificate locations for secure logins.
• Added mod_proxy directives to forward application server stuff

  • Web/vectorbase/ensembl/conf/perl.startup
    • Commented out Apache2::SizeLimit (does not work on Mac OS X).
    • Modified BioMart configuration to point to the VectorBase install of BioMart.
    • Added EnsEMBL::Web::SpeciesDefs to allow dynamic access to the organism names while in the configuration stage.
    • Changed startup output to reflect multiple Listen ports.

  • Web/vectorbase/ensembl/conf/ini-files/DEFAULTS.ini
    • Set the default EnsEMBL database host, username, and password.
    • Set the directory for EnsEMBL helper binaries to /sw/bin.
    • Disabled pfetch sequence fetching
    • Set the font path for image rendering to Web/vectorbase/ensembl/conf/fonts/.
    • Set AFFY_FASTAVIEW default link to point to base.vectorbase.org.
- Web/vectorbase/ensembl/conf/ini-files/MULTI.ini
  - Set the database name for the Compara database.
APPENDIX B:

SEARCH ENGINE DOCUMENTATION

B.1. Overview

This documentation was generated with the Sun javadoc tool, using the ApiGuideDoclet doclet to convert to HTML, with some formatting applied by hand.

B.2. Packages Overview

The API consists of the following packages:

- org.vectorbase.www.search
- org.vectorbase.www.search.analysis.chado
- org.vectorbase.www.search.indexer
- org.vectorbase.www.search.indexer.chado
- org.vectorbase.www.search.indexer.cv
- org.vectorbase.www.search.lookup
- org.vectorbase.www.search.lookup.chado
- org.vectorbase.www.search.ser

The API packages contain classes and interfaces as outlined below.

- org.vectorbase.www.search
  - Classes
• **PaddedNumString** - `PaddedNumString` is a wrapper for integers that can be used to get zero-padded versions of that integer.

• **org.vectorbase.www.search.analysis.chado**
  
  • **Interfaces**
  
  • `ChadoTokenizerConstants` - No description.
  
  • `CharStream` - This interface describes a character stream that maintains line and column number positions of the characters.

  • **Classes**

  • `ChadoTokenizer` - A grammar-based tokenizer constructed with JavaCC.
  
  • `ChadoTokenizerTokenManager` -
  
  • `FastCharStream` - An efficient implementation of JavaCC's `CharStream` interface.
  
  • `Token` - Describes the input token stream.

  • **Exceptions**

  • `ParseException` - This exception is thrown when parse errors are encountered.

  • **Errors**

  • `TokenMgrError` - No description.

• **org.vectorbase.www.search.indexer**

  • **Interfaces**

  • `Aggregator` - Interface for aggregators that generate Lucene
    

  • **Classes**

  • `Indexer` - `Indexer` generates a new Lucene index file from the database.

  • **Exceptions**

  • `IndexerException` - `IndexerException` is a simple extension of the `Exception` class that has no added functionality.

• **org.vectorbase.www.search.indexer.chado**

  • **Classes**

  • `ChadoFeatureAggregator` - `ChadoFeatureAggregator` is the implementation of `Aggregator` used to index genomic features from a Chado database.
- **ChadoFeatureAnalyzer** - *ChadoFeatureAnalyzer* is an extension of the *Analyzer* class for tokenizing and analyzing Chado genomic features.

- **org.vectorbase.www.search.indexer.cv**

  - **Classes**
    - **CVAggregator** - *CVAggregator* is the implementation of *Aggregator* used to index controlled vocabulary terms from a Chado database.

- **org.vectorbase.www.search.lookup**

  - **Classes**
    - **Lookup** - *Lookup* is a test class for the *Search* class.
    - **NumRangeQuery** - *NumRangeQuery* is a numeric range query for Lucene that can work with large ranges.
    - **Search** - *Search* is the class that creates actual searches against the index and returns the results.
    - **SearchResult** - *SearchResult* is the most basic search result, extended by anything that needs more fields. It is hopefully generic for any sort of search result that you want to return.
    - **WildcardTermFilter** - *WildcardTermFilter* is an extension of the basic Lucene *Filter* class for filtering a set of search results based on a search term containing a wildcard.

- **org.vectorbase.www.search.lookup.chado**

  - **Classes**
    - **ChadoArchiveIdSearchResult** - *ChadoArchiveIdSearchResult* extends *ChadoSearchResult* to add a flag indicating that the feature is archived.
    - **ChadoBACImageQueryController** - *ChadoBACImageQueryController* is an extension of *ChadoQueryController* used to build a single query from a set of *BooleanClauses* for finding "BAC_clone" features with images.
    - **ChadoBACImageSearchResult** - *ChadoBACImageSearchResult* extends *ChadoSearchResult* to add a flag indicating that the feature has an image.
    - **ChadoMicroArraySearchResult** - *ChadoMicroArraySearchResult* extends *ChadoSearchResult* to add a flag indicating that the feature is a microarray spot.
    - **ChadoQueryController** - *ChadoQueryController* is used to build a single query from a set of *BooleanClauses*. 
• ChadoSearchResult - ChadoSearchResult extends SearchResult to add Chado genomic feature fields.

• org.vectorbase.www.search.ser

Classes

• ChadoBeanSerializer - ChadoBeanSerializer implements serialization of a ChadoSearchResult (and known subclasses ChadoBACImageSearchResult, ChadoMicroArraySearchResult, and ChadoArchiveIdSearchResult for use by Axis when converting to a SOAP object.

• ChadoBeanSerializerFactory - ChadoBeanSerializerFactory implements a serializer factory for Axis to create a ChadoBeanSerializer for serializing org.vectorbase.www.search.lookup.chado.ChadoSearchResult and subclasses.

The remaining chapters of this document describe the contents of each package in detail.


Package org.vectorbase.www.search contains the following interfaces, classes, exceptions and errors.

B.3.1. Class: PaddedNumString

PaddedNumString is a wrapper for integers that can be used to get zero-padded versions of that integer. By default, integers are padded to 10-character strings, so a value of 123 would be padded to "0000000123". This object is used in two ways within the search application. First, it is used to wrap basic integers into a toStringable object (much as could be done with just an Integer object) so that they can be stored as indexed values by the Lucene indexer. Second, it is used to pad strings for the org.vectorbase.www.search.lookup.NumRangeQuery query parser, which uses a string-based sort. Sorting integers in a stringlike way works fine, as long as the integers are padded, hence this class.
Several constructors exist that take the input number as a String, Integer, or int.

The fields, constructors, and methods of the class follow.

- **Fields**
  - `protected static final int DEFAULT_LENGTH = 10` - the default length of the resulting padded integer
  - `protected int value` - the integer value that is being padded
  - `protected int length` - the length of the resulting padded string, if one has been set; otherwise, `DEFAULT_LENGTH` is used.

- **Constructors**
  - `public PaddedNumString(java.lang.String value)` - Given a String containing an int that can be parsed out using `Integer.valueOf()`, create a new `PaddedNumString` with the padding length set to `DEFAULT_LENGTH`.
  - `public PaddedNumString(java.lang.String value, int length)` - Given a String containing an int that can be parsed out using `Integer.valueOf()`, and a padding length, create a new `PaddedNumString` with the given padding length.
  - `public PaddedNumString(java.lang.Integer value)` - Given an Integer, create a new `PaddedNumString` with the padding length set to `DEFAULT_LENGTH`.
  - `public PaddedNumString(int value)` - Given an int, create a new `PaddedNumString` with the padding length set to `DEFAULT_LENGTH`.
  - `public PaddedNumString(int value, int length)` - Given an int, and a padding length, create a new `PaddedNumString` with the given padding length.

- **Methods**
  - `public int getLength()` - Gets the current padding length; defaults to 10 unless set otherwise during construction or by `setLength`.
  - `public void setLength(int length)` - Sets the padding length; fetching the padded number after this will return the number padded with the new length.
  - `public java.lang.String paddedString()` - Returns the value, padded to the length
  - `public java.lang.String toString()` - Returns the value as a String.
  - `private java.lang.String zeroPadNumber(int inval, int length)` - Private method for actually padding an integer. Given an int value and an int length, return a zero-padded String with the value padded on the left by zeros ("0").
• public int intValue() - Returns the integer value of this PaddedNumString.


Package org.vectorbase.www.search.indexer contains the following interfaces, classes, exceptions and errors.

B.4.1. Interface: Aggregator

Aggregator is the interface for aggregators that generate Lucene
org.apache.lucene.document.Documents. All aggregators must implement the aggregate method. Aggregators are expected to take an IndexWriter and add Lucene Documents to it.

• Methods


B.4.2. Class: Indexer

Indexer generates a new Lucene index file from the database. It creates a new Lucene IndexWriter, connects to the Chado database and VectorBase user interface database, and then passes both to ChadoFeatureAggregator to create the indexed content.

When aggregation has finished, the index file is saved. The fields, constructors, and methods of the class follow.

• Fields

• private static org.apache.lucene.index.IndexWriter writer - The Lucene IndexWriter that will be used to generate the index file.

• private static java.lang.String INDEXPATH - The relative path to the index file. The absolute path is generated by appending this to the indexpath value from the configuration in config.xml.

• Constructors

• public Indexer() - Default constructor.
• Methods

• public static void main(java.lang.String[] argv) - Generates the index file. If there are to be new indices, the code to create them should be added here. The location of the output file, and the connection information for the Chado and VectorBase user interface database are read from the configuration file config.xml.

B.4.3. Exception: IndexerException

IndexerException is a simple extension of the Exception class that has no added functionality. In effect, we are merely renaming the Exception class for better handling in the Indexer.

• Constructors

• public IndexerException(java.lang.String message) - Initialize the parent Exception with the given message.


Package org.vectorbase.www.search.indexer.chado contains the following interfaces, classes, exceptions and errors.

B.5.1. Class: ChadoFeatureAggregator

ChadoFeatureAggregator is the implementation of {Aggregator} used to index genomic features from a Chado database. It is responsible for opening a connection to a Chado database (given connection information) and for creating Lucene Document objects to add to the given IndexWriter. The fields, constructors, and methods of the class follow.

• Fields

• private java.util.HashMap chadoInfo - Connection information for the Chado database. The Chado database is the source for all of the data that will be indexed. The chadoInfo hash should have three keys:
  • url - JDBC connection URL in the format jdbc:postgresql://host/chado_database
  • username - Database username (a read-only account is fine)
• password – Database password

• private java.util.HashMap.vectorbase_uiInfo - Connection information for the VectorBase UI database. The UI database is used to read the organism abbreviations for the website, which are then used during creation of URLs for features. The vectorbase_uiInfo hash should have three keys:

  • url - JDBC connection URL in the format
    jdbc:postgresql://host/ui_database

  • username - Database username (a read-only account is fine)

  • password – Database password

• Constructors

  • public ChadoFeatureAggregator(java.util.HashMap chado, java.util.HashMap vectorbase_ui) throws org.vectorbase.www.search.indexer.IndexerException - Construct a new ChadoFeatureAggregator given database connection information. Attempts to load the PostgreSQL JDBC driver. If the driver cannot be loaded, an exception is thrown. Note that this function does not actually connect to the databases, it merely loads the driver and caches the database connection information.

• Methods

  • public int aggregate(org.apache.lucene.index.IndexWriter writer) throws org.vectorbase.www.search.indexer.IndexerException, java.io.IOException - Aggregate Chado genomic features and add them to the given IndexWriter. This function first opens the connection to the Chado and VectorBase UI databases. If there is a problem doing so for either, an IndexerException is thrown. If the database connections are established, then actual aggregation begins.

Four queries are initialized:

  • basicFeaturesStatement - Selects all of the basic genomic features with a cvterm_id matching the type given. Includes feature id, name, location, length, cvterm type name, organism genus and species, and analysis name, if any.

  • featureRelationshipStatement - Selects the parent (via the feature_relationship table) of the given feature. Used to get the gene that is the parent of a transcript or protein.

  • featurePropertyStatment - Selects the feature property value (from the featureprop table) for a given feature and feature property type.

  • imageStatment - Select any image references (filename, definition) associated with a given feature.
- **GOStatement**: Select any GO terms (name and definition) associated with a given feature.

For each feature of type gene, chromosome_arm, golden_path, mRNA, protein, pseudogene, contig, supercontig, BAC_clone, microarray_oligo, and predicted_transcript, fetch the feature information. Features are then added to the index. Each `Document` object added to the index can have multiple fields associated with it (for example, name and cvterm), and each field can have multiple terms (for example, multiple names terms associated with a single feature). The following fields are indexed:

- **name**: the name of the feature; if it matches the NAME0000#### EnsEMBL gene ID format, it is indexed both as the original name and as the components NAME and ####. The name is also used to index the `uniquename` and any GO terms associated with the feature.
- **image**: any number of image filenames associated with the feature through the `feature_image` table
- **archived**: if set, indicates that the feature is an EnsEMBL archive feature (replaced by a newer feature)
- **start**: the start location of the feature (fmin)
- **end**: the end location of the feature (fmax)
- **length**: the sequence length of the feature (seqlen)
- **chromosome**: the chromosome the feature lies on
- **organism**: the organism genus and species, concatenated
- **analysis**: any analysis name associated with the feature
- **gene**: the name of the parent gene of the feature (if any)
- **link**: the default URL for the feature (to GeneView, etc.)
- **description**: the description of the parent feature, if any

Once all of the fields have been added to the `Document`, they are pushed onto the `IndexWriter` with `IndexWriter.addDocument(Document)`. Once all of the features have been indexed, the database connections are closed and the number of indexed documents is returned.
B.5.2. Class: ChadoFeatureAnalyzer

ChadoFeatureAnalyzer is an extension of the Analyzer class for tokenizing and analyzing Chado genomic features. It uses a ChadoTokenizer (instead of StandardTokenizer) and the standard Lucene LowerCaseFilter. The fields, constructors, and methods of the class follow.

- **Constructors**
  - public ChadoFeatureAnalyzer() - Default constructor.

- **Methods**
  - public org.apache.lucene.analysis.TokenStream tokenStream(java.lang.String fieldName, java.io.Reader reader) - Given a Reader containing a string, tokenize it with ChadoTokenizer and lowercase all of the tokens using LowerCaseFilter. Return the resulting TokenStream.
  
  - public org.apache.lucene.analysis.TokenStream queryTokenStream(java.lang.String fieldName, java.io.Reader reader) - Given a Reader containing a string, tokenize it with ChadoTokenizer and lowercase all of the tokens using LowerCaseFilter. Return the resulting TokenStream. This function mirrors tokenStream, but could theoretically change in the future. Whereas tokenStream is used by the indexer, queryTokenStream is used by the search engine clients to tokenize search terms rather than text being indexed.

  - public int getPositionIncrementGap(java.lang.String fieldName) - From Analyzer: "Invoked before indexing a Field instance if terms have already been added to that field. This allows custom analyzers to place an automatic position increment gap between Field instances using the same field name. The default value position increment gap is 0. With a 0 position increment gap and the typical default token position increment of 1, all terms in a field, including across Field instances, are in successive positions, allowing exact PhraseQuery matches, for instance, across Field instance boundaries." In our case, the token position increment gap is 0, so we need to set the field position increment gap to 1 so that the sum is still 1 and we increment our field position.


Package org.vectorbase.www.search.indexer.cv contains the following interfaces, classes, exceptions and errors.
B.6.1. Class: CVAggregator

CVAggregator is the implementation of Aggregator used to index controlled vocabulary terms from a Chado database. It is responsible for opening a connection to a Chado database (given connection information) and for creating Lucene Document objects to add to the given IndexWriter. The fields, constructors, and methods of the class follow.

- **Fields**
  - private java.util.HashMap chadoInfo - Connection information for the Chado database. The Chado database is the source for all of the controlled vocabulary terms that will be indexed. The chadoInfo hash should have three keys:
    - url - JDBC connection URL in the format 
      jdbc:postgresql://host/chado_database
    - username - Database username (a read-only account is fine)
    - password - Database password

- **Constructors**
  - public CVAggregator(java.util.HashMap chado) throws 
    org.vectorbase.www.search.indexer.IndexerException - Construct a new CVAggregator given database connection information. Attempts to load the PostgreSQL JDBC driver. If the driver cannot be loaded, an exception is thrown. Note that this function does not actually connect to the databases, it merely loads the driver and caches the database connection information.

- **Methods**
  - public int aggregate(org.apache.lucene.index.IndexWriter writer) throws 
    org.vectorbase.www.search.indexer.IndexerException, java.io.IOException - Aggregate Chado genomic features and add them to the given IndexWriter. This function first opens the connection to the Chado and VectorBase UI databases. If there is a problem doing so for either, an IndexerException is thrown. If the database connections are established, then actual aggregation begins. Two queries are initialized.
    - CVStatement - Selects a controlled vocabulary ID (cv.cv_id) given a controlled vocabulary name.
    - CVTermStatement - Selects all of the controlled vocabulary term names and IDs (cvterm.name, cvterm.cvterm_id) for a given controlled vocabulary ID.
For each GO controlled vocabulary (biological_process, molecular_function, cellular_component), fetch all of the CV terms. For each term, create a Lucene {Document} and add two fields: an indexed name for the term name and a non-indexed id for the cvterm_id.

Once all of the terms have been indexed, the database connections are closed and the number of indexed documents is returned.


Package org.vectorbase.www.search.lookup contains the following interfaces, classes, exceptions and errors.

B.7.1. Class: Lookup

Lookup is a test class for the Search class. As a standalone application, it's pretty worthless, but it is a useful tool for debugging indexing and searching problems. In most cases, you'll want to be running it from inside Eclipse or another IDE. All it does is create a new instance of itself in the main function and then call find, which should be edited to perform your test search(es) of interest. The fields, constructors, and methods of the class follow.

- Constructors
  - public Lookup() - Default constructor.

- Methods
  - public static void main(java.lang.String[] argv) - Main method for execution; creates a new instance of this class and then runs #find. Nothing worth changing here, move along.
  - public java.lang.String find() - Does whatever you make it do. Generally, this involves creating creating a {Search} object and passing some arguments to Search.findInChado(String, Boolean, String, Integer, Integer, String[], String[]) and then printing the results to the console.
B.7.2. Class: NumRangeQuery

NumRangeQuery is a numeric range query for Lucene that can work with large ranges. This version is adapted from NumRangeQuery by D. Gilbert. (See authorship.) It provides the following capabilities.

- Full, primitive numeric range (integer) query, can handle large ranges. Unlike lucene.search.RangeQuery which uses Boolean short-cut for each number in a range and blows up on any range > 1.1..100 or so.
- Makes a BitSet of documents within range once, and feeds back to Searcher through score(HitCollector c, final int end) as often as called.
- As per RangeQuery, query semantics are NumField:[1 100] (inclusive) NumField:{1 100} (exclusive)
- Implicit assumptions are:
  - Term first, last have integer values, as does indexed field
  - Indexed field needs to be recoded for alphanumeric lexical sorting; e.g. 2 -> 0000000002, 10 -> 0000000010, -3 -> -0000000003 (as with org.vectorbase.www.search.PaddedNumString)
- Should add options to handle long, float, double numbers
- Fields
  - public static boolean debug
  - public static java.io.PrintStream logp
  - org.apache.lucene.index.Term firstTerm
  - org.apache.lucene.index.Term lastTerm
  - java.lang.String rangefield
  - boolean inclusive
  - int iStart
  - int iStop
  - protected java.util.BitSet docbits
  - protected org.apache.lucene.index.IndexReader lastreader

- Constructors
• public NumRangeQuery(org.apache.lucene.index.Term first, org.apache.lucene.index.Term last, boolean inc) - Construct a new NumRangeQuery with the start and end terms of a range. The start and end are Terms, meaning that they have both a field name (which should be the name for both) and a value (formatted for lexical sorting).

• Methods

• protected static final boolean DEBUG()

• public org.apache.lucene.index.Term getTerm() - Returns the start term of the range query.

• protected int getNum(java.lang.String val) - Returns the numeric (int) value of the input.

• protected final int termCompare(org.apache.lucene.index.Term term) - Compare an input term to the range. If the term falls within the range, return 0. If it falls after the range, return 1. If it falls before the range, return -1.

• public java.lang.String toString(java.lang.String field) - Prints a user-readable version of this query.

• public boolean equals(java.lang.Object o) - Overloads Object.equals equality operator; returns true if the start and end term of this query match the start and end term of another NumRangeQuery.

• public int hashCode() - Returns a hash code value for this object.

• protected org.apache.lucene.search.Weight createWeight(org.apache.lucene.search.Searcher searcher)

• public java.util.BitSet getRangeDocBits(org.apache.lucene.index.IndexReader reader) throws java.io.IOException

B.7.3. Class: Search

Search is the class that creates actual searches against the index and returns the results. All of the public methods are published over SOAP via Axis. It is deployed with:

```xml
<deployment xmlns="http://xml.apache.org/axis/wsdd/
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
  <service name="VectorbaseSearch" provider="java:RPC">
    <parameter name="allowedMethods" value="*"/>
    <parameter name="scope" value="application"/>
    <parameter name="className" value="org.vectorbase.www.search.lookup.Search"/>
    <typeMapping deserializer=""
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```
The fields, constructors, and methods of the class follow.

- **Fields**
  - public static java.util.Hashtable readers - Any number of IndexReaders initialized with an index file. Currently, this is only the Chado reader. More can be added in reloadConfig.
  - private static java.util.Hashtable config - The search configuration, loaded from config.xml. Each key/value pair is created from a single element in the configuration file. Static so that data can persist between instances within the Tomcat server. (Not sure if it really needs to be static or not, depending on Tomcat configuration.)

- **Constructors**
  - public Search() - Constructor for Search; loads the configuration using reloadConfig. Static so that data can persist between instances within the Tomcat server. (Not sure if it really needs to be static or not, depending on Tomcat configuration.)

- **Methods**
  - public java.util.Hashtable getConfig() - Returns the configuration hash; exists so that the configuration can be checked remotely over SOAP.
  - public java.lang.String reloadIndices() - Reloads the index files for every {IndexReader} in readers. Use this to update the search engine indices without restarting Tomcat.
  - public java.lang.String reloadConfig() - Reloads the configuration file. First attempts to read the packed configuration from config.dat. If that doesn't exist, then it attempts to read the configuration from config.xml and then generates the packed config.dat file. If neither exist, an error is printed to the console.

The Hashtable config is populated from the configuration file. For each element inside the root <searchconfig> element, the key is the name of the element (for example, ”indexpath” for the <indexpath> tag), and the value is the content of the
element. Note that this isn't nearly fancy enough to convert nested elements to nested Hashtables; one level only, please.

Since this function initially tries to read from config.dat, and thus will ignore the contents of config.xml, it is necessary to delete config.dat whenever the config.xml file is updated.

Once the configuration has been read, the new IndexReader(s) are created. (Right now, it's just the "Chado" one. This is effectively the same as reloadIndices, so you don't need to run that.

- private org.vectorbase.www.search.lookup.chado.ChadoQueryController prepareQueryController(java.lang.String controllerType, java.lang.String term, java.lang.String organism, java.lang.Integer start, java.lang.Integer end, java.lang.String[] types, java.lang.String[] analyses, java.lang.String chromosome) throws org.apache.axis.AxisFault - Creates a ChadoQueryController (or child of that class) and then adds Query components to it to build a query. Each of the following are added, assuming they are not set to null in the arguments:
  - organism - added as a term that MUST occur
  - type - any one of the types SHOULD occur, but one of them MUST
  - start and end - the features found MUST fall within the range provided
  - analysis - any one of the analysis names SHOULD occur, but one of them MUST
  - chromosome - added as a term that MUST occur


findInChado(String, Boolean, String, Integer, Integer, String[], String[], String, Integer) that starts at a default offset of 0. (Returns the first 10 results of a search.)

B.7.4. Class: SearchResult

SearchResult is the most basic search result, extended by anything that needs more fields. It is hopefully generic for any sort of search result that you want to return. It follows the JavaBean convention of having an accessor for every variable. This is required by Axis to turn an object into a SOAP object using a org.apache.axis.encoding.Serializer. The fields, constructors, and methods of the class follow.

- Fields
  - private java.lang.String name - the name of the item found
  - private java.lang.String text - any additional descriptive text for the item found
  - private java.lang.String link - a link (URL) to a display for the item found
  - private java.lang.String type - the type of feature found; for Chado this is usually the cvterm
  - private java.lang.Integer rank - the order of display for the features (this is just a suggestion, and it may be entirely arbitrary)
  - private java.lang.Integer hitnum - the current result number (as ordered by rank)
  - private java.lang.Integer numhits - the total number of results in the set that this result is a part of

- Constructors
  - public SearchResult(java.lang.String name, java.lang.String text, java.lang.String link, java.lang.String type, java.lang.Integer rank, java.lang.Integer hitNum, java.lang.Integer numHits) - Creates a SearchResult given a value for each of the fields in a SearchResult. The name, text, link, and type fields are all display fields, in that they are to be used by the client as actual content for displaying search results.

The rank, hitnum, and numhits fields are used for sorting hits and displaying summary statistics. The rank (display order) is the determined by the query controller. The hit number is the current number of the hit out of all the hits returned by the search (not just the 10 or so returned at a time). Likewise, the number of hits
should be the total number of hits returned by the search (yes, this data is repeated for every search result, but there's no other easy way to get the total number of hits if you're only getting 10 at a time in).

• Methods
  • public java.lang.String getName() - Returns the name of the item found.
  • public java.lang.String getText() - Returns the descriptive text of the item found.
  • public java.lang.String getLink() - Returns the link to the display for the item found.
  • public java.lang.String getType() - Returns the type of the item found.
  • public java.lang.Integer getRank() - Returns the display order of the item found.
  • public java.lang.Integer getHitNum() - Returns the number of the current result, as ordered by #{rank}.
  • public java.lang.Integer getNumHits() - Returns the total number of results in the set that this result is part of.
  • public java.lang.String toString() - Returns a string representation of this result. Formatted like:

    name(rank)
    text
    link ... type ... hitnum

B.7.5. Class: WildcardTermFilter

WildcardTermFilter is an extension of the basic Lucene Filter class for filtering a set of search results based on a search term containing a wildcard. Uses WildcardTermEnum which is already used by WildcardQuery. The difference between the filter and the query is that the filter can be run on a set of search results after a query has been run. This is necessary in a couple of cases because a WildcardQuery where multiple terms are ANDed together as part of a larger query will cause the whole thing to fail because of too many
results returned. (See org.vectorbase.www.search.lookup.chado.search.) The fields, constructors, and methods of the class follow.

- **Fields**
  - `private org.apache.lucene.index.Term term` - the term containing a wildcard character that is being filtered against. The wildcard characters are "?" (single character) and "*" (many character).

- **Constructors**
  - `public WildcardTermFilter(org.apache.lucene.index.Term term)` - Constructs a new `WildcardTermFilter` with the given Lucene `Term`. The term must contain a wildcard character or the `WildcardTermEnum` will throw an exception when `bits` is called.

- **Methods**
  - `public java.util.BitSet bits(org.apache.lucene.index.IndexReader reader)` - Returns a `BitSet` masking the features that do not match the wildcard `Term` used to create this filter.


Package org.vectorbase.www.search.lookup.chado contains the following interfaces, classes, exceptions and errors.

B.8.1. Class: ChadoArchiveIdSearchResult

ChadoArchiveIdSearchResult extends ChadoSearchResult to add a flag indicating that the feature is archived. The inherited fields are `start`, `end`, `organism`, `chromosome`, and `length`. These fields are used by the web client to generate links to the appropriate genomic regions for ContigView. ChadoArchiveIdSearchResults are created by ChadoQueryController, although other subclasses may create them. The archived field is a flag indicating whether or not the feature's `isObsolete` field was set in the database. The fields, constructors, and methods of the class follow.
• Fields
  • private boolean archived – the archived flag for this result

• Constructors
    - Creates a ChadoArchiveIdSearchResult given a value for each of the fields in a ChadoArchiveIdSearchResult. The name, text, link, and type fields are all display fields, in that they are to be used by the client as actual content for displaying search results.

    The rank, hitnum, and numhits fields are used for sorting hits and displaying summary statistics. The rank (display order) is the determined by the query controller. The hit number is the current number of the hit out of all the hits returned by the search (not just the 10 or so returned at a time). Likewise, the number of hits should be the total number of hits returned by the search (yes, this data is repeated for every search result, but there's no other easy way to get the total number of hits if you're only getting 10 at a time in).

    The start, end, organism, and chromosome fields locate the Chado feature on the genome of an organism, and are used to generate links to appropriate regions of the genome in ContigView. The length is the length of the Chado feature.

    The archived field is used to indicate that this is an archived feature. Somewhat redundant, but since this field is encoded into the SOAP object sent to the client, it's an easy way to figure out the result type even if the SOAP object is a collection of generic ChadoSearchResult objects.

• Methods
  • public java.lang.String toString() - Returns a string representation of this result. Formatted like:

    name(rank)
    text
    link ... type ... hitnum

    Position: start, end (length) on chromosome arm or contig chromosome of organism organism

    with class ChadoArchiveIdSearchResult with archived "archived"
• public boolean getArchived() - Returns the value of the is_obsolete flag. This will indicate that this is, for sure, an archived feature.

B.8.2. Class: ChadoBACImageQueryController

ChadoBACImageQueryController is an extension of {ChadoQueryController} used to build a single query from a set of {BooleanClause}s for finding "BAC_clone" features with images. It is also responsible for running the query against the index and generating the {ChadoBACImageSearchResult}s for the results of that query. The fields, constructors, and methods of the class follow.

• Constructors

• public ChadoBACImageQueryController(org.apache.lucene.index.IndexReader reader) – Build a query controller with the index attached to the IndexReader reader.

• Methods

• public org.vectorbase.www.search.lookup.chado.ChadoSearchResult[] search(org.apache.lucene.search.Query query, java.lang.String term, java.lang.Boolean andTerms, java.lang.Integer resultOffset) throws org.apache.axis.AxisFault - Runs a search against the IndexReader this ChadoBACImageQueryController was created with using the query passed in as query. The input query will then be tied to a TermQuery created from term, and the input query and term query will be joined with a boolean org.apache.lucene.search.BooleanClause.Occur#MUST. Expects that part of query is a query against the type field that requires the results to have cvterms of type "BAC_clone". Adds its own query that requires that there be an entry in feature_image and an associated image filename.

Returns the first 10 results after the resultOffsetth position.

First, a new IndexSearcher and SortField are created; the IndexSearcher to run the actual search, and the SortField to ensure that results are returned from earliest to latest on the genome. Next, if there is no term, just run the query in query under the assumption that org.vectorbase.www.search.lookup.Search.prepareQueryController has required enough fields to ensure a somewhat limited set of results.

If there is a term, it is tokenized using ChadoFeatureAnalyzer. If there are multiple tokens and andTerms is true, the tokens are added to they query as BooleanClauses with the org.apache.lucene.search.BooleanClause.Occur#MUST operator. If
andTerms is false, they are added with the 
org.apache.lucene.search.BooleanClause.Occur#SHOULD operator. For each token 
in the term, if it is contains a "?" or a "*", then it is assumed to be a wildcard token 
and is added as a WildcardQuery or WildcardTermFilter. A WildcardQuery is used if 
the terms are being ORed for speed, but a WildcardTermFilter is used to AND 
queries to allow more features to be returned (WildcardQuery fails due to restrictions 
within Lucene once we get past 100 or so results).

Once the term has been tokenized, it and the input query are tied together with a 
boolean MUST. Unlike ChadoQueryController another query is added at this point 
that requires the presence of an entry in the image field. After this query is added, 
the query is passed to the IndexSearcher along with the SortField created at the top 
of the function. Out of the resulting array of hits, the results from resultsOffset to 
resultOffset + 10 are parsed from a Lucene Document into one of several 
ChadoSearchResult types.

The result Document is always transformed into a ChadoBACImageSearchResult, even 
if it isn't an image of a BAC hybridization, as long as there is an image associated 
with the feature. It is therefore necessary to make sure that there is a component of 
the query forcing the type field to "BAC_clone". This is done by 
org.vectorbase.www.search.lookup.Search#findBACImageInChado, but still, 
beware.

B.8.3. Class: ChadoBACImageSearchResult

ChadoBACImageSearchResult extends ChadoSearchResult to add a flag indicating 
that the feature has an image. The inherited fields are are start, end, organism, 
chromosome, and length. These fields are used by the web client to generate links to the 
appropriate genomic regions for ContigView. ChadoBACImageSearchResults are created by 
ChadoQueryController and ChadoBACImageQueryController, although other subclasses may 
create them. The image field is a flag indicating whether or not the feature has an 
associated image in the feature_image table. The fields, constructors, and methods of the 
class follow.

• Fields
  • private boolean image – the image flag for this result
• Constructors

  • public ChadoBACImageSearchResult(java.lang.String name, java.lang.String text, java.lang.String link, java.lang.String type, java.lang.Integer rank, java.lang.Integer hitNum, java.lang.Integer numHits, java.lang.Integer start, java.lang.Integer end, java.lang.Integer length, java.lang.String organism, java.lang.String chromosome, boolean image) - Creates a ChadoBACImageSearchResult given a value for each of the fields in a ChadoBACImageSearchResult. The name, text, link, and type fields are all display fields, in that they are to be used by the client as actual content for displaying search results.

  The rank, hitnum, and numhits fields are used for sorting hits and displaying summary statistics. The rank (display order) is the determined by the query controller. The hit number is the current number of the hit out of all the hits returned by the search (not just the 10 or so returned at a time). Likewise, the number of hits should be the total number of hits returned by the search (yes, this data is repeated for every search result, but there's no other easy way to get the total number of hits if you're only getting 10 at a time in).

  The start, end, organism, and chromosome fields locate the Chado feature on the genome of an organism, and are used to generate links to appropriate regions of the genome in ContigView. The length is the length of the Chado feature.

  The image field is used to indicate that this is a feature with an associated feature_image entry. Somewhat redundant, but since this field is encoded into the SOAP object sent to the client, it's an easy way to figure out the result type even if the SOAP object is a collection of generic ChadoSearchResultS.

• Methods

  • public boolean getImage() - Returns the value of the image flag. This will indicate that this is, for sure, an image feature.

  • public java.lang.String toString() - Returns a string representation of this result. Formatted like:

    name(rank)
    text
    link ... type ... hitnum
    Position: start, end (length) on chromosome arm or contig chromosome of organism organism
    with class ChadoBACImageSearchResult with image "image"
B.8.4. Class: ChadoMicroArraySearchResult

ChadoMicroArraySearchResult extends ChadoSearchResult to add a flag indicating that the feature is a microarray spot. The inherited fields are are start, end, organism, chromosome, and length. These fields are used by the web client to generate links to the appropriate genomic regions for ContigView. ChadoMicroArraySearchResults are created by {ChadoQueryController}, although other subclasses may create them. The microarray field is a flag indicating that the feature's cvterm is "microarray_oligo". The fields, constructors, and methods of the class follow.

• Fields
  • private boolean microarray – the microarray flag for this result

• Constructors
  • public ChadoMicroArraySearchResult(java.lang.String name, java.lang.String text, java.lang.String link, java.lang.String type, java.lang.Integer rank, java.lang.Integer hitNum, java.lang.Integer hitNum, java.lang.Integer numHits, java.lang.Integer start, java.lang.Integer end, java.lang.Integer length, java.lang.String organism, java.lang.String chromosome, boolean microarray) - Creates a ChadoMicroArraySearchResult given a value for each of the fields in a ChadoMicroArraySearchResult. The name, text, link, and type fields are all display fields, in that they are to be used by the client as actual content for displaying search results.

  The rank, hitnum, and numhits fields are used for sorting hits and displaying summary statistics. The rank (display order) is the determined by the query controller. The hit number is the current number of the hit out of all the hits returned by the search (not just the 10 or so returned at a time). Likewise, the number of hits should be the total number of hits returned by the search (yes, this data is repeated for every search result, but there's no other easy way to get the total number of hits if you're only getting 10 at a time in).

  The start, end, organism, and chromosome fields locate the Chado feature on the genome of an organism, and are used to generate links to appropriate regions of the genome in ContigView. The length is the length of the Chado feature.

  The microarray field is used to indicate that this is a microarray feature. Somewhat redundant, but since this field is encoded into the SOAP object sent to the client, it's an easy way to figure out the result type even if the SOAP object is a collection of generic ChadoSearchResultS.
• Methods
  • public boolean getMicroarray() - Returns whether or not this feature's cvterm was "microarray_oligo". This will indicate that this is, for sure, an archived feature.
  • public java.lang.String toString() - Returns a string representation of this result. Formatted like:

  name(rank)
  text
  link ... type ... hitnum

  Position: start, end (length) on chromosome arm or contig chromosome of organism organism

  with class ChadoMicroArraySearchResult with microarray "microarray"

B.8.5. Class: ChadoQueryController

ChadoQueryController is used to build a single query from a set of BooleanClauses.

It is also responsible for running the query against the index and generating the ChadoSearchResults for the results of that query. It is extended to run slightly different sets of queries or to return results of a more specific type (such as org.vectorbase.www.search.lookup.chado.ChadoBACImageSearchResult). The fields, constructors, and methods of the class follow.

• Fields
  • protected org.apache.lucene.index.IndexReader reader - the IndexReader to search against
  • protected org.apache.lucene.search.BooleanQuery query - the query to run. Although there are many types of query, they can all be encapsulated within a BooleanQuery.

• Constructors
  • public ChadoQueryController(org.apache.lucene.index.IndexReader reader) - Creates a new ChadoQueryController with the given IndexReader. The IndexReader will be used as the source to search against.
• Methods

  • public void add(org.apache.lucene.search.BooleanClause query) - Adds a new BooleanClause to the query that will be run. A BooleanClause includes a boolean term (org.apache.lucene.search.BooleanClause.Occur#MUST, org.apache.lucene.search.BooleanClause.Occur#SHOULD, or org.apache.lucene.search.BooleanClause.Occur#MUST_NOT) and another query clause. The other query clause can be another BooleanClause, or any other query (such as a TermQuery).

  BooleanClause queries should have multiple clauses within them to define the set of requirements to be applied. A BooleanClause with a single entry where the boolean is MUST and the query is a TermQuery for the name field equal to "BAC100" will find features named "BAC100". A BooleanClause with two entries both
  BooleanQuery.add(org.apache.lucene.search.BooleanClause)ed as a MUST, with one where (field = name, term = "BAC100") and the other (field = type, term = "BAC_clone") will find only features that are named "BAC100" of type "BAC_clone". For more information, see the documentation for Query and BooleanClause.

  NOTE: Don't add the final TermQuery with this function; you'll want to pass it in as the second argument to search.

  • public org.vectorbase.www.search.lookup.chado.ChadoSearchResult[] search(org.apache.lucene.search.Query query, java.lang.String term, java.lang.Boolean andTerms, java.lang.Integer resultOffset) throws org.apache.axis.AxisFault - Runs a search against the IndexReader this ChadoQueryController was created with using the query passed in as query. The input query will then be tied to a TermQuery created from term, and the input query and term query will be joined with a boolean
  org.apache.lucene.search.BooleanClause.Occur#MUST.

  Returns the first 10 results after the resultOffsetth position.

  First, a new IndexSearcher and SortField are created; the IndexSearcher to run the actual search, and the SortField to ensure that results are returned from earliest to latest on the genome. Next, if there is no term, just run the query in query under the assumption that
  org.vectorbase.www.search.lookup.Search.prepareQueryController has required enough fields to ensure a somewhat limited set of results.

  If there is a term, it is tokenized using ChadoFeatureAnalyzer. If there are multiple tokens and andTerms is true, the tokens are added to the query as BooleanClauses with the org.apache.lucene.search.BooleanClause.Occur#MUST operator. If andTerms is false, they are added with the
  org.apache.lucene.search.BooleanClause.Occur#SHOULD operator. For each token in the term, if it is contains a "?" or a "*", then it is assumed to be a wildcard token.
and is added as a WildcardQuery or WildcardTermFilter. A WildcardQuery is used if the terms are being ORed for speed, but a WildcardTermFilter is used to AND queries to allow more features to be returned (WildcardQuery fails due to restrictions within Lucene once we get past 100 or so results).

Once the term has been tokenized, it and the input query are tied together with a boolean MUST, and then the query is passed to the IndexSearcher along with the SortField created at the top of the function. Out of the resulting array of hits, the results from resultsOffset to resultOffset + 10 are parsed from a Lucene Document into one of several ChadoSearchResult types.

If the result Document contains anything in the image field, it is transformed into a ChadoBACImageSearchResult. If it contains anything in the archived field, it is transformed into a ChadoArchiveIdSearchResult. If it contains anything in the microarray field, it is transformed into a ChadoMicroArraySearchResult. Any other results are transformed into generic ChadoSearchResult objects. Since all of these objects extend from ChadoSearchResult, they are all put into a ChadoSearchResult[] array, which is returned.

- public java.lang.String toString() - Returns the a string containing the query (converted to a string).
- public org.apache.lucene.search.BooleanQuery query() - Returns the current query stored by this controller.
- public void setQuery(org.apache.lucene.search.BooleanQuery query) - Set the query for this query controller. If you want to reuse a query controller, just set this to an empty BooleanQuery. If you set this to something besides a BooleanQuery, make sure that it supports the add(BooleanClause) method (unlikely!).

**B.8.6. Class: ChadoSearchResult**

ChadoSearchResult extends SearchResult to add Chado genomic feature fields.

These are start, end, organism, chromosome, and length. These fields are used by the web client to generate links to the appropriate genomic regions for ContigView.

ChadoSearchResults are created by ChadoQueryController, although other subclasses may create them. As with SearchResult, ChadoSearchResult is expected to provide a generic enough set of fields to cover any feature stored in Chado. The fields, constructors, and methods of the class follow.
• Fields
  • private java.lang.Integer start - the start location of the feature on the chromosome/supercontig
  • private java.lang.Integer end - the end location of the feature on the chromosome/supercontig
  • private java.lang.String organism - the organism this feature belongs to
  • private java.lang.String chromosome - the chromosome on which this feature falls
  • private java.lang.Integer length - the length of this feature, read from feature.seqlen

• Constructors
  • public ChadoSearchResult(java.lang.String name, java.lang.String text, java.lang.String link, java.lang.String type, java.lang.Integer rank, java.lang.Integer hitNum, java.lang.Integer numHits, java.lang.Integer start, java.lang.Integer end, java.lang.Integer length, java.lang.String organism, java.lang.String chromosome) - Creates a ChadoSearchResult given a value for each of the fields in a ChadoSearchResult. The name, text, link, and type fields are all display fields, in that they are to be used by the client as actual content for displaying search results.

The rank, hitnum, and numhits fields are used for sorting hits and displaying summary statistics. The rank (display order) is the determined by the query controller. The hit number is the current number of the hit out of all the hits returned by the search (not just the 10 or so returned at a time). Likewise, the number of hits should be the total number of hits returned by the search (yes, this data is repeated for every search result, but there's no other easy way to get the total number of hits if you're only getting 10 at a time in).

The start, end, organism, and chromosome fields locate the Chado feature on the genome of an organism, and are used to generate links to appropriate regions of the genome in ContigView. The length is the length of the Chado feature.

• Methods
  • public java.lang.Integer getStart() - Returns the start location of the feature on the chromosome/supercontig.
  • public java.lang.Integer getEnd() - Returns the end location of the feature on the chromosome/supercontig.
  • public java.lang.String getOrganism() - Returns the organism this feature belongs to in the format "Genus species" as in "Anopheles gambiae".
• public java.lang.String getChromosome() - Returns the chromosome on which this feature falls.

• public java.lang.Integer getLength() - Returns the length of this feature, read from feature.seqlen.

• public java.lang.String toString() - Returns a string representation of this result. Formatted like:

  name(rank)
  text
  link ... type ... hitnum

  Position: start, end (length) on chromosome arm or contig chromosome of organism organism

  with class ChadoSearchResult-or-subclass
APPENDIX C:

CHAPOLLO DOCUMENTATION

C.1. Module: Chapollo::Main

C.1.1. Summary

Chapollo::Main - Main program code for Chapollo Chado->ChadoXML conversion tool.

C.1.2. Included modules

- Bio::SeqIO
- Bio::SeqIO::chadoxml_apollo
- Chapollo::DB::Chado

C.1.3. Example code listing

```perl
use Chapollo::Main; use Bio::SeqFeature::Generic;
my $chapollo = new Chapollo::Main({
  'dbstr'     => 'dbi:Pg:dbname=chado;host=localhost;',
  'user'      => 'chado_user',
  'password'  => 'chado_password'
});
my $regions = [
  { -seq_id => 'CHROMOSOMEX', -start => 1, -end => 100000 },
  { -seq_id => 'CHROMOSOMEX', -start => 200000, -end => 300000 }
];
my $data_types = [
  'DrosophilaBlast' =>
  { 'type' => 'analysis',
    'filters' =>
  }
];```
C.1.4. Description

This class contains the working code for running the Chapollo database dumper.

Chapollo attempts to connect to a FlyBase/VectorBase-style Chado database, and generates an Apollo-compatible ChadoXML document for a given region, and with given feature types. There are two major types of features:

- **Analysis features**
  - These are displayed in the "evidence" track in Apollo. Chapollo can export all feature types as analysis features. This includes both real analysis features (like DrosophilaBlast) and genes that are not analysis features. The reasoning is that you may want existing genes to be visible in the evidence track and/or the feature track.

- **Features**
  - These types include just genes (along with associated transcripts and exons). The type name is pulled from the gene's `<i>source</i>` featureprop. At VectorBase, this will be one of `<i>ensembl</i>`, `<i>community annotation pipeline</i>`, or `<i>manual annotator</i>`.

  Analysis features can be filtered in several ways to allow easier use in Apollo; these are passed in as part of the `<i>filters</i>` hash. (See example, above.)

- **noise**
  - Noise filtering is used to remove low-scoring hits from the ends of otherwise high-scoring matches. (For example, a BLAST hit may be very good for much of its
length, then have a largish gap and a tiny little low-scoring piece that's just an artifact of the BLAST algorithm.

- **min_gap_size**
  - The minimum distance that a hit must be away from other hits to the same query sequence for it to be considered as noise

- **significance(1)**
  - The significance cutoff. If a hit is far enough away from other hits, and has a significance value less-than-or-equal-to this value, then remove it.

- **identity(1)**
  - The identity cutoff. If a hit is far enough away from other hits, and has an identity value less-than-or-equal-to this value, then remove it.

- **score**
  - Score filtering is used to remove hits scoring below a certain threshold. For example, setting the I<identity> value to 0.001 removes all hits with an e-value <= 0.001. Note that some analysis types to not have score data associated with them; this filter will not touch them and they will be displayed in Apollo as having a score of 0.

- **rawscore**
  - The score cutoff. If a hit has a raw score value less-than-or-equal-to this value, then remove it.

- **significance(2)**
  - The significance cutoff. If a hit has a significance value less-than-or-equal-to this value, then remove it.

- **identity(2)**
  - The significance cutoff. If a hit has an identity value less-than-or-equal-to this value, then remove it.

- **compress_gap**
  - The compression gap filter will join hits that are closer than the compress_gap value apart in order to reduce the load on Apollo and visual clutter. For instance, if the compress_gap is 1, any two hits from the same query sequence that are 1 or 0 bp apart will be joined into a single hit. Since there are frequently many overlapping hits from the same query sequence, this can mean a huge reduction in number of hits (and corresponding increase in Apollo's usability). Note that hits that are combined
will not include any sort of score data. They’ll still get filtered by score appropriately, but you won’t see the actual values in Apollo.

C.1.5. Methods

• new

**Arguments:** $args

The constructor takes a hash containing database connection information a Chado database. The dbstr item holds the database connectionstring, the user entry holds the database username, and the password entry holds the database password.

```perl
$args = {    'chado' => {      'dbstr' => 'dbi:Pg:dbname=chado;host=localhost;',      'user' => 'chado_user',      'password' => 'chado_password'    },    'manual_chado' => {      'dbstr' => 'dbi:Pg:dbname=manual_chado;host=localhost;',      'user' => 'manual_chado_user',      'password' => 'manual_chado_password'    }, }
```

Feature types can be filtered only by selecting a particular instance of a feature to extract. This is done by setting the item field of the feature hash. Only one item filter can be specified for a given feature type. This is primarily useful when you want to select out a particular feature for editing in Apollo. If no item is specified, all features for the given region(s) are exported.

• dump

**Arguments:** $organism, $regions, $data_types

This function is the primary work function for Chapollo, and is responsible for generating hashes of feature data and passing them to the XML writer. It expects three arguments: the organism (genus and species), the regions (chromosome and start/end), and the data sources desired (type and filters). For more explanation of data type format, see the noise, score, and compress_gap sections above.

```perl
my $organism = { 'genus' => 'Anopheles', 'species' => 'gambiae' };  my $regions = [    { -seq_id => 'CHROMOSOMEX', -start => 1, -end => 100000 },    { -seq_id => 'CHROMOSOMEX', -start => 200000, -end => 300000 },  ];  my $data_types = [    [ 'DrosophilaBlast' =>       { 'type' => 'analysis',   'filters' => {    'noise' => { 'min_gap_size' => 50 },       'score' => { 'rawscore' => 100 },       'compress_gap' => 3   }      }    ],    [ 'manual annotator' =>      {        'type' => 'feature', 'item' => 'ManualGeneA'      }    ]  ];
```

**NOTE:** The chromosome/supercontig must be the same for all regions provided. Apollo does not support ChadoXML files containing features from regions on different chromosomes/supercontigs, although it can be coerced into showing disjoint regions on the same chromosome. When multiple regions are specified, the ChadoXML file claims to cover the range from the beginning of the first region to the end of the last region, but features are only exported for the regions specified; the inter-region gaps are left blank.
• **get_residues**

  **Arguments:** $feature_name, $organism_id, $start, $length

  Returns the residues (sequence) for a Chado feature given by $feature_name (generally a chromosome or supercontig) from the given start position for the given length. (Uses the Postgres SUBSTR function; includes the $startth character, and the total resulting length is $length.)

• **get_analysis_features_by_region**

  **Arguments:** $region, $organism_id, $analyses, $filters

  Return a hash of feature hashes in a format appropriate to pass to chadoxml_apollo->write_feature given a Bio::SeqFeature::Generic as theregion to read from, the Chado organism_id, an array of names of the analysis feature types to fetch (matched to the Chado field analysis.program), and a hash of filters in the format:

  ```
  'analysis_name' => {    'noise' => { 'min_gap_size' => 50, etc.},    'score' => { 'rawscore' => 100, etc.},    'compress_gap' => 3  }
  ```

  This function doesn't actually do much of the work of building the feature hash; it relies on get_analysis_features_by_location for that. It is, however, responsible for applying the filters to the features returned by that function. The order of filters is somewhat important... remove_noise is called first, because if we remove low scoring hits first, then call remove noise, some hits that were formerly adjacent to others may suddenly be further away and appear to be noise. remove_low_scoring_hits is called next, because gap compression will remove the score data from hits that are compressed together, making it impossible to tell if they should be removed or not. compress_analyses is called last, because it's the only thing left.

• **get_analysis_features_by_location**

  **Arguments:** $region_name, $organism_id, $start, $end, $analysis_name

  Return a hash of feature hashes (where the key is the uniquename) for all the analysis features for the given $organism_id, $region_name, $start, and $end associated with the analysis $analysis_name. Basically, fetches all the analysis features and hits associated with that analysis feature and returns them in a manner ready for output using chadoxml_apollo->write_feature. An explanation of the feature hash is in order. Basically, for each analysis (e.g. DrosophilaBLAST), there is one entry in the analysis table. For each query sequence tested against the genome, there are one or more match features for each continuous set of hits. For each match feature, there are generally several hit features (often called high scoring pairs or HSPs) for each match of the query sequence vs. the genome within the match feature's region. To some extent, these can be transliterated into gene, transcript, and exons, and they exist in the feature table side-by-side with them, and
are related using feature_relationship, just like ordinary genomic features, except with an additional relationship to an analysis through the analysisfeature table.

- get_transcripts_as_analyses_for_region

  **Arguments:** $region, $organism_id, $feature_types

  Return a hash of feature hashes in a format appropriate to pass to chadoxml_apollo->write_feature given a Bio::SeqFeature::Generic as the region to read from, the Chado organism_id, and an array of names of the feature types to fetch (matched to the feature's cvterm name). Unlike get_analysis_features_by_region, it doesn't really do much more than separate the components of the Bio::SeqFeature into individual arguments for get_transcripts_as_analyses_by_location, which does all the work of generating the feature hashes.

- get_transcripts_as_analyses_by_location

  **Arguments:** $region_name, $organism_id, $start, $end, $feature_type

  Return a hash of feature hashes (where the key is the uniquename) for all the non-analysis genomic features for the given $organism_id,$region_name, $start, and $end where the feature's type_id points to a cvterm of type $feature_type. Basically, fetchestranscripts and associated exons, munging them into a form that makes Apollo treat them as evidence features. The feature hash is in the same format as with get_analysis_features_by_location, except in this case, the analysis and analysisfeature are made up, the match is actually a transcript, and the hits are actually exons. No scores are associated with any of the features.

- get_manual_genes_by_region

  **Arguments:** $region, $organism_id, $feature_types, $items

  Read from the manual_chado database and return a hash of feature hashes in a format appropriate to pass to chadoxml_apollo->write_feature given a Bio::SeqFeature::Generic as the region to read from, the Chado organism_id, an array of names of the feature types to fetch (matched to the feature's cvterm name), and an optional hash limiting the features selected by uniquename - note that you can only specify one item to select out per feature type; not specifying an item selects all features of that type in a region. ($items is there mostly to allowselection of a single gene out for editing by a curator as part of the VectorBase manual annotation pipeline.) $items should be in the format:

  ```
  { 'manual gene' => 'MyManualGENE', etc. }
  ```

  As with the other get_xxx_by_region functions, this function doesn't actually do the work of building the feature hash; it relies on get_manual_genes_by_location for that. It does do some modification of the generated feature hash. In particular, if $items containsan entry for the given feature type, it clears our features with non-
matching names. It also tosses out features with a type that isn't listed in $feature_types. Further, it resizes protein subfeatures to match Apollo's (as opposed to everything else's) expectation that the stop codon will not be included in the CDS sequence. If the feature is on the forward (+) strand, this means subtracting three (3) from the fmax of the protein feature if is_fmax_partial is false. On the reverse (-) strand, it adds three (3) to the fmin of the protein feature if is_fmin_partial is false.

- **get_manual_genes_by_location**
  Arguments: $region_name, $organism_id, $start, $end

  Return a hash of feature hashes (where the key is the uniqname) for all the manually curated genes for the given $organism_id, $region_name, $start, and $end that reside in the manual_chado database. Doesn't do any filtering by type at the database stage under filtering by type at the database stage in order to make the SQL faster; pulling out the features and filtering in Perl is quicker than making multiple database requests (probably).

- **get_genes_by_region**
  Arguments: $region, $organism_id, $feature_types, $items

  Read from the main chado database and return a hash of feature hashes in a format appropriate to pass to chadoxml_apollo->write_feature given a Bio::SeqFeature::Generic as the region to read from, the Chado organism_id, an array of names of the feature types to fetch (matched to the feature's cvterm name), and an optional hash limiting the features selected by uniqname - note that you can only specify one item to select out per feature type; not specifying an item selects all features of that type in a region. ($items is there mostly to allow selection of a single gene out for editing by a curator as part of the VectorBase manual annotation pipeline.) $items should be in the format:

  ```
  { 'ensembl' => 'ENSANGG00000000001', etc. }
  ```

  As with the other get_xxx_by_region functions, this function doesn't actually do the work of building the feature hash; it relies on get_genes_by_location for that. It does do some modification of the generated feature hash. In particular, if $items contains an entry for the given feature type, it clears our features with non-matching names. It also tosses out features with a type that isn't listed in $feature_types. Unlike get_manual_genes_by_region, this function does not modify the length of the protein feature to remove the stop codon. Genes loaded into Chado from Ensembl by default include the stop codon, but there is no differentiation (no is_fmin/max_partial), so we can't make the decision to remove the 3bp at the beginning or end.

- **get_genes_by_location**
  Arguments: $region_name, $organism_id, $start, $end
Return a hash of feature hashes (where the key is the uniquename) for all the genes for the given $organism_id, $region_name,$start, and $end that reside in the main chado database. Doesn't do any filtering by type at the database stage in order to make the SQL faster; pulling out the features and filtering in Perl is quicker than making multiple database requests (probably).

- **remove_noise**

  **Arguments:** $analyses, $filters

  Given a hash of analysis features in $analyses, and a hash of filters in $filters as defined in the SYNOPSIS containing any entries under "noise", remove hits from features that are at least min_gap_size away from other hits and have significance below the filter's significance and/or identity below the filter's identity. See also noise.

- **remove_low_scoring_hits**

  **Arguments:** $analyses, $filters

  Given a hash of analysis features in $analyses, and a hash of filters in $filters as defined in the SYNOPSIS containing any entries under "score", remove hits from features that have a raw score below the filter's rawscore, and/or a significance below the filter's significance, and/or an identity below the filter's identity value. See also score.

- **compress_analyses**

  **Arguments:** $analyses, $filters

  Given a hash of analysis features in $analyses, and a hash of filters in $filters as defined in the SYNOPSIS containing any entries under "compress_gap", combine hits within features that are closer together than the filter's compress_gap. Hit-specific data is removed (such as rawscore, identity, and significance), and the resulting hit starts and ends at the minimum and maximum points of the combined hits. See also compress_gap.

- **remove_null_elements**

  **Arguments:** $feature_hash

  Remove null elements from a hash. When we build feature hashes, various fields are added (such as feature_relationship, an array) that are never populated. If they are allowed to be exported as empty tags into the XML, Apollo's XML parsing fails, so we remove them here. This function removes empty arrays and hash keys with undef values.

- **remove_empty_analysisfeatures**

  **Arguments:** $analyses
A higher-level version of remove_null_elements, this function is used to clean up analysis features that no longer contain any hits because filtering has removed all of them. remove_null_elements is insufficient, because these analysis features still contain the mid-level match features, feature properties, etc. It takes in a hash containing all analyses to be filtered and removes those with an empty feature_relationship array.

- **deep_copy**
  
  **Arguments:** $var
  
  Returns a deep copy of a Perl variable made up of arrays, hashes, and scalars. Doesn't work on user-defined object types. Used internally to create duplicates of feature hashes so that they aren't internally self-referential.

- **chado**
  
  **Arguments:** [$chado]
  
  Returns (or sets and returns, given $chado) the Chapollo::DB::Chado object used to pull features out of the main Chado database.

- **manual_chado**
  
  **Arguments:** [$manual_chado]
  
  Returns (or sets and returns, given $manual_chado) the Chapollo::DB::Chado object used to pull features out of the Chado database used by the manual annotation pipeline.

- **get_feature_hash**
  
  **Arguments:** $db, $feature_id
  
  Given the database name to connect to in $db (either "chado" or "manual_chado"), recursively build and return a hash for the given $feature_id including sub-features related through feature_relationship.

- **get_featurerelationships_hash**
  
  **Arguments:** $db, $feature_id
  
  Given the database name to connect to in $db (either "chado" or "manual_chado"), build and return an array for all of the features related to the given feature, where the given feature_id specifies the object_id in feature_relationship. Uses get_feature_hash to recurse.

- **get_featureprops_hash**
  
  **Arguments:** $db, $feature_id
Given the database name to connect to in $db (either "chado" or "manual_chado"),
build and return an array for all of the feature properties (from featureprop) of the
given $feature_id.

• get_featuredbxsrefs_hash
  Arguments: $db, $feature_id
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  build and return an array for all of the feature dbxrefs(from feature_dbxref) of the
given $feature_id.

• get_dbxref_hash
  Arguments: $db, $dbxref_id
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  build and return a hash for a single dbxref (from the dbxref table) for the given
  $dbxref_id.

• get_featurecvterms_hash
  Arguments: $db, $feature_id
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  build and return an array for all of the feature's cvterms (from feature_cvterm) for the
given $feature_id.

• get_cvterm_hash
  Arguments: $db, $cvterm_id
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  build and return a hash for a single cvterm (from the cvterm table) for the given
  $cvterm_id.

• get_featureloc_hash
  Arguments: $db, $feature_id
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  build and return a hash for a single feature location(from the featureloc table) for the
given $feature_id.

• get_organism_id_by_genus_and_species
  Arguments: $db, $genus, $species
  Given the database name to connect to in $db (either "chado" or "manual_chado"),
  and a genus and species, return the associated organism_id (from the organism
  table).
• get_organism_hash

Arguments: $db, $organism_id

Given the database name to connect to in $db (either "chado" or "manual_chado"), and an organism_id, return an organism hash suitable for use by chadoxml_apollo->write_feature.

C.2. Module: Chapollo::DB::Chado

C.2.1. Summary

Chapollo::DB::Chado - Database connection and data-fetching functionality for Chapollo.

C.2.2. Included modules

• DBI

C.2.3. Example code listing

use Chado::DB::Chado;

$chado = new Chapollo::DB::Chado(
    'dbi:Pg:dbname=chado;host=localhost;', 'username', 'password'
);

# Some functions (singular results) return a single row as a hash
$row_hash = $chado->get_feature_with_organism($feature_name, $organism_id);
print $row_hash->{feature_id} . "\n";

# Some functions (plural results) return a statement handle
$sth = $chado->get_genes_by_location($srcfeature_id, $start, $end);
while ($row = $sth->fetchrow_hashref()) {
    print $row->{uniquename} . "\n";
}

C.2.4. Description

This class is used by Chapollo::Main for access to a Chado database. (It is actually used twice; once for the main Chado database, and once for the Chado database containing manually curated genes.) It provides a set of functions for getting data from the database as
either a hash of a single row (in cases where only a single row's worth of data is necessary, such as requesting an organism_id for a single organism), or as a database statement handle returned by DBI::execute. For more information on how to determine the return type of a given function, see do_query.

Part of the work done by this function is initial preparation of queries during the creation of the constructor. As such, this class is only useful when working with these predefined queries; it is not a general-purpose database class.

C.2.5. Methods

- **new**
  
  **Arguments:** $dbstr, $user, $password
  
  The constructor takes a PostgreSQL connection string (see DBD::Pg), and a username and password that grant read access to the database. It then opens a database connection and prepares all of the queries that will be used by the class. Since many of these queries are run many times, preparation speeds execution significantly.

- **connect**
  
  **Arguments:** $dbstr, $user, $password
  
  Attempt to connect to the given database; only used by the constructor, and just sets some extra options and throws an error on failure.

- **dbh**
  
  Return a reference to the internal DBD::Pg database connection handle.

- **do_query**
  
  **Arguments:** $sth_name, $input
  
  This function wraps the actual query execute calls in some error handling code and provides the option to return either a single row as a hash or a reference to the statement handle that has just been executed.
The $sth_name variable is the key in the hash of queries defined in the constructor. The $input variable is a hash containing two options. The first is $bind_params which is an array of the values to be passed to the query, if any. The second is $return_ssth, which determines the return type. If false, a hash of the first row returned by the query is returned. If true, a reference to the executed query is returned.

- **get_feature_with_organism**

  **Arguments:** $feature_name, $organism_id

  Given a feature name and organism_id, return the feature_id, uniquename, type (cvterm name), and organism genus and species. Used in Chapollo::Main for fetching information about chromosomes/supercontigs given an organism and chromosome/supercontig name.

  Returns a single-row hash.

- **get_analysis_features_by_location**

  **Arguments:** $srcfeature_id, $start, $end, $analysis_id

  Given a feature_id for a chromosome/supercontig, a start and end position, and an analysis_id for a single analysis, return all analysisfeatures for the given $analysis_id that fall on the given region ($start, $end) of the given source feature.

  The fields queried are genus and species (which should be the same for all features), and the "match" features' name, uniquename, timeaccessioned, and timelastmodified (in the SQL, each of these fields is aliased to match_<fieldname>; for instance, name becomes match_name). The "hit" features' attributes included are name, uniquename, timeaccessioned, timelastmodified, rawscore, normscore, significance, identity, fmin, fmax, rank, strand, is_fmin_partial, is_fmax_partial, and locgroup.

  Returns a database statement handle.

- **get_transcripts_by_location**

  **Arguments:** $feature_type, $exon_type, $srcfeature_id, $start, $end

  Given a $feature_type (the cvterm name for the transcript's type; either "mRNA" or "predicted_transcript"), an $exon_type (the cvterm name for the exon's type; either "exon" or "predicted_exon"), a feature id (in $srcfeature_id) for a chromosome/supercontig, and a start and end locations on that source feature, return all matching transcripts as Apollo evidence-track features.

  The fields queried are genus and species (which should be the same for all features), and the value of the transcripts "source" feature property (from featureprop) as "sourcename". The transcripts' fields returned are name, uniquename, timeaccessioned, and timelastmodified. These fields are aliased to match_<fieldname>, as with get_analysis_features_by_location; though these are
transcripts and not analyses, we still prefix with "match." for consistency. The exons' fields returned are feature_id, name, uniquename, timeaccessioned, timelastmodified, fmin, fmax, rank, strand, is_fmin_partial, is_fmax_partial and locgroup. A "rawscore" field is also included for compatibility with functions that work with real analysis features, but it is always set to null.

Returns a database statement handle.

• get_genes_by_location

Arguments: $srcfeature_id, $start, $end

Given a feature_id for a chromosome/supercontig and a $start and $end on that source feature, return all of the genes falling on that region. Unlike get_transcripts_by_location and get_analysis_features_by_location, this function does not return all of the data necessary to describe each gene with transcript and exon. Rather, it just queries the feature_id and uniquename fields of the actual genes. The feature_ids can then be passed to get_feature_by_id to get the rest of the features' information.

Returns a database statement handle.

• get_analysis_info

Arguments: $analysis_name

Given an analysis's program name in $analysis_name, return all of the fields for the matching record in the analysis table. They are analysis_id, name, description, program, programversion, algorithm, sourcename, sourceversion, sourceuri, and timeexecuted.

Returns a hash of the first (hopefully only) record retrieved.

• get_feature_by_id

Arguments: $feature_id

Given a $feature_id, return all of the fields for the associated record in the feature table. They are feature_id, dbxref_id, organism_id, name, uniquename, residues, seqlen, md5checksum, type_id, is_analysis, is_obsolete, timeaccessioned, and timelastmodified.

Returns a hash of the only record retrieved (feature_id is unique).

• get_cvterm_and_cv_by_id

Arguments: $cvterm_id

Given a $cvterm_id, return the name of the cvterm and the name of the CV it is part of as "name" and "cv_name", respectively.
Returns a hash of the only record retrieved (cvterm_id is unique).

- get_dbxref_and_db_by_id

  Arguments: $dbxref_id

  Given a $dbxref_id, return the accession of the dbxref and the name of the DB it is associated with.

  Returns a hash of the only record retrieved (dbxref_id is unique).

- get_feature_dbxrefs_by_id

  Arguments: $feature_id

  Given a $feature_id, return all of the dbxref_ids associated with that feature in the feature_dbxref table. The dbxref_ids can then be passed to get_dbxref_and_db_by_id to get the rest of the dbxrefs' information.

  Returns a database statement handle.

- get_feature_cvterms_by_id

  Arguments: $feature_id

  Given a $feature_id, return all of the cvterm_ids associated with that feature in the feature_cvterm table. The cvterm_ids can then be passed to get_cvterm_and_cv_by_id to get the rest of the cvterms' information.

  Returns a database statement handle.

- get_featureprops_by_id

  Arguments: $feature_id

  Given a $feature_id, return all of the records in the featureprop table associated with that feature. The fields queried are featureprop_id, feature_id, type_id, value, and rank. The get_cvterm_and_cv_by_id function can then be used with the type_id to get the name of the feature property.

  Returns a database statement handle.

- get_featureloc_by_id

  Arguments: $feature_id

  Given a $feature_id, return all fields for the record in the featureloc table that provides location information for that feature. The fields queried are featureloc_id, feature_id, srcfeature_id, fmin, fmax, is_fmin_partial, is_fmax_partial, strand, phase, rank, residue_info, and loggroup.

  Returns a hash of the first (hopefully only) record retrieved.
get_feature_relationships_by_object_id

Arguments: $object_id

Given a feature_id in $object_id, return all fields for all of the records in the feature_relationship table with $object_id as the "parent" (object) feature. The fields queried are feature_relationship_id, subject_id, object_id, type_id, value, and rank. The subject_id can then be used with get_feature_by_id to get the child features.

Returns a database statement handle.

get_organism_by_genus_and_species

Arguments: $genus, $species

Given an organism's $genus and $species, return the organism_id for that organism. This organism_id can then be used with get_organism_by_id or other functions that need the organism_id rather than name.

Returns a hash of the only record retrieved (genus and species are unique).

get_organism_by_id

Arguments: $organism_id

Given an $organism_id, return the genus and species fields for that organism.

Returns a hash of the only record retrieved (organism_id is unique).

get_residues

Arguments: $feature_id, $start, $length

Given a $feature_id with associated residues (generally a chromosome or supercontig), a $start position in base pair coordinates, and a $length, return the specified sequence. (Uses the Postgres SUBSTR function; includes the $startth character, and the total resulting length is $length.

C.3. Module: Bio::SeqIO::chadoxml_apollo

C.3.1. Summary

Bio::SeqIO::chadoxml_apollo - Extension of Bio::SeqIO::Chado to add complex feature writing.
C.3.2. Included modules

- Bio::SeqIO::chado
- Data::Stag::XMLWriter

C.3.3. Example code listing

```perl
$output = new Bio::SeqIO(-format => 'chadoxml_apollo');
$output->write_appdata('title', "Apollo View of Region");
$output->write_feature({
  'is_analysis' => 1,
  'name' => 'feature_name',
  'organism_id' => {
    'organism' => {
      'genus' => 'Anopheles',
      'species' => 'gambiae',
    },
    'feature_relationship' => [
      {
        'is_analysis' => 1,
        ...
      }
    ]
  }
});
```

C.3.4. Description

This class adds the ability to print hash-defined Chado features (with and without sequence) as Apollo-compatible ChadoXML. It also adds the ability to print the ChadoXML metadata tags (<appdata>) required by Apollo. It does not offer the ability to read these same tags as its parent class (Bio::SeqIO::chado) does.

C.3.5. Methods

- `default_handler_class`

Conform to the interface for Bio::SeqIO writers; return our preference for the Data::Stag output writer. We choose Data::Stag::XMLWriter since we expect to be outputting ChadoXML.

- `recursive_hash_to_stag`

Arguments: $hash

Given a reference to a hash in $hash, this function recursively translates it into the Data::Stag format. It also does some simple tag reordering to provide compatibility with Apollo's picky implementation of ChadoXML.
Translation:

The following hash:

```perl
$hash = {
    'name' => 'SomeFeature',
    'feature_relationship' => [
        'subject_id' => {
            'name' => 'SomeOtherFeature'
        }
    ]
}
```

Is translated to:

```perl
$stag = [
    ['name', 'SomeFeature'],
    ['feature_relationship' => [
        ['subject_id' => {
            ['feature' => {
                ['name' => 'SomeOtherFeature']
            ]
        ]
    ]
    ]
];
```

Which will be translated to XML as:

```xml
<name>SomeFeature</name>
<feature_relationship>
    <subject_id>
        <feature>
            <name>SomeOtherFeature</name>
        </feature>
    </subject_id>
</feature_relationship>
```

Reordering:

The Apollo specific reorderings apply to `<dbxref>` and `<feature_relationship>` tags. `<dbxref>` tags contain two tags: `<accession>` and `<db_id>`. Apollo always expects the `<accession>` tag first, and will fail to load ChadoXML where this is not the case. Likewise, `<feature_relationship>` tags can two several kinds of features for transcript features: exons and proteins. Apollo always expects to see the the exon features first and the protein features last. It also expects the exon features to be sorted by location on the genome.

- **write_appdata**
  - **Arguments**: $name, $value
_appdata tags are Apollo-specific entries in ChadoXML that are used for things like the title and region's residues. This function writes out a ChadoXML _appdata tag in the form:

```xml
<appdata name="$name">$value</appdata>
```

- **write_feature**
  - **Arguments:** $feature_hash
  - This function uses recursive_hash_to_stag to turn a Chado feature provided as a hash (by Chapollo::Main) into a structure compatible with the Data::Stag system, and then writes it. It is also responsible for wrapping the pieces of a feature returned by recursive_hash_to_stag in an actual <feature> tag.
D.1. Module: ChadoLoader::Main

D.1.1. Summary

ChadoLoader::Main - Entry point for ChadoLoader application. Provides a framework for loading and connecting ChadoLoader data modules and organism modules.

D.1.2. Included modules

- ChadoLoader::Util

D.1.3. Example code listing

```perl
use ChadoLoader::Main;

@ARGV = (
    '-dbname',    'chado',
    '-dbhost',    'localhost',
    '-dbuser',    'user',
    '-dbpass',    'password',
    '-enshost',   'localhost',
    '-ensuser',   'ens_user',
    '-enspass',   'ens_pass',
    '-organisms', 'Anopheles_gambiae,Aedes_aegypti',
    '-modules',   'EnsEMBLBasicFeatures,EnsEMBLMicroarrays'
);

$loader = new ChadoLoader::Main(-args => \@ARGV);
$loader->load();
```
D.1.4. Description

This class is responsible for turning an @ARGV-style array of arguments into useful input arguments for the database connection and for loading (at runtime) the user-specified data-type modules (e.g. ChadoLoader::Modules::EnsEMBLBasicFeatures) and organism modules (e.g. ChadoLoader::Organism::Anopheles_gambiae). Modules are loaded on an individual basis, and any module that fails will cause the program to return false and print the error to STDERR.

D.1.5. Methods

• new

Arguments: $args

The constructor takes an @ARGV-style array of options (as seen in the SYNOPSIS or the usage arguments for the chadoLoader.pl script. Those arguments must include at least one of the ChadoLoader::Organisms, one of the ChadoLoader::Modules, and the database name and host.

• load

Loads (with ChadoLoader::Util::dynamic_use()) the data and organism modules requested, and then calls load on each of the combinations of organism and data module in turn. If there is an error loading a module (an incorrect name, for instance), the function will print the error and return false, but any modules up to that point that have been requested will have already been run and loaded their data. It also passes in the debugging value (if any) to the data module(s).

• debug

Arguments: [$debug]

Return or set (given $debug) the debugging flags. $debug is an array of strings such as "NO_PRINT_PROGRESS", "NO_PRINT_WARNINGS", "NO_PRINT_NOTICES", or "NO_COMMIT"; these may do different things based on the module in question. (See the module documentation at ChadoLoader::Modules.)

• modules

Arguments: [$modules]
Return or set (given $modules) the data modules to load. $modules is an array of strings such as "EnsEMBLBasicFeatures", "EnsEMBLMicroarrays", etc. For a list of all possible modules, see the documentation at ChadoLoader::Modules or run the chadoLoader.pl script without any arguments.

• organisms

Arguments: [$organisms]

Return or set (given $organisms) the organism modules to load. $organisms is an array of strings such as Anopheles_gambiae, Aedes_aegypti, etc. For a list of all possible organisms, see the documentation at ChadoLoader::Organism or run the chadoLoader.pl script without any arguments.

D.2. Module: ChadoLoader::Organism

D.2.1. Summary

ChadoLoader::Organism - Abstract class for organism modules

D.2.2. Included modules

• ChadoLoader::DB::Chado

D.2.3. Example code listing

package ChadoLoader::Organism::MyOrganism;

use ChadoLoader::Organism;
our @ISA = qw(ChadoLoader::Organism)

sub new { < define organism-specific variables > }

D.2.4. Description

This class is an abstract definition of the ChadoLoader organism modules; it provides all the functionality needed for an organism module, but none of the internal variables that must be set. For an example of an organism module implementation, see ChadoLoader::Organism::Anopheles_gambiae.
D.2.5. Methods

- `_db_update_or_create`
  
  **Arguments:** $chado_dbinfo
  
  Given a hash of information for connecting to a Chado database in $chado_dbinfo (see ChadoLoader::DB::Chado for hash format), fetch (or create if necessary) an entry in that database for this organism and return the organism_id. The criteria for checking if the organism already exists is only based on the genus and species. If other fields have been set for this organism (organism_id, common_name, or abbreviation) but the organism already exists in the database, these fields will be updated to match the database (not the other way around). If the organism does not already exist in the database, these fields will be used to populate the new entry.

- `genus`
  
  **Arguments:** $genus
  
  Return (or set, given $genus) the genus of this organism.

- `species`
  
  **Arguments:** $species
  
  Return (or set, given $species) the species of this organism.

- `common_name`
  
  **Arguments:** $common_name
  
  Return (or set, given $common_name) the common_name of this organism.

- `organism_id`
  
  **Arguments:** $organism_id
  
  Return (or set, given $organism_id) the organism_id of this organism. Note that setting the organism_id will not alter the organism_id in the Chado database (unless the organism has yet to be created, in which case, setting the organism_id here will set the default organism_id for the new organism... not that this is recommended).

- `abbreviation`
  
  **Arguments:** $abbreviation
  
  Return (or set, given $abbreviation) the abbreviation of this organism.

- `ensembl_settings`
  
  Returns this organism's ensembl_settings. This hash specifies things like the basic feature type (chromosome/supercontig), the allowed adaptors, the ensembl database
version, etc. For an example of the ensembl_settings format, see ChadoLoader::Organism::Anopheles_gambiae::new().

D.3. Module: ChadoLoader::Organism::Computational_result

D.3.1. Summary

ChadoLoader::Organism::Computational_result - Organism class for analysis features that have no organism.

D.3.2. Included modules

- ChadoLoader::Organism

D.3.3. Example code listing

```perl
use ChadoLoader::Organism::Computational_result;

$chado_info = {
'dbstr'  => "dbi::Pg:dbname=chado;host=localhost",
'dbuser' => "username",
'dbpass' => "password"
};
$organism = new ChadoLoader::Organism::Computational_result($chado_info)
```

D.3.4. Description

This class is an extension of ChadoLoader::Organism for analysis features that are attached to other organisms but have no organism of their own. It is given an entry in the Chado database, but has no connection to source EnsEMBL databases. The attributes assigned to the Chado entry are:

- 'genus' - 'Computational'
- 'species' - 'result'
- 'common_name' - 'Computational result'
- 'abbreviation' - 'Comp. result'
- 'organism_id' - 0; no organism_id set by default
D.3.5. Methods

• new

Arguments: $chado_dbinfo

Given information for connecting to a Chado database (see the SYNOPSIS), the constructor the "organism"-specific data for computational results, and then fetches or creates (see ChadoLoader::Organism::_db_update_or_create()) the entry in the given Chado database.

D.4. Module: ChadoLoader::Organism::Aedes_aegypti

D.4.1. Summary

ChadoLoader::Organism::Aedes_aegypti – Organism class for Aedes aegypti mosquito.

D.4.2. Included modules

• ChadoLoader::DB::Chado

• ChadoLoader::Organism

D.4.3. Example code listing

use ChadoLoader::Organism::Aedes_aegypti;

$chado_info = {
   'dbstr' => "dbi::Pg:dbname=chado;host=localhost",
   'dbuser' => "username",
   'dbpass' => "password"
};
$organism = new ChadoLoader::Organism::Aedes_aegypti($chado_info)

D.4.4. Description

This class is an extension of ChadoLoader::Organism specific to the Aedes aegypti mosquito. This organism provides the following basic organism attributes:

- 'genus' - 'Aedes'
- 'species' - 'aegypti'
- 'common_name' - 'aedes'
And the following Ensembl attributes:

- **golden_path_unit**: 'contig'; the basic feature type for sliced sequence in Ensembl
- **golden_path_cvterm**: 'contig'; the cvterm name for these golden_path features in Chado
- **base_region**: 'supercontig'; the major genomic feature type in Ensembl
- **chromosome_cvterm**: 'supercontig'; the Chado cvterm name for the major genomic feature
- **ens_version**: '43_1a'; the suffix to Ensembl database names (e.g. aedes_aegypti_core_43_1a)
- **allowed_adaptors**: The ensembl feature types and databases available for this feature. (See ChadoLoader::Util::merge_ensembl_adaptors().)

## D.4.5. Methods

- **new**
  - **Arguments**: $chado_dbinfo
  - Given information for connecting to a Chado database (see the SYNOPSIS), the constructor sets all of the organism-specific data for Anopheles gambiae, and then fetches or creates (see ChadoLoader::Organism::_db_update_or_create()) the entry in the given Chado database.

## D.5. ChadoLoader::Organism::Anopheles_gambiae

### D.5.1. Summary

**ChadoLoader::Organism::Anopheles_gambiae** - Organism class for *Anopheles gambiae* mosquito.

### D.5.2. Included modules

- ChadoLoader::DB::Chado
- ChadoLoader::Organism
D.5.3. Example code listing

```perl
use ChadoLoader::Organism::Anopheles_gambiae;

$chado_info = {
    'dbstr' => "dbi::Pg:dbname=chado;host=localhost",
    'dbuser' => "username",
    'dbpass' => "password"
};
$organism = new ChadoLoader::Organism::Anopheles_gambiae($chado_info)
```

D.5.4. Description

This class is an extension of ChadoLoader::Organism specific to the *Anopheles gambiae* mosquito. This organism provides the following basic organism attributes:

- **'genus'** - 'Anopheles'
- **'species'** - 'gambiae'
- **'common_name'** - 'mosquito'
- **'abbreviation'** - 'A.gambiae'
- **'organism_id'** - 0; no organism_id set by default

And the following EnsEMBL attributes:

- **golden_path_unit** - 'chunk'; the basic feature type for sliced sequence in EnsEMBL
- **golden_path_cvterm** - 'golden_path'; the cvterm name for these golden_path features in Chado
- **base_region** - 'chromosome'; the major genomic feature type in Ensembl
- **chromosome_cvterm** - 'chromosome_arm'; the Chado cvterm name for the major genomic feature
- **ens_version** - '43_3f'; the suffix to EnsEMBL database names (e.g. aedes_aegypti_core_43_3f)
- **allowed_adaptors** - The ensembl feature types and databases available for this feature. (See ChadoLoader::Util::merge_ensembl_adaptors().)

D.5.5. Methods

- **new**
  - **Arguments:** $chado_dbinfo
    - Given information for connecting to a Chado database (see the SYNOPSIS), the constructor sets all of the organism-specific data for Anopheles gambiae, and then fetches or creates (see ChadoLoader::Organism::_db_update_or_create()) the entry in the given Chado database.
D.6. Module: ChadoLoader::DB::Chado

D.6.1. Summary

ChadoLoader::DB::Chado - Chado database interface class for the ChadoLoader application

D.6.2. Included modules

- DBI

D.6.3. Example code listing

```perl
$chado = new ChadoLoader::DB::Chado(
    {
        'dbstr' => "dbi:Pg:dbname=chado;host=localhost",
        'dbuser' => $dbuser,
        'dbpass' => $dbpass,
    }
);
$cvterm_id = $chado->get_cvterm('sequence', 'chromosome')->{'cvterm_id'};
```

D.6.4. Description

This is the class used by the ChadoLoader application for access to a Chado database. Part of the work done by this function is initial preparation of queries during the creation of the constructor. As such, this class is only useful when working with these predefined queries; it is not a general-purpose database class, although it does provide an accessor to the database handle if it's really necessary.

Despite the presence of make_threadsafe and make_not_threadsafe functions, this class is only really safe in a serially fork(ed) environment a la ChadoLoader::Modules::EnsEMBLAnalysisFeatures. Potentially it could be used simultaneously in multiple processes, but there is nothing to prevent attempts at simultaneous insertion of the same data. The only thing that the two thread functions do is
set the InactiveDestroy variable on the DBI handle to prevent child threads from closing the database handle.

This class relies on the Chado database in question being stored in a PostgreSQL database for two features. The first of these is the pg_prepare_now argument used in new to make sure all queries are actually prepared in the database before any potential fork()ing. The second is the use of sub-transaction handling with DBD::Pg::pg_savepoint, DBD::Pg::pg_rollback_to, and DBD::Pg::pg_release. These functions provide a way to recover from single failed queries within a larger transaction without cancelling the whole transaction. Since some of the data loading modules attempt to insert without checking first to see if the feature already exists, this mini-transactioning is what keeps the larger transactions (per-chromosome/supercontig usually) from failing.

D.6.5. Methods

• new

  **Arguments:** $dbinfo

  The constructor takes database connection information in a hash $dbinfo (see the SYNOPSIS), that grants read/write access to the database. It then opens a database connection and prepares all of the queries that will be used by the class. Since many of these queries are run many times, preparation speeds execution significantly.

• connect

  **Arguments:** $dbinfo

  Attempt to connect to the given database given a hash $dbinfo (see the SYNOPSIS). This function should only be called by the new, and just sets some extra options (no automatic committing, and no printing errors to STDERR) and die()s on failure.

• make_threadsafe

  Set the database handle's InactiveDestroy attribute to true so that if a process holding a reference to this database handle exits, the connection to the database is not automatically closed. We do this when we want to use this database handle in several children in sequence (this class is not built to be run in parallel with itself;
see the DESCRIPTION.) See ChadoLoader::Modules::EnsEMBLAnalysisFeatures::load() for an example of the use of this function and make_not_threadsafe.

• make_not_threadsafe

Set the database handle's InactiveDestroy attribute to false so that if a process holding a reference to this database handle exits, the connection to the database is automatically closed. This is the default behavior, so we want to switch to it whenever we don't absolutely need InactiveDestroy to be true. See ChadoLoader::Modules::EnsEMBLAnalysisFeatures::load() for an example of the use of this function and make_threadsafe.

• dbh

Arguments: $newdbh

Returns (or sets, given $newdbh) a reference to the internal DBD::Pg database connection handle.

• do_query

Arguments: $sth_name, $input

This function wraps the actual query execute calls in some error handling code and provides the option to return either a single row as a hash or a reference to the statement handle that has just been executed. It also takes an option allowing queries to be run inside a PostgreSQL savepoint (a subtransaction) so that queries that return an error can be handled properly. These are used (as further explained in the DESCRIPTION) to allow data loading modules to attempt to insert a feature without checking to see if it exists; failure of an insert is made nonfatal (and 0 is returned). Queries that are not wrapped in a savepoint print a partial stacktrace cause the application to die().

The $sth_name variable is the key in the hash of queries defined in the constructor. The $input variable is a hash containing some subset of four options. The first is bind_params which is an array of the values to be passed to the query, if any. The second is return_sth, which determines the return type. If false, a hash of the first row returned by the query is returned. If true, a reference to the executed query is returned. The third is continue, which determines whether savepoints are used (they are if continue is true). The fourth is return_last_id which, if true, makes the return value be the PostgreSQL lastval(); for Chado this will be the feature_id/organism_id/featurelocation_id/etc. (depending on the table last inserted into) for the most recently inserted feature.

• get_cvterm

Arguments: $cv, $cvterm
Given a controlled vocabulary name (cv.name) in $cv, and a controlled vocabulary term name (cvterm.name) in $cvterm, return the cvterm_id of the specified cvterm in a hash (where the only key is "cvterm_id").

- **get_cv**
  
  **Arguments:** $cv

  Given a controlled vocabulary name (cv.name) in $cv return the cv_id of the specified cv in a hash (where the only key is "cv_id").

- **ins_cv**
  
  **Arguments:** $cv, [{$definition}]

  Given a controlled vocabulary name (cv.name) in $cv and optional definition (cv.definition) in $definition, attempt to insert a new controlled vocabulary into the database. If successful, return the cv_id of the new cv.

- **get_cvterm_by_dbxref_id**
  
  **Arguments:** $dbxref_id

  Given a database xref id (dbxref.dbxref_id), return the cvterm_id of the cvterm with that dbxref_id (cvterm.dbxref_id) in a hash (where the only key is "cvterm_id").

- **ins_cvterm**
  
  **Arguments:** $cv_id, $cvterm, [{$definition}], $dbxref_id

  Given a controlled vocabulary id (cv.cv_id) in $cv_id, a controlled vocabulary term name (cvterm.name) in $cvterm, an optional cvterm definition (cvterm.definition) in $definition, and a database xref id (cvterm.dbxref_id) in $dbxref_id, attempt to insert a new controlled vocabulary term into the database. If successful, return the cvterm_id of the new cvterm.

- **get_publication**
  
  **Arguments:** $name

  Given a publication name (pub.uniquename) in $name, return the pub_id of the specified publication in a hash (where the only key is "pub_id").

- **ins_publication**
  
  **Arguments:** $name

  Given a publication name (pub.uniquename) in $name, attempt to insert a new publication into the database. If successful, return the pub_id of the new publication.
• **get_db**

  **Arguments:** $dbname

  Given a database name (db.name) in $dbname, return the db_id of the specified db in a hash (where the only key is "db_id").

• **ins_db**

  **Arguments:** $db, [description]

  Given a database name (db.name) in $db and optional description (db.description) in $description, attempt to insert a new db into the database. If successful, return the db_id of the new db.

• **get_dbxref**

  **Arguments:** $dbid, $dbxref

  Given a database id (db.db_id) in $dbid, and a database xref accession (dbxref.accession) in $dbxref, return the dbxref_id of the specified dbxref in a hash (where the only key is "dbxref_id").

• **ins_dbxref**

  **Arguments:** $db_id, $dbxref

  Given a database id (db.db_id) in $dbid and database xref accession (dbxref.accession) in $dbxref, attempt to insert a new dbxref into the database. If successful, return the dbxref_id of the new dbxref.

• **get_analysis**

  **Arguments:** $program, $programversion, $sourcename

  Given an analysis program name (analysis.program) in $program, an analysis program version (analysis.programversion) in $programversion, and an analysis source name (analysis.sourcename) in $sourcename, return the analysis_id of the specified analysis in a hash (where the only key is "analysis_id").

• **ins_analysis**

  **Arguments:** $program, $programversion, $sourcename, [$name], [$algorithm], [$sourceversion]

  Given an analysis program name (analysis.program) in $program, an analysis program version (analysis.programversion) in $programversion, an analysis source name (analysis.sourcename) in $sourcename, an optional analysis name (analysis.name) in $name, an optional analysis algorithm (analysis.algorithm) in $algorithm, and an optional analysis source version (analysis.sourceversion) in
$sourceversion, attempt to insert a new analysis into the database. If successful, return the analysis_id of the new analysis.

• get_analysisfeature

Arguments: $feature_id, $analysis_id

Given a feature id (feature.feature_id) in $feature_id, and an analysis id (analysis.analysis_id) in $analysis_id, return the analysisfeature_id of the analysisfeature (that ties the feature and analysis together) in a hash (where the only key is "analysisfeature_id").

• ins_analysisfeature

Arguments: $feature_id, $analysis_id, $rawscore, $significance, $identity

Given a feature id (feature.feature_id) in $feature_id, an analysis id (analysis.analysis_id) in $analysis_id, an optional raw score for the analysisfeature (analysisfeature.rawscore) in $rawscore, an optional significance score for the analysisfeature (analysisfeature.significance) in $significance, and an optional identity score for the analysisfeature (analysisfeature.identity) in $identity, attempt to insert an analysisfeature into the database. If successful, return the analysisfeature_id of the new analysisfeature.

• get_feature

Arguments: $organism_id, $uniquename, $cvterm_id

Given an organism id (organism.organism_id) in $organism_id, a feature's uniquename (feature.uniquename) in $uniquename, and a controlled vocabulary term id (cvterm.cvterm_id) in $cvterm_id, return the feature_id of the specified feature in a hash (where the only key is "feature_id").

• ins_feature

Arguments: $dbxref_id, $organism_id, $name, $uniquename, $sequence, $length, $cvterm_id, $is_analysis

Given an optional database xref id (dbxref.dbxref_id) in $dbxref_id, organism id (organism.organism_id) in $organism_id, optional feature name (feature.name) in $name, feature uniquename (feature.uniquename) in $uniquename, optional feature sequence (feature.sequence) in $sequence, optional feature length (feature.length) in $length, controlled vocabulary term id (cvterm.cvterm_id) in $cvterm_id, and optional boolean indicating whether this feature is an analysis hit or not (feature.is_analysis) in $is_analysis, attempt to insert a new feature into the database. If successful, return the feature_id of the new feature.
get_featurelocation

Arguments: $feature_id, $rank

Given a feature id (feature.feature_id) in $feature_id, and a feature location rank (featureloc.rank) in $rank, return the featureloc_id of the specified feature location in a hash (where the only key is "featureloc_id"). Note that featureloc.locgroup is also technically part of the same key on featureloc, but VectorBase's usage of Chado doesn't rely on locgroup (what is it for, anyway?).

ins_featurelocation

Arguments: $feature_id, [$srcfeature_id], [$fmin], [$fmax], [$strand], [$phase], $rank

Given a feature id (feature.feature_id) in $feature_id, optional but recommended source feature (chromosome/supercontig) feature id (featureloc.srcfeature_id) in $srcfeature_id, optional but recommended feature location start (featureloc.fmin) in $fmin, optional but recommended feature location end (featureloc.fmax) in $fmax, optional but recommended feature location strand (featureloc.strand) in $strand, optional feature location phase (featureloc.phase) in $phase, and feature location rank (featureloc.rank) in $rank, attempt to insert a new feature location into the database. If successful, return the featureloc_id of the new featureloc.

get_featurerelationship

Arguments: $subject_id, $object_id, $type_id, $rank

Given a child feature id (feature.feature_id) in $subject_id, a parent feature id (feature.feature_id) in $object_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, and a feature relationship rank (feature_relationship.rank) in $rank, return the feature_relationship_id of the specified feature relationship in a hash (where the only key is "feature_relationship_id").

ins_featurerelationship

Arguments: $subject_id, $object_id, $type_id, $rank

Given a child feature id (feature.feature_id) in $subject_id, a parent feature id (feature.feature_id) in $object_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, and a feature relationship rank (feature_relationship.rank) in $rank, attempt to insert a new controlled vocabulary into the database. If successful, return the feature_relationship_id of the new feature relationship.

get_featureproperty

Arguments: $feature_id, $type_id, $rank

105
Given a feature id (feature.feature_id) in $feature_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, and a feature property rank (featureprop.rank) in $rank, return the featureprop_id of the specified featureprop in a hash (where the only key is "featureprop_id").

• ins_featureproperty

Arguments: $feature_id, $type_id, $rank, $value

Given a feature id (feature.feature_id) in $feature_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, a feature property rank (featureprop.rank) in $rank, and a feature property value (featureprop.value) in $value, attempt to insert a new feature property into the database. If successful, return the featureprop_id of the new feature property.

• get_feature_cvterm

Arguments: $feature_id, $term_id, $publication_id

Given a feature id (feature.feature_id) in $feature_id, a controlled vocabulary term id (cvterm.cvterm_id) in $term_id, and a publication id (pub.pub_id), return the feature_cvterm_id of the specified feature-cvterm relationship (in feature_cvterm) in a hash (where the only key is "feature_cvterm_id").

• ins_feature_cvterm

Arguments: $cv, [definition]

Given a feature id (feature.feature_id) in $feature_id, a controlled vocabulary term id (cvterm.cvterm_id) in $term_id, and a publication id (pub.pub_id), attempt to insert a new feature-cvterm relationship into the database. If successful, return the feature_cvterm_id of the new feature_cvterm.

• get_feature_cvtermprop

Arguments: $feature_cvterm_id, $type_id, $rank

Given a feature-cvterm relationship id (feature_cvterm.feature_cvterm_id) in $feature_cvterm_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, and a feature-cvterm relationship property id (feature_cvtermprop.rank) in $rank, return the feature_cvtermprop_id of the specified feature-cvterm relationship property in a hash (where the only key is "feature_cvtermprop_id").

• ins_feature_cvtermprop

Arguments: $feature_cvterm_id, $type_id, $rank, $value

Given a feature-cvterm relationship id (feature_cvterm.feature_cvterm_id) in $feature_cvterm_id, a controlled vocabulary term id (cvterm.cvterm_id) in $type_id, a feature-cvterm relationship property id (feature_cvtermprop.rank) in $rank, and a
feature-cvterm relationship property value (feature_cvtermprop.value) in $value, attempt to insert a new feature-cvterm relationship property into the database. If successful, return the feature_cvtermprop_id of the new feature_cvtermprop.

• get_organism

Arguments: $genus, $species

Given an organism genus (organism.genus) in $genus, and an organism species (organism.species) in $species, return the organism_id of the specified organism in a hash (where the only key is "organism_id").

• ins_organism

Arguments: [$abbreviation], $genus, $species, [$common_name]

Given an optional organism name abbreviation (organism.abbreviation) in $abbreviation, an organism genus (organism.genus) in $genus, an organism species (organism.species) in $species, and an optional organism common name (organism.common_name) in $common_name, attempt to insert a new organism into the database. If successful, return the organism_id of the new organism.

• get_md5

Arguments: $sequence

Given a string (preferably genomic sequence) in $sequence, return the result of the PostgreSQL md5() function on that string. Used to generate the MD5 hash for the feature.md5 field. If the function fails, return the empty string.

• commit

Call finish() on all of the prepared queries held by this object to prepare for possible database shutdown (although this doesn't prevent them from being started back up again; this function can be called and then the queries can continue to be used; just make sure to call this function once right before closing the database connection or exiting). After the the queries are finish()ed, changes are committed to the database.

• finish

Commit any changes to the database, including finishing all queries (by calling commit), and then disconnect from the database. This should be done just before normal program exit.
D.7. Module: ChadoLoader::DB::EnsEMBL

D.7.1. Summary

ChadoLoader::DB::EnsEMBL - EnsEMBL database interface class for the ChadoLoader application.

D.7.2. Included modules

- Bio::EnsEMBL::DBSQL::DBAdaptor
- DBI

D.7.3. Example code listing

```perl
use ChadoLoader::DB::EnsEMBL;
use ChadoLoader::Organism::Anopheles_gambiae;

$organism = new ChadoLoader::Organism::Anopheles_gambiae(
  {
    'dbstr'  => "dbi:Pg:dbname=chado;host=localhost",
    'dbuser' => $dbuser,
    'dbpass' => $dbpass,
  }
);
$ensembl_dbinfo = {
  'dbhost' => 'localhost',
  'dbuser' => 'ens_user',
  'dbpass' => 'ens_pass'
};
$ensembl = new ChadoLoader::DB::EnsEMBL($ensembl_dbinfo, $organism);
my $archive_ids = $ensembl->get_all_current_ids('gene');
```

D.7.4. Description

This is the class used by the ChadoLoader application as a wrapper for the various EnsEMBL databases, both for getting the standard EnsEMBL Perl API handles (through get_adaptor, core, and est), and direct SQL access (get_all_current_ids, get_assembly_names, and get_supporting_feature).

Besides combining both the EnsEMBL API and SQL queries in one place, it also provides the refresh function to handle reconnection during forking. (See
ChadoLoader::DB::Chado::make_threadsafe() and
ChadoLoader::Modules::EnsEMBLAnalysisFeatures::load().

D.7.5. Methods

• **new**
  
  **Arguments:** $ensembl_dbinfo, $organism
  
  The constructor takes database connection information in a hash $ensembl_dbinfo (see the SYNOPSIS), that grants read access to the EnsEMBL database. It then opens two API connections (one to each of the core and est databases, if available), and two database connection (likewise, one to each major dataset). It also sets up (prepare())s the direct SQL queries that will be used by the ChadoLoader application.

• **refresh**
  
  This function reconnects all of the database handles and queries (much like creating a new instance of the class). It is necessary to call this function if you are trying to use the object within a fork()ed child. (See the DESCRIPTION for more information.)

• **_prepare_queries**
  
  Prepares the queries used for direct SQL access. Needs its own function so we don't have to keep two copies of this code in both new and refresh.

• **get_adaptor**
  
  **Arguments:** $db, $adaptor_name
  
  Given either "est" or "core" as the database name in $db, and an EnsEMBL adaptor name, either return a cached copy of the adaptor, or create a new one if necessary. This is the same as calling $ensembl-$db->get_$adaptor_name>

• **coredbh**
  
  Return a reference to the EnsEMBL core database connection; it is a Bio::EnsEMBL::DBSQL::DBConnection.

• **estdbh**
  
  Return a reference to the EnsEMBL est database connection; it is a Bio::EnsEMBL::DBSQL::DBConnection.
• core

    Return a reference to the Ensembl core database adaptor; it is a Bio::EnsEMBL::DBSQL::DBAdaptor.

• est

    Return a reference to the Ensembl est database adaptor; it is a Bio::EnsEMBL::DBSQL::DBAdaptor.

• organism

    Return a reference to the ChadoLoader::Organism object used to create the instance of this class.

• do_query

    Arguments: $sth_name, $input

    This function wraps the actual SQL queries in some error handling code and provides the option to return either a single row as a hash or a reference to the statement handle that has just been executed. If there is an error in the query, it die()s, halting the application.

• get_all_current_ids

    Arguments: $archive_type

    Given a feature type (gene, transcript, or translation) in $archive_type, this function fetches all of the current Ensembl stable IDs for that feature type and returns an array reference of them.

• get_assembly_names

    Arguments: $archive_id

    Given an Ensembl stable ID in $archive_id, this function fetches any gene build names associated with that stable ID and returns an array reference of them.

• get_supporting_feature

    Arguments: $feature_type, $feature_id, $db

    Given a feature type (dna_align_feature or protein_align_feature), an internal Ensembl feature identifier ($feature->dbID), and a database name (core or est), discover if the feature has any entry in the transcript_supporting_feature table. This table indicates that an analysis feature is actually a sort of meta-feature covering a range of smaller features.

    See also ChadoLoader::Modules::EnsemblAnalysisFeatures::is_final_analysis().
• finish

Run finish() on all of the queries to make sure they are complete and then disconnect from the EnsEMBL databases.

D.8. Module: ChadoLoader::Modules::EnsEMBLBasicFeatures

D.8.1. Summary

ChadoLoader::Modules::EnsEMBLBasicFeatures - ChadoLoader module to load basic EnsEMBL features.

D.8.2. Included modules

• ChadoLoader::DB::Chado
• ChadoLoader::DB::EnsEMBL
• ChadoLoader::Util

D.8.3. Example code listing

use ChadoLoader::Main;
my $loader = new ChadoLoader::Main('modules' => ['EnsEMBLBasicFeatures']);
$loader->load();

or

use ChadoLoader::Modules::EnsEMBLBasicFeatures;
use ChadoLoader::Organism::Anopheles_gambiae;
my $ebf = new EnsEMBLBasicFeatures(
   new ChadoLoader::Organism::Anopheles_gambiae());
$ebf->load();

D.8.4. Description

This module loads the following features from EnsEMBL into Chado:

• chromosomes/supercontigs
• golden path features (generally 'chunks')
• analyses
• chromosome bands
• BAC clone alignments
• genes, transcripts, exons, and proteins
• predicted transcripts

Loading is done in a transactional way, and if there is an unrecoverable error, the script die(). All feature insertions are also done in a way that is compatible with data already being in the database. Running the module twice in a row without cleaning the database should be the same as running the script once, albeit slower the second time around as optimizations have been made for feature insertion. All modules that inherit from EnsEMBLBasicFeatures should also display this functionality for all functions inherited from EnsEMBLBasicFeatures.

This modules requires that the provided organism support SliceAdaptor, AnalysisAdaptor, and GeneAdaptor on the EnsEMBL 'core' database. It will also use KaryotypeBandAdaptor on the 'core' database and GeneAdaptor on the 'est' database, if they are available.

D.8.5. Methods

• new

Arguments: $organism, [$debug]

The constructor takes a ChadoLoader::Organism object in $organism.

It also optionally takes a debugging bitmask in $debug; see the debugging definitions in CONSTANTS.

• load

The workhorse function of this module. Pulls data out of EnsEMBL via the Perl interface, and loads it into Chado. If this class is extended, this is the function that should be overridden by any other Ensembl-connected modules. This function is actually pretty fancy and does things like show progress bars during loading of each data type. It also tries its best to hold off on committing data so that any early errors can be spotted before any changes are made to the database. Sadly, it's impossible to
run this module all the way through with the ChadoLoader::Modules::EnsEMBLBasicFeatures::NO_COMMIT debug option set since the database engine will run out of memory first.

Like the rest of the ChadoLoader modules, this module can theoretically be run over and over; it will not insert data into the database if it already exists, instead opting to use the existing data. This means that if a load is only partially accomplished, the module can be run again without inserting duplicate data. It also means that there won't be problems if another module has already (for example) inserted chromosome features; it will just use the existing ones rather than reinserting new ones. This module is not intelligent enough to update data that has changed, however. If a gene's location is different, this module will keep the old location data, ignoring the new. If you're loading from an updated Ensembl database, you'll need to clean out your database (or start from a new one).

Run time for this function is on the order of hours (3-6 hours for Anopheles gambiae), so beware before you run it.

• organism

Returns the ChadoLoader::Organism object that the module was instantiated with.

• chado

Returns the ChadoLoader::DB::Chado database object used for interacting with the Chado database.

• ensembl

Returns the ChadoLoader::DB::EnsEMBL database object used for interacting with the Ensembl database.

• get_cvterm

Arguments: $cv_name, $cvterm_name, $definition, $dbxref_id

Look up or create a cvterm. Only $cv_name and $cvterm_name are necessary, but the $definition and $dbxref_id fields will used if the cvterm does not exist and must be created.

This functions looks for an existing cvterm entry in the database first, then creates one if an existing one cannot be found.

In either case, it returns a cvterm_id, or die()s in response to an error from SQL.

• get_cv

Arguments: $cv_name, $cvterm_name, $definition, $dbxref_id
Look up or create a cv. Only $cv_name is necessary but the $definition field will used if the cv does not exist and must be created.

This functions looks for an existing cv entry in the database first, then creates one if an existing one cannot be found.

In either case, it returns a cv_id, or die()s in response to an error from SQL.

• get_cvterm_by_dbxref_id

  Arguments: $dbxref_id

  Look up a cvterm by $dbxref_id. This can be used in conjunction with get_dbxref to fetch a cvterm associated with a given accession name, as with GO terms.

  Returns a cvterm_id if found, or die()s otherwise.

• get_publication

  Arguments: $name, $type_id

  Look up or create a publication. Only $name is necessary, but the $type_id field will used if the publication does not exist and must be created.

  This functions looks for an existing publication entry in the database first, then creates one if an existing one cannot be found.

  In either case, it returns a publication_id, or die()s in response to an error from SQL.

• get_db

  Arguments: $name, $description

  Look up or create a database entry. Only $name is necessary, but the $description field will used if the database entry does not exist and must be created.

  This functions looks for an existing database entry in the database first, then creates one if an existing one cannot be found.

  In either case, it returns a db_id, or die()s in response to an error from SQL.

• get_dbxref

  Arguments: $dbname, $accession

  Look up or create a dbxref. Both $dbname and $accession are necessary for either retrieval or creation.

  By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a dbxref. If that fails, it will fall back and try to retrieve the dbxref. If that fails in turn, it will die() with an error message.
In either case, it returns a dbxref_id, or die()s in response to an error from SQL.

• get_analysis

Arguments: $program, $programversion, $sourcename, $name, $algorithm, $sourceversion

Look up or create a analysis. $program, $programversion, and $sourcename are required. $name, $algorithm, and $sourceversion are optional and will be used if the analysis does not yet exist in the database.

This functions looks for an existing analysis entry in the database first, then creates one if an existing one cannot be found.

In either case, it returns a analysis_id, or die()s in response to an error from SQL.

• get_feature

Arguments: [$dbxref_id], $organism_id, [$name], $uniquename, [$sequence], [$length], $cvterm_id, [$is_analysis], $fetch

Look up or create an entry in the feature table. $organism_id, $uniquename, and $cvterm_id are required. $dbxref_id, $name, $sequence, $length, and $is_analysis are optional and will be used if the feature does not yet exist in the database.

By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature. If that fails, it will fall back and try to retrieve the feature. If that fails in turn, it will die() with an error message.

If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature. Failing that, it will then attempt to create the feature. Failing that, it will die() with an error message.

This function will return a feature_id unless it die()s as mentioned.

• get_featurelocation

Arguments: $feature_id, $srcfeature_id, $fmin, $fmax, $strand, $phase, $rank, $fetch

Look up or create an entry in the featureloc table. $feature_id and $rank are required, although rank will be set to a default of '0' if omitted. $srcfeature_id, $fmin, $fmax, $strand, and $phase are optional and will be used if the feature location does not yet exist in the database.

By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature location. If that fails, it will fall back and try to retrieve the feature location. If that fails in turn, it will die() with an error message.

If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature location. Failing that, it will then
attempt to create the feature location. Failing that, it will die() with an error message.

This function will return a featureloc_id unless it die()s as mentioned.

- get_featurerelationship
  
  Arguments: $subject_id, $object_id, $type_id, $rank, $fetch

  Look up or create an entry in the feature_relationship table. $subject_id, $object_id, $type_id, and $rank are all required, although rank will be set to a default of '0' if omitted.

  By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature relationship. If that fails, it will fall back and try to retrieve the feature relationship. If that fails in turn, it will die() with an error message.

  If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature relationship. Failing that, it will then attempt to create the feature relationship. Failing that, it will die() with an error message.

  This function will return a feature_relationship_id unless it die()s as mentioned.

- get_featureproperty
  
  Arguments: $feature_id, $type_id, $value, $rank, $fetch

  Look up or create a feature property. $feature_id, $type_id, and $rank are required, although rank will be set to a default of '0' if omitted. $value is used to set the value of the feature property if it is being created.

  By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature property. If that fails, it will fall back and try to retrieve the feature property. If that fails in turn, it will die() with an error message.

  If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature property. Failing that, it will then attempt to create the feature property. Failing that, it will die() with an error message.

  This function will return a featureprop_id unless it die()s as mentioned.

- get_feature_cvterm
  
  Arguments: $feature_id, $term_id, $publication_id, $fetch
Look up or create a feature_cvterm (for relating multiple cvterms to a single feature). $feature_id, $term_id, and $publication_id are all required, although the publication will be set to the null publication if omitted.

By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature-cvterm relationship. If that fails, it will fall back and try to retrieve the feature-cvterm relationship. If that fails in turn, it will die() with an error message.

If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature-cvterm relationship. Failing that, it will then attempt to create the feature-cvterm relationship. Failing that, it will die() with an error message.

This function will return a feature_cvterm_id unless it die()s as mentioned.

• get_feature_cvtermprop

Arguments: $feature_cvterm_id, $type_id, $value, $rank, $fetch

Look up or create an additional property for a feature-cvterm relationship. $feature_cvterm_id, $type_id, and $rank are required, although rank will be set to a default of '0' if omitted. $value is used to set the value of the feature-cvterm relationship property if it is being created.

By default (when the $fetch argument does not evaluate to true), this function will first attempt to create a feature-cvterm relationship property. If that fails, it will fall back and try to retrieve the feature-cvterm relationship property. If that fails in turn, it will die() with an error message.

If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the feature-cvterm relationship property. Failing that, it will then attempt to create the feature-cvterm relationship property. Failing that, it will die() with an error message. This function will return a featureprop_id unless it die()s as mentioned.

D.9. Module: ChadoLoader::Modules::EnsEMBLAnalysisFeatures

D.9.1. Summary

ChadoLoader::Modules::EnsEMBLAnalysisFeatures - ChadoLoader module to load EnsEMBL microarray features.
D.9.2. Included modules

- Bio::EnsEMBL::DBSQL::DBAdaptor
- ChadoLoader::DB::Chado
- ChadoLoader::Modules::EnsEMBLBasicFeatures
- ChadoLoader::Organism::Computational_result
- ChadoLoader::Util
- Term::ProgressBar

D.9.3. Example code listing

```perl
use ChadoLoader::Main;
my $loader = new ChadoLoader::Main('modules' =>
    ['EnsEMBLAnalysisFeatures']);
$loader->load();

or

use ChadoLoader::Modules::EnsEMBLAnalysisFeatures;
use ChadoLoader::Organism::Anopheles_gambiae;
my $ema = new EnsEMBLAnalysisFeatures(
    new ChadoLoader::Organism::Anopheles_gambiae());
$ema->load();
```

D.9.4. Description

This module loads the following features from EnsEMBL into Chado:

- chromosomes/supercontigs
- microarrays (OligoFeatures)

Loading is done in a semi-transactional way to allow the script to fail an INSERT and recover by trying a SELECT. If there is an unrecoverable error, the script die().s. All feature insertions are done in a way that is compatible with data already being in the database. Running the module twice in a row without cleaning the database should be the same as running the script once, albeit slower the second time around as optimizations have been made for feature insertion. This module requires that the provided organism support
SliceAdaptor on the EnsEMBL core database. It inherits much of its functionality from ChadoLoader::Modules::EnsEMBLBasicFeatures.

D.9.5. Methods

- **new**

  **Arguments:** $organism, $debug

  The constructor takes a ChadoLoader::Organism object in $organism.

  It also optionally takes a debugging bitmask in $debug; see the debugging definitions in ChadoLoader::Modules::EnsEMBLBasicFeatures::CONSTANTS.

- **load**

  The workhorse function of this module. Pulls data out of EnsEMBL via the Perl interface, and loads it into Chado. If this class is extended, this is the function that should be overridden by any other Ensembl-connected modules. As with ChadoLoader::Modules::EnsEMBLBasicFeatures, this function draws pretty progress bars. Unlike ChadoLoader::Modules::EnsEMBLBasicFeatures, however, it does not attempt to avoid committing features for very long. There are so many analysis features that memory limitations quickly become a problem.

  Memory usage is in fact so important that this module actually fork()s for each base feature (chromosomes or supercontigs), and only loads data from EnsEMBL within the forked child. This is necessary because we use lots of circular references, which Perl's garbage collector fails to clean up. Rather than putting tons of work into tracking down references and eradicating them, we just generate each base feature's data structure within a child, so we know nothing about features from other base features. This probably causes some slowdown since we can't cache some repeated features, but at least it doesn't run over 1GB of physical memory and start swapping when loading Aedes aegypti features like it used to.

- **get_analysisfeature**

  **Arguments:** $feature_id, $analysis_id, $rawscore, $fetch

  Look up or create an analysis-feature relationship. $feature_id, and $analysis_id are required. $rawscore is used to set the score (if any) of the analysis-feature relationship that is being created.

  By default (when the $fetch argument does not evaluate to true), this function will first attempt to create an analysis-feature relationship. If that fails, it will fall back and try to retrieve the analysis-feature relationship. If that fails in turn, it will die() with an error message.
If the fetch argument is set to true (e.g. 1, or 'true', or 'fetch', or 'xyzzy'), then the function will first attempt to retrieve the analysis-feature relationship. Failing that, it will then attempt to create the analysis-feature relationship. Failing that, it will die() with an error message.

This function will return an analysisfeature_id unless it die()s as mentioned.

- **is_final_analysis**

  **Arguments:** $analysis_alignment_feature, $database

  This is a function to deal with some quirks in the way Ensembl handles analysis features. For some reason, there is a large analysis feature that overlaps all the hits for a single query, much like we generate "match" features. There's no function in the Ensembl API to figure out if the hit you've pulled out is one of these large overlapping features, but they are listed in the supporting_features table so we can do some actual SQL and filter them out.

  $analysis_alignment_feature should be an Ensembl alignment feature as returned by get_all_SimilarityFeatures, $database should be either "core" or "est".

  See ChadoLoader::DB::EnsEMBL::get_supporting_feature() for the actual SQL.

- **slice_dice**

  **Arguments:** $slice, $chunk_size

  Given an Ensembl "Slice" object in $slice and a length in $chunk_size, refetches the original slice is $chunk_size sized pieces and returns an array of those pieces.

  Used to avoid pulling all the analysis features for a whole chromosome or supercontig, since doing so can easily yield more results than will fit in memory.

---

**D.10. Module:** ChadoLoader::Modules::EnsEMBLArchiveIDs

**D.10.1. Summary**

ChadoLoader::Modules::EnsEMBLArchiveIDs - ChadoLoader module to load

EnsEMBL microarray features.

**D.10.2. Included modules**

- Bio::EnsEMBL::DBSQL::DBAdaptor
- ChadoLoader::DB::Chado
D.10.3. Example code listing

use ChadoLoader::Main;
my $loader = new ChadoLoader::Main('modules' => ['EnsEMBLArchiveIDs']);
$loader->load();

or

use ChadoLoader::Modules::EnsEMBLArchiveIDs;
use ChadoLoader::Organism::Anopheles_gambiae;
my $ema = new EnsEMBLArchiveIDs(
    new ChadoLoader::Organism::Anopheles_gambiae();
);
$ema->load();

D.10.4. Description

This module loads the following features from EnsEMBL into Chado:

- chromosomes/supercontigs
- archive/stable IDs

Loading is done in a transactional way, and if there is an unrecoverable error, the script die()s theoretically without changing the database (some sequence numbers may still be incremented). All feature insertions are also done in a way that is compatible with data already being in the database. Running the module twice in a row without cleaning the database should be the same as running the script once, albeit slower the second time around as optimizations have been made for feature insertion.

This modules requires that the provided organism support SliceAdaptor and ArchiveStableIdAdaptor on the EnsEMBL core database. It inherits most of its functionality from ChadoLoader::Modules::EnsEMBLBasicFeatures.
D.10.5. Methods

• **new**
  
  **Arguments:** $organism, $debug

  The constructor takes a ChadoLoader::Organism object in $organism.

  It also optionally takes a debugging bitmask in $debug; see the debugging definitions in ChadoLoader::Modules::EnsEMBLBasicFeatures::CONSTANTS.

• **load**

  The workhorse function of this module. Pulls data out of EnsEMBL via the Perl interface, and loads it into Chado. If this class is extended, this is the function that should be overridden by any other Ensembl-connected modules. As with ChadoLoader::Modules::EnsEMBLBasicFeatures, this function draws pretty progress bars.

  This function also avoids committing anything until the very end; this might have to be changed if there suddenly become lots of lots of archive IDs for some reason.

• **get_all_current_ids**

  **Arguments:** $archive_type

  Given an EnsEMBL archive type (e.g. 'gene', 'mRNA', or 'protein'), return an array reference of all EnsEMBL stable IDs for that feature type that are current (those that have nothing in the new_stable_id field).

• **get_assembly_names**

  **Arguments:** $archive_id

  Given an EnsEMBL archive/stable ID, return an array reference of the all the gene build names associated with that archive/stable id.

• **get_assembly**

  **Arguments:** $archive_feature_id, $gene_build_name, $rank

  Look up or create an assembly property for a stable/archive ID's feature. This is basically a wrapper around ChadoLoader::Modules::EnsEMBLBasicFeatures::get_featureproperty() that just uses a fixed cvterm. $archive_feature_id, $gene_build_name, and $rank are all required.
This function looks for an existing featureprop entry in the database first, then creates one if an existing one cannot be found.

In either case, it returns a featureprop_id, or die()s in response to an error from SQL.

D.11. Module: ChadoLoader::Modules::EnsEMBLMicroarrays

D.11.1. Summary

ChadoLoader::Modules::EnsEMBLMicroarrays - ChadoLoader module to load

EnsEMBL microarray features.

D.11.2. Included modules

- Bio::EnsEMBL::DBSQL::DBAdaptor
- ChadoLoader::DB::Chado
- ChadoLoader::Modules::EnsEMBLBasicFeatures
- ChadoLoader::Util
- Term::ProgressBar

D.11.3. Example code listing

```perl
use ChadoLoader::Main;
my $loader = new ChadoLoader::Main('modules' => ['EnsEMBLMicroarrays']);
$loader->load();
```

or

```perl
use ChadoLoader::Modules::EnsEMBLMicroarrays;
use ChadoLoader::Organism::Anopheles_gambiae;
my $ema = new EnsEMBLMicroarrays(
    new ChadoLoader::Organism::Anopheles_gambiae);
$ema->load();
```

D.11.4. Description

This module loads the following features from EnsEMBL into Chado:

- chromosomes/supercontigs
- microarrays (OligoFeatures)
Loading is done in a semi-transactional way, so that individual insertions and deletions and transactional and can be recovered from (as with ChadoLoader::Modules::EnsEMBLAnalysisFeatures). Commits happen once per base genome region (e.g. chromosome or supercontig). If there is an unrecoverable error, the script die()s. All feature insertions are also done in a way that is compatible with data already being in the database. Running the module twice in a row without cleaning the database should be the same as running the script once, albeit slower the second time around as optimizations have been made for feature insertion. This modules requires that the provided organism support SliceAdaptor on the EnsEMBL core database. It inherits some of its functionality from ChadoLoader::Modules::EnsEMBLBasicFeatures.

D.11.5. Methods

- **new**

  **Arguments:** $organism, $debug

  The constructor takes a ChadoLoader::Organism object in $organism. It also optionally takes a debugging bitmask in $debug; see the debugging definitions in ChadoLoader::Modules::EnsEMBLBasicFeatures::CONSTANTS.
The workhorse function of this module. Pulls data out of Ensembl via the Perl interface, and loads it into Chado. If this class is extended, this is the function that should be overridden by any other Ensembl-connected modules. As with ChadoLoader::Modules::EnsEMBLBasicFeatures, this function draws pretty progress bars.

This function does one commit per base feature (chromosome/supercontig). Most fatal errors will hopefully occur during the load of the first base feature, meaning that no changes will be made to the database. As with all of the other loading functions, this function can be run repeatedly without inserting duplicate data.


Curwen 2004: Curwen V. et al., The Ensembl automatic gene annotation system., 2004

Lewis 2002: Lewis, S.E. et al., Apollo: a sequence annotation editor, 2002


