VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

QUERY LANGUAGE FOR BIOLOGICAL DATABASES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ BAHUREK

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DOTAZOVACÍ JAZYK PRO DATABÁZE BIOLOGICKÝCH DAT
QUERY LANGUAGE FOR BIOLOGICAL DATABASES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — TOMÁŠ BAHUREK
AUTHOR

VEDOUCÍ PRÁCE — Ing. TOMÁŠ MARTÍNEK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

S rapidně stoupajícím množstvím biologických dat stoupá i důležitost biologických databází. U těchto databází je nezbytné objevování znalostí (nalezení spojitostí, které nebyli známé v čase vkládání dat). K získávání znalostí z biologických databází je nutná konstrukce složitých SQL dotazů, což vyžaduje pokročilou znalost SQL a použitého databázového schématu. Biologové většinou tyto znalosti nemají, proto je potřeba nástroje, který by poskytl intuitivnějšího rozhrání pro tyto databáze. Tato práce navrhuje ChQL, intuitivní dotazovací jazyk pro databázi biologických dat Chado. ChQL umožňuje biologům poskládat dotaz za použití pojmů, které dobře znají bez nutnosti znát SQL nebo použité schéma. Tato práce implementuje aplikaci pro dotazování databáze Chado pomocí ChQL. Webové rozhrání provede uživatele procesem zostavení věty jazyka ChQL. Aplikace přeloží tuto větu do SQL dotazu, odešle jej do databáze Chado a zobrazí vrácená data v tabulce. Výsledky jsou vyhodnoceny testováním dotazů na reálných datech.

Abstract

With rising amount of biological data, biological databases are becoming more important each day. Knowledge discovery (identification of connections that were unknown at the time of data entry) is an essential aspect of these databases. To gain knowledge from these databases one has to construct complicated SQL queries, which requires advanced knowledge of SQL language and used database schema. Biologists usually don't have this knowledge, which creates need for tool, that would offer more intuitive interface for querying biological databases. This work proposes ChQL, an intuitive query language for biological database Chado. ChQL allows biologists to assemble query using terms they are familiar without knowledge of SQL language or Chado database schema. This work implements application for querying Chado database using ChQL. Web interface guides user through process of assembling sentence in ChQL. Application translates this sentence to SQL query, sends it to Chado database and displays returned data in table. Results are evaluated by testing queries on real data.

Klíčová slova

Dotazovací jazyk, Databáze biologických dat, Chado, Gene ontology, Sequence Ontology, Vaadin

Keywords

Query Language, Biological Database, Chado, Gene ontology, Sequence Ontology, Vaadin

Citace

Tomáš Bahurek: Query language for biological databases, diplomová práce, Brno, FIT VUT v Brně, 2015

Query language for biological databases

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Martínka, Ph.D.

....
Tomáš Bahurek
May 24, 2015

Poděkování

Ďakujem Ing. Tomášovi Martínkovi, Ph.D. za podnetné rady pri vypracovávaní tejto práce.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Considering the vast amount of available biological data, biological databases are one of the most important tools used by biologists. These databases are usually used for knowledge discovery. They are queried to gain some new information, which was not known at time of data entry.

Typical questions asked by biologists are :

- Which genes have promoters containing specified element (triplex, quadruplex, palindrome, transcription factor, etc.)?
- How often can be a specified DNA structure (triplex, quadruplex, palindrome, etc.) found near given transcription factor (for example TP53)?
- Which nonB DNA structures are inside gene with specific gene ID (especially in human genome)?

It can be quite complicated to write such queries in SQL, especially for biologists who are often not familiar with this language. In their queries biologists often use terms like promoter, intron, exon, transcription factor and so on. Some of these terms are not exactly identifiable, for example: promoter can have variable length. Sometimes promoter before gene is taken into account, other times promoter before specific transcript is of interest. Adverbs describing positional relationships are ambiguous as well (What is the distance threshold to consider something being near? Does it still count if objects are overlapping as well?). Biologists are often interested, if a specific element exists near other element, if they are overlapping, or if one element is part of other element.

To summarize, there are two problems:

1. Biologists need a more intuitive way to create queries than SQL.
2. Biologists use ambiguous terms in their queries.

This work proposes solution to these problems in form of ChQL. ChQL (Chado Query Language) is a language for querying Biological database Chado. This language uses terms biologists are familiar with, while leaving the unknown technical terms in the background. Wherever possible it uses default values for ambiguous terms with the option of changing them.

Instead of making the user typing words in ChQL, we created a web interface that navigates user through process of assembling sentence in this language. Interface lets user choose next word from the menu, but offers only words that are syntactically correct in current context. It also lets user send the query only if current sentence is completed. This is to avoid situation when user makes a typo or syntax error and has to figure out how to fix it. When user completes sentence in ChQL, application translates it to SQL query, which it sends to database server and displays returned results in a table.

This is work contains 8 chapters. Chapter 2 explains gene ontology, sequence ontology and GFF3 file format. Chapter 3 shows some of current database schemas for biological data and querying tools for biological databases. Chapter 4 explains Chado and its most relevant modules. Chapter 5 proposes design of a language for biological databases including its translation to SQL. Chapter 6 describes implementation of this translation and entire application. Chapter 7 evaluates results and performance of queries translated to SQL.

Chapter 2

Genome Annotation

2.1 Gene Ontology

The Gene Ontology (GO) project [1] is a collaborative effort to address the need for consistent descriptions of gene products across databases. Founded in 1998, the project began as a collaboration between three model organism databases: FlyBase (Drosophila), the Saccharomyces Genome Database (SGD) and the Mouse Genome Database (MGD). The GO Consortium (GOC) has since grown to incorporate many databases, including several of the world's major repositories for plant, animal, and microbial genomes.

The GO project has developed three structured, controlled vocabularies (ontologies) that describe gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent manner. There are three separate aspects to this effort: first, the development and maintenance of the ontologies themselves; second, the annotation of gene products, which entails making associations between the ontologies and the genes and gene products in the collaborating databases; and third, the development of tools that facilitate the creation, maintenance and use of ontologies.

The use of GO terms by collaborating databases facilitates uniform queries across all of them. Controlled vocabularies are structured so they can be queried at different levels; for example, users may query GO to find all gene products in the mouse genome that are involved in signal transduction, or zoom in on all receptor tyrosine kinases that have been annotated. This structure also allows annotators to assign properties to genes or gene products at different levels, depending on the depth of knowledge about that entity.

2.2 Sequence Ontology

The Sequence Ontology [4] is a set of terms and relationships used to describe the features and attributes of biological sequence. SO includes different kinds of features which can be located on the sequence. Biological features are those which are defined by their disposition to be involved in a biological process. Examples are *binding_site* and *exon*. Biomaterial features are those which are intended for use in an experiment such as aptamer and *PCR_product*. There are also experimental features which are the result of an experiment. SO also provides a rich set of attributes to describe these features such as „polycistronic“ and „maternally imprinted“.

The Sequence Ontologies are provided as a resource to the biological community. They have the following uses:

- To provide for a structured controlled vocabulary for the description of primary annotations of nucleic acid sequence, e.g. the annotations shared by a DAS server (BioDAS, Biosapiens DAS), or annotations encoded by GFF3.
- To provide for a structured representation of these annotations within databases. Were genes within model organism databases to be annotated with these terms then it would be possible to query all these databases for, for example, all genes whose transcripts are edited, or trans-spliced, or are bound by a particular protein.
- To provide a structured controlled vocabulary for the description of mutations at both the sequence and more gross level in the context of genomic databases.

2.3 GFF3 format

GFF3 is a standard file format for storing genomic features in a text file. GFF stands for Generic Feature Format. The description of GFF 3 format has been summarized from [4].

Description of the Format

GFF3 files are nine-column, tab-delimited, plain text files. Literal use of tab, newline, carriage return, the percent (%) sign, and control characters must be encoded using RFC 3986 Percent-Encoding; no other characters may be encoded. Backslash and other ad-hoc escaping conventions that have been added to the GFF format are not allowed. The file contents may include any character in the set supported by the operating environment, although for portability with other systems, use of Latin-1 or Unicode are recommended.

Note that unescaped spaces are allowed within fields, meaning that parsers must split on tabs, not spaces. Use of the „+" (plus) character to encode spaces is deprecated from early versions of the spec and is no longer allowed.

Undefined fields are replaced with the „." character.

Column 1: „seqid"

The ID of the landmark used to establish the coordinate system for the current feature.

Column 2: „source"

The source is a free text qualifier intended to describe the algorithm or operating procedure that generated this feature. Typically this is the name of a piece of software, such as „Genescan" or a database name, such as „Genbank." In effect, the source is used to extend the feature ontology by adding a qualifier to the type creating a new composite type that is a subclass of the type in the type column.

Column 3: „type"

The type of the feature (previously called the „method"). This is constrained to be either:

- a term from the „lite" version of the Sequence Ontology - SOFA
- a term from the full Sequence Ontology - it must be an *is_ a* child of *sequence_feature* (SO:0000110)

- a SOFA or SO accession number.

The latter alternative is distinguished using the syntax SO:000000.

Columns 4 & 5: „start“ and „end“

The start and end coordinates of the feature are given in positive 1-based integer coordinates, relative to the landmark given in column one. Start is always less than or equal to end. For features that cross the origin of a circular feature (e.g. most bacterial genomes, plasmids, and some viral genomes), the requirement for start to be less than or equal to end is satisfied by making end = the position of the end + the length of the landmark feature.

For zero-length features, such as insertion sites, start equals end and the implied site is to the right of the indicated base in the direction of the landmark.

Column 6: „score“

The score of the feature, a floating point number. As in earlier versions of the format, the semantics of the score are ill-defined. It is strongly recommended that E-values be used for sequence similarity features, and that P-values be used for ab initio gene prediction features.

Column 7: „strand“

The strand of the feature. + for positive strand (relative to the landmark), - for minus strand, and . for features that are not stranded. In addition, ? can be used for features whose strandedness is relevant, but unknown.

Column 8: „phase“

For features of type „CDS“, the phase indicates where the feature begins with reference to the reading frame. The phase is one of the integers 0, 1, or 2, indicating the number of bases that should be removed from the beginning of this feature to reach the first base of the next codon. In other words, a phase of „0“ indicates that the next codon begins at the first base of the region described by the current line, a phase of „1“ indicates that the next codon begins at the second base of this region, and a phase of „2“ indicates that the codon begins at the third base of this region. This is NOT to be confused with the frame, which is simply start modulo 3.

For forward strand features, phase is counted from the start field. For reverse strand features, phase is counted from the end field.

The phase is REQUIRED for all CDS features.

Column 9: „attributes“

A list of feature attributes in the format `tag=value`. Multiple `tag=value` pairs are separated by semicolons. URL escaping rules are used for tags or values containing the following characters: „,=;“. Spaces are allowed in this field, but tabs must be replaced with the %09 URL escape. Attribute values do not need to be and should not be quoted. The quotes should be included as part of the value by parsers and not stripped.

These tags have predefined meanings. See table 2.1

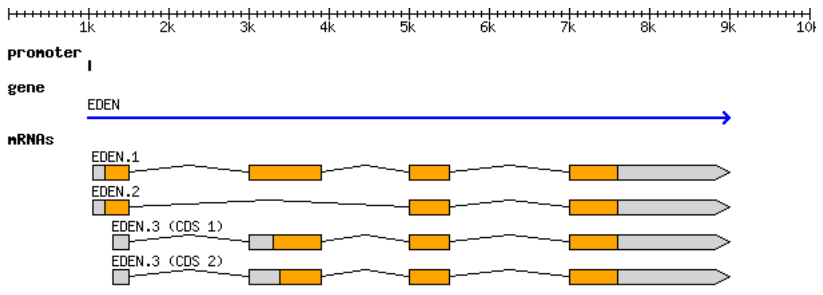


Figure 2.1: Structure of gene [4]

The Canonical Gene

Below is description of the representation of a protein-coding gene in GFF3. To illustrate how a canonical gene is represented, consider Figure 2.1. This indicates a gene named EDEN extending from position 1000 to position 9000. It encodes three alternatively-spliced transcripts named EDEN.1, EDEN.2 and EDEN.3, the last of which has two alternative translational start sites leading to the generation of two protein coding sequences.

There is also an identified transcriptional factor binding site located 50 bp upstream from the transcriptional start site of EDEN.1 and EDEN2.

Here is how this gene should be described using GFF3:

```
##gff-version 3
##sequence-region   ctg123 1 1497228
ctg123  .   gene    1000    9000    .   +   .   ID=gene00001;Name=EDEN
ctg123  .   TF_binding_site 1000    1012    .   +   .   ID=tfbs00001;Parent=gene00001
ctg123 . mRNA            1050   9000  .  +  .  ID=mRNA00001;Parent=gene00001;Name=EDEN.1
ctg123 . mRNA            1050   9000  .  +  .  ID=mRNA00002;Parent=gene00001;Name=EDEN.2
ctg123 . mRNA            1300   9000  .  +  .  ID=mRNA00003;Parent=gene00001;Name=EDEN.3
ctg123 . exon            1300   1500  .  +  .  ID=exon00001;Parent=mRNA00003
ctg123 . exon            1050   1500  .  +  .  ID=exon00002;Parent=mRNA00001,mRNA00002
ctg123 . exon            3000   3902  .  +  .  ID=exon00003;Parent=mRNA00001,mRNA00003
ctg123 . exon            5000   5500  .  +  .  ID=exon00004;Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . exon            7000   9000  .  +  .  ID=exon00005;Parent=mRNA00001,mRNA00002,mRNA00003
ctg123 . CDS             1201   1500  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
ctg123 . CDS             3000   3902  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
ctg123 . CDS             5000   5500  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
ctg123 . CDS             7000   7600  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
ctg123 . CDS             1201   1500  .  +  0  ID=cds00002;Parent=mRNA00002;Name=edenprotein.2
ctg123 . CDS             5000   5500  .  +  0  ID=cds00002;Parent=mRNA00002;Name=edenprotein.2
ctg123 . CDS             7000   7600     .  +  0  ID=cds00002;Parent=mRNA00002;Name=edenprotein.2
ctg123 . CDS             3301   3902  .  +  0  ID=cds00003;Parent=mRNA00003;Name=edenprotein.3
ctg123 . CDS             5000   5500     .  +  1  ID=cds00003;Parent=mRNA00003;Name=edenprotein.3
ctg123 . CDS             7000   7600     .  +  1  ID=cds00003;Parent=mRNA00003;Name=edenprotein.3
ctg123 . CDS             3391   3902  .  +  0  ID=cds00004;Parent=mRNA00003;Name=edenprotein.4
ctg123 . CDS             5000   5500     .  +  1  ID=cds00004;Parent=mRNA00003;Name=edenprotein.4
ctg123 . CDS             7000   7600     .  +  1  ID=cds00004;Parent=mRNA00003;Name=edenprotein.4
```

ID	Indicates the ID of the feature. IDs for each feature must be unique within the scope of the GFF file. In the case of discontinuous features (i.e. a single feature that exists over multiple genomic locations) the same ID may appear on multiple lines. All lines that share an ID collectively represent a single feature.
Name	Display name for the feature. This is the name to be displayed to the user. Unlike IDs, there is no requirement that the Name be unique within the file.
Alias	A secondary name for the feature. It is suggested that this tag be used whenever a secondary identifier for the feature is needed, such as locus names and accession numbers. Unlike ID, there is no requirement that Alias be unique within the file.
Parent	Indicates the parent of the feature. A parent ID can be used to group exons into transcripts, transcripts into genes, an so forth. A feature may have multiple parents. Parent can *only* be used to indicate a part_of relationship.
Target	Indicates the target of a nucleotide-to-nucleotide or protein-to-nucleotide alignment. The format of the value is „target_id start end [strand]“, where strand is optional and may be „+“ or „-“. If the target_id contains spaces, they must be escaped as hex escape %20.
Gap	The alignment of the feature to the target if the two are not collinear (e.g. contain gaps). The alignment format is taken from the CIGAR format.
Derives_from	Used to disambiguate the relationship between one feature and another when the relationship is a temporal one rather than a purely structural „part_of“ one. This is needed for polycistronic genes.
Note	A free text note.
Dbxref	A database cross reference.
Ontology_term	A cross reference to an ontology term.
Is_circular	A flag to indicate whether a feature is circular.

Table 2.1: Tags in GFF3 format [4]

Chapter 3

State of the Art

3.1 Biological databases

BioSQL

BioSQL is a generic unifying relational schema for storing sequences and sequence annotations from different sources, for instance Genbank or Swissprot. It is also well suited to work with phyloninformatics and phylogenetic trees. BioSQL is meant to be a common data storage layer supported by all the different Bio* projects, Bioperl, Biojava, Biopython, and Bioruby. Entries stored through an application written in, say, Bioperl could be retrieved by another written in Biojava [10].

BioSQL is quite focused and is concerned with:

- Sequence
- Sequence annotation
- Phylogeny
- Publications

Extension modules extend the core schema and are optional unless specifically needed for storing or retrieving the respective data types the module accommodates. At present, there is one extension module, PhyloDB, for storing phylogenetic trees and taxonomies. Supported RDBMs are at present PostgreSQL, MySQL, Oracle, HSQLDB, and Apache Derby for the core schema. The current release of the BioSQL core schema is v1.0.1. made in August 2008 [11].

Chado

Chado is a relational database schema now being used to manage biological knowledge for a wide variety of organisms, from human to pathogens, especially the classes of information that directly or indirectly can be associated with genome sequences or the primary RNA and protein products encoded by a genome. Biological databases that conform to this schema can interoperate with one another, and with application software from the Generic Model Organism Database (GMOD) toolkit. Chado is distinctive because its design is driven by ontologies. The use of ontologies (or controlled vocabularies) is ubiquitous across the schema, as they are used as a means of typing entities. The Chado schema is partitioned into integrated subschemas (modules), each encapsulating a different biological domain, and each described using representations in appropriate ontologies [8].

3.2 Biological querying tools

UCSC Genome Browser

The UCSC Genome Browser was created to provide a graphical viewpoint on the very large amount of genomic sequence produced by the Human Genome Project. The client–server model employed by the UCSC Genome Browser has the advantage to the user of offering access to a very large database of information in a uniform interface with no overhead of importing datasets. The footprint of the data underlying the Genome Browser currently amounts to 7.7 Terabytes (TB) of data in flat files (primarily sequence) and 3.3 TB of tables in a MySQL database (annotations).

Database Structure

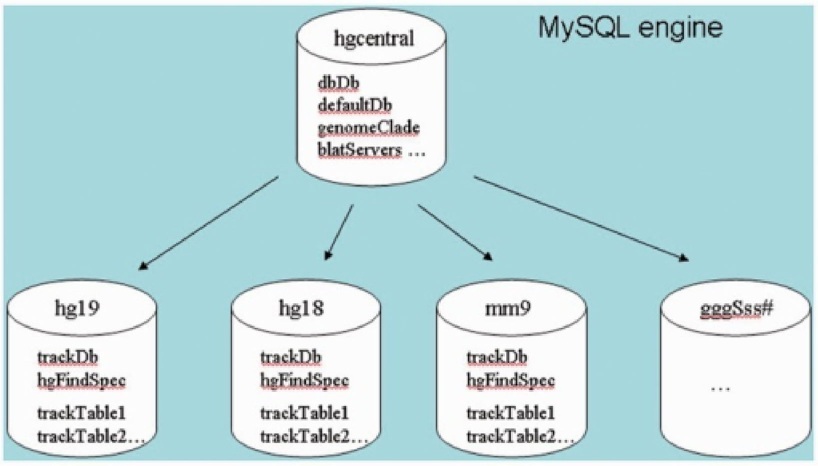


Figure 3.1: Summary database schema for UCSC Genome Browser [6]

The construction of a basic genome browser on a new assembly begins with assigning a name. Because assemblies are given a variety of names by the different sequencing centers and no standard nomenclature exists across all organisms, the UCSC database has standardized on a naming system that is internally consistent, though it has evolved. The first assembly for an organism is given a name in the format `gggSss#` using the first three characters of the genus and species names, with subsequent assemblies incrementing the number. For example, the cow, Bos taurus, currently has assemblies bosTau2 through bosTau6 on the public site (bosTau1 having been archived).

Two other types of names continue to be used for organisms whose initial assemblies predate the introduction in 2003 of the six-letter naming scheme. Human assemblies are called `hg##`, because the database and Genome Browser began when UCSC was generating the human assemblies in-house and the human genome was the only sequence represented. The next several organisms to have browsers were given two-letter names in the format, `gs#`, based on scientific names, until the two-character namespace proved inadequate; mouse (Mus musculus) and rat (Rattus norvegicus) are `mm#` and `rn#`, respectively.

Data for each organism are stored in a separate database in the MySQL database system, which is built in a modular design. Each genome assembly has a database named with the genome name as described above (`hg#`, `gs#` or `gggSss#`). This database contains all the assembly-specific tables needed to create a display in the browser graphic, including individual tables for annotation datasets and several tables of metadata describing display parameters, configuration options, etc., specific to the assembly (see Figure 3.1).

Several additional databases contain information used by more than one assembly (e.g. data needed to show all the organisms in a pulldown menu). Thus, the hgcentral database contains metadata about the assemblies, including genome name, scientific name, location of the .2 bit file, official assembly name and date and other information. Similarly, the hgFixed database (not shown in Figure 3) contains global information, such as restriction enzyme recognition sites [6].

UCSC Table Browser

The UCSC Table Browser provides text-based access to a large collection of genome assemblies and annotation data stored in the Genome Browser Database. A flexible alternative to the graphical-based Genome Browser, this tool offers an enhanced level of query support that includes restrictions based on field values, free-form SQL queries and combined queries on multiple tables. Output can be filtered to restrict the fields and lines returned, and may be organized into one of several formats, including a simple tab-delimited file that can be loaded into a spreadsheet or database as well as advanced formats that may be uploaded into the Genome Browser as custom annotation tracks.

The UCSC Table Browser data retrieval tool is built on top of the Genome Browser Database, a set of MySQL relational databases that each store sequence and annotation data for one genome assembly (1). Tables within the databases may be differentiated by whether the data are based on genomic start-stop coordinates or are independent of position.

Non-positional tables contain data not tied to genomic location. Some non-positional tables relate internal numeric mRNA IDs to extended information such as author, tissue or keyword. Other 'meta' tables contain information about the structure of the database itself or describe external files containing sequence data.

The databases contain optimizations to support range-based queries from the Table Browser and Genome Browser. Smaller tables are indexed on a few critical fields and the data are presorted prior to loading into the database. With larger tables, the data are separated by chromosome into smaller tables, and a binning scheme is implemented on the larger chromosome tables.

In some situations graphical browser may not be optimal tool for working with genomic data. User might wish to view the raw data or examine the relationships between the tables underlying the browser. It is often desirable to filter the display output with greater restrictions than are offered by the Genome Browser, or to output the data in a text-based format that can be imported into other programs.

The UCSC Table Browser provides a powerful and flexible alternative for querying and manipulating the annotation tables within the Genome Browser Database. Using Table Browser form-based or free-form queries, one may quickly and easily extract subsets of the database, in many cases eliminating the need to set up a local copy of the MySQL database. By configuring the tool's output options, the user can generate a custom annotation track that may be automatically added to the graphical browser session, or create a file in one of several output formats that can be used as input into other programs. The Table Browser can also display basic statistics calculated over a selected subset of data.

Basic Data Queries

The Table Browser can be used to retrieve a specific subset of records from a table in a selected genome assembly. The user specifies a position of interest within the assembly (or the keyword ‘genome’ to access data from the entire assembly), selects a table, and then chooses the‘Get all fields’ option. The Table Browser displays the query results in a tab-delimited text format that can be easily downloaded and imported into text editors, spreadsheets and other databases, or may be further processed by the user’s own scripts. For example, a user who is examining alternative splicing in the human genome might be interested in downloading the indices of all mRNA sequences that align to a chromosomal region containing a particular gene. One would set the Table Browser to the gene position, select the *chrN_mrna* positional table, and then click the Get all fields button. This query produces a tab-delimited list of names and positions of mRNAs that align to the specified location.

Advanced queries

Although basic data retrieval is useful, the real power of the Table Browser lies in the ability to filter and refine queries, intersect query results from different tables and configure the resulting output. These options may be accessed through the Table Browser’s set of advanced query features. The available query formats and output options vary by table. Many apply only to tables in which the data is position-oriented, thus preserving the database distinction between positional and non-positional tables. Position-based tables may be further differentiated by the types of data they characterize. For example, alignment tables describe a block structure for each element, but other tables may describe only a starting and ending position. Still others may specify translation start and end positions as well as transcription start and end points.

Filtering

The most flexible feature in the Table Browser is its filtering mechanism. The form-based filter provides a straightforward interface for configuring simple SQL-based queries of the data. By default, a Table Browser search retrieves all records for a specified coordinate range or position. Using the filter, the user may set constraints on the values of some or all of the fields within a table to restrict the set of records retrieved from the query range.

The text fields within the filter support wildcard pattern matching and multiple entries. If any word or pattern within the text field matches the value, then the record meets the constraint on that field. Numeric field comparisons support the operators <, >, and != (not equal) and allow comparisons with ranges of numbers.

To satisfy the needs of advanced users who find the form-based filtering options to be insufficient, the Table Browser also supports free-form queries allowing more complex constraints, typically to relate two or more fields within the selected table. These queries, which use SQL ’where’ clause syntax, can combine simple constraints with AND, OR and NOT, using parentheses as needed for clarity. A basic free-form constraint consists of a field name (or an arithmetic expression of numeric field names), a comparison operator and a value.

For example, when searching for gene models in which a promoter region may be present, the simple free-form query (txStart != cdsStart) on the refGene table will produce a list of genes that have the expected 5' untranslated region (UTR) upstream sequence. Note that if the strand is negative, this will search for cases of 3' UTR downstream sequence. In a more complex version of the previous query, (txStart != cdsStart) AND (txEnd != cdsEnd) AND (exonCount = 1) will return a list of single exon genes with both 5'and 3' flanking UTRs.

Multiple table comparisons

At times one may wish to compare the data between two tables to determine whether any features have positions in common within the genome. The Table Browser provides a simple interface offering the choice of several types of table comparisons based on feature positions. One class of comparisons preserves the gene or alignment structure of the primary table, resulting in output that describes the same type of feature as is shown in that table. Primary table features are kept or discarded based on the amount of positional overlap with features contained in the secondary table. The user controls the query output by specifying the threshold of overlap: any, none or a percentage. For example, one might want to identify all the spliced ESTs that align to a particular region in the Known Genes annotation track. The user would select the location of interest in the Table Browser, choose the chrN_intronEst table, and then proceed to the advanced query options. Intersecting the EST table with the knownGene table results in the desired list. A second class of intersections and unions compares the positions of table features one base position at a time. These queries return only position ranges and do not preserve the structure of the primary table. A base-by-base intersection of two tables will include the base in the output if the nucleotide position is covered by at least one feature of both tables. In a union, the base position need only be covered by the feature of one table.

Retrieving subregions of features

In addition to the SQL constraints on queries, the Table Browser allows the user to specify which subregions of features should be present in the output. For example, someone interested in promoters may want to view the region covered by a gene as well as 5000 additional bases upstream from the 5' end (or downstream from the 3' end on the negative strand). The set of available subregion constraints varies among table types. For instance, gene prediction tables specify both exon structure and translated region. The user may constrain the output to show upstream and downstream regions, exons, introns, or 5', 3', or coding exons. Alternatively, alignment tables, which specify block structure but not translated region, offer only upstream, downstream, blocks or inter-block regions.

Flybase

FlyBase [9], a database of Drosophila genes and genomes, was created in 1992 as a resource for collecting and disseminating Drosophila-related information. The website contains over 2.5 million report pages incorporating data from over 42 000 primary Drosophila research papers and an ever-increasing number of genome-scale projects. As the amount, detail and scope of data in FlyBase has increased, the range of data-retrieval tools has been expanded and improved to ensure that the data in FlyBase are as accessible as possible.

FlyBase curates a variety of data from published biological literature, including phenotype, gene expression, interactions (genetic and physical), gene ontology (GO) information and many others. These data are organized in 31 different data-type reports such as the Gene Report or the Allele Report. The range of data provided increases and changes as new types of data become available.

Data-Mining Tools

The tools Flybase provides range from straightforward general search tools and datatype-specific tools to more sophisticated search tools that facilitate wide-range querying of multiple data types simultaneously. For straightforward searches, the FlyBase homepage contains the search tool QuickSearch, as well as quick links, via large icons, to a variety of other popular tools, such as BLAST and GBrowse. All tools in FlyBase can be found by using the Tools drop-down menu in our navigation bar, found on every FlyBase page.

QueryBuilder

QueryBuilder (http://flybase.org/.bin/qbgui.fr.html) is a sophisticated web-based search tool that allows combinatorial searching of any fields from any report in FlyBase. Its advanced search capability takes maximum advantage of the data field layout in the underlying Chado database, but its easy-to-use interface means that absolutely no knowledge of this table structure is required. Query segments are built one by one to create complex searches. Data fields are chosen that are specific to a given data type (e.g. the Symbol field from the Gene Report, or the Author field from the Reference Report). After the data type of interest is selected, only appropriate fields are shown as options for each data type, and the fields are presented in similar order and format as in FlyBase report pages to aid navigation. Individual query segments can then be combined using Boolean operators (AND, OR and BUT NOT) to build up complex searches. Depending on the fields selected, search criteria can include text strings, CV terms or numbers. Number fields can include calculations and logical functions such as greater than (>) or less than (<), and for many fields an index dictionary is available to allow user to see the most commonly used terms. Wild cards are also allowed, to give user the best chance of carrying out the most appropriate search. An auto-complete function facilitates entry of productive queries. When clicking on the QueryBuilder button on the homepage user is presented with three options: 'Use a query template', 'Import a saved query' or 'Build a new query'. For first time users the 'Use a query template' option is recommended as a starting point. It contains series of pre-constructed templates covering the most commonly used QueryBuilder queries. These templates are divided into sections according to data type and are fully editable allowing user to adapt them to their individual search criteria.

Chapter 4

Chado

Chado is a relational database schema that underlies many GMOD installations. It is capable of representing many of the general classes of data frequently encountered in modern biology such as sequence, sequence comparisons, phenotypes, genotypes, ontologies, publications, and phylogeny. It has been designed to handle complex representations of biological knowledge and should be considered one of the most sophisticated relational schemas currently available in molecular biology. The price of this capability is that the new user must spend some time becoming familiar with its fundamentals [8].

4.1 Modules

The Chado schema is built with a set of modules. A Chado module is a set of database tables and relationships that stores information about a well-defined area of biology, such as sequence or attribution.

Arrows are dependencies between modules. Dependencies indicate one or more foreign keys linking modules.

- General - Identifying things within the DB to the outside world, and identifying things from other databases.
- Controlled Vocabulary (cv) - Controlled vocabularies and ontologies
- Publication (pub) - Publications and attribution
- Organism - Describes species; pretty simple. Phylogeny module stores relationships.
- Sequence - Genomic features and things that can be tied to or descend from genomic features.
- Map - Maps without sequence
- Genetic - Genetic data and genotypes
- Companalysis - Storage of Computational sequence analysis. The key concept is that the results of a computational analysis can be interpreted or described as a sequence feature.

The modules that will be of interest in this work are: sequence and cv.

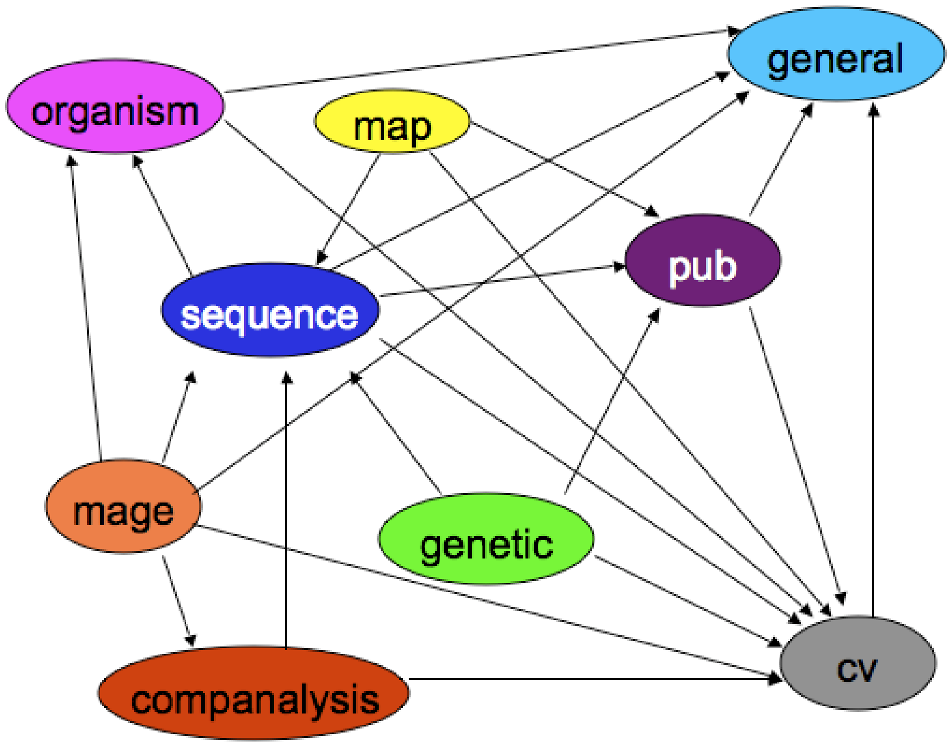


Figure 4.1: Modules in Chado and their relationships [8]

4.1.1 General Module

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

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

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

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

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

4.1.2 Sequence Module

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

Features

Chado does not distinguish between a sequence and a sequence feature, on the theory that a feature is a piece of a sequence, and a piece of a sequence is a sequence. Both are represented as a row in the feature table.

There are many different types of features. Examples include gene, exon, transcript, regulatory region, chromosome, sequence variation, polypeptide, protein domain and cross-genome match regions. Chado does not have a different table for each kind of feature; all features are stored in the feature table.

Feature types are taken from the *Sequence Ontology controlled vocabulary* (see also Controlled Vocabulary module, also known as cv). Types of feature are differentiated using a `type_id` column, which is a foreign key to the cvterm table in the cv (ontology) module, described here. This allows us to type features according to the Sequence Ontology. The use of ontologies to type tables gives Chado a subtyping mechanism, which is absent from the standard relational model. For example, SO tells us that mRNA and snRNA are different kinds of transcript. It can be assumed that any reference to genes, exons, polypeptides, SNPs, chromosomes, transcripts and various kinds of RNAs and so on refers to features of that Sequence Ontology type.

A selection of Chado-relevant types from SO are shown below:

SO Term	**SO id**
Exon	SL:0000025
Intron	SL:0000027
mRNA	SL:0000037
miRNA	SL:0000044
regulatory_element	SL:0000052
transcription_factor_binding_site	SL:0000054

Table 4.1: Sequence ontology terms in Chado [8]

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

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

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

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

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

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

Names of Features

The `name` and `uniquename` columns allow features to be labelled. The `name` column is optional, but it is recommended that all annotated features (as opposed to those that arise from purely computational methods) have names. The name should be a simple, concise, human-friendly display label (such as a gene or gene product symbol, as defined by the nomenclature rules of governing the organism). User interface software (such as GBrowse and Apollo) can use the name column for labelling feature glyphs in user displays. Uniqueness of name within any particular organism or genome project is a desirable characteristic, but is not enforced in the schema, since there are occasions where name clashes are unavoidable. In contrast, the uniquename column is required, and guaranteed to be unique when taken in combination with organism_id and type_id. This is enforced by a constraint in the relational schema. The unique name may be human-friendly (for example, it can be the same as the name); however, it is not guaranteed to be so, and in general should not be displayed to the end user. Its use is mainly as an alternate unique key on the table .

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

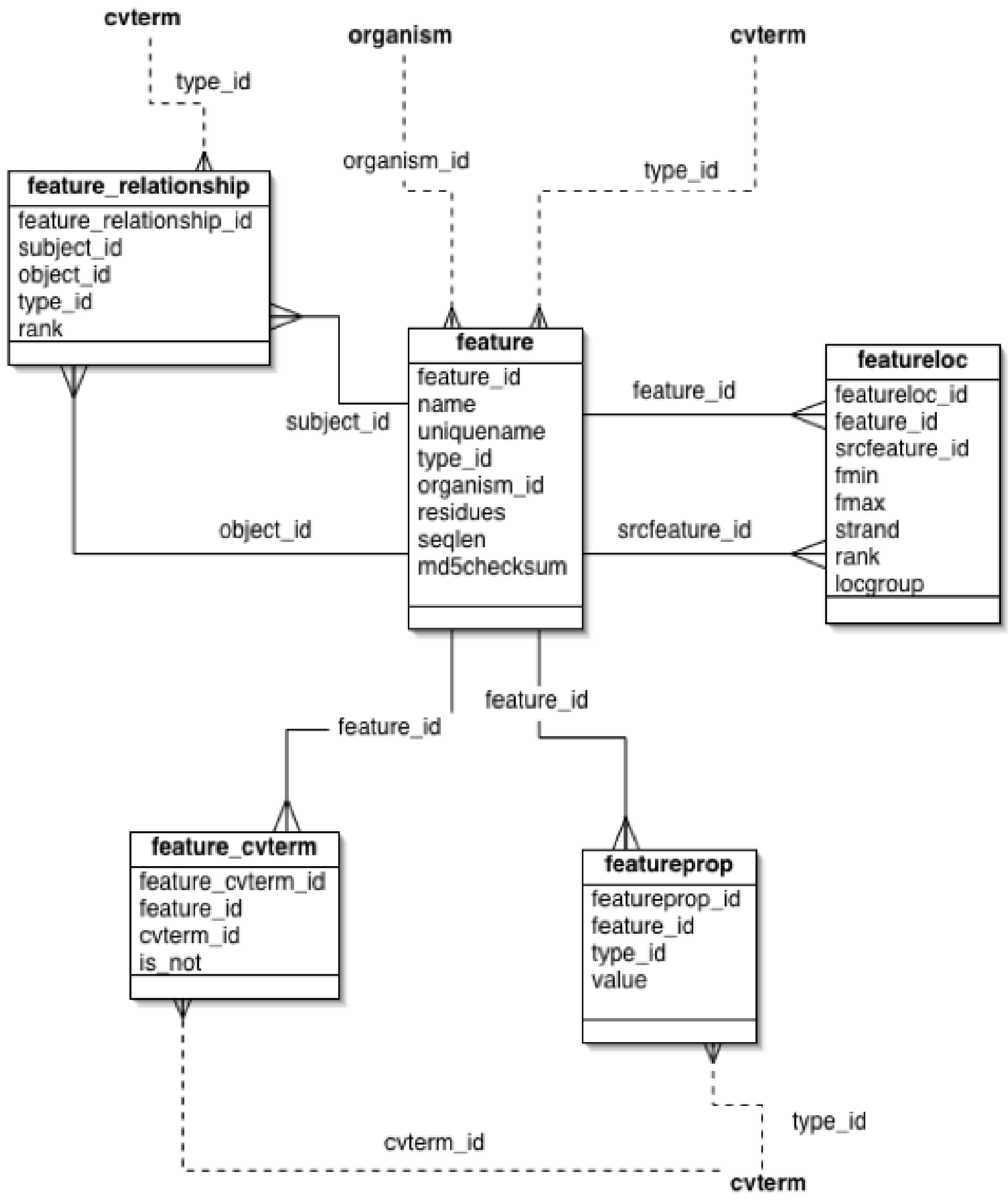


Figure 4.2: Tables in Feature Module [8]

Feature Synonyms

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

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

Feature synonyms are found by joining to `feature_synonym` and synonym. Below is an example query to find gene by name or synonym:

```sql
SELECT feature_id FROM feature
WHERE name = 'name_of_interest'
UNION SELECT feature_id
FROM feature_synonym fs, synonym s
WHERE fs.synonym_id = s.synonym_id
AND s.name = 'name_of_interest'
AND fs.is_current;
```

Feature Locations

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

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

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

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

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

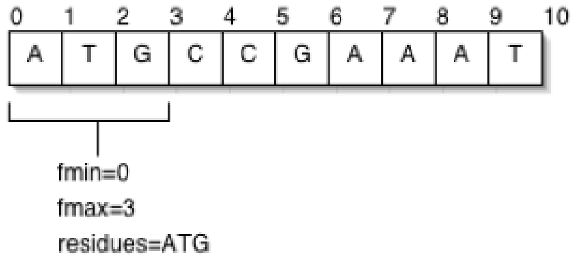


Figure 4.3: Interbase sequence coordinates [8]

The Feature Location Graph

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

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

The schema does not constrain the maximum depth of the LG. This flexibility proves useful when applying Chado to the highly variable needs of multiple different genome projects; however, it can lead to efficiency problems when querying the database. It can also make it more difficult to write software to interoperate with the database, as the software must take into account different contingencies. We can solve this problem by collapsing the LG, in which a graph of arbitrary depth is flattened to a depth of 1, transforming or project-

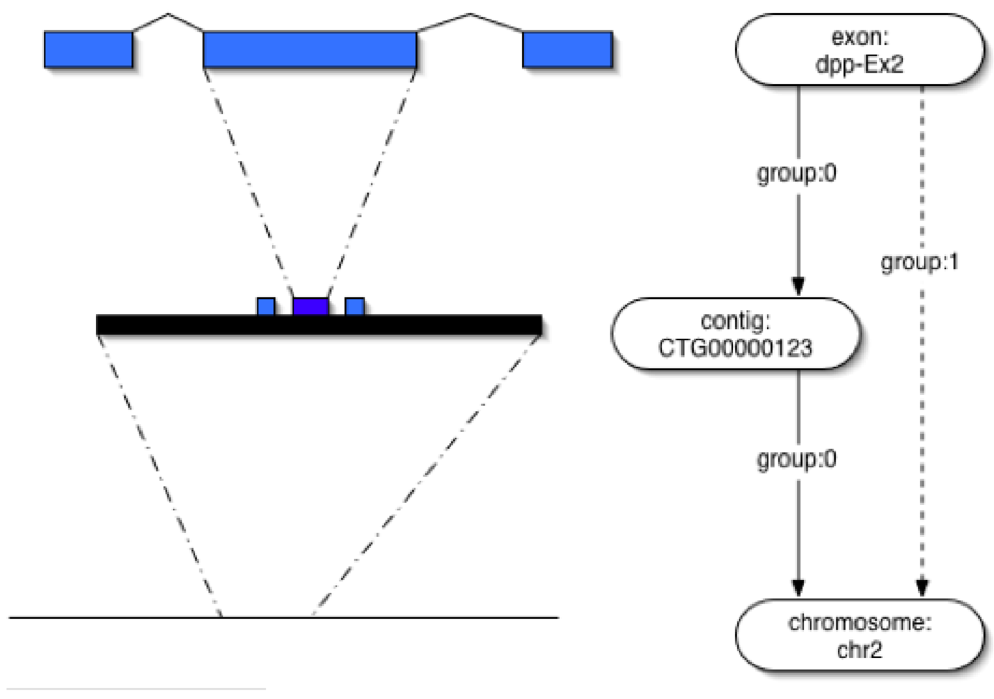


Figure 4.4: Example of featureloc graph [8]

ing featurelocs onto the root features (typically chromosomes or chromosome arms). The original featurelocs are left unaltered in the database, and additional redundant featurelocs between leaf and root features are added to the database. These new featurelocs are known as inferred featurelocs. In the schema inferred featurelocs are differentiated from direct featurelocs using the locgroup column. Direct (non-inferred) localizations are indicated by the locgroup column taking value 0, and transitive localizations are indicated by this column having value !0.

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

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

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

Feature Coordinates

Features are located relative to other features using the featureloc table rows. Features can be located on more than one sequence. For example, a BLAST hit HSP can be a feature of both the query and target sequences. To locate a feature, create a featureloc record with:

- `srcfeature_id` = the id of the sequence on which the feature is being located
- `feature_id` = the id of the feature being located
- `strand` is 1 for the positive strand, -1 for the negative, and 0 for both or indifferent.
- `fmin`, `fmax` – the minimum and maximum coordinates of the interval
- `is_fmin_partial`, `is_fmax_partial` = true if needed to indicate that the sequence is incomplete (e.g. for ESTs or EST assemblies which are known to not go all the way to the 3' or 5' end.)
- `phase` = 0, 1, or 2 – denotes phase of first base pair in a nucleotide feature with respect to a source protein, or the offset of the first nucleotide in its codon.
- `rank`, `locgroup` – these are used to organize groups of feature locations and can be ignored in simple cases (the details are discussed below).

Multiple Locations for a Feature

The ability to have multiple locations for a feature has many uses. For example one can locate a SNP, exon, or protein motif on the genome, on a transcript, and on a protein. A region of similarity between two sequences (HSP) can be located on both of them, so if either is viewed the „hit“ is visible.

Difference Between the Chado Location Model and Other Schemas

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

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

```
CDS       /gene="Acph"
          join(914..1063, 1143..1241, 1297..1536, 1605..2054,
                2667..2925, 3063..3172)
```

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

Feature Rank

The rank field is used when a feature has more than 1 location, otherwise the default rank value of 0 is used. Some features have two locations, for example BLAST hits and HSPs: one location on the query, rank = 0, and one location on the subject, rank = 1.

Extensible Feature Properties

The feature table has a fairly limited set of columns for recording feature data. For example, there is no anticodon column for recording the RNA triplet for the adapter in a tRNA feature (all feature types, including tRNAs, are recorded as rows in the feature table). If we were to add columns such as anticodon then the number of columns in the table would become very large and difficult to manage; most would end up being nullable (for example, anticodon does not apply to non-tRNA features). This is because different organisms, different types of feature and different projects have differing needs regarding what extra data should be attached to any one feature. How then are we to attach both biologically relevant and project specific data to features?

Chado solves this by using an extensible mechanism for attaching attribute-value pairs to features via the featureprop table. The `featureprop.type_id` foreign key column references a property in the Sequence Ontology. The value text column stores the value filler for that property. Sets or lists of values for any property can be stored in the featureprop table, differentiated by the value of the rank column. Provenance for the featureprop assignment is stored using the `featureprop_pub` table in the publications module, allowing multiple publications to be associated with any one assignment.

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

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

Linking Features to External Databases

Public database identifiers are stored in the dbxref table, which holds the database name, the accession number, and an optional version number. Note that this table holds accession numbers published internally by the Chado instance as well as by other databases. A feature can have a primary dbxref, which is linked directly from the feature table. It can also have additional secondary dbxref's linked via `feature_dbxref`. A feature need not have a primary dbxref; e.g. computed features may be considered "lightweight" and not assigned accession numbers. Some groups may wish to set up a trigger to automatically assign primary dbxrefs to features of types that are locally accessioned; a sample trigger is provided with the schema.

Feature Annotations

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

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

Annotations for existing features can also go into the featureprop table using the Chado `feature_property` ontology (defined in `chado/load/etc/feature_property.obo`) and the comment or description terms as appropriate. The purpose of the feature property ontology (and the related `chado/load/etc/genbank_feature_property.obo` file) is to capture terms that are likely to appear in GFF or GenBank sequence files. In theory there is no overlap between these ontologies and the Sequence Ontology.

Relationships Between Features

Biological features are inter-related; exons are part of transcripts, transcripts are part of genes, and polypeptides are derived from messenger RNAs. Relationships between individual features are stored in the `feature_relationship` table, which connects two features via the `subject_id` and `object_id` columns (foreign keys referring to the feature table) and a `type_id` (a foreign key referring to a relationship type in an ontology, either SO, or the OBO relationship ontology, OBO-REL, indicating the nature of the relationship between subject and object features.

The core relationships between features are part-whole (`part_of`) or temporal (`derives_from`). Subject and Object describes the linguistic role the two features play in a sentence describing the feature relationship. In English, many sentences follow a subject, predicate, object syntax, and word order is important. To say that "exons are part of transcripts" is the correct way to describe a typical biological relationship. To say "transcripts are part of exons" is either grammatically or biologically incorrect.

We use this same terminology (which comes from RDF) again in the cv module. The collection of features and feature relationships can be considered as vertices and edges in a graph, known as the Feature Graph (FG). Example feature graphs are shown above and in the Introduction to Chado.

The FG is independent of the LG and in general the FG and the LG should have no edges in common. If there is a featureloc connecting two features, then the addition of a feature relationship between these same two features is redundant. The FG is required in order to query the database for such things as alternately spliced genes, exons shared between transcripts, etc.

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

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

Compliance

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

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

Chado Compliance Layers Proposal for levels of compliance.

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

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

Level 2: Graph Level 2 conformance is graph conformance to SO - all feature relationships between a feature of type X and Y must correspond to relationship of that type in SO; for example, mRNA can be part of gene, but mRNA can not be part of golden path region. [more detailed/formal explanation to come]. In practice Level 2 conformance may be undesirable, we may need to make modifications to SO. Orthogonal to these layers are various additional policy decisions. Some of these are more tolerant of non-conformance than others. (there is also some overlaps with levels 1 and 2).

4.1.3 CV Module

This module is for controlled vocabularies (CVs), semantic networks and ontologies, depending on which terminology you prefer. It is intended to be rich enough to encapsulate anything in the Gene Ontology (GO) or OBO family of ontologies. The schema reflects the data model of OBO and of the OBO Edit tool currently used by these projects. This module is also intended to be extensible to richer formalisms such as OWL (Ontology Web Language), but this is outside the current requirements. The schema is similar to the GO database schema, which was also developed by one of the Chado designers.

Overview

An ontology, or controlled vocabulary (CV) is a collection of classes (or concepts or terms, depending on your terminology) with definitions and relationships to other classes. Each class (a word or phrase) can only appear once in a controlled vocabulary and has a defined meaning within that vocabulary. The controlled vocabularies are chosen so that the contents do not overlap; if the same text string is used to describe two different concepts in two different CVs, these are distinct classes. These terms are housed in the cvterm table in the

Chado schema. CVterms are related to one another via cvter_relationship. This can be thought of as a graph, or semantic network. The relationship types (the labels on the arcs of the graph) are also stored in the cvterm table. The relationship types are extensible, but the type is a (subtyping relationship) is assumed to be present; many OBO ontologies use the part of relationship, and GO also uses the regulates relation. Relationship types also come from a controlled vocabulary, the OBO Relation Ontology. The cvterm_relationship can be thought of as specifying sentences about the cvterms. These sentences have 3 parts - a subject term, an object term, and a verb or type. For example in the phrase „an exon is part of a transcript“ the subject of the sentence is „exon“ and the object is „transcript“. If you prefer to think of it as a directed graph), then you can think of the subject as the child node, and the object as the parent node.

Associating features to cvterms

This module is used by most of the Chado modules. But it is useful to describe here how this module would be used in the context of the sequence module. It is often desired to attach cvterms to features. One example is typing features with SO - this is central to the sequence module. Each feature has one primary type, stored in featuretype_id. We can also attach an arbitrary number of non-primary cvterms to a feature. For example, we may want to attach GO annotations to gene or protein features. We may also want to attach phenotypic terms to gene features (although the preferred way to do this is via a genotype using the genetics module).

Complex annotations

The sequence module makes extensive use of terms taken from various ontologies such as SO and the OBO Relations Ontology, using the `type_id` foreign key column. In addition, features can be annotated using ontologies such as GO using the feature_cvterm linking table. These terms are modelled using the cv module, the core of which is the cvterm table. The chado cv module is based on the GO Database schema. Terms are stored in the cvterm table, and relationships between terms are stored in the cvterm_relationship table. This table follows an analogous structure to the feature_relationship table, in that it has columns subject_id, object_id and `type_id`. Here, all three of these foreign keys refer to rows in the cvterm table. A brief treatment of relationship types in biological ontologies can be found here. Of particular interest to Chado is the is_a relation, which specifies a subtyping relationship between two terms or classes. Recall that tables in the sequence module frequently (such as the feature table) defined a `type_id` foreign key column to indicate the specific type or class of entity for each row in that table. The combination of the `type_id` column and the is_a relationship gives Chado a data sub-classing system, beyond what is possible with traditional SQL database semantics. The collection of cvterms and cvterm_relationships can be considered to constitute vertices and edges in a graph. This graph is typically acyclic (a DAG), though it is not guaranteed to be as certain relationship types are allowed to form cycles.

Chapter 5

Design

5.1 Schema and interface

Before designing language for querying biological database two decisions needed to be made: (1) Which biological database to use, and (2) What kind of user interface (or style) to use for querying.

For database, UCSC Genome browser was considered mainly for its popularity. Unfortunately it uses its own custom schema, hence language designed for this schema would work only for UCSC Genome browser.

Two other relational database schemas for biological data considered for were BioSQL and Chado. While BioSQL is more simple and better interacts with Bio* projects (Bioperl, Biojava, Biopython, and Bioruby), Chado offers more complex schema, connecitvity with other GMOD tools, more detailed documentation and capability to import data from GFF3 files. Chado is also successfully used as schema for Flybase project, for which it was originally designed. That doesn't mean it can be used only for Drosophila. Since it was made generic and extensible, it can be used for any model organism data. For these reasons Chado was chosen to be used as a database schema.

Regarding querying style two tools were considered: UCSC Table Browser and Flybase QueryBuilder.

UCSC Table browser offers filtering (returning only those records where selected columns have selected values) with wildcard pattern matching, comparisons (<, >, and !=) for numeric fields and simple user-defined free-form queries. However it doesn't offer pre-defined queries or templates.

Flybase has QueryBuilder, that allows to build queries either from scratch or from prepared template queries. Its interface is user-friendly and useful even for users who are not familiar with underlying database structure.

Common queries by biologists can have certain ambiguities. For example, biologist can be searching for something related to promoter. Promoters are usually not included in organism data sets and this creates uncertainty in how to translate such query to SQL. For example: one gene can have more transcripts and we can consider promoters before each of these transcripts. Unfortunately existing applications do not deal with such uncertainties.

5.2 Solution proposal

We will propose a simple language that biologists can use to query the database. This language will also deal with some of the mentioned uncertainties, that existing applications don't consider. Our application will translate query in this language to SQL. The language will be designed to use Chado as its underlying database schema. It will mainly use Sequence Module, which can be considered Chado's main module. It will focus on queries regarding sequence features, feature localizations and relationships between features from localization point of view. Application will offer a web interface for navigating the user in assembling the query.

5.2.1 Creation of queries

Before designing our language we'll show how to manually create SQL queries for Chado. For this purpose we have 4 queries written in natural language that will be translated to SQL.

- Find nonB DNA element in promoter of (specific) gene.
- Find nonB DNA element near (specific) transcription factor.
- Find nonB DNA element in nth intron of (specific) gene.
- Find nonB DNA element in (specific) gene.

We will describe how to construct SQL queries for Chado database representing the 4 example queries in natural language. We will explain each term for each query excluding terms that were already explained in some of the previous queries. When speaking about specific column of table following naming convention will be used: `column_name` (`table_name`). The meaning is the same as `table_name.column_name` in SQL command.

Find nonB DNA element inside promoter of a gene

In our example query (see apendix B.2.1) we chose triplex as element (specifying only „ss_type“), used promoter 100bp long and used strict definition of word inside.

Non-B DNA element is a an entry in table `feature`, which has a corresponding entry in `cvterm` table with column `name` specifying the Non-B DNA element. These two tables are joined on equal values of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). Sometimes terms in Chado are not specific enough to distinguish different non-B DNA elements, but we can insert custom feature property to database for this purpose. This property would be called `ss_type`. In this case, if we were looking for **triplex**, we would look for entry in `cvterm` table with column `name` having value „DNA_sequence_secondary_structure“. We would also join table `featureprop` on same value of columns `feature_id` (`feature`) and `feature_id` (`featureprop`) where value of column `value` (`featureprop`) is „triplex“. We would then join `cvterm` table on same values of `type_id` (`featureprop`) and `cvterm_id` (`cvterm`) where value of column `name` (`cvterm`) equals „ss_type“. In case we use „ss_type“, joining `cvterm` table is not necessary, since results will be the same.

Gene is a an entry in table `feature`, which has a corresponding entry in `cvterm` table with `name` having value „gene“. These two tables are joined on equal values of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). If we were looking for specific gene, the value of column `name` (`feature`) would be same as the name of this gene.

Promoter is a region of DNA that initiates transcription of a particular gene. Promoters usually aren't stored in database, but it's known they are located near the transcription start sites of genes, on the same strand and upstream on the DNA and that their length can be about 100–1000 base pairs. With this information an arbitrary number can be chosen (for example: 100) and any sequence of this length ending at the beginning of a gene can be considered a promoter. If we join entry in `feature` table for gene with its corresponding entry in `featureloc` table, we will get location of the gene. These two tables are joined on equal values of columns `feature_id` (`feature`) and `feature_id` (`featureloc`). Column `fmin` (`featureloc`) marks start of the gene and column `fmax` (`featureloc`) marks end of the gene. If promoter has a length of 100 base pairs, then beginning of promoter is represented by `fmin` (`featureloc`) `- 100` and end of the promoter is represented by `fmin` (`featureloc`).

Inside in a less strict sense means the two features overlap. If we are looking for overlapping features, we are looking for their corresponding entries in `featureloc` tables, which have same value of `srcfeature_id` (this means they both lie on same feature, usually chromosome or contig). In case of „nonB DNA element in promoter of a gene“, we are looking for two entries in `featureloc` table (let's assign them aliases `lg` and `le`), one for gene, one for non-B DNA element, where intervals defined by <`le.fmin, le.fmax`> and <`lg.fmin - 100, lg.fmin`> overlap. In more strict sense, it would mean that interval <`le.fmin, le.fmax`> begins after and ends before interval <`lg.fmin - 100, lg.fmin`>.

Find nonB DNA element near transcription factor

In our example query (see apendix B.2.2) we chose triplex as element (specifying only „ss_type“) and used 100bp long border from both sides to define word near (with accepting also overlap with transcription factor itself).

Transcription factor is an entry in table `feature`, which has a corresponding entry in `cvterm` table with column `name` having value „TF_binding_site“. These two tables are joined on equal values of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`).

Near means that one feature is overlapping a defined border around other feature. We choose an arbitrary number (for example 100) saying how many base pairs this border is from the actual feature. If we are looking for one feature near other feature, we are looking for their corresponding entries in `featureloc` tables, which have same value of `srcfeature_id` (this means they both lie on same feature, usually chromosome or contig). In case of „nonB DNA element near transcription factor“, we are looking for two entries in `featureloc` table (let's assign them aliases `lt` and `le`), one for transcription factor, one for non-B DNA element, where sequences defined by `<le.fmin,le.fmax>` and `<lt.fmin - 100, lt.fmax + 100>` overlap. Alternatively we could also define the border only from one side (left or right) or be more strict and consider one feature near other only in case one feature overlaps defined border of another feature, but does not overlap the feature itself.

Find nonB DNA element inside nth intron of gene

In our example query (see apendix B.2.3) we chose triplex as element (specifying only „ss_type"), first intron of gene and used strict definition of word inside.

Intron could be an entry in `table` feature, which has a corresponding entry in `cvterm` table with name „intron". These two tables are joined on equal values of columns `featuretype_id` and `cvterm.cvterm_id`. However in most cases introns are not stored in the database like this, because their position can be calculated from exons

Exon is a an entry in table `feature`, which has a corresponding entry in `cvterm` table with name „exon". These two tables are joined on equal values of columns `featuretype_id` and `cvterm_id` (`cvterm`).

Nth Exon is a an entry in table `feature` for exon having feature property „exon_number" with value of `N` (N is an integer greater than zero). That is an entry in table `feature`, which has a corresponding entry in `cvterm` table with `name` „exon". These two tables are joined on equal values of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). It also has corresponding entry in `featueprop` table. These tables are joined on equal values of columns `feature_id` (`featureprop`) and `feature_id` (`feature`). The `featureprop` table has a corresponding entry in `cvterm` table with `name` „exon_number". These tables are joined on equal values of columns `type_id` (`featureprop`) and `cvterm_id` (`cvterm`). The value of column `value` (`featureprop`) is `N`.

Nth Intron is located between „Nth Exon" and „(N+1)th Exon". To get location of intron we need to get 2 exons that have the same source feature, have exon numbers of `N` and `N+1` respectively, and belong to the same gene. To see if exon belongs to a certain gene, we join `feature` table with table `feature_relationship` on same value of columns `feature_id` (`feature`) and `subject_id` (`feature_relationship`). We can then join `feature` for gene on same value of columns `feature_id` (`feature`) and `object_id` (`feature_relationship`).

Let there be 3 entries in `featureloc` table. Frist entry (with alias `lge1`) represents location of first exon of a gene. Second entry (with alias `lge2`) represents location of second exon of the same gene. Third entry (with alias `le`) represents location of non-B DNA element lying on same feature (chromosome, contig...) as first two entries. if sequence defined by interval `<le.fmin, le.fmax>` overlaps sequence defined by `<lge1.fmax, lge2.fmin>`, then `le` represents location of „NonB DNA element in 1st intron". This is in case of the less strict interpretation of **inside**, where any overlap is taken into account. In the more strict interpretation interval `<le.fmin, le.fmax>` would have to begin after and end before interval `<lge1.fmax, lge2.fmin>`.

Find nonB DNA element inside specific gene

In our example query (see apendix B.2.4) we chose triplex as element (specifying only „ss_type") and used strict definition of word inside.

This query is similar to query „Find nonB DNA element inside promoter of a gene". The difference is that we directly compare locations of gene and element instead of calculating position of gene's promoter. If we used aliases defined before, we would compare intervals $<$`le.fmin, le.fmax`$>$ and $<$`lg.fmin, lg.fmax`$>$

Summary

This section covered creation of four example SQL queries for Chado based on queries in natural language. The SQL queries used 9-20 joins (including the first table) and 10-20 where conditions. To construct such queries a person needs to have knowledge not only of SQL, but also of Chado database schema. Even for person with such knowledge, manually constructing these queries is inconvenient and takes a lot of time. It's clear that biologist without SQL knowledge won't be able to construct such queries and needs a more convenient solution.

5.3 Language Proposal

We will propose a language that would allow biologists to create their queries in a convenient way. This language should resemble natural language with use of biological terms. It will be heavily based on locations of features and comparison of these locations. It will allow to look for features based on feature types (gene, intron, exon, . . .) and positional relationships (inside, near, . . .). We will call it **Chado Query Language** or **ChQL**.

5.3.1 Functions and operators

Here we will explain the basic idea of functions, operators and their parameters. **Element**, **feature**, **gene** and **transcription factor** are *basic functions* returning respective type of feature. **Exon of**, **intron of** and **promoter of** are *secondary functions* that can only be applied to function **gene**. **Inside**, **intersect** and **near** are positional operators that can only be used on two basic functions. **Nth** is a *tertiary function* that can only be applied to functions **Exon of** and **intron of**. **Source_feature** and **query** are *special functions* that can be used only once at the end of sentence, but don't have to be used at all.

Feature is a basic function returning a feature.

- Parameter **type** allows to specify type of feature. If this parameter is not present, function returns any type of feature.

Gene is a basic function returning feature of type „gene“

- Parameter **name** allows to specify name of the gene
- Parameter **function** allows to specify function of the gene.

Element is a basic function returning feature representing non-B DNA element.

- Parameter **type** allows to specify type of non-B DNA element. Possible values are „triplex“, „quadruplex“ and „palindrome“

Transcription factor is a basic function returning feature of type 'TF_binding_site'.

- Parameter **name** allows to specify name of the transcription factor

Promoter of is a secondary function calculating promoter position from position of a gene.

- Parameter **length** allows to specify promoter length

Exon of is a secondary function returning feature of type „exon“ in relationship with gene. It has no parameters.

Intron of is a secondary function returning position of intron in relationship to gene. It calculates this position from two consecutive exons in same gene. It has no parameters.

Nth is a tertiary function specifying order of intron or exon

- Parameter **N** specifies order of intron or exon

Inside is a positional operator stating that one feature is inside other feature. It has no parameters

Intersect is a positional operator stating that one feature overlaps other feature

- Parameter **overlap** allows to specify required overlap

Near is a positional operator stating that one feature is near other feature

- Parameter **allow_inside** specifies if one feature is allowed to overlap other feature (not just the defined area around feature)
- Parameter **distance** specifies length of surrounding area in bases, which other feature has to overlap to be considered near the first feature
- Parameter **side** specifies from which side one feature should be near to the other. Possible values are „left“, „right“ and „both“

Source_feature is a special function that can be only used once at the end of sentence (but not after **query**). It allows to specify parameters of source feature

- Parameter **name** allows to specify name of the source feature.
- Parameter **min** allows to specify minimal position on source feature for search.
- Parameter **max** allows to specify maximal position on source feature for search.

Query is a special function that can be only used once at the end of sentence. It allows to specify parameters of query

- Parameter **limit** allows to specify maximum of rows returned by query

The parameters are described in table 5.1. For each parameter it shows which function it belongs to, its data type, if it is mandatory and default value.

in function	**parmeter**	**type**	**mandatory**	**default value**
feature	type	string	no	<none>
gene	name	string	no	<none>
gene	function	string	no	<none>
nth	n	integer	yes	1
element	type	string	yes	triplex
promoter	length	integer	yes	100
transcription factor	name	string	no	<none>
near	allow_inside	bool	yes	true
near	distance	integer	yes	100
near	side	string	yes	both
intersect	overlap	yes	no	1
query	limit	integer	yes	5
source_feature	name	string	no	<none>
source_feature	min	integer	no	<none>
source_feature	max	integer	no	<none>

Table 5.1: Parameters of ChQL language

5.3.2 Language definition

The basic idea of ChQL language (which could be considered ChQL Level 0) is there has to be at least one feature with the possibility of adding positional operator with another feature one or more times. After that there can be zero or one source feature and at the end zero or one query configuration word. If we disregard parameters of functions for the moment, the regular expression for such language would be following:

```
feature( operator feature)*( source_feature)?( query)?
```

Valid sentences of such language then would be for example:

```
feature
feature operator feature
feature operator feature operator feature
feature operator feature query
feature operator feature source_feature
feature source_feature query
```

ChQL Level 1

If we get more specific, „feature“ can be any basic function (feature, gene, element, transcription factor) possibly preceded by appropriate secondary function (promoter of, exon of, intron of) possibly preceded by tertiary function. By this definition valid feature could be „element“, „promoter of gene“, „nth exon of gene“ and so on (see functions in 5.3.1). We'll also use specific positional operators, instead of just word „operator“. This way we can create more advanced language, which we'll call ChQL Level 1. Language ChQL Level 1 is accepted by deterministic finite automaton depicted on figure 5.1. Below is regular expression defining ChQL Level 1 and a few example sentences:

```
((promoter of )?gene|feature|transcription factor|element|(nth )?(
↪ intron of |exon of )gene)( (near|inside|intersect) ((
↪ promoter of )?gene|feature|transcription factor|element|(nth
↪  )?(intron of |exon of )gene))*( source_feature)?( query)?
```

```
gene
promoter of gene
transcription factor near element inside promoter of gene
element intersect gene source_feature query
element inside nth intron of gene
```

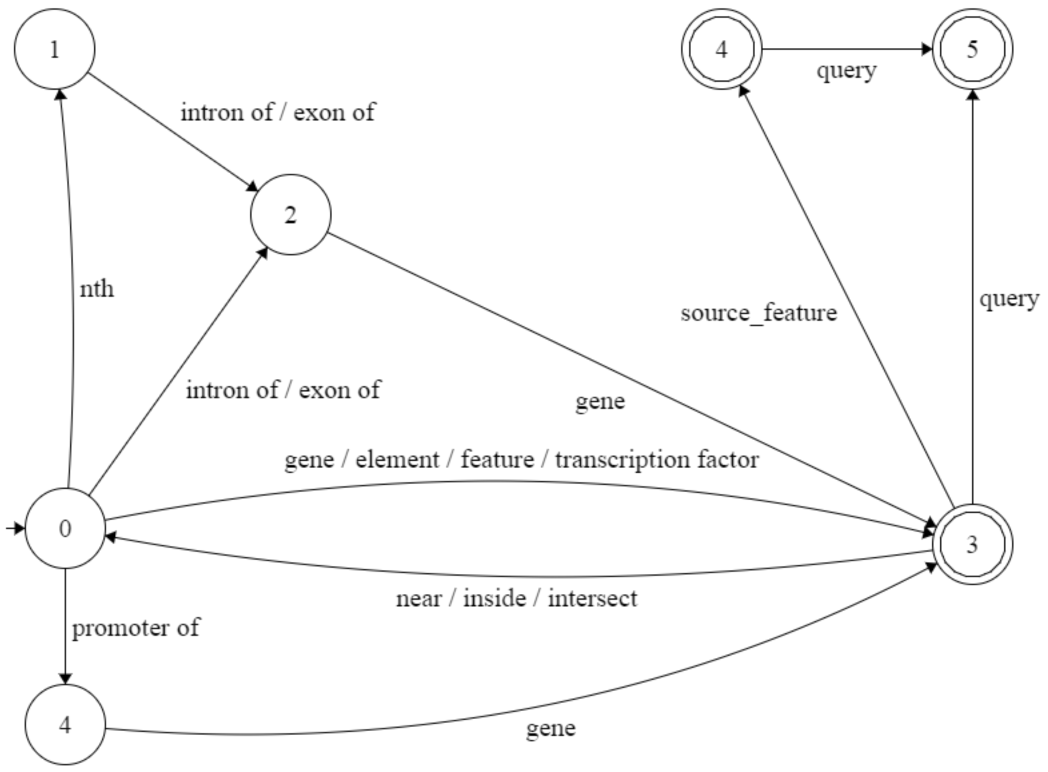


Figure 5.1: Finite State Machine accepting ChQL Level 1

ChQL Level 2

We also want to have parameters for some functions and operators in parentheses. If we take language ChQL Level 1 and add possibility of parameters in parentheses, we get ChQL Level 2. This language can be defined by grammar shown in table 5.2 (Grammar was verified by Grammophone online tool: http://mdaines.github.io/grammophone/). Each table row represents grammar rule with first and second column representing left and right side of the rule respectively. Nonterminals begin with capital letter and are often written in camel case. Terminals never begin with capital letter and are meant literally with these exceptions:

- Character ϵ means empty string.
- The vertical bar (or pipe) symbol (|) means or (alternative right side of rule).
- **string** means data type string.

- **number** means positive integer. In regular expression it would be `[0-9]+`.

5.3.3 Translation to SQL

We will describe how each word of ChSQL language should be translated to SQL. In most queries same table will be used more times for different features. For example feature table for exon, feature table for intron or even feature table for one gene and feature table for other gene. To be able to distinguish for which exact feature is table used, we need to number words in sentence and use these numbers when assembling a query. The number for word will be called wid (word id) and will be increased for basic features as shown in table 5.3. Sometimes same table will be used more times for same feature in different context. For example, cvterm table can be used to determine type of feature, but also type of particular feature property or term relevant to feature (usually describing function of the feature). For this reason we need to use different table aliases for same table in different context. To be able to assemble SQL query correctly, we need to establish standard aliases based on table and context, in which table is used, and adhere to this established naming convention (see table 5.4). In snippets from SQL queries any word starting with symbol `@` illustrates a variable. If some part of query will be directly replaced by variable, it's represented by `{variable}`. Each SQL query will be assembled from three parts: SELECT, FROM and WHERE clause. We designed SELECT clause for each feature type to select only certain columns and rename them for better readability (described in appendix B.1). Only FROM and WHERE clauses of queries will be described here, since they influence which rows are retrieved (and therefore are most important).

Query in general

Each query will have source feature at the beginning of its FROM clause. We will use the defined alias `ft_src` for feature representing the source feature. Beginning of FROM clause of any query will look like this: `FROM feature ft_src`.

Source_feature

If parameter name is used a condition will be added to query to search only on source feature with that name: `WHERE feature ft_src.uniquename = @name`. If at least one of parameters min and max was used, for each feature a condition will be added to check whether this feature lies in the boundaries specified by min and max. Depending on which of the parameters min, max (min/max/both) were entered, there are 3 possible options of comparison.

1. Both @min and @max were entered.

```
-- WHERE clause --
{exampleFeatureLoc}.fmin BETWEEN @min AND @max
{exampleFeatureLoc}.fmax BETWEEN @min AND @max
```

2. Only @min was entered.

```
-- WHERE clause --
{exampleFeatureLoc}.fmin >= @min
{exampleFeatureLoc}.fmax >= @min
```

Left side of rule	**Right side of rule**
Start	Continue MaybeSrcFeature MaybeQueryConfigPart
Continue	Promoter Gene MaybeLocation
Continue	MaybeNth IntronExon Gene MaybeLocation
Continue	OtherFeatures MaybeLocation
OtherFeatures	Gene | Element | Feature | TranscriptionFactor
MaybeLocation	Operator Continue | ϵ
Operator	inside | Near | Intersect
IntronExon	intron of | exon of
Promoter	promoter of (PromoterParams)
Element	element (ElementParams)
Feature	feature | feature (FeatureParams)
TranscriptionFactor	transcription factor | transcription factor (TFParams)
PromoterParams	length: number
MaybeNth	Nth | ϵ
Nth	nth (n: number)
Gene	gene | gene (GeneParams)
GeneParams	GeneParam | GeneParams , GeneParam
GeneParam	name: string | function: string
ElementParams	ElementParam | ElementParams , ElementParam
ElementParam	type: ElementType
ElementType	triplex | quadruplex | palindrome
FeatureParams	FeatureParam | FeatureParams , FeatureParam
FeatureParam	type: string | min: number | max: number
TFParams	TFParam | TFParams , TFParam
TFParam	name: string
Near	near (NearParams)
NearParams	NearParam | NearParams , NearParam
NearParam	allow_inside: TrueFalse
NearParam	distance: number
NearParam	side: SideOption
SideOption	left | right | both
Intersect	intersect (IntersectParams)
IntersectParams	IntersectParam
IntersectParams	IntersectParams , IntersectParam
IntersectParam	overlap: number
TrueFalse	true | false
MaybeQueryConfigPart	query (limit: number) | ϵ
MaybeSrcFeature	source_feature (SrcFeatureParams) | ϵ
SrcFeatureParams	SrcFeatureParam | SrcFeatureParams , SrcFeatureParam
SrcFeatureParam	name: string | min: number | max: number

Table 5.2: Grammar for ChQL Level 2 (Nonterminals begin with capital letters. Terminals representing data types are *string* and *number*. Character ϵ means empty string. Character | means OR, in other words alternative right side of rule. Other terminals are taken literally)

word	**increases wid**
gene	yes
element	yes
feature	yes
transcription factor	yes
near	no
intersect	no
inside	no
query	no
source_feature	no
exon	no
intron	no
promoter	no
nth	no

Table 5.3: Words of ChQL with relationship to increasing wid (word id)

Table	**Alias**
feature (original)	ft
feature (source)	ft_src
featureloc (original)	loc
cvterm (feature type)	cvt
feature_relationship	rel
feature_cvterm	ftcvt
cvterm (connected to original feature through feature_cvterm)	cvt_ftcvt
featureprop (connected to original feature on same feature_id)	prop
cvterm (conncted through featureprop `type_id`)	cvt_prop

Table 5.4: Aliases for tables based on context

3. Only @max was entered.

   ```
   -- WHERE clause --
   {exampleFeatureLoc}.fmin <= @max
   {exampleFeatureLoc}.fmax <= @max
   ```

Basic features

The following applies to basic functions (**element**, **feature**, **gene**, **transcription factor**). Different features will be assembled differently, but every feature type will be localized. This means we will join `featureloc` on same value of its column `srcfeature_id` and column `feature_id` of source feature.

```
-- FROM clause --
join featureloc loc{wid}
on loc{wid}.srcfeature_id = ft_src.feature_id
join feature ft{wid}
on loc{wid}.feature_id = ft{wid}.feature_id
```

Feature

When used without parameter, feature can be any type of feature and is created the same way as described above. In case parameter type is used (type: @feature_type)

```
-- FROM clause --
join cvterm cvt{wid}
on ft{wid}.type_id = cvt{wid}.cvterm_id

-- WHERE clause --
cvt{wid}.name = @feature_type
```

Gene

When translating word **gene** into SQL we need to connect `feature` table with table `cvterm` on same value of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). The value of `name` (`cvterm`) has to be „gene“.

```
-- FROM clause --
join cvterm cvt{wid}
on ft{wid}.type_id = cvt{wid}.cvterm_id

-- WHERE clause --
cvt{wid}.name = 'gene'
```

In case parameter name is used (name: @gene_name), we need to add condition to WHERE clause checking if feature.name is same as value of variable @gene_name.

```
-- WHERE clause --
ft{wid}.name = @gene_name
```

In case parameter function is used (function: @gene_function), we need to join table `feature_cvterm` on same value of columns `feature_id` (`feature`) and `feature_id` (`feature_cvterm`). We also need to join table cvterm on same values of columns `cvterm_id` (`feature_cvterm`) and `cvterm_id` (`cvterm`). Finally we need to add condition to WHERE clause that column `name` (`cvterm`) has same value as variable @gene_function and that `is_not` (`feature_cvterm`) equals 'f'. Otherwise the connection could mean that gene does not have the function.

```
-- FROM clause --
join feature_cvterm ftcvt{wid}
on ftcvt{wid}.feature_id = ft1.feature_id
join cvterm cvt_ftcvt{wid}
on ftcvt{wid}.cvterm_id = cvt_ftcvt{wid}.cvterm_id

-- WHERE clause --
and ftcvt{wid}.is_not = 'f'
and  cvt_ftcvt{wid}.name = @gene_function
```

Promoter of gene

The only difference from using gene without promoter is that in case of comparison, the start position will be represented by `fmin - @promoter_length` and end position will be represented by `fmin`.

Transcription factor

When translating word **transcription factor** into SQL we need to connect feature table with table cvterm on same value of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). The value of `name` (`cvterm`) has to be „TF_binding_site".

```
-- FROM clause --
join cvterm cvt{wid}
on ft{wid}.type_id = cvt{wid}.cvterm_id

-- WHERE clause --
cvt{wid}.name = 'TF_binding_site'
```

In case parameter name is used (name: @TF_name), we need to add condition to WHERE clause checking if feature.name is same as value of variable @gene_name.

```
-- WHERE clause --
ft{wid}.name = @TF_name
```

Element

When translating word **element** into SQL we need to connect feature table with table cvterm on same value of columns `type_id` (`feature`) and `cvterm_id` (`cvterm`). The value of `name` (`cvterm`) has to be „gene". Since the terms available in `cvterm` table for feature type are sometimes not detailed enough, we use our custom feature property called „ss_type". For this reason we also need conenct table `featureprop` and `cvterm` on equal values of `type_id` (`featureprop`) and `cvterm_id` (`cvterm`).

```
-- FROM clause --
join cvterm cvt{wid}
on ft{wid}.type_id = cvt{wid}.cvterm_id
join featureprop prop{wid}
on prop{wid}.feature_id = ft{wid}.feature_id
join cvterm cvt_prop{wid}
on prop{wid}.type_id = cvt_prop{wid}.cvterm_id
```

Based on exact value of parameter type, there are thre possible WHERE clauses of SQL query for element. In case type is a triplex cvterm representing feature type is „DNA_sequence_secondary_structure“ and value of feature property „ss_type“ is „triplex“

```
-- WHERE clause --
cvt{wid}.name = 'DNA_sequence_secondary_structure'
and cvt_prop{wid}.name = 'ss_type'
and prop{wid}.value = 'triplex'
```

In case type is a quadruplex cvterm representing feature type is „G_quartet“ and value of feature property „ss_type“ is „quadruplex“

```
-- WHERE clause --
cvt{wid}.name = 'G_quartet'
and cvt_prop{wid}.name = 'ss_type'
and prop{wid}.value = 'quadruplex'
```

In case type is a quadruplex cvterm representing feature type is „inverted_repeat“ and value of feature property „ss_type“ is „palindrome“

```
-- WHERE clause --
cvt{wid}.name = 'inverted_repeat'
and cvt_prop{wid}.name = 'ss_type'
and prop{wid}.value = 'palindrome'
```

Performance improvement Our tests (see 7.2.4) showed that determining type of element by checking both type and feature property negatively impacted performance. Since custom property is more specific than type using standard SO terms, implementation only checks the feature property. In this new version table with alias `cvt{wid}` is never joined and therefore never used in WHERE clause.

Exon of gene

Translation of **Exon of gene** is a special case and won't use the part of query that is common for basic features, although it uses similar principles. Basic part joins `featureloc` table for exon location and `feature` table for exon. It also joins `feature` table for gene through `feature_relationship` table. Finally it joins `cvterm` table for exon and gene. Were clause of query checks if `name` (`cvterm`) for feature type of exon and gene is „exon“ and „gene“ respectively

```
-- FROM clause --
join featureloc loc{wid}_exon1
on loc{wid}_exon1.srcfeature_id = ft_src.feature_id
join feature ft{wid}_exon1
on loc{wid}_exon1.feature_id = ft{wid}_exon1.feature_id
join feature_relationship rel{wid}_gene_exon1
on rel{wid}_gene_exon1.subject_id = ft{wid}_exon1.feature_id
join feature ft{wid}_gene
on rel{wid}_gene_exon1.object_id = ft{wid}_gene.feature_id
join cvterm cvt{wid}_gene
on ft{wid}_gene.type_id = cvt{wid}_gene.cvterm_id
join cvterm cvt{wid}_exon1
on ft{wid}_exon1.type_id = cvt{wid}_exon1.cvterm_id
```

```
-- WHERE clause --
cvt{wid}_gene.name = 'gene'
and cvt{wid}_exon1.name = 'exon'
```

Nth exon of gene

To translate **Nth exon of gene** (n: @n) same procedure will be used as for **exon of gene**. Additionaly `featureprop` table will be joined to `feature` table for exon. Finally `cvterm` table will be joined through featureprop table. In WHERE clause of query it will be checked if feature property „exon_number“ has value of @n.

```
-- FROM clause --
join featureprop prop{wid}_exon1
on prop{wid}_exon1.feature_id = ft{wid}_exon1.feature_id
join cvterm cvt_prop{wid}_exon1
on cvt_prop{wid}_exon1.cvterm_id = prop{wid}_exon1.type_id

-- WHERE clause --
and cvt_prop{wid}_exon1.name = 'exon_number'
and prop{wid}_exon1.value = @n
```

Intron of gene

Translating **intron of gene** is very simillar to the **exon of gene**, but instead of having one exon (and all related table entries), we have two exons for the same gene.

```
-- FROM clause --
join featureloc loc{wid}_exon1
on loc{wid}_exon1.srcfeature_id = ft_src.feature_id
join feature ft{wid}_exon1
on loc{wid}_exon1.feature_id = ft{wid}_exon1.feature_id
join feature_relationship rel{wid}_gene_exon1
on rel{wid}_gene_exon1.subject_id = ft{wid}_exon1.feature_id
join feature ft{wid}_gene
on rel{wid}_gene_exon1.object_id = ft{wid}_gene.feature_id
join cvterm cvt{wid}_gene
on ft{wid}_gene.type_id = cvt{wid}_gene.cvterm_id
join cvterm cvt{wid}_exon1
on ft{wid}_exon1.type_id = cvt{wid}_exon1.cvterm_id
join feature_relationship rel{wid}_gene_exon2
on rel{wid}_gene_exon2.object_id = ft{wid}_gene.feature_id
join feature ft{wid}_exon2
on rel{wid}_gene_exon2.subject_id = ft{wid}_exon2.feature_id
join cvterm cvt{wid}_exon2
on ft{wid}_exon2.type_id = cvt{wid}_exon1.cvterm_id
join featureloc loc{wid}_exon2
on ft{wid}_exon2.feature_id = loc{wid}_exon2.feature_id
join featureprop prop{wid}_exon1
on prop{wid}_exon1.feature_id = ft{wid}_exon1.feature_id
join cvterm cvt_prop{wid}_exon1
on cvt_prop{wid}_exon1.cvterm_id = prop{wid}_exon1.type_id
join featureprop prop{wid}_exon2
on prop{wid}_exon2.feature_id = ft{wid}_exon2.feature_id
join cvterm cvt_prop{wid}_exon2
```

```
on cvt_prop{wid}_exon2.cvterm_id = prop{wid}_exon2.type_id
```

In this case the word **nth** wasn't used in front of word **intron**. It needs to be chcecked that features for first exon, second exon and gene have correct types (through `cvterm` table). Area of intron is defined as area between two consecutive exons of same gene, which means WHERE clause needs to check that exon number of second exon is same as exon number of first exon + 1.

```
-- WHERE clause --
and cvt{wid}_gene.name = 'gene'
and cvt{wid}_exon1.name = 'exon'
and cvt{wid}_exon2.name = 'exon'
and cvt_prop{wid}_exon1.name = 'exon_number'
and cvt_prop{wid}_exon2.name = 'exon_number'
and cast(prop{wid}_exon1.value as integer) = cast(prop{wid}_exon2.value
    ↪ as integer) - 1
```

Nth intron of gene

To translate **Nth intron of gene** (n: @n) same procedure will be used as for **intron of gene**. In WHERE clause of query it will be checked if feature property „exon_number“ has value of @n for first exon and value of @n + 1 for second exon.

```
-- WHERE clause --
and cvt{wid}_gene.name = 'gene'
and cvt{wid}_exon1.name = 'exon'
and cvt{wid}_exon2.name = 'exon'
and cvt_prop{wid}_exon1.name = 'exon_number'
and prop{wid}_exon1.value = @n
and cvt_prop{wid}_exon2.name = 'exon_number'
and prop{wid}_exon2.value = @n + 1
```

Operators in general

Operators need to get information from the features they are trying to compare, because different type of feature could mean comparing different @start and @end values. Below is a description of how values for comparison will be assembled for different feature types. Operators will only appear as conditions in WHERE clause of final query.

- basic functions without additional functions
  - `@start = "loc"+{wid}+".fmin"`
  - `@end = "loc"+{wid}+".fmax"`
- promoter of gene
  - `@start = "loc"+{wid}+".fmin - "+@length`
  - `@end = "loc"+{wid}+".fmin"`
- exon of gene
  - `@start = "loc"+{wid}+"_exon1.fmin"`
  - `@end = "loc"+{wid}+"_exon1.fmax"`

- intron of gene
  - `@start = "loc"+{wid}+"_exon1.fmax"`
  - `@end = "loc"+{wid}+"_exon2.fmin"`

Intersect

Positional operator **intersect** selects features that overlap each other. Two features (1 and 2) don't overlap if @start1 > @end2 or if @end1 < @start2. From this we can conclude that two features overlap when @start1 <= @end2 and @end1 >= @start2. This means following condition needs to be added to WHERE clause of SQL query:

```
-- WHERE clause --
and start2 <= end1
and end2 >= start1
```

if minimal overlap is greater than 1, then following line will also be added to WHERE clause of SQL query.

```
-- WHERE clause --
and greatest(0,least(loc1.fmax,loc2.fmax)
-greatest(loc1.fmin,loc2.fmin)) > @overlap
```

Inside

For positional operator **inside** we will consider the most strict definition. This operator will select only such results where entire feature 1 is in feature 2. In other words feature 1 begins after beginning of feature 2 and ends before ending of feature 2. Following condition will be added to WHERE clause:

```
-- WHERE clause --
and loc{wid1}.fmin >= loc{wid2}.fmin
and loc{wid1}.fmax <= loc{wid2}.fmax
```

Near

Positional operator **near** defines an area surrounding feature 2 and checks if feature overlaps this area. Values of @startBorder2 and @endBorder2 are based on parameters *side* and *allow_inside* (see table 5.5 for details).

```
-- WHERE clause --
ft1.feature_id != ft2.feature_id
and @startBorder2 <= @end1
and @endBorder2 >= @start1
```

In case of patameter values side: both and allow_inside: false, this part will be in the WHERE clause as well

```
-- WHERE clause --
and not (@start2 <= @end1
and @end2 >= @start1)
```

side	**variables**	**allow_inside = 1**	**allow_inside = 0**
left	`@startBorder2`	`@start2 - @distance`	`@start2 - @distance`
left	`@endBorder2`	`@end2`	`@start2`
right	`@startBorder2`	`@start2`	`@end2`
right	`@endBorder2`	`@end2 + @distance`	`@end2 + @distance`
both	`@startBorder2`	`@start2 - @distance`	`@start2 - @distance`
both	`@endBorder2`	`@end2 + @distance`	`@end2 + @distance`

Table 5.5: Near parameters: values of start and end variables

Chapter 6

Implementation

The application is implemented in Vaadin, an open source Web application framework featuring a server-side architecture. Client-side uses Ajax technology and is build on top of Google Web Toolkit. Application in Vaadin can be developed in very similar way to Java Swing application without need to separately program server side, client side and communication between them. Because of this it was chosen for development of our application.

6.1 Web Interface

We considered letting the user type the sentence in ChQL. This approach could lead to typos or syntax errors, which could lead to confusion and frustration of user and go against our goal of making the language user-friendly for biologists.

Instead, we decided to implement our application as a web interface (see figure 6.1), that would navigate the user through assembling a sentence in ChQL language. User creates sentence in ChQL by choosing words from drop down menu.

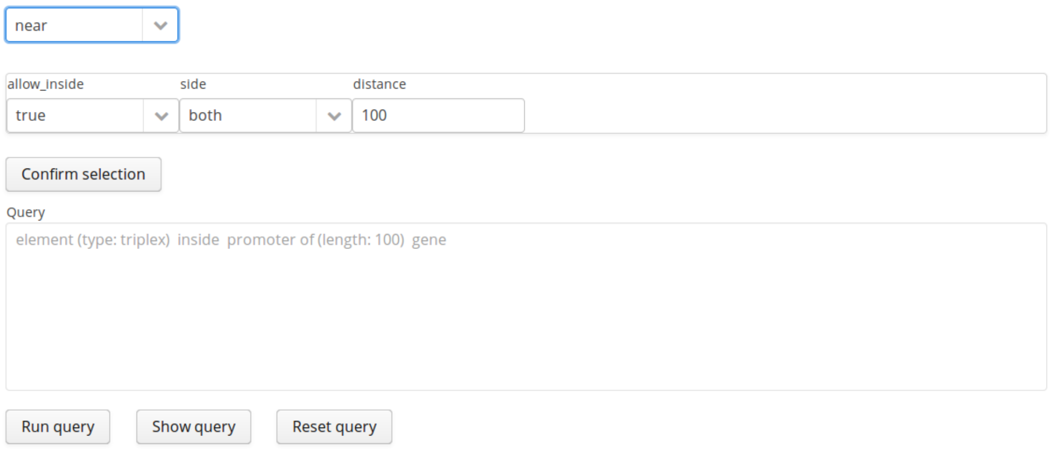
When user chooses word from drop down menu, additional drop down menus or text fields appear to allow user set values of parameters. For most parameters there are already pre-entered default values that user can keep, if they do not wish to use their own values.

After parameters are filled (or left at default values), user clicks „Confirm selection“ button, which adds word to sentence in ChQL Level 2 along with chosen parameters. The content of the main drop down menu then changes. The application uses deterministic finite state machine accepting language ChQL Level 1 (see figure 5.1) to determine which group of words to offer next. This way user is unable to assemble sentence that would have incorrect syntax. The state machine also checks if current state is an accepting state. If yes, the user can click „Show query“ button to see message box with generated SQL query, click „Run query“ button to send generated SQL query to server and get results, or continue assembling the sentence. If current state is not an accepting state, „Show query“ and „Run query“ buttons are disabled and user has to finish sentence before being able to use them.

6.2 Core Implementation

Class T0_ChadoQueryBuilder

Class *T0_ ChadoQueryBuilder* implements user interface as well as logic of the application. When user inserts a word into sentence, an object of class *Word* representing word with parameters is inserted into list `wordList`. After user clicks „Show query“ or „Run query“,

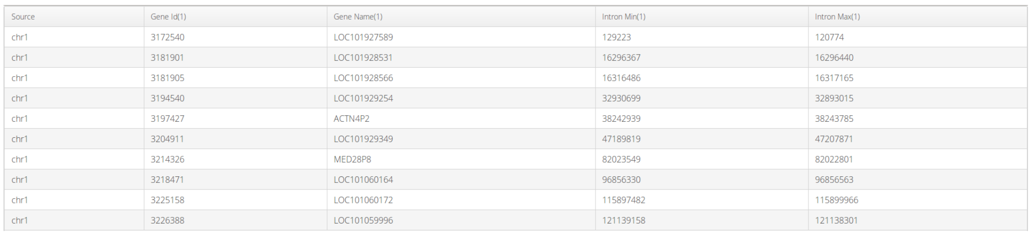


allow_inside	side	distance
true	both	100

Query

element (type: triplex) inside promoter of (length: 100) gene

Figure 6.1: Web interface for creating sentence in ChQL language



Source	Gene Id(1)	Gene Name(1)	Intron Min(1)	Intron Max(1)
chr1	3172540	LOC101927589	129223	120774
chr1	3181901	LOC101928531	16296367	16296440
chr1	3181905	LOC101928566	16316486	16317165
chr1	3194540	LOC101929254	32930699	32893015
chr1	3197427	ACTN4P2	38242939	38243785
chr1	3204911	LOC101929349	47189819	47207871
chr1	3214326	MED28P8	82023549	82022801
chr1	3218471	LOC101060164	96856330	96856563
chr1	3225158	LOC101060172	115897482	115899966
chr1	3226388	LOC101059996	121139158	121138301

Figure 6.2: Query results in web interface

query translation starts. The list of commands in ChQL language is looped through and new objects are created and added to list `wordObjectList`. Based on value of variable `command` in the list, different objects can be created. Object of class *Word* representing **gene** is grouped with objects representing secondary functions (**promoter of**, **intron of**, **exon of**) and tertiary functions (**nth**) to create object of class *Gene*. For values of `command` **element**, **feature** and **transcription factor** new objects will have class *Element*, *Word* and *TranscriptionFactor* respectively.

The initial parts of SELECT, FROM and WHERE clause are created and list of objects is looped through. If object is of class extending class *Operator*, its methods *setLeft* and *setRight* are called to prepare prerequisities for positional comparison based on class of objects that is in the list before (left) and after (right) the operator object. For each object its methods for filling its part of query are called. At the end, SELECT, FROM and WHERE clauses of query are joined to one string representing the finished SQL query. In case user had clicked the button „Run query“, class *DatabaseHelper* will send the generated SQL query to database and results will be displayed (see figure 6.2).

Class DatabaseHelper

Class *DatabaseHelper* manages connection to PostgreSQL server running Chado database and sends generated SQL queries to database.

Class Word

Class *Word* is used to store words of ChSQL language that user has added to the sentence along with parameters. String `command` stores name of function in ChQL language. Hash map `parameters` stores pairs of parameter and value (for example <length, 100>).

Class Operator

Class *Operator* is used for all operators (see figure 6.3). It has String `wherePart`, stroring generated WHERE clause for SQL query. Its has variables `left` and `right` of type *Operand* pointing to features (objects of class *Operand* left and right from the *Operator*). Method *setQueryPart* is used to determine what string to use for location comparison based on type of feature (see „Operators in general“ in 5.3.3).

Class Inside

Class *Inside* extends class *Operator*. Its method *setQuery* generates part of WHERE clause that checks if feature pointed to by `left` is located inside feature pointed to by `right`.

Class Intersect

Class *Intersect* extends class *Operator*. Its variable `overlap` represents minimal required overlap. Method *setQuery* generates part of WHERE clause that checks if feature pointed to by `left` overlaps feature pointed to by `right`. If `overlap` is greater than 1, it also adds part that checks if overlap of features is at least the value of variable `overlap`.

Class Near

Class *Near* extends class *Operator*. Its method *createQuery* generates part of WHERE clause that checks if feature pointed to by `left` overlaps defined reagion around feature pointed to by `right`. Variables `distance`, `isAllowedInside` and `side` determine region that will be used for comparison (check table 5.5 for details).

Class Operand

Class *Operand* is used by class *Operator* to create appropriate positional comparison. Its variables `localMin` and `localMax` store strings that would be used in comparison (see „Operators in general“ in 5.3.3). Variable `op` points to object of class *GeneralFeature*.

Class GeneralFeature

Class *GeneralFeature* extends class *Operand* and is used for all features. Variable `index` represents the feature's wid (word id, as described in 5.3.3). Variables `selectPart`, `fromPart` and `wherePart` store query's SELECT clause, FROM clause and WHERE clause respectively. Variable `interval` stores string for check on minimum and maximum values of all feature locations.

Class Feature

Class *Feature* extends class *GeneralFeature*. Its method `createQuery` generates SELECT, FROM and WHERE clause for feature. If variable `type` is not null, check for particular feature type will be also incorporaed into WHERE clause.

Class Element

Class *Element* extends class *GeneralFeature*. Its method *createQuery* generates SELECT, FROM and WHERE clause. The WHERE clause also contains check on specific type of element based on value of variable `type`.

Class TranscriptionFactor

Class *TranscriptionFactor* extends class *GeneralFeature*. Its method *createQuery* generates SELECT, FROM and WHERE clause. If variable *name* is not null, check for name of transcription factor will also be incorporated into WHERE clause.

Class Gene

Class *Gene* extends class *GeneralFeature*. Variable `predecessor` determines in which context the word gene was used. If its value is null, gene was used as a standalone word. If it's „promoter of", all logic will be the same from point of view of class *Gene* with only exception of adding extra column for promoter position to SELECT clause. Otherwise it only influences positional comparison, which is taken care of by class *Operator* using variable `promoterLength`. For values „exon of" and „intron of" it means the word gene was used as „exon of gene" and „intron of gene" respectively. If also variable `N` is not null, then gene was used as „nth intron of gene" or „nth exon of gene". Based on value of variable `predecessor`, method *createQuery* runs method *createQueryIntron* (for „intron of", *createQueryExon* (for „exon of") or *createQueryGene* (for „promoter of" or null value). These methods will construct SELECT, FROM and WHERE clause for intron, exon and gene. In any of these methods of variable `name` is not null, condition checking name of gene will be added to WHERE clause. Same applies to variable `function` and check for function of gene.

Class SourceFeature

Class *SourceFeature* adds condition to WHERE clause of query (variable `wherePart`) to look only for source feature with name specified by variable `name`. In case at least one of variables `min`, `max` is not null, it will also add condition for each feature that it has to be in boundaries defined by either `min`, `max` or both. This condition is stored in variable `interval` in class *GeneralFeature*.

Class Query

Class *Query* adds line at the end of query (variable `endPart`) that will limit number of returned rows based on value of variable `limit`.

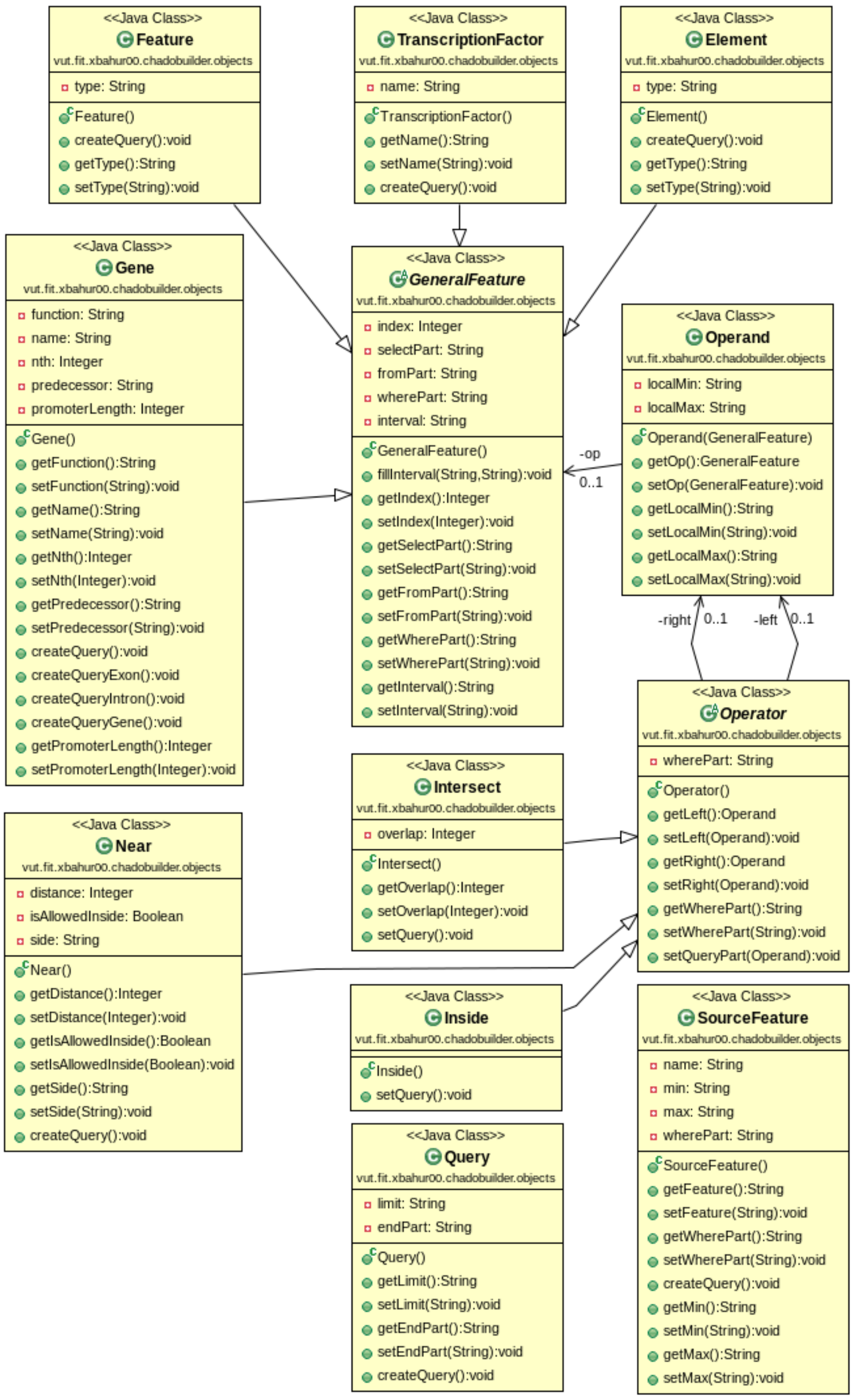


Figure 6.3: UML diagram for ChQL Query Builder

Chapter 7

Evaluation and Results

The application and database ran on Orcale VM VirtualBox virtual machine with 4500 MB RAM. This machine was running operating system Ubuntu 14.04 LTS (Trusty Tahr). For database we used PostgreSQL 9.1 with Chado database schema. The web application was running on Tomcat 8.0 server.

7.1 Data

Before testing any SQL queries on Chado database, we filled the database with data. Instead of using dummy test data, we used authentic data from various sources.

Data sources

All data was imported to database using GFF3 files. We imported annotation data for human genome, triplexes, quadruplexes, palindromes and transcription factors.

- GFF3 file for human genome (hg19 version) was downloaded from website of ensembl project [2]. This file did not contain features for chromosomes.
- For chromosomes in human genome we made and imported custom GFF3 file based on infromation from NCBI website.
- GFF3 files for transcription factors were downloaded from MotifMap website [3].
- GFF3 file for triplexes was generated using R/Bioconductor package Triplex [5].
- GFF3 file for quadruplexes was created based on [7].
- GFF3 file for palindromes was generated by R script provided by supervisor of this work.

Data Import and Fixes

The GFF3 files were imported into Chado database using script `gmod_bulk_load_gff3.pl` (Chado bulk loader script), which is part of standard Chado installation.

Most of the GFF3 files (see section 2.3 for details on columns and attributes of GFF3 file) contained some problems preventing the bulk loader script from successful import into database. Because of huge size of some files (over 1 GB) it would be impossible to fix these errors manually. We created shell scripts to fix these errors.

Below is description of various errors and problems that occured during data import using the bulk loader script and solutions to these problems.

Problems with directives using double hash This problem happened because GFF3 file used double hash for some entries instead of single hash. It can be solved by replacing the double hash with single hash (can be done manually as problem is only once at the beginning of a GFF3 file).

Error `don't know what do do with directive: '##species'`

Bad reference to source feature This problem occured, because GFF 3 file used directive `##sequence-region` to define source feature. Bulk loader script does not support such way of declaring a reference sequence for a GFF3 file. The `##sequence-region` directive is not expressive enough to define what type of feature the sequence is. If GFF3 file uses a `##sequence-region` directive in this way, it must be converted to a full GFF3 line (see C.2 fro details).

Error `Unable to find srcfeature <feature> in the database`

Inconsistent sequence ids Another problem is when sequence definition has different value of seqid (first column) and attribute ID. Sequence definition should have same values of seqif and attribute ID. In some GFF3 files attribute ID used some automatically generated value that was different than seqid. A script was made to fix this problem. (see appendix C.1 for details).

Splitting data into multiple chunks

Importing big files was inefficient, because it took considerable amount of time (importing entire human genome into database took several days). Bulk loader script often found error in the imported file after it had already imported considerable part of file and had to rollback entire import. Such situations lead to losing even the part of import that contained no errors. To prevent this, we created shell script to split GFF3 files into smaller chunks. These chunks could then be imported individually.

7.2 Experiments

7.2.1 Recreating 4 template queries in ChQL language

We tested our web application by recreating the 4 template queries in ChQL language and translating them to SQL.

Element inside promoter of gene

There are more possible variants of this query based on specific element (triplex, quadruplex, palindrome), strictness of word inside (Does entire element have to be inside promoter or is a certain overlap enough?) and parameters of word gene (Are we looking for gene with specific

Source	Element id	El. start	El. end	Gene id	Gene name	G. start	G. end
chr1	4660802	910482	910559	3172968	C1orf170	910478	910578
chr1	4660803	910508	910554	3172968	C1orf170	910478	910578

Table 7.1: Triplexes in promoter of gene „C1orf170“

name, specific function or just any gene?). We can also include specific source feature and query options. In this example we'll be looking for triplex that lies strictly inside promoter (with length 100) of gene named C1orf170. Below is the query written in ChQL language. Each function is in its own row for better readability:

```
element (type: triplex)
inside
promoter of (length: 100)
gene (name: C1orf170)
```

Below are instructions for assembling this sentence in ChQL language using the application web interface.

1. From main drop down menu choose „element“.
2. From drop down menu „type“ choose „triplex“.
3. Click „Confirm selection“. Sentence below now shows:
   `element (type:triplex)`
4. Choose „inside“ from main drop down menu.
5. Click „Confirm selection“. Following part is now added to the sentence:
   `inside`.
6. Choose „promoter of“ from main drop down menu.
7. Keep the default value of „100“ in text field labeled „length“
8. Click „Confirm selection“. Following part is now added to the sentence:
   `promoter of (length:100)`
9. Choose „gene“ from main drop down menu.
10. Type „C1orf170“ inside text field labeled „name“
11. Click „Confirm selection“. Following part is now added to the sentence:
    `gene (name:C1orf170)`
12. Click „Show query“ to see generated SQL or „Run query“ to send query to SQL server and get results.

Generated SQL query (see appendix B.3.1) will return results listed in table 7.1. Only some columns from returned data set were chosen for better readability.

Element near transcription factor

There are more possible variants of this query based on specific element (triplex, quadruplex, palindrome), parameters of word near (From which side? Is overlap with transcription factor allowed? What is maximum distance considered?) and parameters of word transcription factor (Are we looking for transcription factor with specific name or just any transcription factor?). We can also include specific source feature and query options. In this example we'll be looking for quadruplex that overlaps distance of 500 bases around the transcription factor from left or right side and could (but doesn't have to) overlap the transcription factor itself. Each function is in its own row for better readability:

```
element (type: triplex)
near (side: both, allow_inside: true, distance: 500)
transcription factor
query (limit: 10)
```

Below are instructions for assembling this sentence in ChQL language using the application web Interface.

1. From main drop down menu choose „element".
2. From drop down menu labeled „type" choose „quadruplex".
3. Click „Confirm selection". Sentence below now shows:
   `element (type:quadruplex)`
4. Choose „near" from main drop down menu.
5. In drop down menu labeled „side" keep the default value of „both"
6. In drop down menu labeled „allowed_inside" keep the default value of „true"
7. Type „500" into text field labeled „distance"
8. Click „Confirm selection". Following part is now added to the sentence:
   `near (side:both, distance:500, allow_inside:true)`
9. Choose „transcription factor" from main drop down menu.
10. Click „Confirm selection". Following part is now added to the sentence:
    `transcription factor`
11. Click „Show query" to see generated SQL or „Run query" to send query to SQL server and get results.

Generated SQL query (see appendix B.3.2) will return results listed in table 7.2. Only some columns from returned data set were chosen for better readability.

Source	El. start	El. end	TF feature_id	TF name	TF start	TF end
chr1	6418793	6418813	6202255	LM1_RFX1	6419106	6419125
chr1	6418793	6418813	6203052	LM1_RFX1	6419106	6419125
chr1	6419501	6419543	6202255	LM1_RFX1	6419106	6419125
chr1	6419501	6419543	6203052	LM1_RFX1	6419106	6419125
chr1	6550198	6550239	6203291	LM2_CTCF	6550096	6550116
chr1	6550578	6550618	6203291	LM2_CTCF	6550096	6550116
chr1	6673144	6673172	6204306	LM4_M2	6673619	6673642
chr1	6673314	6673339	6204306	LM4_M2	6673619	6673642
chr1	6673314	6673339	6203989	LM4_M2	6673678	6673701
chr1	6673778	6673802	6204306	LM4_M2	6673619	6673642

Table 7.2: Quadruplexes near transcription factors

Element inside nth intron of gene

There are more possible variants of this query based on specific element (triplex, quadruplex, palindrome), parameters of word nth and parameters of word gene (are we looking for gene with specific name, specific function or just any gene?). We can also include specific source feature and query options. In this example we'll be looking for quadruplex that overlaps (any overlap is enough) the first intron of any gene. We'll limit the results to 10 rows. Each function is in its own row for better readability:

```
element (type: quadruplex)
inside
nth (n:1)
intron of
gene
query (limit:10)
```

Below are instructions for assembling this sentence in ChQL language using the application web Interface.

1. From main drop down menu choose „element".
2. From drop down menu „type" choose „quadruplex".
3. Click „Confirm selection". Sentence below now shows:
   `element (type:quadruplex)`
4. Choose „intersect" from main drop down menu.
5. Keep the default value of „1" in text field labeled „overlap"
6. Click „Confirm selection". Following part is now added to the sentence:
   `intersect (overlap:1)`
7. Choose „nth" from main drop down menu.
8. Keep the default value of „1" in text field labeled „n"
9. Click „Confirm selection". Following part is now added to the sentence:
   `nth (n:1)`

Source	Gene name	Int. start	Int. end	El. start	El. end
chr1	LOC101927589	133748	129054	131383	131410
chr1	LOC101927589	133748	129054	132967	132992
chr1	LOC101927589	133748	129054	133326	133348
chr1	LOC101060126	11822378	11821435	11822206	11822230
chr1	LOC101928566	16302700	16316365	16304445	16304479
chr1	LOC101928566	16302700	16316365	16312786	16312834
chr1	LOC101928566	16302700	16316365	16315749	16315784
chr1	LOC101928566	16302700	16316365	16315919	16315937
chr1	LOC101928566	16302700	16316365	16316028	16316048
chr1	ACTN4P2	38232179	38242339	38237916	38237938

Table 7.3: Quadruplexes in first intron of gene

10. Choose „intron of" from main drop down menu.
11. Click „Confirm selection". Following part is now added to the sentence:
    `intron of`
12. Choose „gene" from main drop down menu. Leave the text fields „name" and „function" empty
13. Click „Confirm selection". Following part is now added to the sentence:
    `gene`
14. Choose „query" from main drop down menu.
15. Type „10" inside text field labeled „limit".
16. Click „Confirm selection". Following part is now added to the sentence:
    `query(limit:10)`
17. Click „Show query" to see generated SQL or „Run query" to send query to SQL server and get results.

Generated SQL query (see appendix B.3.3) will return results listed in table 7.3. Only some columns from returned data set were chosen for better readability.

Element inside gene

There are more possible variants of this query based on specific element (triplex, quadruplex, palindrome), strictness of word inside (Does entire element have to be inside gene or is a certain or any overlap enough?) and parameters of word gene (Are we looking for gene with specific name, specific function or just any gene?). We can also include specific source feature and query options. In this example we'll be looking for triplex that lies strictly inside gene named „RERE":

```
element (type:triplex) inside gene (name:RERE) query (limit:10)
```

Below are instructions for assembling this sentence in ChQL language using the application web interface.

Source	Element start	Element end	Gene name	G. start	G. end
chr1	8426921	8427000	RERE	8412463	8877699
chr1	8426963	8427016	RERE	8412463	8877699
chr1	8429531	8429596	RERE	8412463	8877699
chr1	8429538	8429605	RERE	8412463	8877699
chr1	8429568	8429610	RERE	8412463	8877699
chr1	8446028	8446091	RERE	8412463	8877699
chr1	8446045	8446108	RERE	8412463	8877699
chr1	8461077	8461127	RERE	8412463	8877699
chr1	8540147	8540217	RERE	8412463	8877699
chr1	8542742	8542791	RERE	8412463	8877699

Table 7.4: Triplexes in gene „RERE“

1. From main drop down menu choose „element“.
2. From drop down menu „type“ choose „triplex“.
3. Click „Confirm selection“. Sentence below now shows:
   `element (type:triplex)`
4. Choose „inside“ from main drop down menu.
5. Click „Confirm selection“. Following part is now added to the sentence:
   `inside`
6. Choose „gene“ from main drop down menu.
7. Type „RERE“ inside text field labeled „name“
8. Click „Confirm selection“. Following part is now added to the sentence:
   `gene (name:RERE)`
9. Choose „query“ from main drop down menu.
10. Type „10“ inside text field labeled „limit“.
11. Click „Confirm selection“. Following part is now added to the sentence:
    `query(limit:10)`
12. Click „Show query“ to see generated SQL or „Run query“ to send query to SQL server and get results.

Generated SQL query (see apendix B.3.4) returned results listed in table 7.4. Only some columns from returned data set were chosen for better readability.

7.2.2 Manual vs automatic queries

We verified that queries created manually (see subsection 5.2.1) have same performance and give same output as queries created by application. Only difference is in aliases for tables and order of joins and conditions. These things do not affect result or performance of the query. Queries created manually can have joins and conditions in a way that makes sense

Entry Type	Count	Query runtime
Feature (Any)	2 636 040	263 s
Gene	39 867	3 s
Promoter of gene (*)	39 867	4 s
Exon	879 841	72 s
Exon of gene	3 854	4 s
Intron of gene (*)	1 798	5 s
Transcription factor	3 422	1 s
Triplex	380 346	49 s
Quadruplex	429 491	42 s
Palindrome	714 960	76 s

Table 7.5: Types of features in database. Query runtime states how long does it take to list all items of that type. Asterisk symbol (*) marks items that are calculated ad-hoc from other items in database.

to person who created the query. Also aliases made by person can be more readable than aliases created by application. Those created by application have to comply with established naming convention, while those manually created can be named differently based on context.

7.2.3 Data statistics

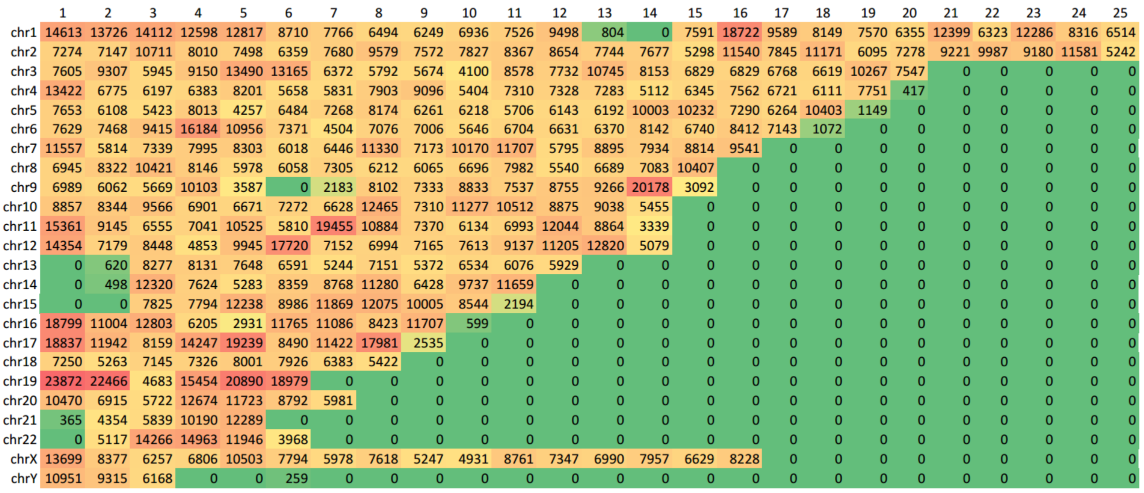
It's good to have an idea of how much data there is in the database. This is especially important when making complicated queries with lots of joined tables. The most performance heavy are comparisons of locations, which are used often since our language is based on positional relationships of features. In our database we have over 2 million features. The counts and query run times of specific feature types are in table 7.5.

In some situations it's good to limit query by choosing only a certain part of specific chromosome defined by number of base pairs. We checked each chromosome by 10Mb chunks and counted how many features each chunk contains (see figure 7.1). Each number represents amount of 10 Mb at which the chunk ends. For example number 3 represents chunk on interval (20 Mb, 30 Mb>.

From database point of view size of chunk in base pairs is not as important as how many features it contains. For example chromosome chrY contains only 259 features on chunk 6, while chromosome chr19 contains more than 23 thousand features on chunk 1. Even though these chunks have same length in base pairs, searching on the first one could be 85 times faster than on second one. It's good to keep this in mind when creating queries for specific chunks.

7.2.4 Performance

When creating queries in ChQL language, it's important to have an idea of how long could each query take. Some queries take less than second, some take several minutes and others are bound to run for hours. Counting any type of feature without an operator usually takes at most few seconds. Actually listing the items (instead of just returning count) creates certain performance overhead, but even then, listing one type of feature is fairly quick (see table 7.5). Listing all genes (or promoters of genes), exons of genes, introns of genes, or transcription factors takes at most 5 seconds. Listing elements (triplexes, quadruplexes,



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
chr1	14613	13726	14112	12598	12817	8710	7766	6494	6249	6936	7526	9498	804	0	7591	18722	9589	8149	7570	6355	12399	6323	12286	8316	6514
chr2	7274	7147	10711	8010	7498	6359	7680	9579	7572	7827	8367	8654	7744	7677	5298	11540	7845	11171	6095	7278	9221	9987	9180	11581	5242
chr3	7605	9307	5945	9150	13490	13165	6372	5792	5674	4100	8578	7732	10745	8153	6829	6829	6768	6619	10267	7547	0	0	0	0	0
chr4	13422	6775	6197	6383	8201	5658	5831	7903	9096	5404	7310	7328	7283	5112	6345	7562	6721	6111	7751	417	0	0	0	0	0
chr5	7653	6108	5423	8013	4257	6484	7268	8174	6261	6218	5706	6143	6192	10003	10232	7290	6264	10403	1149	0	0	0	0	0	0
chr6	7629	7468	9415	16184	10956	7371	4504	7076	7006	5646	6704	6631	6370	8142	6740	8412	7143	1072	0	0	0	0	0	0	0
chr7	11557	5814	7339	7995	8303	6018	6446	11330	7173	10170	11707	5795	8895	7934	8814	9541	0	0	0	0	0	0	0	0	0
chr8	6945	8322	10421	8146	5978	6058	7305	6212	6065	6696	7982	5540	6689	7083	10407	0	0	0	0	0	0	0	0	0	0
chr9	6989	6062	5669	10103	3587	0	2183	8102	7333	8833	7537	8755	9266	20178	3092	0	0	0	0	0	0	0	0	0	0
chr10	8857	8344	9566	6901	6671	7272	6628	12465	7310	11277	10512	8875	9038	5455	0	0	0	0	0	0	0	0	0	0	0
chr11	15361	9145	6555	7041	10525	5810	19455	10884	7370	6134	6993	12044	8864	3339	0	0	0	0	0	0	0	0	0	0	0
chr12	14354	7179	8448	4853	9945	17720	7152	6994	7165	7613	9137	11205	12820	5079	0	0	0	0	0	0	0	0	0	0	0
chr13	0	620	8277	8131	7648	6591	5244	7151	5372	6534	6076	5929	0	0	0	0	0	0	0	0	0	0	0	0	0
chr14	0	498	12320	7624	5283	8359	8768	11280	6428	9737	11659	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr15	0	0	7825	7794	12238	8986	11869	12075	10005	8544	2194	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr16	18799	11004	12803	6205	2931	11765	11086	8423	11707	599	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr17	18837	11942	8159	14247	19239	8490	11422	17981	2535	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr18	7250	5263	7145	7326	8001	7926	6383	5422	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr19	23872	22466	4683	15454	20890	18979	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr20	10470	6915	5722	12674	11723	8792	5981	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr21	365	4354	5839	10190	12289	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chr22	0	5117	14266	14963	11946	3968	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chrX	13699	8377	6257	6806	10503	7794	5978	7618	5247	4931	8761	7347	6990	7957	6629	8228	0	0	0	0	0	0	0	0	0
chrY	10951	9315	6168	0	0	259	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 7.1: Number of features on chromosomes (per 10 Mb chunks)

palindromes) or exons (not necessarily in relationship with gene) takes around 1 minute.

Situation is different for queries using operators. These queries take longer time than queries only listing certain type of feature. To test this, we looked for triplex inside different types of features (see table 7.6). If these queries are run for entire genome, the runtime is usually at least 1 hour (or significantly more depending on type of query). Because of that it is worth considering checking only a certain chunk on chosen chromosome. In our tests we used first 10Mb chunk on chromosome chr1. This chunk contains more than 14 thousand features. Run time for such queries ranges from under 1 second to approximately 1 minute. For example looking for triplex inside gene, promoter of gene or exon of gene takes around 1 second, while looking for triplex in intron of gene takes around 1 minute. Looking for triplex is fastest in gene. In promoter of gene it's slightly slower. For exon of gene it's approximately 4 times slower than for gene. For intron of gene it's 50 times slower than for exon of gene.

Searching entire genome by non-overlapping chunks would neglect results that lie on multiple chunks. Regardless of that, this method is useful for estimating runtime of different queries. If we use time measured for first 10Mb chunk of chromosome chr1, we can estimate how much time would checking entire genome by 10Mb chunks take. Around one minute for gene (or promoter of gene), 4 minutes for exon of gene and around 3 hours for intron of gene.

Searching entrire genome at once does not take the same time as searching it by chunks. We found out that checking the entire genome is 40-65 times slower depending on specific query. It took 63 minutes for gene, only 7 more minutes for promoter of gene and around 4 hours for exon of gene. For intron the query consumed all space left on VM's drive (around 80 GB) and crashed because of insufficient space. If there would be enough resources for this query, we estimate it would run for more than 8 days.

All tested queries confirmed that queries searching in promoter of gene are only slightly slower than in gene. This is understandable, as promoters are calculated ad-hoc from gene position during query execution, which creates only a slight performance overhead. Searching in exon takes approximately 4 times more than gene. This is probably, because exon has to be connected to gene through `feature_relationship` table and even though there are only 3k exons connected to gene, there are more than 800k exons. Search in

Triplex inside	**chunk (14.6k fts.)**	**genome (by chunks)**	**genome (whole)**
gene	394 ms	70 s (*)	63 min
promoter (100)	539 ms	96 s (*)	67 min
exon	1341 ms	4 min (*)	4 h 20 min
intron	64 s	3 h 11 min (*)	8 d 15 h (*)

Table 7.6: Runtimes of queries searching for triplex in different feature types. Asterisk symbol (*) marks values that were only estimated.

intron of gene runs approximately 50 times longer than in exon of gene. One could ask why do intron related queries with operator have such bad performance, while only listing introns doesn't take much longer than other types of features. When introns are listed by themselves, they are calculated based on pairs of exons of the same gene with consecutive exons numbers. Locations are only listed, but not used for any kind of comparison. However, when intron is used with operator, locations are used also for comparison. To determine if a feature is in some kind of positional relationship with intron, we need to check its positional relationship for first exon and then for second exon in exon pair. Performance-wise this is the equivalent of comparing not two, but three features with positional operators. When we take this into account, it's understandable that queries using intron of gene with operator have significantly worse performance than other queries.

On the other hand, some things have negligible effect on performance. For example, same queries with different operators (inside, intersect, near) have very similar runtimes. Specific type of element (triplex, quadruplex, palindrome) didn't have noticeable effect on query runtime. Changing promoter length had almost no effect on runtime of queries.

Performance Improvements

When manually constructing queries in the beginning, we noticed poor performance in those containing word **element**. These queries used both feature type and feature property to determine exact type of element. We tried to check only one of these options and found out that checking only feature property is much faster. Additionally, custom defined feature property is more specific than terms available in Chado database. We chose to use this faster method of determining the element type in the implementation of our application.

In Chado there is an alternative way of comparing feature intervals. Instead of doing standard comparisons like ((loc1.fmin >= loc2.fmin) and (loc1.fmax <= loc2.fmin)), one can use boxrange. **Boxrange** is a built-in function that converts `featureloc` entry to a rectangle in two dimensional space. This rectangle ranges on y-axis from value in `fmin` column to value in `fmax` column. On x-axis, rectangle begins and ends on same value, represented by `srcfeature_id` column (`sfid` for short). So coordinates of this rectange are ((`sfid`, `fmin`), (`sfid`, `fmax`)). For example if a feature is located on feature with id 1 and has values of `fmin` and `fmax` of 100 and 200 respectively, it is defined by a rectangle starting at point (1,100) and ending at (1,200). Since features with different source feature lie in different „columns" (different value of x), they cannot overlap, which makes sense. When locations of features are converted into rectangles in two dimensional space, geometric operators of PostgreSQL can be used to check their positional relationships (for example overlap or containment). This solution utilizes index binloc_boxrange_src on feaureloc table. In our tests it made queries up 2x faster than original approach, but in some cases could be also more than 10x slower. For this reason we decided to use the standard method of comparison.

Sometimes queries can be slow because SQL server does not have enough resources. PostgreSQL is preconfigured with very low values for some of these resources. We tried to configure PostgreSQL server to use more resources (for example 650MB shared memory instead of default 24 MB). This however did not bring any noticable improvement in performance of queries.

Chapter 8

Conclusion

Chado is a very useful database schema for storing genome annotation data. Constructing queries for Chado is no easy task even for person familiar with SQL and Chado database schema and is not an option for a biologist without this knowledge. We designed a language that uses biological terms and words referring to positional relationships between features.

Writing queries in this language would probably have been intuitive enough, but we decided to go further. We implemented a web interface, which guides user through the process of assembling a sentence in ChQL. This way instead of wondering what to type next, the user is always given choice of words, which can be used in current context. It also prevents the user from making typos or syntax errors.

There are improvements, that could be made. For example option of typing the commands for experienced users, more feature types and operators (not just for position, but different types of relationships), feature grouping with option of applying the operator to the entire group. Queries could be programmed to automatically run on overlapping chunks to improve performance. ChQL could also use more tables as it currently uses only 6 tables. However, it's debatable if there would be enough annotation data using these highly specialized tables to make such addition worthwhile.

Even in its current form, ChQL can be used to create some interesting queries, especially for positional comparison of features. ChQL doesn't offer same level of freedom as SQL, but it could be useful for biologists without SQL knowledge working with Chado.

Bibliography

[1] M. Ashburner. Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

[2] Fiona Cunningham, M. Ridwan Amode, Daniel Barrell, et al. Ensembl 2015. *Nucleic Acids Research*, 43(D1):D662–D669, 2015.

[3] Kenneth Daily, Vishal R. Patel, Paul Rigor, Xiaohui Xie, and Pierre Baldi. Motifmap: integrative genome-wide maps of regulatory motif sites for model species. *BMC Bioinformatics*, 12:495, 2011.

[4] Karen Eilbeck, Suzanna Lewis, Christopher Mungall, Mark Yandell, Lincoln Stein, Richard Durbin, and Michael Ashburner. The sequence ontology: a tool for the unification of genome annotations. *Genome Biology*, 6(5):R44, 2005.

[5] Jiří Hon, Tomáš Martínek, Kamil Rajdl, and Matej Lexa. Triplex: an r/bioconductor package for identification and visualization of potential intramolecular triplex patterns in dna sequences. *Bioinformatics*, 29, 2013.

[6] Robert M. Kuhn, David Haussler, and W. James Kent. The ucsc genome browser and associated tools. *Briefings in Bioinformatics*, 14(2):144–161, 2013.

[7] Matej Lexa, Pavlína Šteflová, Tomáš Martínek, Michaela Vorlíčková, Boris Vyskot, and Eduard Kejnovský. Guanine quadruplexes are formed by specific regions of human transposable elements. *BMC Genomics*, 15(1032):1–12, 2015.

[8] Christopher J. Mungall, David B. Emmert, and The FlyBase Consortium. A chado case study: an ontology-based modular schema for representing genome-associated biological information. *Bioinformatics*, 23(13):i337–i346, 2007.

[9] Susan E. St. Pierre, Laura Ponting, Raymund Stefancsik, Peter McQuilton, and the FlyBase Consortium. Flybase 102—advanced approaches to interrogating flybase. *Nucleic Acids Research*, 42(-):780–788, 2013.

[10] Website. Generic model organism database (gmod). `http://gmod.org`, [Accessed: 2014-07-07].

[11] Website. Biosql. `http://www.biosql.org`, [Accessed: 2015-09-01].

Appendix A

Contents of USB flash drive

```
/
├── doc/ ...          Directory containing pdf files and latex source files of this work.
├── ChadoVM.zip ...   Zip file containing virtual machine used for this work. It contains
│                     everything one needs to test/use the application (database filled with
│                     data is included).
└── README.pdf ...    Manual for running the virtual machine and using/testing the
                      application.
```

Appendix B

SQL queries

B.1 SELECT clause for different feature types

Each query will have source feature in select clause.

```
-- SELECT clause --
ft_src.uniquename AS "Source"
```

Feature

```
-- SELECT clause --
ft{wid}.feature_id AS "Feature␣Id({wid})",
ft{wid}.name AS "Feature␣Name({wid})",
loc{wid}.fmin AS "Feature␣Min({wid})",
loc{wid}.fmax AS "Feature␣Max({wid})"
```

Element

```
-- SELECT clause --
ft{wid}.feature_id AS "Element␣Id({wid})",
ft{wid}.name AS "Element␣Name({wid})",
loc{wid}.fmin AS "Element␣Min({wid})",
loc{wid}.fmax AS "Element␣Max({wid})"
```

Transcription factor

```
-- SELECT clause --
ft{wid}.feature_id AS "TF␣Id({wid})",
ft{wid}.name AS "TF␣Name({wid})",
loc{wid}.fmin AS "TF␣Min({wid})",
loc{wid}.fmax AS "TF␣Max({wid})"
```

Gene

```
-- SELECT clause --
ft{wid}.feature_id AS "Gene␣Id({wid})",
ft{wid}.name AS "Gene␣Name({wid})",
loc{wid}.fmin AS "Gene␣Min({wid})",
loc{wid}.fmax AS "Gene␣Max({wid})"
```

Promoter of gene

Variable `@length` represents promoter length.

```
-- SELECT clause --
ft{wid}.feature_id AS "Gene Id({wid})",
ft{wid}.name AS "Gene Name({wid})",
loc{wid}.fmin - @length AS "Promoter Min({wid})",
loc{wid}.fmin AS "Gene Min({wid})",
loc{wid}.fmax AS "Gene Max({wid})"
```

Exon of gene

```
-- SELECT clause --
ft{wid}_gene.feature_id AS "Gene Id({wid})",
ft{wid}_gene.name AS "Gene Name({wid})",
ft{wid}_exon.name AS "Exon Name({wid})",
loc{wid}_exon.fmin AS "Exon Min({wid})",
loc{wid}_exon.fmax AS "Exon Max({wid})"
```

Intron of gene

```
-- SELECT clause --
ft{wid}_gene.feature_id AS "Gene Id({wid})",
ft{wid}_gene.name AS "Gene Name({wid})",
loc{wid}_exon{wid}.fmax AS "Intron Min({wid})",
loc{wid}_exon2.fmin AS "Intron Max({wid})"
```

B.2 Manually created SQL queries

B.2.1 Element inside promoter of gene

```
SELECT *
FROM feature chromosome_feature
  join featureloc element_location
  on element_location.srcfeature_id = chromosome_feature.feature_id
  join feature element_feature
  on element_location.feature_id = element_feature.feature_id
  join featureprop element_property
  on element_property.feature_id = element_feature.feature_id
  join cvterm element_property_type
  on element_property.type_id = element_property_type.cvterm_id
  join featureloc gene_location
  on gene_location.srcfeature_id = chromosome_feature.feature_id
  join feature gene_feature
  on gene_location.feature_id = gene_feature.feature_id
  join cvterm gene_type
  on gene_feature.type_id = gene_type.cvterm_id
 WHERE true
  and element_property_type.name = 'ss_type'
  and element_property.value = 'triplex'
  and element_location.fmin >= gene_location.fmin - 100
  and element_location.fmax <= gene_location.fmin
  and gene_type.name = 'gene'
```

B.2.2 Element near transcription factor

```
SELECT *
FROM feature chromosome_feature
join featureloc element_location
on element_location.srcfeature_id = chromosome_feature.feature_id
join feature element_feature
on element_location.feature_id = element_feature.feature_id
join featureprop element_property
on element_property.feature_id = element_feature.feature_id
join cvterm element_property_type
on element_property.type_id = element_property_type.cvterm_id
join featureloc TF_location
on TF_location.srcfeature_id = chromosome_feature.feature_id
join feature TF_feature
on TF_location.feature_id = TF_feature.feature_id
join cvterm TF_type
on TF_feature.type_id = TF_type.cvterm_id
WHERE true
and TF_type.name = 'TF_binding_site'
and element_property_type.name = 'ss_type'
and element_property.value = 'triplex'
and TF_location.fmin - 100 <= element_location.fmax
and TF_location.fmax + 100 >= element_location.fmin
```

B.2.3 Element intersect nth intron of gene

```
SELECT *
FROM feature chromosome_feature
join featureloc element_location
on element_location.srcfeature_id = chromosome_feature.feature_id
join  feature element_feature
on element_location.feature_id = element_feature.feature_id
join  featureprop element_property
on element_property.feature_id = element_feature.feature_id
join  cvterm element_property_type
on element_property.type_id = element_property_type.cvterm_id
join  featureloc exon_1st_location
on exon_1st_location.srcfeature_id = chromosome_feature.feature_id
join  feature exon_1st_feature
on exon_1st_location.feature_id = exon_1st_feature.feature_id
join  feature_relationship rel2_gene_exon1
on rel2_gene_exon1.subject_id = exon_1st_feature.feature_id
join  feature gene_feature
on rel2_gene_exon1.object_id = gene_feature.feature_id
join  cvterm gene_type
on gene_feature.type_id = gene_type.cvterm_id
join  cvterm exon_1st_type
on exon_1st_feature.type_id = exon_1st_type.cvterm_id
join  featureprop exon_1st_property
on exon_1st_property.feature_id = exon_1st_feature.feature_id
join  cvterm exon_1st_property_type
on exon_1st_property_type.cvterm_id = exon_1st_property.type_id
join  feature_relationship rel2_gene_exon2
on rel2_gene_exon2.object_id = gene_feature.feature_id
```

```
join  feature exon_2nd_feature
on rel2_gene_exon2.subject_id = exon_2nd_feature.feature_id
join  cvterm exon_2nd_type
on exon_2nd_feature.type_id = exon_1st_type.cvterm_id
join  featureloc exon_2nd_location
on exon_2nd_feature.feature_id = exon_2nd_location.feature_id
join  featureprop exon_2nd_property
on exon_2nd_property.feature_id = exon_2nd_feature.feature_id
join  cvterm exon_2nd_property_type
on exon_2nd_property_type.cvterm_id = exon_2nd_property.type_id
WHERE true
and element_property_type.name = 'ss_type'
and element_property.value = 'triplex'
and gene_type.name = 'gene'
and exon_1st_type.name = 'exon'
and exon_2nd_type.name = 'exon'
and exon_1st_property_type.name = 'exon_number'
and exon_1st_property.value = '1'
and exon_2nd_property_type.name = 'exon_number'
and exon_2nd_property.value = '2'
and element_location.fmin >= exon_1st_location.fmax
and element_location.fmax <= exon_2nd_location.fmin
```

B.2.4 Element inside gene

```
SELECT *
FROM feature chromosome_feature
 join featureloc element_location
 on element_location.srcfeature_id = chromosome_feature.feature_id
 join feature element_feature
 on element_location.feature_id = element_feature.feature_id
 join featureprop element_property
 on element_property.feature_id = element_feature.feature_id
 join cvterm element_property_type
 on element_property.type_id = element_property_type.cvterm_id
 join featureloc gene_location
 on gene_location.srcfeature_id = chromosome_feature.feature_id
 join feature gene_feature
 on gene_location.feature_id = gene_feature.feature_id
 join cvterm gene_type
 on gene_feature.type_id = gene_type.cvterm_id
 WHERE true
 and element_property_type.name = 'ss_type'
 and element_property.value = 'triplex'
 and element_location.fmin >= gene_location.fmin
 and element_location.fmax <= gene_location.fmax
 and gene_type.name = 'gene'
```

B.3 ChQL translated to SQL

B.3.1 Element inside promoter of gene

```
SELECT *
FROM feature ft_src
join featureloc loc1
```

```
on loc1.srcfeature_id = ft_src.feature_id
join feature ft1
on loc1.feature_id = ft1.feature_id
join featureprop prop1
on prop1.feature_id = ft1.feature_id
join cvterm cvt_prop1
on prop1.type_id = cvt_prop1.cvterm_id
join featureloc loc2
on loc2.srcfeature_id = ft_src.feature_id
join feature ft2
on loc2.feature_id = ft2.feature_id
join cvterm cvt2
on ft2.type_id = cvt2.cvterm_id
WHERE true
and cvt_prop1.name = 'ss_type'
and prop1.value = 'triplex'
and loc1.fmin >= loc2.fmin -100
and loc1.fmax <= loc2.fmin
and cvt2.name = 'gene'
and ft2.name = 'C1orf170'
```

B.3.2 Element near transcription factor

```
SELECT *
FROM feature ft_src
join featureloc loc1
on loc1.srcfeature_id = ft_src.feature_id
join feature ft1
on loc1.feature_id = ft1.feature_id
join cvterm cvt1
on ft1.type_id = cvt1.cvterm_id
join featureprop prop1
on prop1.feature_id = ft1.feature_id
join cvterm cvt_prop1
on prop1.type_id = cvt_prop1.cvterm_id
join featureloc loc2
on loc2.srcfeature_id = ft_src.feature_id
join feature ft2
on loc2.feature_id = ft2.feature_id
join cvterm cvt2
on ft2.type_id = cvt2.cvterm_id
WHERE true
and cvt1.name = 'G_quartet'
and cvt_prop1.name = 'ss_type'
and prop1.value = 'quadruplex'
and loc2.fmin - 500 <= loc1.fmax
and loc2.fmax + 500 >= loc1.fmin
and cvt2.name = 'TF_binding_site'
limit 10
```

B.3.3 Element intersect nth intron of gene

```
SELECT *
FROM feature ft_src
join featureloc loc1
```

```
on loc1.srcfeature_id = ft_src.feature_id
join feature ft1
on loc1.feature_id = ft1.feature_id
join featureprop prop1
on prop1.feature_id = ft1.feature_id
join cvterm cvt_prop1
on prop1.type_id = cvt_prop1.cvterm_id
join featureloc loc2_exon1
on loc2_exon1.srcfeature_id = ft_src.feature_id
join feature ft2_exon1
on loc2_exon1.feature_id = ft2_exon1.feature_id
join feature_relationship rel2_gene_exon1
on rel2_gene_exon1.subject_id = ft2_exon1.feature_id
join feature ft2_gene
on rel2_gene_exon1.object_id = ft2_gene.feature_id
join cvterm cvt2_gene
on ft2_gene.type_id = cvt2_gene.cvterm_id
join cvterm cvt2_exon1
on ft2_exon1.type_id = cvt2_exon1.cvterm_id
join featureprop prop2_exon1
on prop2_exon1.feature_id = ft2_exon1.feature_id
join cvterm cvt_prop2_exon1
on cvt_prop2_exon1.cvterm_id = prop2_exon1.type_id
join feature_relationship rel2_gene_exon2
on rel2_gene_exon2.object_id = ft2_gene.feature_id
join feature ft2_exon2
on rel2_gene_exon2.subject_id = ft2_exon2.feature_id
join cvterm cvt2_exon2
on ft2_exon2.type_id = cvt2_exon1.cvterm_id
join featureloc loc2_exon2
on ft2_exon2.feature_id = loc2_exon2.feature_id
join featureprop prop2_exon2
on prop2_exon2.feature_id = ft2_exon2.feature_id
join cvterm cvt_prop2_exon2
on cvt_prop2_exon2.cvterm_id = prop2_exon2.type_id
WHERE true
and cvt_prop1.name = 'ss_type'
and prop1.value = 'quadruplex'
and loc2_exon1.fmax <= loc1.fmax
and loc2_exon2.fmin >= loc1.fmin
and cvt2_gene.name = 'gene'
and cvt2_exon1.name = 'exon'
and cvt2_exon2.name = 'exon'
and cvt_prop2_exon1.name = 'exon_number'
and prop2_exon1.value = '1'
and cvt_prop2_exon2.name = 'exon_number'
and prop2_exon2.value = '2'
```

B.3.4 Element inside gene

```
SELECT *
FROM feature ft_src
join featureloc loc1
on loc1.srcfeature_id = ft_src.feature_id
join feature ft1
```

```
on loc1.feature_id = ft1.feature_id
join cvterm cvt1
on ft1.type_id = cvt1.cvterm_id
join featureprop prop1
on prop1.feature_id = ft1.feature_id
join cvterm cvt_prop1
on prop1.type_id = cvt_prop1.cvterm_id
join featureloc loc2
on loc2.srcfeature_id = ft_src.feature_id
join feature ft2
on loc2.feature_id = ft2.feature_id
join cvterm cvt2
on ft2.type_id = cvt2.cvterm_id
WHERE true
and cvt1.name = 'G_quartet'
and cvt_prop1.name = 'ss_type'
and prop1.value = 'quadruplex'
and loc2.fmin - 500 <= loc1.fmax
and loc2.fmax + 500 >= loc1.fmin
and cvt2.name = 'TF_binding_site'
limit 10
```

Appendix C

Data fixes

C.1 Inconsistent sequence ids

Principle of script is to create list with good ids and bad ids. Each occurence of each item from `BadId` list will be replaced by item in `GoodId` list on same position

```
GoodID=($(cat ${fileName}.gff3 | grep "##sequence-region" -A 2 | grep "
    ↪ ID=" | awk '{print $1}'))

BadID=($(cat ${fileName}.gff3 | grep "##sequence-region" -A 2 | grep "ID
    ↪ =" | awk '{print $9}' | awk -F ';' '{print $1}' | awk -F '=' '{
    ↪ print $2}'))

cp ${fileName}.gff3 ${fileName}_fixed_id_0.gff3

for ((i=0;i<${#GoodID[@]};i++)); do
cat ${fileName}_fixed_id_$i.gff3 | sed "s/${BadID[i]};/${GoodID[i]};/g"
    ↪ > ${fileName}_fixed_id_$((i + 1)).gff3
rm -rf ${fileName}_fixed_id_$i.gff3
done
```

C.2 Bad reference to source featue

Line replacement example

Problematic line:

```
##sequence-region chr1 1 249250621
```

Fixed line:

```
chr1    .    chromosome  1    249250621    .    .    .    ID=chrI
```