Developing a Database for GenBank Information

By
Nathan Mann
B.S., University of Louisville, 2003

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed Scientific School
As Partial Fulfillment of the Requirements
For the Professional Degree

Master of Engineering

Department of Computer Engineering and Computer Science

December 2004

Developing a Database for GenBank Information

Submitted by: ____

Nathan Mann

A Thesis Approved on

____

(Date)

By the Following Reading and Examination Committee:

____

Dr. Dar-Jen Chang, Thesis Director

____

Dr. Eric Rouchka

____

Dr. Nigel G. Cooper

Acknowledgment

I would like to acknowledge that this thesis would not be possible without the help of my thesis director, Dr. Chang. The help of Dr. Eric Rouchka has been immensely helpful in understanding the role and usage of GenBank and how it affects the University of Louisville and the bioinformatics research group. I would also like to thank Dr. Nigel Cooper for taking time out of his schedule to serve on my thesis committee.

This thesis is also not possible without the support of my friends and family, for helping through all of my school endeavors.

Finally, but not least, I would like to express my earnest gratitude to my fiancé, for her support and inspiration throughout the entire Masters Program and completion of this thesis.

Abstract

The thesis project, Gene Database, was done to create a way for the bioinformatics research group at the University of Louisville to have access to GenBank EST information in the form of a database. This database allows for a programmable front end to be used to conduct further research with the use of EST information. The database backend used is Oracle and was populated through a custom Java program. The loader was created in lieu of using Oracle's SQL*Loader because of the limitations in SQL*Loader.

Previous ways of accessing the GenBank information included downloading the compressed files and using them locally as raw file formats or using the NCBI Website remotely. This Gene Database allows for a central location for bio-information of the GenBank to be kept at the University of Louisville.

The database was initially populated with the human EST information. The database is versatile enough to allow for other organisms to be stored in the database as well. It also allows for custom queries for specific research goals that are spawned by having this information readily available for researchers.

Table of contents

ACKNOWLEDGMENT III

ABSTRACT IV

TABLE OF CONTENTS V

I. INTRODUCTION 1

II. GENBANK 4

A. FILE FORMAT 5
B. FEATURES TABLE FORMAT 9
C. ISSUES OF GENBANK RAW FILE FORMAT 10

III. DATABASE SCHEMA 12

A. OBJECT RELATIONAL WITH ORACLE 12
B. DECIDING ON THE SCHEMA 14
C. DATABASE SCHEMA 15
D. FEATURES TABLE 19

IV. PARSER/IMPORTER 21

A. PARSER 22
B. IMPORT TOOL 23
C. LOADING DATA 24

V. CONCLUSIONS 25

A. SQL*LOADER 25
B. PERFORMANCE OF PROGRAM 26
C. RESULTS TABLES 28
D. FUTURE PROJECTS AND ENHANCEMENTS 28
E. CONCLUSIONS 29

REFERENCES 30

APPENDIX I: SAMPLE EST 31

APPENDIX II: RESULTS DATABASE DIAGRAM 32

APPENDIX III: SQL STATEMENTS FOR GENBANK DATABASE SCHEMA 33

APPENDIX IV: SQL STATEMENTS FOR THE RESULTS DATABASE SCHEMA 41

APPENDIX V: PARSER SOURCE CODE 43

APPENDIX VI: LOADER SOURCE CODE 51

APPENDIX VII: CONNECTION SOURCE CODE 60

APPENDIX VIII: BASH SCRIPT TO POPULATE DATA 61

VITA 62

I. Introduction

GenBank is a data store containing over 100 gigabytes of compressed information of DNA and protein sequences. Expressed Sequence Tags (EST) information is one type of data housed within GenBank. Access to EST information is in one of two main forms. The first is through the National Center for Biotechnology (NCBI) Entrez web interface. The web interface is a starting point at looking at information that is in human readable. The other method of accessing EST information is in FASTA format, highly condensed information that has the Gene Sequence, and a little bit more. Other than those two methods, there is no structured application programming interface to aid in querying GenBank information directly. Researching EST through a computer program for the Bio-informatics research group (BRG) at the University of Louisville has been difficult, because of the amount of time involved when querying and the inability for specific queries. The objective of this thesis project is to design an object-relational database called Gene Database to store GenBank data. The versatility of Gene Database will allow the database to be populated not only with human EST information but also with the genomic and EST information of other species. A computer interface for Gene Database will allow more specific research goals to be met. The ultimate goal of Gene Database is aiding in the research of bioinformatics.

The database that is created will initially be used in pulling information to be matched against the genomic sequences by the BRG that will perform a sequence of operations to look for a specified result that is defined in the external experiment ran by other researchers, and those results are stored back in the database created in the project. These results are organized in such a fashion to be searchable for latter use in finding some particular sequence that is interesting for the BRG. What the results mean and how

they are useful is part of research done by another PhD student. The role that this project has on the BRG is initially to increase access performance versus parsing each file to gather the information required and to find a way to store the results in a database. The project makes it possible to pull other information from it that can be used in the future by the BRG, or by any other U of L group that has a need to access this kind of information, using their own front end to better suit their own needs. If there is something that is useful to the rest of the community, then the entire EST entry can be recreated, and the interesting information can be inserted into the file, and sent to GenBank for submission of new research work. The other role is to provide a database that can handle the data from NCBI and house it in an integrated database instead of multiple compressed flat files. This project will provide a way of populating the data not only for human EST data but also for other data in the same file format, after verifying the space and time requirements for populating the database with the requested information from GenBank.

This project has several constraints due to the specific needs of the current BRG research interests. The problem has been constrained to human EST files. The files from NCBI originally contained more than just human, so the files were parsed, and the human entries were extracted. Those extracted files are recompressed and imported to the database. The other constraint is that the database is initially populated with just information that is of use to the PhD student, as a way to show the proof-of-concept for the student's work.

Chapter II gives background information about GenBank itself. A sample flat file is given in Appendix I to clarify any questions about the file format or any rules that are specified.

Chapter III explains the process behind developing the database. It also explains the issues behind it as well as why certain decisions were made. The SQL statements to create the database with Oracle are given in Appendix III.

Chapter IV reviews the software used to populate the database. It also explains how the user can run the program and a shell script in UNIX to automate the process. The source code is given in Appendices IV through VI.

Chapter V gives the results, and does analysis on the time requirements and size of the database. It also explains why Oracle's bulk loader was not used to import the data. It gives the conclusions of this thesis project, and recommendations for future projects.

II. GenBank

The U.S. government established the National Center for Biotechnology Information (NCBI) in 1988 as a central repository of molecular biology data used in research projects, analyzing genome data, and communicating new discoveries within a subject area. NCBI was setup as a division of the National Library of Medicine (NLM), which is housed by the National Institute of Health (NIH). NCBI hosts several different databases that are used worldwide by countries such as the United States, Japan, and England. NCBI is used as a clearinghouse for genomic information, molecular information, and more.

GenBank was formed as a data warehouse of EST information, as part of NCBI. It was meant to be an easily searchable database of EST information, making it useful for researchers around the world to use this information. By 1992, the amount of submitted EST information become overwhelming, and NCBI recognized a way of filtering out already sequenced genetic information. NCBI created dbEST as a database for screened EST information and allowing the FASTA format. The FASTA format contains the genetic sequence, accession number, and a few other bits of information useful in sequencing the data. FASTA format is used to help recreate the genomic structure.

The information being used to populate the Gene Database is from nucleotide and proteomic information stored in the NCBI's GenBank. The area of this thesis project is only concerned with human Expressed Sequence Tags (EST) data, but there are many more specie EST data stored in the GenBank file format. As such, only the EST division of GenBank is considered in this project. EST allows researchers to isolate genome sequences that have been worked on previously on a particular problem, and to help isolate a particular sequence of proteins on a particular genome strand. EST typically

report shorter nucleotide sequences, usually between 200 and 500 base pairs[5]. Sequencing is done for either one or both ends of an expressed gene, which generates the nucleotide information.

A gene is expressed through characters that have special meanings. By convention, lower-case letters a, c, g, and t are actual sequences, and capital characters can be referenced in a table that are abbreviations for longer names, or they may have a meaning such as don't care or repeat sections. Repeating sequences that are expressed have a separate file to allow the reassembling of the data if the laboratory desires that information about the genome.

GenBank produces updated EST information several times a year. These updates incorporate the new information that researchers have reported back. It is possible to get the updated EST data along with the current release of EST data from NCBI GenBank website.

A. File Format

EST data that is retrieved from GenBank is stored as compressed files that are organized in structured format. EST data can be retrieved via GenBank's ftp site and downloading the EST files of interest. The file format has a lot of information that is easily recognizable with a quick glance. The EST entries have six basic rules when a full record is created. The file is organized with keywords and their associated descriptions. Each line is considered a record, and the first 10 characters allow for the keyword or sub-keywords. The rest of the line, from position 13 (unless it is part of the nucleotide sequence then it is position 11) to position 80, contains the information pertaining to its keyword. The rules of the keywords are[1]:

1. A keyword begins at position 1.

2. A sub-keyword begins in either position 3 or 4 (depending on the keyword) with the pre-columns left as blank.
3. Blank characters in the first 10 positions indicate that the record is a continuation of the information under the keyword or sub-keyword above it.
4. If there is a code that starts in position 6, it is part of the features table, and those codes are described in Section 3.4.12.1 of ftp://ftp.ncbi.nih.gov/genbank/docs/. The features table version that was the standard during the project's work is 6.2.
5. In the nucleotide sequence, there is a number instead of a word indicating its line number for ordering purposes.
6. The end of an entry is marked with "//" in positions 1 and 2[1].

There is a list of 18 different keywords that make up an EST entry. Those keywords include LOCUS, DEFINITION, REFERENCE, FEATURES, and ORIGIN. Each keyword has a different meaning and usage, which is determined by GenBank. Some keywords are allowed to repeat such as the REFERENCE keyword. Other keywords are used to make up a complex group such as PUBMED and AUTHORS. The following is a quick reference to all the fields that make up an EST entry. A complete document containing all the information about an EST entries file format is on the GenBank website[1], and is rather lengthy.

The LOCUS field holds basic information of the GenBank record and a short name describing it. It contains how many base pairs are in the nucleotide snippet, what kind of sequence it is (DNA, mRNA, etc.), what direction it has, the accession, and when it was submitted to GenBank. This record is exactly one record long, and is a mandatory field.

The DEFINITION field allows for a concise definition of the sequence. This is a mandatory field, with one or more records.

The ACCESSION field contains the records accession number, as well as any previous record's accession numbers that this record used. The primary accession number is used for citations. This field is mandatory with one or more records.

[1] ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt section 3.4.2

The VERSION field is a two part complex field with the primary accession number and a numeric version. The second part of the field starts with the keyword GI, and a numeric number follows. That number is assigned to the sequence by NCBI for better machine parsing. This field is mandatory with exactly one record.

The KEYWORDS field is for all annotated entries. It is a complex field by allowing one or more keywords to be in the field separated by a semi-colon. The last keyword in the list has a period instead. This is a mandatory field if the entry is annotated then it can have one or more records.

The SOURCE field provides the common name or mostly commonly used name for the organism in which the nucleotide came from. This field is mandatory on all annotated entries. It contains one or more records and one sub-keyword.

The ORGANISIM field contains the formal scientific name and on subsequent lines the taxonomic classification levels. This field is a mandatory sub-keyword for all annotated entries and contains two or more records.

The REFERENCE field allows the record to have citations of the publications of where it came from and where to find more information. This is a mandatory field with one or more records. It includes up to seven sub-keywords and can be repeated for multiple citations.

The AUTHORS field lists the authors of the citation. This field is an optional sub-keyword with one or more records.

The CONSRTM field allows for a consortium of organizations for submissions rather than listing all the authors that worked on the sequence. It is an optional sub-keyword with one or more records.

The TITLE field is the full title of the citation. This optional sub-keyword is mandatory in all but unpublished citations with one or more records.

The JOURNAL field lists the journal name, volume, year, and page number from the citation. This sub-keyword is mandatory with one or more records.

The MEDLINE field allows the citation to include the Medline number; which allows it to be looked up at the National Medicine's MEDLINE unique identifiers. This field is an optional sub-keyword with one record.

The PUBMED field is used to reference the PubMed database at http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=PubMed for a citation. This field is an optional sub-keyword with one record.

The REMARK field specifies the relevance of this citation to the entry. It is an optional sub-keyword with one or more records.

The COMMENT field allows for any additional comments, notes, comparisons to other collections, and other remarks. This field is an optional keyword with one or more records, and records may be left blank.

The FEATURES field contains information of biological interest to researchers, process of experiment, location within the genomic sequence, and information on portions of the sequence. This field is an optional keyword with one or more records, and will be explained in the following section.

The ORIGIN field details the point of origination in the genomic map and how the first base is reported. This field is mandatory with one record, but the record may be blank. The ORIGIN keyword is followed by the sequence data; which has one or more records.

B. Features Table Format

The features table is used to define and provide information about the results of the experiments ran and what was of biological interest. This table will contain one or more feature keys.

The features table in each entry contains multiple parts. The FEATURES field first record contained just the string "Location/Qualifiers." Each record thereafter contains information with a feature key, location, descriptor, and a description. The feature key is a label with words such as source, CDS, gene, etc. These are all in positions 6 through 20. In position 21 through 80 is the location in which the particular feature is describing. The descriptor starts on the next record at position 22, and is formatted as /label=. The label is used to help describe why the description is being used. The description is followed by the descriptor and is one or more records long.

The feature table is complex through having one or more feature keys. Each feature key contains a location, and can have zero or more descriptors with descriptions. Each descriptor must have a description. There is a listing of feature keys and descriptors that are already defined by NCBI that can be found at ftp://ftp.ncbi.nih.gov/genbank/docs/FTv6_2.html. An example of the features table is in Figure 1.

This is a quick walk-through of the features table. See the related material for more specific information. Such information that was not included is the list of possible descriptors and the related labels that correspond to a particular descriptor. The other information is what the various lengths mean and the notation's meaning.

```
FEATURES             Location/Qualifiers
     source          1..289
                     /organism="Aotus azarai"
                     /mol_type="genomic DNA"
                     /db_xref="taxon:30591"
     sig_peptide     134..193
     exon            <134..200
                     /number=1
     intron          201..>289
                     /number=1
```

Figure 1[2] Sample Features Table Description

C. Issues of GenBank Raw File Format

Some issues of using GenBank's raw data are that all the GenBank files available for download are compressed. The other issue pertains to search of features stored in the features table that each EST entry contains.

The compressed files that can be retrieved from GenBank are fairly large. Moreover, the file needs to be uncompressed, and that file would need to be parsed to find the information that is of interest. If the wrong file was uncompressed and searched, then another guess as to what file should be parsed would have to be made; this process may be repeated several times until desired results were obtained. Looking at just human EST data, some compressed files are only a few megabytes big, some are bigger in the 20-30 MB size range, and some are huge with 70+ MB size. Since each file is compressed text as well, it makes the uncompressed size quickly go into the 80+ Mb size for a single EST file. There are over 300 EST files. It quickly becomes apparent that why some research labs have incorporated some kind of database to store the GenBank information, since it helps getting the required information quick and easy. This project

[2] Figure 2 was generated from the GenBank entry AF032092, from the Night Monkey.

for the BRG is to allow a better way to access the information starting with the human EST information.

The other issue that is of interest to the BRG concerns with the searching of the features table of any EST entry. The storage, searching, and recreation of the features portion of any particular EST entry is useful because some of the possible information the features contain pertains to an experiment, how it was conducted, and what was accomplished through the experiment. The features portion is referred to as the features table; it holds many records and is set up with several relationships. The features table along with the database schema is discussed in the next chapter.

III. Database Schema

The database is the main focus for this project. It will allow for faster access for the full GenBank record, for specifically queried fields. The major focus is on the features table, which allows features information to be searchable. As seen from the discussion about the features table, it is the most complex field in an EST. The rest of the design is briefly discussed in sections B and C based on the GenBank file description, which is available at the GenBank's website[3].

A. Object Relational With Oracle

One of the more powerful features of Oracle is that it is an Object-Relational Database Management System (ORDBMS). Some of the other commonly used DBMS are Microsoft's SQL Server and open-source MySql, which are Relational Database Management Systems (RDBMS).

A RDBMS allows for multiple entities and relationships to be defined in the schema. Those relationships are constraints in which the database manager enforces and sends back error messages if a statement violates the constraint. An assumption is made that basic knowledge about designing databases that are relational is inferred or can be obtained through a quick search on the Internet.

Object-Relational DBMS is a relatively new type of database management. It allows for more complex object-oriented relationships to be defined and modeled for complex data-centric problems. An ORDBMS also supports relational data as well, so the two designs can be used in the same database[7].

A type, which is an object within the database, allows for one or more fields (columns) to be included. An object can be inherited to create more complex objects. This object can be used as a regular data-type for the rest of the design. An example of

this is the storage of names. If the name contains a first and a last name, the traditional relational approach would use two fields in an entity to hold that information. With an object approach, a type could be created, called tName, containing two fields, first and last. The tName data type then can be included in an entity as a single field and treated the same as any single field in the relational data model.

Object tables are unique tables in the sense that they cannot exist on their own without data. These tables are referenced by entities and are used to hold complex information. An entity with an object-table has a unique table for each column in which the table was specified as the data-type. An example of this is to extend the tName data type presented above. If the requirement is to track all the names in the EST's REFERENCE field, which contains a list of author names, then a possible design is to store names in a single field (i.e. multi-valued field). The data type tName can be created, and then nested within that entity as a table. That table would contain the names of all the authors that worked on the EST. Compared with the relational data model approach to storing multi-valued fields, this nested-table approach is much more intuitive to the users.

Both types and object-tables follow object-oriented rules; which can be extended through object inheritance. They allow for a more complex solution than a traditional relational database can model. Oracle performs faster retrievals using the nested tables than using traditional join statements to retrieve the same information.

Oracle also includes the object-type array. Since the array is static in size, it requires the knowledge of how many elements need to be stored for any particular row using the type array. An array can also use object-data-types.

Oracle's storage mechanism of types and tables is somewhat unique in that it does not actually store the nested information in the table itself. Oracle creates a unique table that is external to the entity, which contains the information and how the two entities link. This linkage is through Oracle row management. Every row in Oracle contains a unique id and a globally unique id. These ids are Oracle generated so they are easier and faster for Oracle to process the information than making a surrogate key.

Oracle is a powerful database management system. Its ability to run on any platform makes it versatile for deciding what platforms are best suited for a particular application. It also integrates with Java, making applications that use Java even easier. Oracle contains everything that is specified in the latest database standard, and some extra features that differentiate it from the rest of the database management systems.

B. Deciding on the Schema

Several different approaches were explored creating the schema for GenBank information. Some of the ways were to create the schema in at least third normal form but preferably in Boyce/Codd Normal Form (BCNF.) The other was to break normalization and use an object-relational approach to the database schema design.

Creating databases in at least third normal form is useful for removing duplicate information, compressing the data by only storing one instance of the data, and having references to it in any pertinent tables that need it. Each entity contains only information that pertains to that entity. Any information that does not pertain to it is moved into an appropriate entity, which helps in storage, and maintaining the desired data integrity constraints. With the textual columns, if one of the rows that are part of a reference is changed, all the references to it will be updated immediately. It is a design issue, and is solvable by knowing what kind of data needs to be stored. If storing data that is case-

sensitive, the approach of normalization should keep the text in the same entity and allow it to repeat[8]. An example of this is in the features table; one entry's descriptor maybe source; and another is Source. This is a subtle difference, but in trying to recreate the GenBank entry exactly, it will fail, and possibly cause it to get a completely new accession number, which should not have occurred in this case.

The object-oriented aspect of Oracle was examined. Some of the features of object-oriented aspects of Oracle made the database schema simpler than a relational data model approach; other aspects of the object-oriented aspects in Oracle would make the database harder to maintain and overly complicated. It would not make sense to create an object to hold what division of GenBank a particular entry is from, whereas the features table of an EST entry is complex. Creating the features table through normalization would become very complex and hard to maintain all the information and links to the information that is in the correct case.

C. Database Schema

Through going in iterative steps, and looking at the composition of the files being imported, several ideas of how to store the data were generated. The first proposed solution had any type of text as its own table, which is not shown here for simplicity. The next solution is to use a hybrid approach; minimizing where it was possible, but some of the tables needed uniqueness as well. The use of an object-relational approach was followed to develop the schema that is used in the project. The use of objects in a database removes the ability of the database to be normalized. This is inherent from the database design, because there will be fields that repeat and are part of a larger entity that a normalization process would break up. There is not a proposed method of calculating how efficient an object-oriented database is; therefore, it is not discussed here.

The process of using smaller tables to hold string data, and pointers of that table are in the main tables is one solution to keeping the amount of text in the database to a minimum. When holding variable length data in an environment with a high level of insertions and deletions, the database easily fragments and slows down performance. The hit in performance of the fragmentation is justifiable for using the smaller tables and joining them with queries. The smaller tables also go along with normalization in terms of minimizing data duplications.

An object-relational model for designing databases goes against the normalization process that is used in any good database schema. A summary of the entities that were decided on is shown in Table I. Most of the entities follow a similar style to an entry. The modifications were made to make the database more efficient than the flat file.

The entities from Table I are the basis of the database diagram shown in Figure 2, drawn to be able to generate the SQL create-table statements for the defining the database schema. There are three things not shown in the schema is that any key that is supposed to be auto-incrementing required a trigger to be attached to the table. Oracle does not provide this functionality. In addition, is nested tables are entities that have a composite type. Finally, the data types for each column were left out for making the database schema more compact. This database diagram was created with Visio 2000, which does not support object-relational database schema design. The database schema is included in Figure 2.

Table I

Entities in the Gene Database

Entity	**Description**
Locus	Central location that holds the basic information of an entry.
Sequences	Holds the nucleotide sequence as well as the number of a’s, c’s, g’s and t’s.
Accession	Creates a way to reference multiple accessions with different versions.
LocusAccession	Allows for a multiple format, along with keeping track of the secondary accessions.
Sources	Holds the scientific and common name of an entry.
Taxonomic	Holds the scientific taxonomic information provided by an entry.
Locus_Sources	This joiner entity allows a way to re-create the sources and taxonomic information.
Definitions	This holds the 0 or more definitions that can occur in any entry.
LocusDefinition	This allows a way to reassemble the definition field.
Features	This is the features table. More information is provided further in the thesis.
Reference	This allows the keeping track of all the references an entry has.
LocusReference	This allows multiple references to be included for an entry.

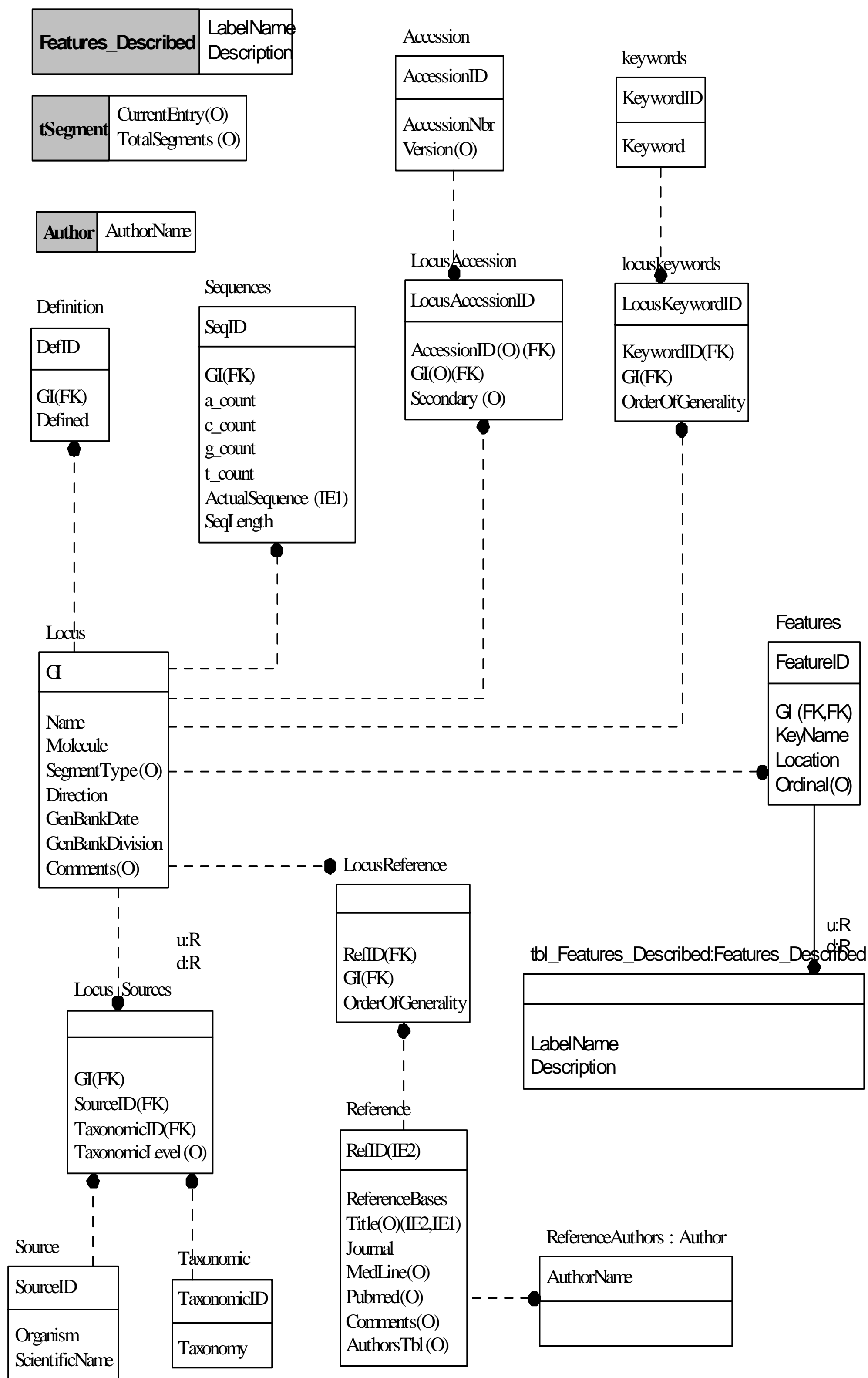


Figure 2 Entire Gene Database Schema

D. Features Table

There are many different aspects of the features table of an entry to be considered. It is the most complex field for any entry and requires the most time in developing a schema to accommodate the complexity. Some of those include how to effectively organize and hold the data allowing for the features while being fast and easy to do searches within the description. Taking an object-relational approach produces a different schema than a traditional relational approach.

A relational database approach would separate out the descriptor and description as one entity and the keywords as either another entity or in a joiner table that allows zero or more descriptor-description pairs. That joiner table would also need the GI to be included and the GI would be used to find all the entries of the features table. There would also need to be a way to keep track of how to reassemble the features table to keep consistent with the flat file format.

The object-relational database approach still requires the separation of the descriptor and description, and keeps track of how to organize the features table to be consistent with any entry. Taking the descriptor and description pair, it was made into a data-type. That data-type was used to create an object-table that is nested in a column. That would allow the features to be included for each row, which speeds up queries and creation of the entry once a match is found. Only one row is needed to hold all the possible descriptor and description pairs, allowing the key name and location to be stored in the same row as well. The location field is left intact since there are numerous ways to report a location and it is not a priority to have a better way of searching the locations. The two extra fields in the features table are for the GI and an ordinal for showing the order to re-assemble all the features for a particular entry.

The database schema for the features table is shown in Figure 3. The entity tbl_Features_Described is a nested table of Features.

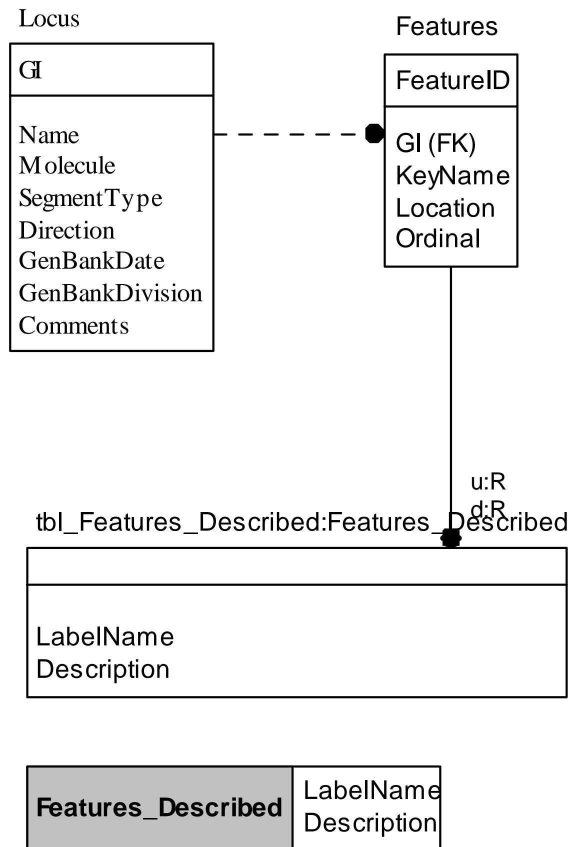


Figure 3 Features Table Schema

The features table is one of the most complex fields in a GenBank entry. The use of an object-oriented database made it simpler to model than the flat-file approach. Accessing the information in the table requires queries to use dot-notation in getting to the sub-data type.

IV. Parser/Importer

The parsing and loading an entry into the database is crucial for the project. Without data in the database, this would just be an exercise in creating a database schema that may or may not work. The program to load the data is broken down into a parser for each entry and an importer for that entry. The parser reports back what it has done and if it was successful. If the entry parsed successfully, the import tool would then proceed to import that entry. After finishing an entry, the parser could work again to get the next entry. This will continue on until the end of file. If multiple files are specified, then the process is repeated for each file. The parser contains information on how the GenBank file will be formatted and what fields are required and what ones are optional. The import tool contains the information of how to put the parser object into the database.

In order for this program to work, the Sun's Java JDK 1.4 or greater is required. The other requirement is to have the Oracle Thin client JDBC installed on the machine as well. Oracle provides several ways to connect to any database that is housed with an Oracle server depending on the environment. The thin client allows calls to Oracle without having any other Oracle programs installed on the machine.

The machine that housed all the data for the BRG is kybrin0, which is on the spd subnet of the Louisville.edu domain. The global version of Java running on kybrin0 is 1.3, which means getting around that limitation. The way that was opted for was to do a local install of Java, and set the class path to look in the local directory first for Java, then the rest of the class path. The class path is Java's version of PATH, which is a pointer to where the compiled classes for the language are. This path is also used when setting up packages, such as JDBC, and Java will not know where to look without the class path

being set. In either Windows or Unix type environment, the class path is set by using the profile, and making a variable called CLASSPATH=”location.”

In order to get Oracle’s JDBC to work the proper driver was downloaded from Oracle’s site. Java does not come bundled with any JDBC, just the interface, so it is required to get the correct JDBC from the vendor. The version of Oracle that was used for the project is 9i. The JDBC was installed locally, and the class path was modified so that it also looked for the JDBC package.

Invoking Java’s compiler and passing it the parameter OracleImportTool.java will compile the program. OracleImportTool.java is the class that contains a main method, which is where Java will start, and in this case where the program’s execution needs to begin. The file OracleConnection.java contains the database connection information as well as the username and password to access the data. To run the program, invoke Java and pass the parameter OracleImportTool and the file name(s) that need to be loaded into the database.

A. Parser

The parser is called first in order to read the file that is passed into the object and to gather the information about each entry. The parser is called repeatedly until all the records in the file have been worked on. The parser uses the regular expression class that is found in Java 1.4, which is used to get all the information up to the “//”, and then breaks it down into 3 main sections. The first section gets all the information from all the fields from LOCUS up to but not including the REFERENCE. The second section just gathers all the REFERENCE fields. The third section is the rest of an entry.

In each section, the use of patterns to help match the individual field is used. This pattern matching is used then to either get the individual information in a simple field, or

to further split up complex fields. All of this is hard-coded into the parser so there is no way to change the behavior if a future release changes or adds in new fields.

The parser is invoked by the calling parseFile method, and passing in a parser listener object. The parser listener object is used as a way to pass back the information of a success status of a record parse and saying that it has the information ready to be used. It allows the caller to use the information before returning to the parser for the next record.

The parser is like any other basic parser. It breaks the file up and puts the required field found at some position into a variable to be used by another process. Regular expressions made it more manageable not only in just general breaking up the entry but also in finding repeating fields.

B. Import Tool

The import tool uses the parser that was created for this project and described in the previous section and sends commands to load the data into Oracle. There are bulk loader tools that work in some of the cases for Oracle but not in the case of this schema. Oracle's bulk loader, SQL*Loader, is discussed in the conclusions of this thesis. The import tool also handles uniqueness and does not insert multiple entries that already exist in the database.

The import tool uses the schema in a different order than what is defined. It loads the information that another entity needs and stores the results in a local variable. Those results are then used to populate the next entity that needs the results. Before it does an insert statement, it queries the database to see if the values are in the database. If they already exist then it will either skip over that entity, or get unique id for that entity, and use it for future entities that need that unique id.

The information used by OracleImportTool is obtained by the invocation of a Parser object. The Parser object notifies the importer that it has a record ready and waits until it is invoked again. If there is data left in the GenBank record the parser will get the information, and the importer will continue to import until there are no more records to process in the file.

The connection to Oracle is established from another class that handles the connection object. The class sets up any parameters for the database connection and keeps from having multiple connections to the database.

The Import Tool is a simple importer. The most difficult task that it has is to gather all the information that the parser acquired and use it to populate the database. The connection is made from the other class as a way to break down the code into more specific tasks.

C. Loading Data

With the tools created to import the record, a simple bash script was written to get all the files of a particular directory. The script goes into a loop where it first unzips the files to standard output. The standard output was piped into a temporary file, and the import tool ran with the files name as the only argument. The actual implementation is just a little more complicated by needing the data files to be in the user's local directory to run the program, and it goes back to the directory for the next file.

V. Conclusions

A. SQL*Loader

SQL*Loader is an Oracle's tool for doing bulk loads of data. It allows the disabling of constraint checking, and re-enabling the constraint checking after it is done. SQL*Loader allows multiple-table inserts in the same file, object tables, limited conditional statements, and basic SQL statements. SQL*Loader parses the file, creates a bulk insert statement that Oracle understands, and reports back the records with errors, success, and general information. SQL*Loader provides nice flexibility for importing old data that does not use auto-generated keys in entities. The loader will also go through all data that are present and keep doing the insertions until there are no data left.

This tool works great with tables that do not refer to other table's surrogate key or use objects. SQL*Loader can generate the surrogate key through the built-in sequence generator.

SQL*Loader was not used in this project because of the lack of looking up surrogate keys that are auto-generated. The use of auto-generated surrogate keys is a choice made for duplication reasons. It allows going through any entity that needs to be changed and has a key that is truly unique. If for some reason a row was duplicated that does not have an auto-generated number, a deletion to remove the one row will remove both rows due to Oracle not being able to differentiate between the two.

SQL*Loader does not provide a means to do SQL statements on filler columns. A filler column in the control portion is a column that does not map to the database, but can be used to hold other information or to skip records. Without a way to find uniqueness and their associated index, it became obvious that Oracle's SQL*Loader would not work.

It would not take a lot to get around this limitation. The issue then becomes space of the generated files to hold all the information.

B. Performance of program

The use of a custom import tool made importing data slower. This was only considered because of the data schema for holding the GenBank information. Future versions of SQL*Loader may allow it to be used. The space required for unzipping all the files sequentially versus all of them at one time was a factor in the custom import tool. The time it took was the other factor. The tests that were performed were on a small subset of data that was stripped to only human EST information. The filename convention from GenBank for EST information is gbestXXX.seq.gz, where XXX is a number. The stripped data uses a similar naming convention of humanestXXX.seq.gz, where XXX is a number.

The compression that the files have from GenBank is roughly 91% smaller. For a file that is 19 Megabytes from GenBank, it would require approximately 212 Megabytes uncompressed. The GenBank information that is human EST requires 1800 Megabytes. Uncompressed human EST files would require approximately 20199 Megabytes, or 21 Gigabytes of hard drive space. The sample data that was generated is in Table 2. Looking at the size of the compressed file, and comparing it to the uncompressed size, making the ratio, acquired the information. The average ratio was calculated and the estimates for the uncompressed size were made and are summarized in Table 2.

Table 2
Compression Ratio of GenBank Information

File:	Compressed Size (M)	Uncompressed Size (M)	Compression Ratio (C/U)
humanest108.seq.gz	19	212	0.0896
humanest275.seq.gz	1.3	14	0.0928
humanest303.seq.gz	14	166	0.0843
humanest304.seq.gz	13	136	0.0955
humanest305.seq.gz	7.9	95	0.0831
		Average Ratio	0.0891
		Directory Size (M)	1800
		Uncompressed Size (M)	20199.1518
		Compressed %	91.08873472

The time that the import tool required to run was calculated by taking how long several different files took to import. The files used were of different size, and an analysis of the number of records in the file was calculated. The number of records that the import tool can handle is 6.2 records per second. If there were a million records to be inserted, it would take approximately 9 days to do all the insertions. That does not take into account the time it takes to uncompress the file. A summary of how the average number of records for insertions is included in Table 3.

The uncompressed space requirement is attainable by any academic institution; performing searches on the raw flat file format is inefficient. This analysis is only the human EST information; the entire GenBank contains more than 147 Gigabytes of information.

Table 3
Database Loading Information

EST File	# Human ESTs	Seconds	Records/ Second	Uncompressed File Size
Humanest.108.seq	71,160.00	18000	3.953	212
Humanest.155.seq	57,096.00	8424	6.778	195
Humanest.275.seq	4,172.00	1046	3.989	14
Humanest.304.seq	44,164.00	6120	7.216	136
Humanest.305.seq	32,603.00	3480	9.369	95
		Average	6.261	

C. Results Tables

This aspect of the project has not been discussed mostly from this being more of a future extension and each person that may be running experiments on it may or may not need the tables that exist, or need to store other information, which may or may not be uploaded to the same database. The user that will initially be using this database is comparing a sequence to other sequences in the EST to find how closely they are. Some of the information that is being stored is where the pattern starts and stops, what EST it came from, the percentage covered, and the matching value. This is of interest to the individual's research project.

The results database that is included is for the projects initial use, another member of the BRG will link to the Gene Database and use it for their research. What it does incorporate is how to expand the information, and use it for other research-based storage.

D. Future Projects and Enhancements

There were several things that can be improved upon this project. The first is to extend the database to include the human Genome sequence. A more robust front-end user interface would be good for someone trying to access the Gene Database with external links to other biological databases. An implementation of a programmable

library or class that can be hooked into by any program that gives the Gene Database information and is self-contained to internal knowledge of the database can be added as a way to encourage more research at the University of Louisville using the database holding GenBank information. Verifying the data being populated in the database is another step in the future to be taken. The verification needs to make sure that the flat file format can be replicated, and possibly to check the sequence's going in are the right species. To verify the flat file, a program is needed to query the database, and gather the information t reconstruct the database. With that data, a regular expression tool, such as grep, can be used to find the differences, and if there is no difference then it was a successful load of the file. When the GenBank structure does change, it will be necessary to update the structure of the database to handle the changes in the GenBank format. This can be done because the Locus table does not contain any information about the other segments of the GenBank. Adding or removing new fields will be a matter of adding in the new table's schema to the existing database schema. Then populate the table with the new information.

E. Conclusions

The Gene Database was a success in the standpoint that it imported the data that can be used by the BRG to do further research dealing with genomic data. The human EST data importing does take about a week to complete. However, GenBank only releases updated information every quarter, so a week it takes to populate the database with the new information is acceptable. This project is the initial step to helping make searches and doing computer algorithms dealing with GenBank information easier.

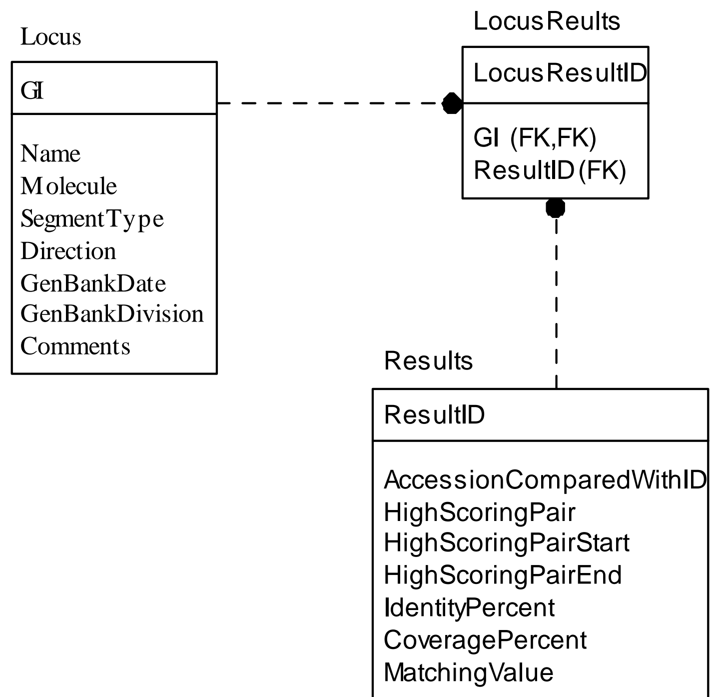
References

1. NCBI. 2004. Genetic Sequence Data Bank. ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt
2. NCBI. 2004. The DDBJ/EMBL/GenBank Feature Table: Definition ftp://ftp.ncbi.nih.gov/genbank/docs/FTv6_2.html
3. NCBI. 2004. A Science Primer. National Center for Biotechnology Information. http://www.ncbi.nlm.nih.gov/About/primer/est.html
4. NCBI. 2004. http://www.ncbi.nlm.nih.gov/
5. NCBI. 2004. GenBank Overview. http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html
6. NCBI. 2004. NCBI dbEST. http://www.ncbi.nlm.nih.gov/dbEST/index.html
7. Ulman, Jeff. 1997. Oracle SQL. http://www-db.stanford.edu/~ullman/fcdb/oracle/or-nonstandard.html. Stanford
8. Wyllys, R. E. 2002. Overview of Normalization. http://www.gslis.utexas.edu/~l384k11w/normover.html

Appendix I: Sample EST

```
LOCUS       AA000972                 449 bp    mRNA    linear   EST 29-NOV-1996
DEFINITION  ze46c04.r1 Soares retina N2b4HR Homo sapiens cDNA clone
            IMAGE:362022 5' similar to gb:X61499 NUCLEAR FACTOR NF-KAPPA-B P49
            SUBUNIT (HUMAN);contains Alu repetitive element;, mRNA sequence.
ACCESSION   AA000972
VERSION     AA000972.1  GI:1437057
KEYWORDS    EST.
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 449)
  AUTHORS   Hillier,L., Clark,N., Dubuque,T., Elliston,K., Hawkins,M.,
            Holman,M., Hultman,M., Kucaba,T., Le,M., Lennon,G., Marra,M.,
            Parsons,J., Rifkin,L., Rohlfing,T., Soares,M., Tan,F.,
            Trevaskis,E., Waterston,R., Williamson,A., Wohldmann,P. and
            Wilson,R.
  TITLE     The WashU-Merck EST Project
  JOURNAL   Unpublished (1995)
COMMENT     Contact: Wilson RK
            Washington University School of Medicine
            4444 Forest Park Parkway, Box 8501, St. Louis, MO 63108
            Tel: 314 286 1800
            Fax: 314 286 1810
            Email: est@watson.wustl.edu
            This clone is available royalty-free through LLNL ; contact the
            IMAGE Consortium (info@image.llnl.gov) for further information.
            Insert Length: 3119   Std Error: 0.00
            Seq primer: mob.REGA+ET
            High quality sequence stop: 342.
FEATURES             Location/Qualifiers
     source          1..449
                     /organism="Homo sapiens"
                     /mol_type="mRNA"
                     /db_xref="GDB:1278725"
                     /db_xref="taxon:9606"
                     /clone="IMAGE:362022"
                     /sex="male"
                     /tissue_type="retina"
                     /clone_lib="Soares retina N2b4HR"
                     /dev_stage="55 year old"
                     /lab_host="DH10B (ampicillin resistant)"
                     /note="Organ: eye; Vector: pT7T3D (Pharmacia) with a
                     modified polylinker; Site_1: Not I; Site_2: Eco RI; 1st
                     strand cDNA was primed with a Not I - oligo(dT) primer [5'
                     TGTTACCAATCTGAAGTGGGAGCGGCCGCGCTTTTTTTTTTTTTTTTTTT 3'],
                     double-stranded cDNA was size selected, ligated to Eco RI
                     adapters (Pharmacia), digested with Not I and cloned into
                     the Not I and Eco RI sites of a modified pT7T3 vector
                     (Pharmacia). The retinas were obtained from a 55 year old
                     Caucasian  and total cellular poly(A)+ RNA was extracted 6
                     hrs after their removal. The retina RNA  was kindly
                     provided by Roderick R. McInnes M.D. Ph.D. from the
                     University  of Toronto. Library constructed by Bento
                     Soares and M.Fatima Bonaldo."
ORIGIN
        1 attcggcact aggcataaag caaacagaat catcacttct actgcagcct aaaggtgttg
       61 ataattatcc agaaaagcaa atctcctttc tataaggttc agcggtttga gagttgtgat
      121 gnggaaggca ctccttcgga aatcatcctc actatctaga gatatttcag ggctggggta
      181 ctctcctgtg aacaatagga aaatacatac ttaattgaaa gtaataaacc agggttctga
      241 cttttttttt tttntttctt tgagacggaa tttcactctc ttgcccaagg ctggagtgca
      301 gcaagcgcta atctccgctc actgcagcct ctaacctccc ngggttcaaa gcaatntctc
      361 ctgccttcaa gntcttccaa gaatangctg gggattacag ggcacccaan caggaanggc
      421 ccggggntaa attttttngg gtggttttt
//
```

Appendix II: Results Database Diagram



Appendix III: SQL Statements for GenBank Database Schema

```
/*
   Author: Nathan Mann
   Connection: EST Database
   Host: Kybrinwb

   This script will generate the database according to the
   schema described, and set-up the auto-numberings.
   This will drop any tables that exist that are named the
   same as the ones for this database.  Use with CAUTION!

   Drop all tables, and the types so that new types can be created
   for this database.
 */
drop table Sequences cascade constraints;
drop table Accession cascade constraints;
drop table LocusAccession cascade constraints;
drop table Definitions cascade constraints;
drop table Features cascade constraints;
drop table Reference cascade constraints;
drop table LocusReference cascade constraints;
drop table Keywords cascade constraints;
drop table LocusKeywords cascade constraints;
drop table Taxonomic cascade constraints;
drop table Sources cascade constraints;
drop table Locus_Sources cascade constraints;
drop table Locus cascade constraints;

drop type tbl_Features_Described;
drop type ReferenceAuthors;

/*
   Type: tSegment
   Holds information about a segment of an entry.
 */
create or replace type tSegment as object
(
   CurrentEntry number,
   TotalSegments number
)
/

/*
    Table: Locus
   This will hold all the basic information,
   And will allow for a complete GenBank type to be found, and created.
*/
create table Locus
(
   GI number,
   Name varchar2(10) not null,
   Molecule varchar2(7) not null,
   SegmentType tSegment,
   Direction varchar2(7) not null,
   GenBankDate date not null,
   GenBankDivision char(3) not null,
   Comments varchar2(2000),
   constraint pk_GI primary key(GI)
);

/*
    Table: Sequences
   This is what will hold the orgin information, and basic
   aggregate functions that can be of use to researchers.
 */
create table Sequences
(
    SeqID number,
   GI number,
   a_count number not null,
   c_count number not null,
```

```
   g_count number not null,
   t_count number not null,
   ActualSequence clob not null,
   SeqLength number not null,
   constraint pk_SeqID primary key (SeqID),
   constraint fk_Sequences_GI foreign key (GI) references Locus (GI)
);

/*
    Table: Definitions
   This holds the definition field
   of an entry.
 */
create table Definitions
(
   DefID number,
   GI number,
   Defined varchar2(2000) not null,
   constraint pk_DefID primary key(DefID),
   constraint fk_Definitions_GI foreign key (GI) references Locus (GI)
);


/*
   Section: Accession Information
   This will set-up the tables used to hold information
   about the accession numbers, the version, and any secondary accessions.

   Table: Accession
   This is the part that keeps the various versions straight
   Human readable, uses a ##.# for major and minor revisions.
*/
create table Accession
(
   AccessionID number,
   AccessionNbr char(8) not null,
   Version number,
        constraint pk_AccessionID primary key (AccessionID)
);

/*
   Table: LocusAccession
   This will allow for a versioning of the different
   Genbank entries
 */
create table LocusAccession
(
   LocusAccessionID number,
   AccessionID number,
   GI number,
   Secondary number(2),
   constraint fk_LocusAccession_AccessionID foreign key (AccessionID)
      references Accession (AccessionID),
   constraint fk_LocusAccession_GI foreign key (GI) references Locus (GI)
);

/*
    Section: Source Information
   This section sets-up the tables needed to keep track of the source,
   the taxonomic information of the orginims, and how to
   reassamble the the entry.

   Table: Sources
   This holds the name of the thing the dna came from
   such as homo sapiens (human)
 */
create table Sources
(
    SourceID number,
   Organism varchar2(200) not null,
   ScientificName varchar2(500) not null,
   constraint pk_SourceID primary key (SourceID)
);
```

```
/*
   Table: Taxonomic
   This table is used for knowing more information about the
   orginism, and its orgins, and what other species contains
   similiar information
 */
create table Taxonomic
(
   TaxonomicID number,
   Taxonomy varchar2(50) not null,
   constraint pk_TaxonomicID primary key (TaxonomicID)
);

/*
   Table: Locus_Sources
   This is a joiner table between Locus, and the source
   for knowing what a species came from.  It is used in
   recreating entries.
 */
create table Locus_Sources
(
   GI number,
   SourceID number,
   TaxonomicID number,
   TaxonomicLevel number,
   constraint fk_LocusSource_SourceID foreign key (SourceID)
      references Sources (SourceID),
   constraint fk_LocusSource_GI foreign key (GI) references Locus (GI),
   constraint fk_LocusSource_TaxonomicID foreign key (TaxonomicID)
      references Taxonomic (TaxonomicID)
);

/*
   Section: Keyword
   These are the words that describe a particular entry.
   There can be 0 to more keywords in any entry.  It will
   allow the entry to be reassembled in the correct order.

   Table: Keywords
   This table holds the word that needs to be defined, and
   the id can be used to reference the word.
 */
create table Keywords
(
   KeywordID number,
   Keyword varchar2(200) not null,
   constraint pk_KeywordID primary key(KeywordID),
   constraint uq_Keyword unique (Keyword)
);

/*
   Table: LocusKeywords
   This table allows the combination of the locus table
   to attach to the keyword, and specify the order it goes.
 */
create table LocusKeywords
(
   LocusKeywordID number,
   KeywordID number,
   GI number,
   OrderOfGenerality number not null,
   constraint pk_LocusKeywordID primary key (LocusKeywordID),
   constraint fk_LocusKeywords_GI foreign key (GI) references Locus (GI),
   constraint fk_LocusKeywords_KeywordID foreign key (KeywordID)
      references Keywords (KeywordID)
);
/*
   Section: References
   This section sets-up the way to store the reference information
   of the entry.
   It allows for multiple references with multiple configurations, for
   a nice flexible storage, and retrievals.
```

```
   Type: Authors
   This type allows the name of the author to be defined.  It is to be used
        with creating a nested table inside of the Reference table.
 */
create or replace type Author as object
(
        AuthorName varchar2(75)
)
/

/*
   Type: ReferenceAuthors
   This type is of table, with the columns defined in the type Author.
   This will be nested inside each row of the Reference table.
 */
create or replace type ReferenceAuthors as table of Author
/

/*
   Table: Reference
   This will hold the reference information.
   The information is any that may be included in any reference.
 */
create table Reference
(
   RefID number,
   ReferenceBases varchar2(50) not null,
   Title varchar2(200), --Optional
   Journal varchar2(200) not null,
   MedLine number,   --Optional - links to a website
   PubMed number, --Optional - links to a website
    Comments varchar2(1000),  --Optional
   AuthorsTbl ReferenceAuthors,
   constraint pk_RefID primary key(RefID)
)
nested table AuthorsTbl store as ReferenceAuthors_Table;

/*
   Table: LocusReference
   Joiner table to hold 1 entry to many reference entries.
   Each reference entry
 */
create table LocusReference
(
   GI number,
   RefID number,
   OrderOfGenerality number not null,
   constraint fk_RefID foreign key(RefID) references Reference (RefID),
   constraint fk_Locus foreign key(GI) references Locus(GI)
);

/*
   Section: Features
   This section sets up the features table, with all the objects
   to hold the feature information.  It allows for multiple
   decriptor's/description combinations.  It also allows for
   multiple keynames for the features table.

   Type: Feature_Described
   This holds the description and the descriptor (label)
   for a single feature.
 */
create or replace type Feature_Described as object
(
    LabelName varchar2(50),
   Description varchar2(1000)
)
/

/*
   Type: tbl_Features_Described
   This type allows the features to include every descriptor/description
```

```
   inside the row, keeping it straight for retrievals.
 */
create or replace type tbl_Features_Described as table of Feature_Described
/

/*
   Table: Features
   This table holds all the information to replicate the features table
   of an entry.  This makes some use of Oracle's object-oriented
   database features.
 */
create table Features
(
   GI number not null,
   FeatureID number not null,
   KeyName varchar2(50) not null,
   Location varchar2(50) not null,
   Ordinal number not null,
   FeaturesTbl tbl_Features_Described,
   constraint pk_FeatureID primary key(FeatureID),
   constraint fk_Features_GI foreign key (GI) references Locus(GI)
)
nested table FeaturesTbl store as Features_Described_Table;

/*
   Drop al the sequences that could conflict within the Oracle DB.
   The sequences will be used when generating the index.
   It takes a unique sequence to do the auto-incrementing,
   then a trigger will be added to auto-magically create the autonumbering
   effect to the tables that need auto-numbering.
 */
drop sequence Accession_Seq;
drop sequence Titles_Seq;
drop sequence References_Seq;
drop sequence Definition_Seq;
drop sequence Taxonomic_Seq;
drop sequence Source_Seq;
drop sequence Sequences_Seq;
drop sequence Features_Seq;
drop sequence Locus_Seq;
drop sequence LocusAccession_Seq;
drop sequence Keywords_Seq;
drop sequence LocusKeywords_Seq;

/*
   Sequences: Each one referes to a different table.  Past experience
   has shown that it is easier to do deletes when there is a unique ID
   easily retrieved.
   Having the start with 1 will reset any information about the index that
   Oracle may have in it's cache.
 */
create sequence Accession_Seq start with 1;
create sequence Features_Seq start with 1;
create sequence Titles_Seq start with 1;
create sequence References_Seq start with 1;
create sequence Definition_Seq start with 1;
create sequence Taxonomic_Seq start with 1;
create sequence Source_Seq start with 1;
create sequence Sequences_Seq start with 1;
create sequence Locus_Seq start with 1;
create sequence LocusAccession_Seq start with 1;
create sequence Keywords_Seq start with 1;
create sequence LocusKeywords_Seq start with 1;
/*
   Triggers: These have been added to create the autonumbering.
   Each one is basically the same.
   Create the trigger, say that it is to be done before an insert
   and put in the column of the table the next sequence value.
 */
create or replace trigger Accession_Bir
before insert on Accession
for each row
begin
```

```
   select Accession_Seq.NEXTVAL
   into :new.AccessionID
   from dual;
end;
/

create or replace trigger LocusAccession_Bir
before insert on LocusAccession
for each row
begin
   select LocusAccession_Seq.NEXTVAL
   into :new.LocusAccessionID
   from dual;
end;
/

create or replace trigger References_Bir
before insert on Reference
for each row
begin
   select References_Seq.NEXTVAL
   into :new.RefID
   from dual;
end;
/

create or replace trigger Definition_Bir
before insert on Definitions
for each row
begin
   select Definition_Seq.NEXTVAL
   into :new.DefID
   from dual;
end;
/

create or replace trigger Source_Bir
before insert on Sources
for each row
begin
   select Source_Seq.NEXTVAL
   into :new.SourceID
   from dual;
end;
/

create or replace trigger Taxonomic_Bir
before insert on Taxonomic
for each row
begin
   select Taxonomic_Seq.NEXTVAL
   into :new.TaxonomicID
   from dual;
end;
/

create or replace trigger Sequences_Bir
before insert on Sequences
for each row
begin
   select Sequences_Seq.NEXTVAL
   into :new.SeqID
   from dual;
end;
/

create or replace trigger Features_Bir
before insert on Features
for each row
begin
   select Features_Seq.NEXTVAL
   into :new.FeatureID
   from dual;
```

```
end;
/

create or replace trigger Keywords_Bir
before insert on Keywords
for each row
begin
   select Keywords_Seq.NEXTVAL
   into :new.KeywordID
   from dual;
end;
/

create or replace trigger LocusKeywords_Bir
before insert on LocusKeywords
for each row
begin
   select LocusKeywords_Seq.NEXTVAL
   into :new.LocusKeywordID
   from dual;
end;
/

/*
   These procedures are useful, and are not needed for this database.
   For speed considerations, they should be used before, and after the
   running of an import tool, if there is a huge amount of data to be stored
   at one time.

   Procedure: EnableConstraints
   This will enable all the constraints within the user's schema.
   i.e. checking of keys will be started, and used if this procedure
    is called.
   Usage: execute EnableConstraints
 */
create or replace procedure EnableConstraints
 as
begin
  begin
  declare
  v_sql varchar2(1000);

  begin
    for rec in (select * from user_constraints)
    loop
      v_sql := 'alter table '||rec.table_name||
        ' disable constraint '||rec.constraint_name;
     execute immediate v_sql;
    end loop;

    exception
    when others then
      dbms_output.put_line('Exception:'||sqlerrm);
    end;
  end;
end;
/

/*
   Procedure: DisableConstraints
   This will disable all the constraints within the user's schema.
   i.e. checking of keys will be stopped if this procedure is called.
   Usage: execute DisableConstraints
 */
create or replace procedure DisableConstraints
 as
begin
  begin
  declare
  v_sql varchar2(1000);

  begin
    for rec in (select * from user_constraints)
```

```
    loop
      v_sql := 'alter table '||rec.table_name
        ||' disable constraint '||rec.constraint_name;
     execute immediate v_sql;
    end loop;

    exception
    when others then
      dbms_output.put_line('Exception:'||sqlerrm);
    end;
  end;
end;
/
```

Appendix IV: SQL Statements for the Results Database Schema

```
/*
   Author: Nathan Mann
   Connection: EST Database
   Host: Kybrinwb

   This database is used to hold results from research being done
   by Elizebeth, of the Bio-informatics group.  It holds
   the results of that research, and links back to the
   GenBank information that exists in the database.

   Drop all tables, and the types so that new types can be created
   for this database.
 */
drop table Results;
drop table LocusResults;

/*
   Table: Results
   This table holds the results from Elizebeth's findings.
 */
create table Results
(
   ResultID number,
   AccessionComparedWithID number not null,
   HighScoringPair number not null,
   HighScoringPairStart number not null,
   HighScoringPairEnd number not null,
   IdentityPercent number(2,10) not null,
   CoveragePercent number(2, 10) not null,
   MatchingValue number(2, 10) not null,
   constraint pk_ResultID primary key(ResultID)
);

/*
   Table: LocusResults
   This table joins the results to the rest of the database
 */
create table LocusResults
(
   LocusResultID number,
   GI number,
   ResultID number,
   constraint pk_LocusResultID primary key(LocusResultID),
   constraint fk_LocusResults_GI foreign key (GI)
      references LocusResults(GI),
   constraint fk_LocusResults_ResultID foreign key (ResultID)
      LocusResults(ResultID)
);

/*
   Drop al the sequences that could conflict within the Oracle DB.
   The sequences will be used when generating the index.
   It takes a unique sequence to do the auto-incrementing,
   then a trigger will be added to auto-magically create the autonumbering
   effect to the tables that need auto-numbering.
 */
drop sequence Results_Seq;
drop sequence LocusResults_Seq;

/*
   Sequences: Each one referes to a different table.  Past experience
   has shown that it is easier to do deletes when there is a unique ID
   easily retrieved.
   Having the start with 1 will reset any information about the index that
   Oracle may have in it's cache.
 */
create sequence Results_Seq start with 1;
create sequence LocusResults_Seq start with 1;

/*
```

```
   Triggers: These have been added to create the autonumbering.
   Each one is basically the same.
   Create the trigger, say that it is to be done before an insert
   and put in the column of the table the next sequence value.
 */
create or replace trigger Results_Bir
before insert on Results
for each row
begin
    select Results_Seq.NEXTVAL
   into :new.ResultID
   from dual;
end;
/

create or replace trigger LocusResults_Bir
before insert on LocusResults
for each row
begin
    select LocusResults_Seq.NEXTVAL
   into :new.LocusResultID
   from dual;
end;
/
```

Appendix V: Parser Source Code

```
import java.io.*;
import java.util.*;
import java.util.regex.*;

/**
 * This class is for parsing plain text files in the GenBank format
 * @see
 * <a href="http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html">
 * GenBank Sample Record</a><br>
 *
 * This class is modified from MWING, a class project.
 *
 */
public class Parser
{

   interface ParserListener
   {
      void callback(boolean isSuccess, String error);
    }

   /**
    * Amount of data to be read from the disk at one time.
    */
    public int IOBUFFER = 350000;

    /**
    * The constructor will set-up the pattern to be used,
    * With the pattern's set, it makes it a lot easier
    * to parse the GenBank file.
    */
   public Parser()
   {
      // pre-compile these patterns because they are
        // used to parse every record.
        _Pattern_Section1 = Pattern.compile("^REFERENCE", Pattern.MULTILINE);
        _Pattern_Section2 =
            Pattern.compile("^(COMMENT|FEATURES)", Pattern.MULTILINE);

        _Pattern_Section1_Fields =
            Pattern.compile(
                "^(LOCUS|DEFINITION|ACCESSION|VERSION|KEYWORDS|SOURCE)|GI:",
                Pattern.MULTILINE);
        _Pattern_Section2_Fields =
         Pattern.compile("^REFERENCE",Pattern.MULTILINE);
        _Pattern_Section3_Fields =
         Pattern.compile(
            "^(COMMENT|FEATURES|ORIGIN)",
                Pattern.MULTILINE);
    }

    /**
      * Parses GenBank records one at a time. A ParserListener
      * object is notified whether or not the parse of the record
      * was successful.
      */
    public void parseFile(String seqFileName, ParserListener listener)
    {
        String[] records = null;
        String str = "", str2 = "";
        char[] cbuf = new char[IOBUFFER];//for the data to be loaded
        BufferedReader bfr = null;//file reader
         File flSize = new File(seqFileName);
        int bytesRead = 0;
        Pattern p = Pattern.compile("^//", Pattern.MULTILINE);

        try
        {
         //Read the file, and get the information, if there is more than 0 bytes.
            bfr = new BufferedReader(new FileReader(seqFileName), cbuf.length);
```

```
      if (!bfr.ready())
      {
         listener.callback(false, "File is 0 length - " + seqFileName);
         return ;
      }
     }
     catch (IOException e)
     {
         listener.callback(false, e.toString());
         return;
     }

   // buffer 350 K at a time
    while (bytesRead != -1)
   {
      try
         {
            bytesRead = bfr.read(cbuf, 0, cbuf.length);
       }
      catch (Exception e)
         {
            listener.callback(false, e.toString());
           return;
      }

         if (bytesRead != -1)
        {
           str = new String(cbuf);
         // put the records in an array
             records = p.split(str);
         // wraparound the end of the previous record
         records[0] = str2 + records[0];
             str2 = new String(records[records.length - 1]);
            for (int i = 0; i < records.length - 1; i++)
           {
              try
             {
               // parse each record and notify listener if successful
                 parseRecord(records[i]);
                listener.callback(true,"");
                 }
                catch (IndexOutOfBoundsException e)
               {
                  listener.callback(false, "Parse Failure - GI: " + gi
                       + e.toString());
             }

             }
        }
   }
     try
    {
       bfr.close();
     }
    catch (IOException ignored)
   {
     }
 }

 /*   This is the mean of the parser.
 * It is where all the fields are tooken out
 * and put into the variables, which can
 * be used for other applications.
 */
private void parseRecord(String gbRecord)
{
   gi = "-1";
     Matcher m;
     Pattern p;
     int prevMatchPos = 0;
     record = new String(gbRecord);
     String []temp=null;
   String strTemp = "";
```

```
//Setup the sequence information data structure
sequence = new SequenceInformation();

  //This will place a double quote, which keeps
//the structure of the string for any database.
record = record.replaceAll("'", "''");

  //For split record into the main sections
  String _Section1;
  String _Section2;
  String _Section3;

  //Get everything up to (but exclude) REFERENCE
  m = _Pattern_Section1.matcher(record);
  if (!m.find())
      return;
  _Section1 = record.substring(prevMatchPos, m.start() - 1);
  prevMatchPos = m.start();

  //Get Everything Between REFERENCE and ^(COMMENT|FEATURES)
  m = _Pattern_Section2.matcher(record);
  if (!m.find())
      return;
  _Section2 = record.substring(prevMatchPos, m.start() - 1);
  prevMatchPos = m.start();

  //Get everything Else
  _Section3 = record.substring(prevMatchPos);

  // Split main sections into subsections
  String[] _Section1_Fields = _Pattern_Section1_Fields.split(_Section1);
  String[] _Section2_Fields = _Pattern_Section2_Fields.split(_Section2);
  String[] _Section3_Fields = _Pattern_Section3_Fields.split(_Section3);

  //////////////////////////////////////////////////
  // Parse section 1 - ^LOCUS thru ^REFERENCE
  /////////////////////////////////////////////////
  String _Locus_Field = new String(_Section1_Fields[1].trim());
  String keywords_Field = new String(_Section1_Fields[6].trim());
  String _Source_Field = new String(_Section1_Fields[7].trim());

  //LOCUS information
  String[] _Locus_Sections = _Locus_Field.split("\\s+");
  int len = _Locus_Sections.length;

  locusName = new String(_Locus_Sections[0].trim());
  sequence.sequenceLength = new String(_Locus_Sections[1].trim());
  moleculeType = new String(_Locus_Sections[3].trim());
  geneDirection = new String(_Locus_Sections[4].trim());
  genbankDivision = new String(_Locus_Sections[len - 2].trim());
  modificationDate = _Locus_Sections[len - 1].trim();

//Get all the accession's, including the secondary ones
strTemp = _Section1_Fields[3].trim();
accession = strTemp.split(" ");
if (accession[0] == "")
{
   accession[0] = _Section1_Fields[3].trim();
}

  //Version
  temp=_Section1_Fields[4].split("\\.");
  version=new String(temp[1].trim());
  gi = new String(_Section1_Fields[5].trim());

  //KEYWORDS
  keywords_Field = keywords_Field.replaceAll("\\n|\\.|SEGMENT.*", "");
  keywords = null;
  if (!keywords_Field.equals(""))
{
      keywords_Field = keywords_Field.replaceAll("\\s+", " ");
      keywords = keywords_Field.split(";");
```

```
      for (int i = 0; i < keywords.length; i++)
          keywords[i] = keywords[i].trim();
  }

//Definitions
  definition = _Section1_Fields[2].replaceAll("\\s+", " ").trim();

  //SOURCE
  String _Source_Sections[] = _Source_Field.split("ORGANISM");
  commonName = new String(_Source_Sections[0]).trim();
temp = commonName.split("\\(");
temp = temp[1].split("\\)");
commonName = temp[0];

String _Sci_Sections[] = _Source_Sections[1].split("\n", 2);
  scientificName = _Sci_Sections[0].trim();
  scientificFamilyTree = _Sci_Sections[1].split(";|\\.");
  for (int i = 0; i < scientificFamilyTree.length; i++)
      scientificFamilyTree[i] = scientificFamilyTree[i].trim();

  //////////////////////////////////////////////////////////
  // Parse section 2 - ^REFERENCE thru ^(COMMENT|FEATURES)
  //////////////////////////////////////////////////////////
  references = new Reference[_Section2_Fields.length - 1];
  p = Pattern.compile("TITLE|JOURNAL|MEDLINE|PUBMED|REMARK");
  for (int i = 0; i < references.length; i++)
{
      references[i] = new Reference();

   //Get the bases
   temp = _Section2_Fields[i + 1].split("AUTHORS");
   references[i].bases = temp[0].trim();

   String _Reference = _Section2_Fields[i + 1].replaceAll("\\s+", " ");
      String temp1;
      String[] temp2;

      if (_Reference.indexOf("AUTHORS") != -1)
   {
          temp1 =
              _Reference.substring(
                  _Reference.indexOf("AUTHORS") + "AUTHORS".length(),
                  _Reference.length());
          temp2 = p.split(temp1);

          String _Authors_Field = temp2[0].trim();
          _Authors_Field = _Authors_Field.replaceAll("\\. and", ".,");
          _Authors_Field = _Authors_Field.replaceAll("III, ", "III., ");

          references[i].authors = _Authors_Field.split("\\., ");
          for (int j = 0; j < references[i].authors.length - 1; j++)
      {
              references[i].authors[j] = references[i].authors[j] + ".";
          }
      }
      if (_Reference.indexOf("TITLE") != -1)
   {
          temp1 =
              _Reference.substring(
                  _Reference.indexOf("TITLE") + "TITLE".length(),
                  _Reference.length());
          temp2 = p.split(temp1);
          references[i].title = temp2[0].trim();
      }
      if (_Reference.indexOf("JOURNAL") != -1)
   {
          temp1 =
              _Reference.substring(
                  _Reference.indexOf("JOURNAL") + "JOURNAL".length(),
                  _Reference.length());
          temp2 = p.split(temp1);
          references[i].journal = temp2[0].trim();
      }
```

```
      if (_Reference.indexOf("MEDLINE") != -1)
   {
          temp1 =
              _Reference.substring(
                  _Reference.indexOf("MEDLINE") + "MEDLINE".length(),
                  _Reference.length());
          temp2 = p.split(temp1);
          references[i].medline = temp2[0].trim();
      }
   if (_Reference.indexOf("PUBMED") != -1)
   {
      temp1 =
         _Reference.substring(
            _Reference.indexOf("PUBMED") + "PUBMED".length(),
            _Reference.length());
      temp2 = p.split(temp1);
      references[i].pubmed = temp2[0].trim();
   }
   if (_Reference.indexOf("REMARK") != -1)
   {
      temp1 =
         _Reference.substring(
            _Reference.indexOf("REMARK") + "REMARK".length(),
            _Reference.length());
      temp2 = p.split(temp1);
      references[i].remark = temp2[0].trim();
   }
  }

  //////////////////////////////////////////////////////////
  // Parse section 3 - ^(COMMENT|FEATURES) thru ^//
  /////////////////////////////////////////////////////////
  if (_Section3.indexOf("COMMENT") != -1)
{
      comment = new String(_Section3_Fields[1].trim());
      comment = comment.replaceAll("\\s+", " ");
      strTemp = new String(_Section3_Fields[2].trim());
      sequence.origin = new String(_Section3_Fields[3].trim());
  }
else
{
      strTemp = new String(_Section3_Fields[1].trim());
      sequence.origin = new String(_Section3_Fields[2].trim());
  }
  sequence.origin = sequence.origin.replaceAll("(\\s|[0-9])+", "");
//Get a count for the A's, T's, C's, and G's
for (int i=0; i<sequence.origin.length(); i++)
{

   switch (sequence.origin.charAt(i))
   {
      case 'a':
         ++sequence.a_count;
         break;
      case 'c':
         ++sequence.c_count;
         break;
      case 'g':
         ++sequence.g_count;
         break;
      case 't':
         ++sequence.t_count;
         break;
   }

}

//Seperate out the features
int i, j, k, l = 0;
String [] _Section4_Fields = null;
String _Section4 = strTemp;
String []strFeature = null;
StringBuffer strBuff = null;
```

```
String strDescription = "";
Vector featuresRaw = new Vector();
Vector containerRaw = new Vector();

_Section4_Fields = _Section4.split("\\n");

for (i=1; i<_Section4_Fields.length; i++)
{
   temp = _Section4_Fields[i].split("\\n");
   if (temp.length > 0)
   {
      for (j=0; j<temp.length; j++)
      {
         //This will trim off all the excess white space, making it possible
         //to exam the first charecter for each line
         strBuff = new StringBuffer(temp[j]);
         temp[j] = strBuff.substring(5);

         //look at the 1st charecter, to determine if it is a slash or not
         if (temp[j].charAt(0) == ' ')
         {
            //Loop through, looking for the /'s
            temp[j] = temp[j].trim();
            if (temp[j].charAt(0) == '/')
            {
               strFeature = temp[j].split("=");
               strDescription = strFeature[1];
               strFeature = strFeature[0].split("/");
               containerRaw.add(strFeature[1]);
               containerRaw.add(strDescription);
               ++l;
            }
            else
            {
               //Need to append this to the end of the description
               strDescription =
                        (String)containerRaw.remove(
                        containerRaw.size() - 1);
               strDescription += temp[j];
               containerRaw.add(strDescription);
            }
         }
         else
         {
            //if the vector is not empty, place the number of
            //elements gathered so it can be orginized later.
            if(!containerRaw.isEmpty())
            {
               featuresRaw.add(new Integer(l));
            }
            l = 0;
            //it is a keyword!
            //format: keyword w+ location
            strFeature = temp[j].split("\\s+");
            featuresRaw.add(strFeature[0]);
            featuresRaw.add(strFeature[1]);
         }
      }
   }
}
if (!containerRaw.isEmpty())
{
   featuresRaw.add(new Integer(l));
}

//Construct the datastructure to hold the features
//This will make inserting the values into the database easier
features = new Feature[featuresRaw.size() / 3];
j = 0;
for(i = 0; i < featuresRaw.size(); i++)
{
   switch (i % 3)
   {
```

```
        case 0:
           features[j] = new Feature();
           features[j].keyname = new String ((String) featuresRaw.elementAt(i));
           break;
        case 1:
           features[j].location = new String ((String) featuresRaw.elementAt(i));
           break;
        case 2:
               features[j].container = new Feature_Described[((Integer)
                   featuresRaw.elementAt(i)).intValue()];
           ++j;
           break;
     }
  }

  //Go through and put together each label and description
  j = 0;
  k = 0;
  for(i = 0; i < containerRaw.size(); i++)
  {
     //Reset k, and advance J
     if (features[j].container.length == k)
     {
        k = 0;
        ++j;
     }

     if (features[j].container.length > k)
     {
        if ((i % 2) == 0)
        {
           features[j].container[k] = new Feature_Described();
           features[j].container[k].label = new String
              ((String) containerRaw.elementAt(i));
        }
        else
        {
           features[j].container[k].description = new String
              ((String) containerRaw.elementAt(i));
           ++k;
        }
     }
     else
        --i;
  }
}

 /**
 * This holds the information that can be found
 * in the reference section of the GenBank entry.
 */
 public class Reference
{
    String[] authors;
    String title = "", journal = "", medline = "";
  String pubmed = "", remark = "";
  String bases = "";
 }

/**
 * This contains the information about the ORIGIN
 * field and the count of characters.
 */
public class SequenceInformation
{
  int a_count = 0, c_count = 0, g_count = 0, t_count = 0;
  String origin = "", sequenceLength = "";
}

/**
 * This will allow the features to be used,
 * and not just a long field with all the information
 * in a long string.
```

```
    */
   public class Feature
   {
      String location = "", keyname = "";
      Feature_Described container[] = null;
   }

   /**
    * This allows the descriptor's and descriptions
    * and is used within the Feature class
    * for the many features.
    */
   public class Feature_Described
   {
      String label = "", description = "";
   }

    Pattern _Pattern_Section1;
    Pattern _Pattern_Section2;
    Pattern _Pattern_Section1_Fields;
    Pattern _Pattern_Section2_Fields;
    Pattern _Pattern_Section3_Fields;

    String record = "";

    //information about LOCUS
    String locusName = "";
    String moleculeType = "";
    String genbankDivision = "";
    String modificationDate = "";
    String geneDirection = "";

    String definition = "";
    String[] accession = null;
    String version = "";
    String gi = "";
    String[] keywords = null;
    String commonName = "";
    String scientificName = "";
    String[] scientificFamilyTree = null;

    Reference[] references = null;
   Feature[] features = null;

    String comment = "";
   SequenceInformation sequence = null;
}
```

Appendix VI: Loader Source Code

```
import java.util.regex.*;
import java.sql.*;
import java.io.*;

/**
 *  This class provides the mechanism to
 *  send the message to the parser with the filenames,
 *  get the information, and insert each record from the parser.
 *  This will insert each entry individually, which is slower.
 *
 * This class was modified from MWING.
 * A class project from databases.
 */
public class OracleImportTool
{
   //This provides a way to report back any errors, and the times it took to
   //parse the file at hand.
   private FileOutputStream errorsfos;

   /**
    * This is what is called by java to run the program.
    * It requires at least one filename to be passed into
    * the program.
    */
   public static void main(String[]args)
    {
      if (args.length < 1)
        {
         System.err.println("Usage: java OracleImportTool file1 file2 ...");
        }
       else
        {
         //Set up a new instance of this class.
         OracleImportTool dbWriter = new OracleImportTool();
            for (int i=0; i < args.length; i++)
            {
                dbWriter.insertFile(args[i]);
           }
      }
    }

   /**
    * This constructor sets up the parser, retrieves the
    * database connection, and the file to write the log
    * into.
    */
   public OracleImportTool()
   {
      parser = new Parser();
        getConnection();
      counter = 0;
      try
      {
         errorsfos = new FileOutputStream("oracle_import_errors.txt",true);
        }
        catch (IOException e)
      {
            System.err.println(e.getMessage());
        }
    }

   /**
    * This will do the work of taking the file specified,
    * parsing the information, and inserting the information
    * into Oracle.
    * It also provides a time tracking method.
    */
    public void insertFile(String fileNamed)
   {
      failures = 0;
```

```
      fileName = fileNamed;
      //Get the time it takes to do the operation
      long startTime = System.currentTimeMillis();
      log("Initial time: "+ startTime);
    //create a listner for when the parser gets done parsing an entry,
    //Insert that entry into the database, and continue.
      parser.parseFile(fileName,
       new Parser.ParserListener()
          {
          public void callback(boolean isSuccess, String error)
              {
             if (isSuccess)
                 {
                insertRecord();
                //provide a way to see if it is still working, and how long
                //it took to insert 1,000 records at a time.
                if (counter >= COUNTERINT)
                {
                   log("GI: " + parser.gi);
                   counter = 0;
                }
                  }
                  else
                  {
                failures ++;
                      log(error);
                  }
              }
       }
      );
    //Log how long it took.
    log("End time - " + System.currentTimeMillis());
    if (counter > 0)
    {
       //Close the db connection, it messes w/oracle to have open connections
       try
       {
          rs.close();
          stmt.close();
          conn.close();
       }
       catch (SQLException e)
       {
          handleSqlException(e);
       }

    }

      long endTime = System.currentTimeMillis();
      long min = (endTime - startTime) / (1000 * 60);
      long sec = (endTime - startTime) / 1000 - 60 * min;

      String results = fileName + " - Finished Processing - "
          + min + " minutes " + sec + " seconds.";
      if (failures == 0)
    {
          results = results + " All records inserted successfully.";
      }
    else
    {
          results = results + " Some records failed to be inserted,"
              + " possibly due to inconsistencies in the sequence file"
              + " or bad queries.";
      }
      log(results);
  }

 /**
  * This is where the information from the parser is used.
  * It takes that information, and constructs the appropriate
  * SQL Queries to put in the information.
  * This part can take a while with all the information.
  */
```

```
private void insertRecord()
{
  //These all hold values that are used latter.
  //Mostly for tables that need an ID from one table for
  //another table.
    String _Keywords_ID = "";
    String _Source_ID = "";
    String _Ref_ID = "";
  String _Accession_ID = "";
  String strQueryTmp = "";
  String strTmp = "";
  String _Taxonomic_ID = "";
  int i, j;
  startTransaction();

  //Locus and Sources
  try
  {
     //Source information
     strQuery = "select sourceid from sources where SCIENTIFICNAME = "
        + quoted(parser.scientificName);
        rs = stmt.executeQuery(strQuery);
        if (rs.next())
            _Source_ID = rs.getString("SourceID");
        else
     {
        strQueryTmp = "insert into sources "
           + "(organism, scientificname) values ("
                + quoted(parser.commonName) + ", " + quoted(parser.scientificName)
                + ")";

            stmt.executeUpdate(strQueryTmp);
            rs = stmt.executeQuery(strQuery);
            if (rs.next())
           _Source_ID = rs.getString("SourceID");

     }
     rs.close();

     //Check to see if the GI # is in the table Locus
     strQuery = "select name from locus where gi = " + parser.gi;
     rs = stmt.executeQuery(strQuery);
     if (!rs.next())
     {
        //Locus Table
        strQuery = "INSERT INTO locus (GI, Name, Molecule, "
           + "Direction, GenBankDate, "
           + "GenBankDivision, Comments) VALUES ("
                + parser.gi + ", " + quoted(parser.locusName)
                + ", " + quoted(parser.moleculeType) + ", "
           + quoted(parser.geneDirection) + ", "
           + quoted(parser.modificationDate) + ", "
           + quoted(parser.genbankDivision) + ", "
           + quoted(parser.comment) + ")";
        stmt.executeUpdate(strQuery);
     }

     //Accession
     //This will see if the accession and version is in the database
     for (i=0; i<parser.accession.length; i++)
     {
        strQuery = "select accessionid from accession where accessionnbr like "
           + quoted(parser.accession[i]);
        rs = stmt.executeQuery(strQuery);
        if (rs.next())
           _Accession_ID = rs.getString("accessionid");
        else
        {
           rs.close();
           //Need to insert the accession information
           strQueryTmp = "insert into accession (AccessionNbr, Version) values ("
              + quoted(parser.accession[i]) + ", " + parser.version + ")";
           stmt.executeUpdate(strQueryTmp);
```

```
            rs = stmt.executeQuery(strQuery);
            if (rs.next())
               _Accession_ID = rs.getString("accessionid");
         }
         rs.close();

         //Now update the joiner table
         strQuery = "select LocusAccessionID from LocusAccession where "
            + "AccessionID = " + _Accession_ID + " and GI = " + parser.gi;
         rs = stmt.executeQuery(strQuery);
         if (!rs.next())
         {
            rs.close();
            //Need to enter in the data
            strQuery = "insert into LocusAccession (AccessionID, GI, Secondary) "
               + " values (" + _Accession_ID + ", " + parser.gi + ", " + i + ")";
            stmt.executeQuery(strQuery);
         }
         rs.close();
      }

      //Locus_Sources, and Taxonomic
      for (i=0; i < parser.scientificFamilyTree.length; i++)
      {
         //See if the taxonomic is in the table
         strQuery = "select taxonomicid from taxonomic where "
            + " taxonomy = " + quoted(parser.scientificFamilyTree[i]);
         rs = stmt.executeQuery(strQuery);
         if (rs.next())
            _Taxonomic_ID = rs.getString("taxonomicid");
         else
         {
            rs.close();
            //insert the information
            strQueryTmp = "insert into taxonomic (taxonomy) values ("
               + quoted(parser.scientificFamilyTree[i]) + ")";
            stmt.executeUpdate(strQueryTmp);
            //Get the value from the table
            rs = stmt.executeQuery(strQuery);
            if (rs.next())
               _Taxonomic_ID = rs.getString("taxonomicid");
         }
         rs.close();

         //See if for this GI, that it doesnt exist in the locus_sources table
         strQuery = "select GI from Locus_Sources where"
            + " SourceID = " + _Source_ID
            + " and TaxonomicID = " + _Taxonomic_ID
            + " and GI = " + parser.gi;
         rs = stmt.executeQuery(strQuery);
         if (!rs.next())
         {
            //Insert the record, with the current level it is up to
            strQueryTmp = "insert into Locus_Sources values ( "
               + parser.gi + ", " + _Source_ID + ", "
               + _Taxonomic_ID + ", " + i + ")";
            stmt.executeUpdate(strQueryTmp);
         }
      }

      //Keywords
      if (parser.keywords[0].length() > 0)
      {
         for (i = 0; i < parser.keywords.length; i++)
         {
            _Keywords_ID = "";
            //See if the keyword exists in the database
                 strQuery = "select keywordid from keywords where Keyword = "
                         + quoted(parser.keywords[i]);
                 rs = stmt.executeQuery(strQuery);
                 if (rs.next())
                     _Keywords_ID = rs.getString("KeywordID");
                 else
```

```
        {
           //insert the keyword into the the table.
           rs.close();
                 strQueryTmp = "insert into keywords (Keyword) values ("
              + quoted(parser.keywords[i]) + ")";
           stmt.executeUpdate(strQueryTmp);
                 rs = stmt.executeQuery(strQuery);
                 if (rs.next())
                     _Keywords_ID = rs.getString("KeywordID");
             }
        rs.close();

        //Does it exist in the joiner table
             strQuery = "select LocusKeywordID from LocusKeywords where"
           + " GI = " + parser.gi
           + " and KeywordID = " + _Keywords_ID;
             rs = stmt.executeQuery(strQuery);
        if (!rs.next())
        {
           //insert the value into the joiner table
           strQuery = "insert into LocusKeywords (KeywordID, GI, "
              + " OrderOfGenerality) values ("
              + _Keywords_ID + ", " + parser.gi
              + ", " + i + ")";
           stmt.executeUpdate(strQuery);
        }
        rs.close();
         }
  }

  //Definition
  //See if the definition exists
  strQuery = "select defid from definitions where "
     + "defined = " + quoted(parser.definition);
  rs = stmt.executeQuery(strQuery);
  if (!rs.next())
  {
     //insert the value, since it does not exist.
     strQueryTmp = "insert into definitions (GI, defined) values ("
        + parser.gi + ", " + quoted(parser.definition) +")";
     stmt.execute(strQueryTmp);
  }
  rs.close();

  //Reference's
  for(i=0; i<parser.references.length; i++)
  {
     //See if this record already exists
     strQuery = "select refid from reference where"
        + " referencebases = " + quoted(parser.references[i].bases)
        + " and journal like " + quoted(parser.references[i].journal);
     if (parser.references[i].title.length()>0)
        strQuery += " and title like " + quoted(parser.references[i].title);
     else
        strQuery += " and title is null";
     rs = stmt.executeQuery(strQuery);
     if (rs.next())
        _Ref_ID = rs.getString("refid");
     else
     {
        rs.close();
        //Set up the authors table first.
        j=0;
        for (j=0; j< parser.references[i].authors.length-1;++j)
        {
           strTmp = "Author(" + quoted(parser.references[i].authors[j])
              + "), ";
        }
        strTmp = "Author (" + quoted(parser.references[i].authors[j])
           + ")";
        //Now the rest of the record
        strQueryTmp = "insert into reference (referencebases, title, journal, “
               + “medline, "
```

```
            + "pubmed, comments, authorstbl) values ("
            + quoted(parser.references[i].bases);

         if (parser.references[i].title.length() > 0)
            strQueryTmp += ", " + quoted(parser.references[i].title);
         else
            strQueryTmp += ", null";

         strQueryTmp += ", " + quoted(parser.references[i].journal);
         if (parser.references[i].medline.length() > 0)
            strQueryTmp += ", " + quoted(parser.references[i].medline);
         else
            strQueryTmp += ", null";
         if (parser.references[i].pubmed.length() > 0)
            strQueryTmp += ", " + quoted(parser.references[i].pubmed);
         else
            strQueryTmp += ", null";
         if (parser.references[i].remark.length() > 0)
            strQueryTmp += ", " + quoted(parser.references[i].remark);
         else
            strQueryTmp += ", null";

         strQueryTmp += ", ReferenceAuthors("
            + strTmp + "))";
         stmt.executeUpdate(strQueryTmp);
         rs = stmt.executeQuery(strQuery);
         if (rs.next())
            _Ref_ID = rs.getString("refid");
      }
      rs.close();
      //Now add that reference to the joiner table
      strQuery = "select refid from locusreference where "
         + " gi = " + parser.gi + " and refid = "
         + _Ref_ID;
      rs = stmt.executeQuery(strQuery);
      if (!rs.next())
      {
         strQueryTmp = "insert into locusreference values ("
            + parser.gi + ", " + _Ref_ID + ", " + i + ")";
         stmt.executeUpdate(strQueryTmp);
      }
   }

   //Sequence
   strQuery = "select seqid from sequences where gi = "
      + parser.gi;
   rs = stmt.executeQuery(strQuery);
   if (!rs.next())
   {
      //Insert the value into the table.
      strQueryTmp = "insert into sequences (gi, a_count, c_count,"
         + " g_count, t_count, ActualSequence, SeqLength) values ("
         + parser.gi + ", "
         + parser.sequence.a_count + ", "
         + parser.sequence.c_count + ", "
         + parser.sequence.g_count + ", "
         + parser.sequence.t_count + ", "
         + quoted(parser.sequence.origin) + ", "
         + parser.sequence.sequenceLength + ")";
      stmt.executeUpdate(strQueryTmp);
   }
   rs.close();

   //Features
   for (i = 0; i < parser.features.length; i++)
   {
      //See if the features are already there
      strQuery = "select featureid from features where "
         + "keyname = " + quoted(parser.features[i].keyname)
         + " and location = " + quoted(parser.features[i].location)
         + " and gi = " + parser.gi;
      rs = stmt.executeQuery(strQuery);
      if (!rs.next())
```

```
        {
           //need to insert the information
           strQuery = "insert into features (gi, keyname, location, "
              + "ordinal, featurestbl) values (" + parser.gi
              + ", " + quoted(parser.features[i].keyname)
              + ", " + quoted(parser.features[i].location)
              + ", " + i + ", ";
           //set up the nested features table
           strQueryTmp = "tbl_Features_Described(";
           for (j = 0; j < parser.features[i].container.length - 1; j++)
           {
              strQueryTmp += "Feature_Described("
                 + quoted(parser.features[i].container[j].label)
                 + ", "
                 + quoted(parser.features[i].container[j].description)
                 + "), ";
           }
           if (parser.features[i].container.length > 0)
           {
              strQueryTmp += "Feature_Described("
                 + quoted(parser.features[i].container[j].label)
                 + ", "
                 + quoted(parser.features[i].container[j].description)
                 + ")))";
           }
           else
           {
              strQueryTmp = "null)";
           }
           strQuery += strQueryTmp;
           stmt.executeUpdate(strQuery);
        }
        rs.close();
     }
  }
  catch (SQLException e)
  {
     handleSqlException(e);
  }

  //Make sure database knows that transaction is done.
  completeTransaction();
  ++counter;
 }

/**
 * This will take a string and put quotes around it.
 * Simplifies writing queries with strings in it, and
 * needing quotes around them.
 * e.g. Char String is turned into 'Char String'
 */
public String quoted (String strQuote)
{
  return "'" + strQuote + "'";
}

/* This provides a uniform error handling
 * if anything throws an error.  It will write
 * to the log what the error is, and report
 * what part of it failed.
 */
 private void handleSqlException(SQLException e)
{
  String exMessage = e.getMessage();
    String exNotify = "";
    // Many fields, such as Source, will be duplicates
    // and throw a SQLException when re-inserting.
    // This gets ignored here.
    if (exMessage.indexOf("Duplicate") != -1)
  {
        return;
    }
  else
```

```
    {
       // Handle queries exceeding the maximum size limit
          if (exMessage.indexOf("PacketTooBigException") != -1)
       {
          // don't log the actual query because it's huge
              log("File: " + fileName + "GI: " + parser.gi
             + " - SQLException - "
                  + "PacketTooBigException - "
                  + "Query Length: " + strQuery.length());

              // reconnect to database
              getConnection();
          }
       else
       {
              log("File: " + fileName + "\nGI: " + parser.gi + "\nCounter: " + counter
                      + "\nBad Query: " + strQuery + "     "
                      + e.toString());
          }
          this.isTransactionSuccess = false;
          ++failures;
       completeTransaction();
      }
 }

/* This provides a way to get the connection
 * to the database.  It can be re-written
 * to have the connection information in here.
 * This way is using a class outside this java
 * source code to create the connection, and
 * returns it.
 */
 private void getConnection()
 {
    try
    {
       //Done by an external class
       conn = OracleConnection.getConnection();
       stmt = conn.createStatement();
      }
    catch (SQLException e)
    {
          e.printStackTrace();
      }
      if (conn == null)
    {
          System.err.println("Could not connect to database");
          System.exit(-1);
      }
 }

 /*   This provides a way to do batch updates,
 * which is more efficient, by saying there
 * will be some more.  If there is an error
 * the transaction can be rolled back, not
 * having any information in the database.
 */
private void startTransaction()
 {
    isTransactionSuccess = false;
    try
    {
       conn.setAutoCommit(false);
       isTransactionSuccess = true;
    }
    catch (SQLException e)
    {
       log(fileName + " - Failed to start transaction "
          + e.toString());
    }
 }

/* This will commit the statements that
```

```
     * are available, if it was a success,
     * and rollback the statements if it
     * there was an error.
     */
     private void completeTransaction()
     {
       try
       {
            if (isTransactionSuccess)
          {
             conn.commit();
            }
          else
          {
             conn.rollback();
           }
         }
       catch (SQLException e)
       {
             log(fileName + " - Failed to complete transaction "
                     + e.toString());
         }
     }

    /**
     * This will write the message to the file
     * and also to the screen.  It logs it into
     * the file created in the constructor.
     */
     public void log(String msg)
     {
         System.err.println(msg);
         try
       {
             errorsfos.write((msg + "\n").getBytes());
         }
         catch (IOException e)
       {
             System.err.println(e.getMessage());
         }
     }

     private boolean isTransactionSuccess;
     private int failures;  //a count of the number of failures
     private Parser parser; //A parser object
     private Connection conn = null;//A connection to Oracle
     private Statement stmt;   //The statements that are used.
     private ResultSet rs;  //Result sets from the statements
     private String strQuery;//query string for statements
    private int counter; //keep track of number done.
    private String fileName;
    private static int COUNTERINT = 1;
}
```

Appendix VII: Connection Source Code

```
/*
 ** +-----------------------------------------------+
 ** | FILE        : OracleConnection.java            |
 ** | AUTHOR      : Jeff Hunter                      |
 ** | DATE        : 06-DEC-2001                      |
 ** +-----------------------------------------------+
*/

import java.sql.*;
import java.util.*;
import oracle.jdbc.driver.*;

public class OracleConnection
{
   /*
    ** +---------------------------------------+
    ** | Return a JDBC connection appropiately  |
    ** | either inside or outside the database. |
    ** +---------------------------------------+
    */
   public static Connection getConnection() throws SQLException
   {
      String   username = "your user name";
      String   password = "your password";
      String   driver_class = "oracle.jdbc.driver.OracleDriver";

      // +-----------------------------------------------+
      // | You can use either of the two URLs to obtain a  |
      // | default connection.                             |
      // |   "jdbc:default:connection:"                    |
      // |   "jdbc:oracle:kprb:"                           |
      // +-----------------------------------------------+
      String   defaultConn  = "jdbc:default:connection:";
      String   thinConn     = "jdbc:oracle:thin:@kybrinwb.spd.louisville.edu:1521:bio";
      Connection conn = null;
      try
      {
         Class.forName (driver_class).newInstance();
         if (connectedToDatabase())
         {
            conn = DriverManager.getConnection(defaultConn);
         }
         else
         {
            conn = DriverManager.getConnection(thinConn, username, password);
         }
         conn.setAutoCommit(true);
         return conn;
      }
      catch (Exception e)
      {
         throw new SQLException("Error loading JDBC Driver");
      }
   } // METHOD: (getConnection)

   /*
    ** +---------------------------------------------------------+
    ** | Method to determine if we are running in the database. |
    ** | If oracle.server.version is not null, we are running   |
    ** | in the database.                                        |
    ** +---------------------------------------------------------+
    */
   public static boolean connectedToDatabase()
   {
      String version = System.getProperty("oracle.server.version");
      return (version != null && !version.equals(""));
   } // METHOD: (connectedToDatabase)
} // CLASS: (OracleConnection)
```

Appendix VIII: Bash Script to Populate Data

```
BIODIR=/biodata/iecha001/estSeq/humanest
HOMEDIR=~
TEMPFILE=ndm.tmp

cd $BIODIR
for i in `ls -1 *.seq.gz`
do
  echo "`date` $i"
  gunzip -c $i > ~/$TEMPFILE

  cd $HOMEDIR
  java OracleImportTool $TEMPFILE

  cd $BIODIR
done
```

The Linux bash profile required to run the script look likes this:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$HOME/j2sdk1.4.2_06/bin:$PATH:$HOME/bin
CLASSPATH=$CLASSPATH:.:/~

export PATH
unset USERNAME
```

Vita

Nathan Mann was born and raised in Louisville, Kentucky. While a youth, he participated in the Boy Scouts of America, and earned his Eagle Scout award. One of the many merit badges he did earn was the Computers. With the Boy Scout heritage and his dad's influence Nathan knew he would go into the computer field.

Nathan went to Rose-Hulman Institute of Technology in 1998 and was there the first year and a half of college. Then he transferred to the University of Louisville. He received his Bachelors of Computer Engineering and Computer Science in the fall of 2003 and went directly into the Masters program. In the fall of 2004 Nathan is completing the work for the thesis and to graduate with the Masters of Engineering of Computer Engineering and Computer Science.

While Nathan has been at both Rose-Hulman and U of L, he participated in some level of band, the Association of Computing Machinery (ACM), and Alpha Phi Omega (APO). Band was an outlet of an artistic side that came out occasionally while playing percussion instruments. In ACM Nathan server as vice-president and has been an active member. Alpha Phi Omega is a co-ed service fraternity based on service.

Nathan plans on staying in Louisville and getting married and working the Computer Engineering field. More specifically, Nathan is looking to keep to the Software Engineering side of the degree, and is looking to stay with Industrial Services of America doing in-house software engineering for their business.