Protein Geometry Database Documentation

Release 1.0.2

Oregon State University Open Source Lab

July 15, 2015

Contents

Contents:

CHAPTER 1

Installation

This is a manual installation guide for the Protein Geometry Database (PGD).

1.1 Install dependencies

Certain operating-system packages are required to run the PGD. As an example, here are the additional packages required for a Centos 7 server with EPEL:

- bzip2
- cairo-devel
- gcc
- gcc-c++
- libffi
- libffi-devel
- mysql
- mysql-devel
- nodejs
- npm
- python-devel
- python-setuptools
- tar

Also required is `dsspcmbi`, compiled from the DSSP software.

Python 2.7 or greater is required, but Python 3.x is not currently supported.

1.2 Get the Code

1. Make sure you have Git installed.
2. Check it out from the repository:

```
git clone https://github.com/osuosl/pgd
```

1.3 Configuration

The easiest way to spin up an instance of the PGD for testing purposes is with Docker. If Docker is unavailable, an instance of the PGD can be spun up locally following these instructions.

1. Construct a `settings.ini` file in the top level of the application. This file must contain values for `SECRET_KEY`, with other values optional. Here is an example:

```
[settings]
MYSQL_ENV_MYSQL_DATABASE=pgd
MYSQL_ENV_MYSQL_USER=pgd
MYSQL_ENV_MYSQL_PASSWORD=kjwb_if4hgkujpb3*7(_8
MYSQL_PORT_3306_TCP_ADDR=mysql.example.org
MYSQL_PORT_3306_TCP_PORT=3306
GOOGLE_ID=UA-8675309-1
SECRET_KEY=2nWoBbgLb1bVbOzM0PaW/q0jScKKcP5j2nWoBbgLb1bVbOzM0P
MEDIA_ROOT=/opt/django/pgd/media
STATIC_ROOT=/opt/django/pgd/static
EMAIL_HOST=smtp.example.org
DEFAULT_FROM_EMAIL=registration@pgd.example.org
SERVER_EMAIL=pgd@pgd.example.org
```

2. Initialize the database:

```
$ python manage.py syncdb
```

3. Collect the static files:

```
$ python manage.py collectstatic
```

4. Now the server can be run:

```
$ python manage.py runserver
```

See the importing data section for instructions on how to import data.

More information about deploying with WSGI can be found here.

CHAPTER 2

Using PGD with Docker

PGD ships with a Dockerfile to make development easier. Consult the docker documentation for instructions on how to use docker. http://docs.docker.com/reference/

2.1 Quick Start: Demonstrating the PGD with Docker Compose and Docker

This is for folks who are already familiar with Docker Compose and Docker. If you are new to either of these tools, please skip ahead to the next section of the documentation.

There are three steps to this process: building the containers, populating them with content, and starting the web server.

2.1.1 Build the containers

Note: The repository contains a script named *dev-setup.sh* which builds the containers following the same instructions found in this script. Use at your own risk as the script may not be updated as often as the documentation. When in doubt, trust the docs!

This is pretty straightforward.

```
$ docker-compose build
```

The database containers (for MySQL and PDB files) need to be brought up next.

```
$ docker-compose up -d mysql pdb
```

Now create the necessary database tables. For this version of Django, syncdb is still required. The PGD does not use the admin site at this time, so there's no need to create an account.

```
$ docker-compose run web python manage.py syncdb --noinput
```

This command may fail the first time with a lack of connection due to docker-compose's not-yet-mature orchestration functionality. Simply run it again and it should succeed.

2.1.2 Install the content

Note: The repository contains a script named *integration_test.sh* which tests the scripts mentioned in this section.

Use at your own risk as the script may not be updated as often as the documentation. Again, when in doubt, trust the docs!

The next step is to create a selection file. It's possible to use a subset of an existing selection file (say the first hundred lines) but if you need to generate a new one, use this command:

```
$ docker-compose run web python ./pgd_splicer/dunbrack_selector.py --pipeout > selection.txt
$ sed 100q selection.txt > top-100-selection.txt
```

The selected proteins must be retrieved from the worldwide PDB collection. This command may take some time!

```
$ docker-compose run web python ./pgd_splicer/ftpupdate.py --pipein < top-100-selection.txt
```

To list the proteins that were successfully downloaded, run this command:

```
$ docker-compose run web ls /opt/pgd/pdb
```

You should see 100 files with names like *pdb1ae1.ent.gz*.

Finally, all the retrieved proteins must be imported into the database. This command will definitely take some time: a full update currently consists of over twenty-six thousand proteins, and can take upwards of eight hours to process.

```
$ docker-compose run web python ./pgd_splicer/ProcessPDBTask.py --pipein < top-100-selection.txt
```

To confirm the number of proteins in the database, use the Django shell:

```
$ docker-compose run web python manage.py shell
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from pgd_core.models import Protein
>>> Protein.objects.count()
100
>>>
```

2.1.3 Start it up!

Looking good! Now it's time to actually start the web server.

```
$ docker-compose up
```

This will generate a screen or two of output from the different containers. Once that output stabilizes, open a web browser to *http://localhost:8000* (or a different host, depending on where you're running Docker) and you should see the PGD! Select 'Search', remove the default search constraints on omega from the search page, and select 'Submit', and you should see a Ramachandran plot with results. Success!

2.2 Using Docker Compose

Docker Compose is a command line tool to automate using multiple docker containers. Usually multiple long incantations of docker commands are necessary to get a working development environment. Using docker-compose, a simple instance of the PGD without any content can be started from scratch with a simple command:

```
$ docker-compose up
```

The application will be available on `http://localhost:8000`.

Note: Docker-Compose is a developer tool, and this early version is prone to certain kinds of race conditions. It is possible for the web container to come up before the database container, and if the web container can't find the database it will fail.

Similarly, to run all the tests in the PGD code base, the following command can be very useful:

```
$ docker-compose run web python manage.py test
```

Consult the docker-compose documentation for details on how to modify the *docker-compose.yml* file, and other commands you can use with docker-compose.

The following sections will not be necessary if you use docker-compose.

2.3 Building an Image

To build an image with PGD installed, run this command:

```
$ docker build -t osl_test/pgd .
```

The *-t* option specifies the tag for the image. We use *osl_test* here for testing.

2.4 Running a MySQL Container

PGD relies on a MySQL database. We use the default *mysql* image. Docker will fetch the *mysql* image automatically. The *-e* option passes an environment variable to the image. In this example we set a series of necessary environment variables to a simple default. The *–name* option gives this new container a name so it is easier to remember and reference when using the docker command.

```
$ docker run --name pgd_mysql \
 -e MYSQL_ROOT_PASSWORD=pgd_root_password \
 -e MYSQL_USER=pgd_user \
 -e MYSQL_PASSWORD=pgd_user_password \
 -e MYSQL_DATABASE=pgd_db \
 -d mysql
```

2.5 Running an Image and Linking it

Once the MySQL container is running, we can run the PGD container we built and link it with MySQL. Linking it means that the pgd container will be able to transparently access it. We will also forward the container's port 8000 to the host's port 8000.

```
$ docker run -d --name pgd -p 8000:8000 --link pgd_mysql:mysql osl_test/pgd
```

This should result in an instance of the PGD running on localhost at port 8000.

2.6 Mounting the PGD Code as a Volume

Some developers may find the following to be convenient:

```
$ docker run -d --name pgd \
-p 8000:8000 \
-v /path/to/code:/opt/pgd \
--link pgd_mysql:mysql \
osl_test/pgd
```

Be warned: this may clash with the Dockerfile's treatment of *settings.py* depending on whether one already exists in the checkout.

CHAPTER 3

Importing Data

3.1 Running Splicer From The Command Line

Splicer can be run from the command line. It requires that several steps be run separately.

All commands should be run from the project root (directory with settings.py in it).

3.1.1 Selecting Proteins

Proteins must first be selected. Default filtering settings will be used for threshold, resolution and r_factor.

```
1 ./pgd_splicer/dunbrack_selector.py
```

This will return information about the the parameters used, the files proteins were selected from, and a list of proteins in the following format:

```
code chains threshold resolution rfactor rfree
```

Save the selection into a file:

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selection.txt
```

^^^^^^- Options ^^^^^^-

- –pipeout - will only output the data. This should be used if you would like to create output suitable for input into one of the other steps

3.1.2 Downloading PDB Files

PDB files are downloaded from an FTP site using **ftpupdate.py**. This script will synchronize **./pdb** with the remote FTP server. Only new files will be downloaded, but it will check the timestamps on all files.

This is a time-consuming step. Be prepared to wait for approximately two days for this to complete on a fresh local copy, or one day on an update.

```
1 ./pgd_splicer/ftpupdate.py code [code...]
```

To only grab the proteins which are selected (and cut down massively on consumed bandwidth and time), try:

```
1./pgd_splicer/ftpupdate.py --pipein < selection.txt
```

3.1.3 Processing PDB Files

PDB files can be imported into the database with **ProcessPDBTask.py**. Multiple proteins can be fed as commands to be imported. Errors will be written to **ProcessPDB.log**.

```
1 ./pgd_splicer/ProcessPDBTask.py code chains threshold resolution rfactor rfree [repeat]
```

As before, a selection can be piped in:

```
1 ./pgd_splicer/ProcessPDBTask.py --pipein < selection.txt
```

Expect this to take a few days as well.

3.1.4 Parameters

Parameters are all required, and may be repeated for multiple proteins.

- **code** - protein code to import, should be all uppercase
- **chains** - list of chains to import, should be a string of chain ids. (ie. ABCDEF). The string should not have quotes around it.
- **threshold, resolution, rfactor, rfree** - the value for these fields. These properties are retrieved from the selection script so they are included as input for processing the protein.

3.1.5 Options

- –pipein - input will be read from a pipe instead of arguments. proteins in the list should be separated by newlines.

3.2 Example

Some examples. Intermediate output is saved to a text file so that it can be examined later.

3.2.1 Full Import

Update all proteins regardless of whether the file was downloaded by **ftpupdate**. *ProcessPDBTask* will still check the update date and exclude pdbs that are not new.

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2 ./pgd_splicer/ftpupdate.py --pipein < selected_proteins.txt
3 ./pgd_splicer/ProcessPDBTask.py --pipein < selected_proteins.txt
```

3.2.2 Update Only New

Update only proteins for which we have a new FTP file.

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2 ./pgd_splicer/ftpupdate.py --pipein --pipeout < selected_proteins.txt > updated_proteins.txt
3 ./pgd_splicer/ProcessPDBTask.py --pipein < updated_proteins.txt
```

3.2.3 Update Skipping Download

If all the pdb files are already downloaded you may skip the FTP step to save time.

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2 ./pgd_splicer/ProcessPDBTask.py --pipein < selected_proteins.txt
```

or as a single command:

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout | ./pgd_splicer/ProcessPDBTask.py --pipein
```

CHAPTER 4

Updating Protein Database

The PGD database is updated approximately four times per year. The following process should be followed to update the database.

One week before update: generate new selection file and download majority of updates.

From the pgd-staging Django site:

```
1 > python manage.py fetch --report=date-report.txt --selection=date-selection.txt
```

One day before update: reset staging database and load new selection file.

From the pgd-staging Django site:

```
1 > python manage.py shell


1 > from pgd_core.models import Protein
2 > for p in Protein.objects.all():
3 >   p.delete()

1 > ./pgd_splicer/ProcessPDBTask.py --pipein < date-selection.txt
```

On the update day: do the update!

From the pgd-staging Django site:

```
1 > ./pgd_splicer/dunbrack_selector.py --pipeout  > selection.txt
2 > ./pgd_splicer/ftpupdate.py --pipein < selection.txt
3 > ./pgd_splicer/ProcessPDBTask.py --pipein < selection.txt
```

That last command should be run within script so the output can be examined for particular failure modes once the command is complete. Error messages and recommended actions:

- "CRC check failed", "local variable 'i' referenced before assignment", "KeyError": collect all codes with these message, delete the files associated with these codes, retrieve them again from the server, and attempt to import the files again. Example command lines for codes 1B12 and 4AMW:

```
1 > (cd ./pdb && for code in 1b12 4amw; do rm pdb$code.ent.gz; done)
2 > egrep \(1B12\|4AMW\) selection.txt | ./pgd_splicer/ftpupdate.py --pipein
3 > egrep \(1B12\|4AMW\) selection.txt | ./pgd_splicer/ProcessPDBTask.py --pipein
```

If the errors persist, document offending codes in bug reports as appropriate and copy the retrieved files aside for testing and comparison. * "Structure/DSSP mismatch", "No chains were parsed!": document offending codes in update post to PGD mailing list.

Cross-check database against selection file.

From the pgd-staging Django site:

```
1 > python manage.py crosscheck --selection=selection.txt
```

The staging site is now ready for customer preview.

Promote database from staging to production.

From the pgd-prod Django site:

```
1 > ./update-from-staging.sh
```

Rename the selection file for archival purposes.

```
1 > mv selection.txt 201310-selection.txt
```

On production, in settings.py, update the DATA_VERSION to the date used for the selection file.

CHAPTER 5

Site Specific Information

5.1 Virtual environments

The virtual environments for PGD on zeus are located under the PGD user's home directory.

- pgd-staging: /home/pgd/.virtualenvs/pgd-staging/
- pgd-prod: /var/lib/django/pgd-prod/pgd (will move to /home/pgd/.virtualenvs/pgd-prod/ ASAP)

In addition to the traditional contents of a virtual environment as directed by the requirements file, the bin directory in the virtual environment also contains the binary:

```
dsspcmbi
```

which is used by ProcessPDBTask.py to import data into the database.

5.2 Files generated during the update process

Text files pertaining to previous updates (selection files, reports, crosschecks, etc.) should be moved to the following directory after the update is complete:

```
/home/pgd/update-records
```

These files should follow a naming format like this before moving:

```
20131231-selection.txt
```

Compressed SQL files generated by the update tool should be renamed in the same manner after the update is complete, but they are too large to be moved at this time.

5.3 Differences between staging and production

The staging directory has two directories which are not found in the production directory. These directories hold the files downloaded from the WWPDB and CMBI which are imported into the staging database. Their names are:

```
pdb
dssp
```

They may move, possibly to the home directory, at a later date.

The production directory has a number of files which are not present in the staging directory and are also not in source control.

- Historical copies of the staging and production databases for the pgd_core app.
- There are large compressed SQL files which represent the state of the staging and production databases for the pgd_core app in the production directory. These are used during the update process, and can be used for post-update analysis. Any files of this type older than one year can be deleted.
- lib64
- It is not clear to me what this directory is for – it may be part of the virtualenv.
- searchConvert.py
- This file was used to convert certain data structures from version 0.9.2 to 0.9.3 and can probably be deleted.
- static/pdf/2011_tronrud-shelxl.pdf
- This is a local copy of the paper which is cited as a source when referencing PGD in publications. There are five other papers in that directory which are in the repository, so this one should probably be added.
- update-from-staging.sh
- This script is executed to update the production database from the staging database.

In addition to these differences, the settings.py for staging and production is different with regard to database configuration and version settings.

CHAPTER 6

Technologies Used By Protein Geometry Database

6.1 Django

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Django provides some of the following features that are useful to PGD:

- Object Relational Mapping (ORM) - maps python classes to SQL tables, including a comprehensive query engine.
- Form API for validation of user input
- Templating system for layout of presentation

Read more at: djangoproject.com

6.2 jQuery

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. jQuery is used extensively in the front end to provide a "web 2.0" experience with dynamically updating pages.

Read more at: jquery.com

6.3 Raphael

Raphaël is a small JavaScript library that should simplify your work with vector graphics on the web. Raphael is cross browser supported and is used to render graphs within PGD.

Read more at: raphaeljs.com

6.4 Mysql

Mysql is an opensource database

CHAPTER 7

Terminology (WIP)

Protein *

Sidechain *

Residue

- **Each residue has many attributes**
  - **Conformation Angles (Angles of Rotation)**
    - phi, psi, ome (omega), omep (omega prime)
  - **Bond Angles**
    - a1, a2, a3, a4, a5, a6, a7
  - **Bond Lengths**
    - L1, L2, L3, L4, L5
  - **Secondary Structure**
    - ss
  - **Chi Angles, properties vary per Residue**
    - chi1, chi2, chi3, chi4, chi5
  - bm
  - bs
  - bg
  - h_bond_energy
  - zeta
  - terminal_flag
  - xpr

Plots

- In the plot each hit is sorted into a box based on the values of its attributes listed as “X Axis” and “Y Axis”. Any attribute can be chosen as X or Y. The “Plotted Attribute” of all the hits that end up in the same box are averaged (and the standard deviation is calculated). If no hits fall in a box there is no meaningful average so the box is left blank. The occupied boxes are colored based on the average value in that box.

- The exception is the default, when the “Plotted Attribute” is “Observations”. Then the color is simply calculated from the number of hits that end up in the box, with zero being black. Regardless of the “Plotted Attribute” the same boxes should always be black.

Search

- Returns a set of residues
- When the “Plotted Attribute” is “Observations” the color is simply calculated from the number of hits that end up in the box, with zero being black. Regardless of the “Plotted Attribute” the same boxes should always be black.

CHAPTER 8

Code Modules

The Protein Geometry Database makes use of Django "apps". Apps are synonymous with a module or plugin. They are module bits of an application that are, for the most part, portable between other apps.

PGD is divided into three apps to allow code to be portable.

8.1 PGD Core

PGD Core defines the core data structures. There is no functionality contained within this app. It is intended to contain only the database so that it can be reused in other applications

8.2 PGD Search

PGD Search contains logic for searching and displaying data from PGD Core. This includes search logic, models, and the web front end.

8.3 PGD Splicer

PGD Splicer contains all of the code required for importing data from PDB files into objects defined within PGD Core

CHAPTER 9

Protein Data Models

Protein data model is composed of three classes: **Protein, Chain,** and **Residue**. The models are designed to represent the protein in the most compact way. With proper SQL indexes this is also the most efficient method of storing the data for search queries when the database contains greater than 2 million residues.

The models are defined as Django models meaning the data exists as both Python classes and SQL Tables. This allows the data to be accessed through SQL, or preferably as Python objects through the Django Query API. Django models can be used from any program provided your *Django environment is configured*

For more information on Django models:

- Django Model Reference
- Django Query Reference

9.1 Protein

Represents a Protein.

- **code** - 4 letter code for protein as imported from pdb.
- **rfactor**
- **rfree**
- **resolution**
- **threshold**
- **residues** - related field collection, returns queryset of all residues related to this protein
- **chains** - related field colleciton, returns queryset of all chain related to this protein

9.2 Chain

Chain id's are stored to allow importing of multiple chains from a single protein. The search interface does not currently include selection of chains.

- **protein** (protein_id) Foreign Key relation to the protein this residue belongs to. May be retrieved as a Protein object or identifier
- **residues** - related field collection, returns queryset of all residues related to this protein

9.3 Residue

Represents a Residue (Amino Acid) belonging to a protein

- **id** - unique identifier for residue, internal to database
- **oldID** - identifier as listed in PDB file. May include icodes appended to the end.
- **chainIndex** - numerical index of residue in the chain. All chain breaks are represented as a single gap in numbers. IE. 1,2,4,5.
- **protein** (protein_id) Foreign Key relation to the protein this residue belongs to. May be retrieved as a Protein object or identifier
- **prev** (prev_id) - Foreign Key relation to the previous residue in the chain, this may be retrieved as a Residue object or identifier
- **next** (next_id) - Foreign Key relation to the next residue in the chain, this may be retrieved as a Residue object or identifier
- **Bond Lengths**
  - L1 -
  - L2 -
  - L3 -
  - L4 -
  - L5 -
- **Bond Angles**
  - A1 -
  - A2 -
  - A3 -
  - A4 -
  - A5 -
  - A6 -
  - A7 -
- **Dihedral Angles**
  - Omega
  - Phi
  - Psi
  - Zeta
- **Sidechain**
  - **X1 through X4** as defined by mmlib.
- **B-Factor**
  - **Bm** - Average b-factor of mainchain atoms
  - **Bs** - Average b-factor of sidechain
  - **Bg** - Average b-factor of Cg atom if present

CHAPTER 10

Optimization

PGD has been optimized using the following techniques

- SQL Indexes
- SQL Aggregate Functions
- In Memory Tables
- Denormalized Search Table

CHAPTER 11

Optimization: SQL Indexes

Like any database PGD relies on SQL Indexes for improved performance. This is a description of indexes used, and some that didn't work.

Please see the section on Memory Table Indexes for more information about the types of indexes used with memory tables.

11.1 Protein

11.1.1 Primary Key Index

The primary key index is used when specific proteins have been selected by Code (primary key)

11.1.2 Resolution Index

The index on resolution is used in most cases. It filters large sets of proteins. The default query, with resolution <= 1.2 reduces the number of proteins from 16,000 to 2500.

As the number of proteins nears the total number of proteins MySQL will switch to performing a full table scan. Even with indexes on other fields it does not appear to use them.

11.1.3 Failed Indexes

We also attempted to create indexes with resolution and other fields. No noticeable increase was detected, MySQL always opted for the individual Resolution Index.

11.1.4 Protein Joined to Residue

When joining a Residue with its Protein an index on Residue.protein_id is used

11.1.5 Failed Indexes

We attenmpted to add additional fields to the protein_id index. It was actually slower than the protein_id index alone.

11.2 Residue Joined to Residue

Residues are joined to Residues for the previous and next relationships using the Primary Key index on Residue.

11.2.1 Note on Join Direction for Previous and Next

Residues join from **residue_0.next** to **residue_1.id**

```
SELECT * FROM pgd_core_residue r0 INNER JOIN pgd_core_residue r1 ON (r0.next = r1.id)
```

instead of **residue_0.id** to **residue_1.prev**

```
SELECT * FROM pgd_core_residue r0 INNER JOIN pgd_core_residue r1 ON (r0.next = r1.id)
```

The latter appeared to be a faster query but is not possible with Django. The custom clause requires adding the where clause with queryset.extra(). But django will also add the original clause.

CHAPTER 12

SQL Aggregates

SQL aggregates are functions that run on the server. They can perform statistics such as **Count, Average, Standard Deviation, Min,** and **Max**. Aggregate functions can also be paired with **GROUP BY** to calculate statistics for different groupings of data.

Django supports aggregate functions as of 1.1. Read more about it here: Django Aggregates

```
SELECT COUNT(L1), MIN(L1), MAX(L1), AVG(L1), STDDEV(L1) FROM pgd_core_residue GROUP BY aa;
```

Aggregate functions increase the speed of calculations because they are run on the data in place. Transferring data between the database server and application server requires significant overhead.

12.1 Statistics for Dihedral Angles

Dihedral angles require special function for average and standard deviation. The special function takes into account that the angles may wrap around from 180 to -180. Both of these functions work like any other aggregate functions. They have also been wrapped in a custom Django Aggregate function to work with django querysets.

12.2 Average

::

```
IF(DEGREES(ATAN2(
        -AVG(SIN(RADIANS(ome))) ,-AVG(COS(RADIANS(ome))))
    ) < 0 ,DEGREES(ATAN2(
        -AVG(SIN(RADIANS(ome))) ,-AVG(COS(RADIANS(ome))))
    ) + 180 ,DEGREES(ATAN2(
        -AVG(SIN(RADIANS(ome))) ,-AVG(COS(RADIANS(ome))))
    ) - 180
) AS ome_avg
```

This works by converting the value into vectors. It adjusts the angles by +180 or -180 depending on whether it is a positive or negative angle. This shifts the vectors into the same space so that they may be averaged.

12.3 Standard Deviation

::

SQRT(

IF (((ome+360)%360 - avgs.ome_avg) < 180 ,SUM(POW((ome+360)%360-avgs.ome_avg, 2))
,SUM(POW(360-((ome+360)%360-avgs.ome_avg),2))

)/(COUNT(ome)-1)) AS OME_STDDEV

This function is very similar to a normal standard deviation calculation. The only difference is that a dihedral angle can have two deviations, the short and long way around the circle. We always want to use the shortest distance.

12.3.1 Average Selection

This standard deviation aggregate requires that the average be passed in. There are only two ways to match a list of averages to groups, subqueries or case logic. Neither is an ideal solution but case logic is the lesser of two evils.

```
CASE SS WHEN 'B' THEN 'foo' WHEN 'H' THEN 'bar' END
```

This results in queries that are very long (text size), but execution time is fast enough.

CHAPTER 13

Optimization: In Memory Tables

Loading PGD tables in memory, if indexed properly, will greatly outperform on-disk tables. It is a solution dependent on a capable enough server.

13.1 Indexing Memory Tables

Memory tables will only outperform a properly indexed on-disk table if it is also indexed. Above a certain threshold a full table scan of a memory table is still slower than a binary search of an index, even when it is on disk.

Memory tables support indexing, but mysql does not correctly index when using btree. Btree is the default for on-disk tables, and is generally faster than a Hash index. however, mysql will generate an empty index when btree is used with memory tables. This means that any table joined with a memory table must also be in memory.

13.2 Parallelization of Queries

Memory addresses can be read by multiple threads simultaneously, unlike disks which require seeking back and forth. Without seek times to slow down simultaneous reads, multiple queries can be run on an in memory table at the same time. This can reduce the time required for a query to the longest in the set of queries

Diagram

Note that this becomes a limitation of CPU cores and the number of concurrent threads the server is capable of handling.

13.3 Startup and Django Configuration Issues

Memory tables do not persist through mysql restarts. They must be recreated and indexed every time the server starts. This needs to be automated so that when the server first starts it is able to check and create the tables if needed.

Django must also be told what this special table is. The choices are:

- rename the table during the creation process. It might be impossible to determine the state of the tables though.
- change the name of the table in the django configuration to match the in memory table. this can only be done prior to django loading, the django orm cannot be reinitialized

13.4 Growth Concerns

Growth of the database is a major concern when dealing with large memory tables. Ram is cheap, but not as cheap as disk space. The current PGD database requries about 1.5 gigabytes (1.1 for Residues) of ram to load the Protein and Residue table in memory. Two factors will increase growth:

- Additional Fields - We have at least a 2 dozen new properties to add, which will add around 80% growth in the short term. There may be more fields added later also.
- Additional Proteins added - Expected to be 10% growth per year.

The works out to the following projects:

- current - 1.5 gigabytes
- 6 Months - 2.4 gigabytes
- 12 Months - 2.64 gigabytes
- 2 Years - 2.9 gigabytes
- 3 Years - 3.19 gigabytes
- 4 Years - 3.51 gigabytes
- 5 Years - 3.86 gigabytes

A server purchased now with 4 gigabytes of ram allocated for the just the memory table would last 5 years. This is two years past what is normally “end of life” for a server.

CHAPTER 14

Attempted Optimization: De-normalizing Residue Table

Searching segments of residues requires joining the residue table on itself, numerous times. Even when indexed properly joins can be slow. To remove the need for joins we de-normalized **pgd_core_residue** into **pgd_search_segment**. This table contained all properties, for each residue, for each possible segment in a protein.

This optimization greatly out-performed joins with less than 20,000 results. However, when the result set increased to 300,000 or more results it was twice as slow. The reason was that the table was too large to perform an entire table scan. It only was sped up when there was an index touching every field a search had a clause for. It was not feasible to build an index touch all 200+ fields to allow quick searching.

14.1 Table Size

The table size was not completely unmanageable, it was only 7 gigabytes. This was fine for on-disk, but would not scale well in memory.

CHAPTER 15

Search Workflow

PGD Search workflow is designed to take advantage of all of the features provided by Django: Models, Forms, and QuerySets. Search Models and Forms are interchangeable via methods provided by PGD.

15.1 Models

The Search class closely mimics a protein segment. It is composed of a three classes: Search, SearchResidue, and SearchCode. By creating Search as django models they can be stored and retrieved from the database.

15.1.1 Search

Search model contains all fields present in a protein.

15.1.2 SearchCode

SearchCode is a list of protein IDs to include in the search

15.1.3 SearchResidue

SearchResidue mimics a Residue object. It contains all fields that a

- Fields are all strings and can use the PGD Query Syntax
- Type fields such as AA_Type and SS_Type are stored as an integer with choices encoded in binary to conserve space.

15.2 Forms

SearchForms mimic Search Models exactly.

15.3 Conversions

Search and SearchForm are interchangeable.

- Use **function name** to convert a **Search** to a **!SearchForm**
- Use **function name** to convert a **!SearchForm** to a **Search**

Search can generate Django QuerySets

Use **search.queryset()**

15.4 Workflow

When submitting the search form, if the search is successful a **Search** object will be created and stored in the user's session. The **Search** can be retrieved and converted back into a **SearchForm** to edit the search.

This **Search** object becomes the basis for rendering all other pages. PGD is taking advantage of two features of querysets:

- **Lazy-execution of SQL Queries** - the query is not performed until the results are requested from the object.
- **Querysets can be further customized** - the query can be refined further to filter the fields returned, statistical calculations, etc.

These features allow PGD apply view specific logic without recreating the **base query** for every **view**. The **base query** contains the set of records to operate on. The **view** applies its specific logic and calculations.

15.4.1 Examples

- Plot page only requires 3 properties to perform its calculation
- Statistics only needs to return Averages, Standard Deviation and other statistics instead of the residue values.

CHAPTER 16

Ramachandran Plots

PGD provides a pseudo ramachandran plot. Our plot uses square bins rather than a more freeflowing plot. This allows quicker generation of graphs.

The Plot page uses django querysets, but makes use of [SQL Aggregates]. For ease in explaining how the data is processed this page will refer to the SQL generated by the querysets

16.1 Data Selection

The Ramachandran plot is generated using a specialized SQL query to group data points into bins.

```
1. Each coordinate has the minimum value subtracted and then divided by the BIN_SIZE and rounded down
2. GROUP BY is applied to both the X and Y coordinates.  This sorts the residues into a grid.

select FLOOR((phi-PHI_MIN)/10) as X, FLOOR((psi-PSI_MIN)/10) as Y from pgd_core_residue GROUP BY X, Y
```

Subtracting the minimum value from the coordinate shifts the start of the bins to the minimum value. The first bin will always be the same size as other bins. The last bin may be a different size if BIN_SIZE does not divide MAX-MIN evenly

16.2 Statistics Calculation

Once records are selected statistical calculations are performed depending on the input. By default the ramachandran plot z-axis displays observations, or the count of residues in that bin.

```
select Count(*) as count, FLOOR((phi-PHI_MIN)/10) as X, FLOOR((psi-PSI_MIN)/10) as Y from pgd_core_re
```

Optionally the z-axis can also display the average of user selected attribute.

```
select AVG(a1) as avg, STDDEV(a1) as stddev, FLOOR((phi-PHI_MIN)/10) as X, FLOOR((psi-PSI_MIN)/10) as
```

16.3 Coloring

Graphs are colored to represent values on a z-axis.

- **Reference** - The point at which to calculate distance from for determining colors. Used to adjust focus to a specific value. By default reference is the mean.

- **Outlier Sigma** - Number of sigma (standard deviations) beyond which values are considered outliers. Standard deviation is not recalculated excluding outliers, but they are excluded when calculating the range of colors. This helps keep the range of colors from spreading too widely due to outliers far from the mean.

16.4 Logarithmic Scale

PGD applies a logarithmic scale to all bins. Logarithmic scaled colors allow a greater range of colors choices to be applied closer to the **Reference**. As values approach the extend of **OUTLIER_SIGMA**** SIGMA* the number of colors to choose from lessens. This allows differences within values close to the mean to be more apparent.

16.5 Color Ranges

Plots are colored using a predefined tuple of RGB values:

- Max value
- Adjustment value - used to add a minimum value to the color. ie. the Blue plot adds 75 to all blue values, shifting all colors into a blue hue.

16.6 Algorithm

::

1. The logarithmic scale is first calculated producing a number from 0 to 1.
2. This is multiplied by each value in the RGB MAX tuple
3. each adjustment value in the RGB adjustment is added to the corresponding RGB value

CHAPTER 17

Search Statistics

The statistics page provides statistics of properties across all results for a specific residue index. Results are grouped by Amino Acid Type, and or Secondary Structure Type.

17.1 Queries

The statistics page makes use of aggregate functions to calculate values. Because of the different types of grouping and statistics, it requires several queries to retrieve all of the statistics.

17.1.1 Secondary Structure Counts and Amino Acid Totals

This query produces three values:

- Counts of residues per Secondary Structure Type, per Amino Acid Type.
- Counts of residues per Amino Acid Type.
- Total count of all residues in search.

The SQL required is

```
Select count(*) as count, aa_type, ss_type from pgd_core_residue GROUP BY aa_type, ss_type WITH ROLLU
```

The **WITH ROLLUP** clause instructs mysql to include totals for the fields. This produces the total per amino acid type, and total count of all residues. WITH ROLLUP only affects the first field in the GROUP BY class so totals per Secondary Structure Type are not produced by this query.

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^_ Secondary Structure Total Counts ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^_

Count of residues per Secondary Structure Type.

```
Select count(*) as count, ss_type from pgd_core_residue GROUP BY ss;
```

17.1.2 Field Statistics

Min, Max, Average, and Standard Deviation are calculated for every residue, per Amino Acid Type, include totals.

```
SELECT MIN(a1), MAX(a1), AVG(a1), STDDEV(a1) from pgd_core_residue GROUP_BY aa WITH ROLLUP;
```

17.1.3 Dihedral Angles Statistics

Dihedral angles (ome, zeta, phi, psi, etc.) require the use of Special Statistics for Dihedral Angles. Average is calculated in the field statistics query, but standard deviation requires a second query.

17.2 Optimization

Statistics uses the following optimizations:

- SQL Aggregate Functions to reduce network transport overhead
- Parallel Queries to run calculations simultaneously

CHAPTER 18

Data Dump

Data dump is a dump of all fields of all segments returned by a search.

18.1 Selecting Data

Queries using Django's ORM focus on a single object. Accessing related fields such as **Residue.prev** or **Residue.next** result in a second query to resolve those objects. This means to display a single segment of length 5 you must do 4 additional queries.

The solution is to calculate the list of residues for each segment and select them by range. For example, to retrieve a segment of length 3 where i-3 select all residues from i-1 to i+1 (2 through 4).

```
Select * from pgd_core_residue where chainIndex between 2 and 4
```

This allows all residues in a segment to be retrieved in a single query. This reduces the number of queries to the number of residues in the result set. This may still be a large number of queries but they may be run in parallel and the buffered nature of datadump spreads them out.

18.2 Buffered Response

Datadump uses an iterable class which uses threads to buffer data. The buffering threads use a paginator to split a result set into pieces. This allows downloading to start almost immediately after the button is clicked, rather than waiting for the entire dump to be written to memory.

This is not a perfect solution. Python uses green threads so threads are not truly concurrent. Threads will often be starved and may simply alternate between filling the buffer and emptying it. This is mainly an issue with the threads fighting over the lock. There may be a better solution using double-buffering.

CHAPTER 19

Browse

Browse displays all residues for each segment in a result set. Results a

19.1 Selecting Data

Queries using Django's ORM focus on a single object. Accessing related fields such as **Residue.prev** or **Residue.next** result in a second query to resolve those objects. This means to display a single segment of length 5 you must do 4 additional queries. To display 25 segments per page would require 125 additional queries.

The solution is to calculate the list of residues for each segment and select them by range. For example, to retrieve a segment of length 3 where i-3 select all residues from i-1 to i+1 (2 through 4).

```
Select * from pgd_core_residue where chainIndex between 2 and 4
```

This allows all residues in a segment to be retrieved in a single query. This reduces the number of queries to the number of residues in the result set. This may still be a large number of queries but each page is limited to 25 segments per page.

CHAPTER 20

Splicer

Splicer is the tool used to import data from PDB files.

Splicer is built as a Parallel Pydra Task. Pydra is a framework for parallel and distributed jobs with python. Using Pydra allows splicer run times to be reduced nearly linearly across nodes in the cluster.

20.1 Task Structure

Splicer is made up of several subtasks. It is organized using both ContainerTask (sequential subtasks) and ParallelTask to divide work up and parallelize it It is organized like so:

- Splicer (!ContainerTask)
  - Selector
  - ProcessProteinTask(!ParallelTask)
    - FTPUpdate
    - Processor
    - SegmentBuilder

20.1.1 Selector

Selector downloads lists of pdbs from a source, currently Dunbracks PICSES, and processes the list into a python object.

There are two selectors:

- DunbracksSelector - selects records from dunbracks culled lists
- PDBSelectSelector - selects records from pdbselect lists. This was replaced with dunbrack's selector

Both selectors filter on and parse protein level properties from the list.

20.1.2 FTP Update

Synchronizes a local cache of PDBs with a remote ftp server. By default it will synchronize all files located on the remote directory, optionally it can be given a list of PDB codes to synchronize.

FTP Update compares dates to the milisecond. This requires using the MODTIME (mdtm) FTP command.

20.1.3 Processor

The processor parses a pdb file and extracts properties from it. The current implementation uses *BioPython* library and DSSP. This component is covered in depth.

20.1.4 Segment Builder

This is a defunct subtask that was used to build the *de-normalized segment table*

CHAPTER 21

Developing Splicer

These are some special instructions for developing splicer components.

21.1 Django Settings

Django and its ORM can be used outside of its webserver. The only requirements are

- the directory containing **settings.py** is on the sys.path
- environment variable **DJANGO_SETTINGS_MODULE** is set to the **settings**

Splicer components should all do this automatically using python code to add the correct directory to **sys.path**. This works as long as the components are run from the directory containing **settings.py**

```
import os, sys
#python magic to add the current directory to the pythonpath
sys.path.append(os.getcwd())

# ------------------------------------------------------------
# Setup django environment
# ------------------------------------------------------------
if not os.environ.has_key('DJANGO_SETTINGS_MODULE'):
os.environ['DJANGO_SETTINGS_MODULE'] - 'settings
# ------------------------------------------------------------
# Done setting up django environment
# ------------------------------------------------------------
```

21.2 Running Components from the command line

Splicer Subtasks all contain **main** code that starts the tasks using arguments passed in. All should give you a list of required arguments if you do not pass them in. The main intention of this is to allow easier debugging on tasks.

For example **ProcessPDBTask** requires a pdb code followed by resolution, threshold, rfactor, and rfree. For testing purposing the properties do not need to be real values.

```
~/pgd/pgd $ ./pgd_splicer/ProcessPDBTask.py 12SB 1 2 3 4
```

CHAPTER 22

Splicer Processor Task

The **Splicer Processor Task**, also known as **ProcessPDBTask** accepts a PDB file and processes it into PGD Protein Models.

22.1 Running From the command line

Like all splicer components *ProcessPDBTask* can be run from the commandline to simplify debugging the task. It requires the fo

- PDB Code - 4 character alphanumeric code for a protein.
- Threshold - Float
- Resolution - Float
- Rfactor - Float
- Rfree - Float

When debugging only the PDB code need be a real value. The other values are required, but not validated.

22.2 Libraries

- BioPython - A library that can parse PDB files and contains various functions within its PDB API for extracting data from them.
- DSSP - A program for calculating secondary structure. BioPython has bindings for this program.

22.3 Parsing PDBs

Some properties are available as simple properties using the Residue class within BioPython. Most others require calculations involving individual Atoms within a Residue

22.3.1 Parsing Geometric Properties

Geometric properties must be calculated from raw atom data. BioPython supplies several functions for calculating functions be

- **calc_length(vector, vector)** - Calculates distance between atoms in 3D space (supplied by PGD)
- calc_angle - Calculates the angle between 3 atoms.
- calc_dihedral - Calculates the dihedral (torsion) angle between 4 atoms

These functions require vectors which can be retrieved using Atom.get_vector()

22.4 Example a3

:: N = residue['N'].get_vector() CA = residue['CA'].get_vector() C = residue['C'].get_vector()

a3 = calc_angle(N,CA,C)

22.5 Example: Ome

:: oldCA = prev_residue['CA'].get_vector() oldC = prev_residue['C'].get_vector() N = residue['N'].get_vector() CA = residue['CA'].get_vector()

ome = calc_dihedral(oldCA,oldC,N,CA)

22.5.1 Parsing Averaged Properties

Several properties are presented as averages across the **Main-Chain, *Side-Chain,** and **Carbon Gamma** atoms.

- **Main-Chain** - atoms N, C-alpha, C, O, OXT
- **Side-Chain** - all other atoms excluding Main-Chain, C-gamma, and HETs (water)
- **Carbon-gamma** - also known as C-gamma or Cg. A single atom.

This properties are calculated as the average of an atom level property across the atoms in this group.

22.6 Example B-factor: Bm, Bs, Bg

```
"""
Other B Averages
    Bm - Average of bfactors in main chain.
    Bm - Average of bfactors in side chain.
"""
main_chain = []
side_chain = []
for a in res.child_list:
    if a.name in ('N', 'CA', 'C', 'O','OXT'):
        main_chain.append(a.get_bfactor())
    elif a.name in ('H'):
        continue
    else:
        side_chain.append(a.get_bfactor())

if main_chain != []:
    res_dict['bm'] = sum(main_chain)/len(main_chain)
```

```
if side_chain != []:
    res_dict['bs'] = sum(side_chain)/len(side_chain)
```

22.6.1 Parsing Side Chain Properties

Side chains are different for each type of atom. They require a map of connections to determine which atoms require angles, lengths, and dihedral angles calculated.

Currently **Chi1, Chi2, Chi3, and Chi4*are calculated using a map in *chi.py**. Sidechain lengths and angles will be added later, requiring an additional map of connections between atoms.

22.7 Update Checking

PDB Processor checks for updates when processing proteins. The Protein Model contains a timestamp which corresponds to the update timestamp on the PDB file it was imported from. A protein will only be processed if the PDB file is newer.

CHAPTER 23

Running Splicer

Splicer is intended to be run on a Pydra cluster. These are instructions and notes on running it and dealing with Pydra's immaturity.

Splicer can be deployed and run like any other task on a Pydra cluster.

Components of splicer can also be run manually from the command line

23.1 Slow FTP Issues

The FTP server for PDB files is a very slow, rate limited, server located in the UK. PDB files are currently 1.8 gigabytes total for 16,000 proteins in PGD. It can take a long time to download this much data from the FTP server. This is handled in two ways:

23.2 Maintaining Connections Between Workunits

Each workunit is composed of downloading and processing a PDB file. Rather than disconnecting from the FTP server, connections are maintained until the last work unit is completed. This removes the overhead for connecting and disconnecting from the server

23.3 Only Downloading New Files

Checking dates is very fast, the MODTIME command completes almost instantly. This prevents uneeded downloading

23.4 Storing files in a network share

Pydra can't guarantee that future runs of Splicer will process the same set of proteins on the same hardware. This means that an up-to-date PDB could be mistaken for a PDB that doesn't exist. Storing the files on a shared filesystem ensures that regardless of which Node is assigned the workunit, it will find the same set of PDB files.

Note that this only matters when using

23.5 Workunit Thrashing Problem

There is an outstanding bug in pydra that causes the node to crash when workunits complete too quickly. Splicer includes an option to batch process proteins to ensure that this does not happen. Eventually batching workunits will an automatic feature of Pydra

When running repeat runs of Pydra it is important to increase the workunit size to at least 500-1000. Because the date checks are very fast it will cycle through the existing proteins very quickly.

23.6 Debugging Splicer

Pydra logs most things that happen within it. A full task history can be viewed by clicking the history icon found on the pydra tasks page. Clicking on a task instance gives you more details about the task including which workunits were successful and what their arguments were.

Workunits are logged individually and located in **/var/logs/pydra/archive**. The logs are aggregated from the Nodes after it is done with the entire task

CHAPTER 24

Running Splicer From The Command Line

Splicer can be run from the command line. It requires that several steps be run separately.

All commands should be run from the project root (directory with settings.py in it).

24.1 Selecting Proteins

Proteins must first be selected. Default filtering settings will be used for threshold, resolution and r_factor.

```
1 ./pgd_splicer/dunbrack_selector.py
```

This will return information about the the parameters used, the files proteins were selected from, and a list of proteins in the following format:

```
code chains threshold resolution rfactor rfree
```

Save the selection into a file:

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selection.txt
```

24.2 Options

- –pipeout - will only output the data. This should be used if you would like to create output suitable for input into one of the other steps

24.3 Downloading PDB Files

PDB files are downloaded from an FTP site using **ftpupdate.py**. This script will synchronize **./pdb** with the remote FTP server. Only new files will be downloaded, but it will check the timestamps on all files.

This is a time-consuming step. Be prepared to wait for approximately two days for this to complete on a fresh local copy, or one day on an update.

```
1 ./pgd_splicer/ftpupdate.py code [code...]
```

To only grab the proteins which are selected (and cut down massively on consumed bandwidth and time), try:

```
1 ./pgd_splicer/ftpupdate.py --pipein < selection.txt
```

24.4 Processing PDB Files

PDB files can be imported into the database with **ProcessPDBTask.py**. Multiple proteins can be fed as commands to be imported. Errors will be written to ProcessPDB.log

```
1 ./pgd_splicer/ProcessPDBTask.py code chains threshold resolution rfactor rfree [repeat]
```

As before, a selection can be piped in:

```
1 ./pgd_splicer/ProcessPDBTask.py --pipein < selection.txt
```

Expect this to take a few days as well.

24.5 Parameters

Parameters are all required, and may be repeated for multiple proteins.

- code - protein code to import, should be all uppercase
- chains - list of chains to import, should be a string of chain ids. (ie. ABCDEF). The string should not have quotes around it.
- threshold, resolution, rfactor, rfree - the value for these fields. These properties are retrieved from the selection script so they are included as input for processing the protein.

24.6 Options

- –pipein - input will be read from a pipe instead of arguments. proteins in the list should be separated by newlines.

24.7 Example

Some examples. Intermediate output is saved to a text file so that it can be examined later.

24.8 Full Import

Update all proteins regardless of whether the file was downloaded by **ftpupdate**. ProcessPDBTask will still check the update date and exclude pdbs that are not new.

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2 ./pgd_splicer/ftpupdate.py --pipein < selected_proteins.txt
3 ./pgd_splicer/ProcessPDBTask.py --pipein < selected_proteins.txt
```

24.9 Update Only New

Update only proteins for which we have a new FTP file.

```
1./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2./pgd_splicer/ftpupdate.py --pipein --pipeout < selected_proteins.txt > updated_proteins.txt
3./pgd_splicer/ProcessPDBTask.py --pipein < updated_proteins.txt
```

24.10 Update Skipping Download

If all the pdb files are already downloaded you may skip the FTP step to save time.

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout > selected_proteins.txt
2 ./pgd_splicer/ProcessPDBTask.py --pipein < selected_proteins.txt
```

or as a single command:

```
1 ./pgd_splicer/dunbrack_selector.py --pipeout | ./pgd_splicer/ProcessPDBTask.py --pipein
```

CHAPTER 25

Development workflow

25.1 Create an issue in the issue tracker if one does not already exist.

All modifications to the source code should be associated with an issue for tracking purposes.

25.2 Create a branch from the develop branch based on the issue number.

The branch should be named based on the issue type and number. For now, all issues are considered bugs for naming purposes. The branch for issue #14109 (a feature) should be named 'bug/14109'. At this time, the issue should be updated with the status 'In Progress'.

25.3 Write the code, including tests and release notes entry if necessary.

If the branch is fixing a bug, then a test should be written first if at all possible to confirm the bug exists as well as confirm that the bug fix works.

If the branch makes any user-visible changes, then the release notes (news.html) should be updated to reflect the changes. If you are the first developer to post release notes for the latest development version, add an appropriate header above the existing release notes following the example format – the release engineer will clean it up if necessary when making the next release.

25.4 Commit code and update the issue.

If any billable work is done on an issue, then code should be committed, and the issue should be updated with the number of billable hours spent on the task and a summary of the work that was done. Commits and updates should happen whenever major subtasks are completed and at close of business.

25.5 Request code review.

When the code is complete, the tests if any all pass, and the release notes entry has been added if necessary, mark the issue 'Needs Review' and assign it to another team member for review. That individual will review the code, the tests, the entry, and any other associated changes for accuracy and consistency. If the branch is acceptable, the reviewer will mark the issue 'Resolved'. If the branch is unacceptable, the reviewer will mark the issue 'Needs Work'. In both cases, the reviewer will update the issue with relevant information and reassign the issue back to the original developer.

25.6 Merge changes back into the develop branch.

Once the branch passes review, it should be merged back into develop using 'git merge –no-ff' and then develop should be pushed back into the origin. Once this is complete, the devloper should update the issue to that effect.

CHAPTER 26

Hotfix Workflow

26.1 One or more show-stopping bugs are detected between releases

An example of a show-stopping bug would be something like #14613 where an unexpected side-effect of a bugfix caused incomplete results to be returned on unrelated searches.

26.2 Create a hotfix branch off the master branch

The hotfix branch name will take the form hotfix/x.y.z+1

If the current release is 1.2 the hotfix branch is then hotfix/1.2.1 If the current release is 3.7.2 the hotfix branch is then hotfix/3.7.3

26.3 Create bug branches off the hotfix branch

For each bug that must be fixed, an issue is created and a bug branch named after that issue like all other bugs is created from the hotfix branch instead of the develop branch like all other bugs.

26.4 Merge resolved bugs back to hotfix branch and test on staging

As each individual bug is resolved, its branch is merged back into the hotfix branch. The hotfix branch can then be updated on the staging server for testing.

26.5 When all bugs are resolved and merged, merge hotfix branch into master branch

Do not forget to increment the version, create a new tag, and update the news page with all bug fixes!

26.6 Pull master on production and restart Apache

This should make the new version accessible to the user community.

26.7 Merge hotfix branch back into develop

Once production is back up and running, take the time to merge the hotfix branch back into develop.

CHAPTER 27

Release Workflow

27.1 Announce the upcoming release to the PGD list four weeks before release.

Include the following information:

- version number to be released in x.y.z format
- rough list of features and bugfixes expected to be included
- schedule of events (feature freeze, release branch, etc.)

27.2 Impose a feature freeze on develop three weeks before release.

Features are no longer permitted to be merged into the develop branch, only bugs.

Release engineer checks each resolved ticket to confirm that the branch was indeed merged into the develop branch.

27.3 Start the release branch two weeks before release.

Create a branch from develop named 'release/x.y.z' using the version number mentioned in the release announcement.

Log into the dev site and check out the release branch there.

Bugfixes can only be made from and returned to this branch at this time.

27.4 Freeze the release branch one week before release.

Log into the staging site and check out the release branch there.

Only emergency fixes allowed at this point!

27.5 Release the software, close tickets and unfreeze develop on the release date.

Merge the release branch back into master and develop branches.

Log into the production site and check out the master branch there.

All resolved tickets should be closed at this time.

Any existing branches should be rebased from develop before development continues.

Features may now be merged back into develop at this time.

CHAPTER 28

Management Commands and Possible Redisign

2 Databases:

- Staging (aka Silver)
- Master (aka Gold)

Instead of two databases

```
class Protein(models.Model):
    """
    Same as before
    """
    ...

class GoldProtein(Protein):
    """
    Nothing actually goes here
    """
```

Manage Cammonds:

- **Import (Modifies Staging, Reads from Master)**
  - **Fetches pdb files (like fetch does currently)**
    - –fetch-only as an option
  - Stores the selection in the Audit table
  - Dumps Proteins from staging
  - "ProcessPDBTask"
  - Generates a diff (Total, New, Removed) data and stores it in the Audit table
- **Promote (Reads Staging, Modifies Master)**
  - Dumps Data
  - Copies Staging into Master
  - Updates Master Audit table

CHAPTER 29

Indices and tables

- genindex
- modindex
- search