

WhyNot: A quality control system for PDB-related databases.

Written by Tim te Beek

Under supervision of Gert Vriend

Centre for Molecular and Biomolecular Informatics

Nijmegen, the Netherlands, June 2007

Index

PROBLEM	3
SOLUTION	5
IMPLEMENTATION	6
RELATIONAL DATABASE.....	8
DATA ACCESS OBJECT	11
CRAWLER	12
COMMENTER	13
WEB SERVER.....	14
<i>Reports</i>	14
<i>Collection</i>	15
<i>WhyNot</i>	16
CONCLUSION	17
FUTURE WORK.....	17
REFERENCES.....	18
APPENDIX A: DATABASE MODEL.....	19
APPENDIX B: JDBC.XML EXAMPLE.....	22
APPENDIX C: CRAWLER.....	23
APPENDIX D: COMMENTER	25

Problem

The Protein Data Bank¹ is the single worldwide repository for the storage and distribution of experimentally derived 3D structure data of large molecules like proteins and nucleic acids. Each structure published in the PDB is assigned a unique four-character alphanumeric identifier. The structures in the PDB are stored in flat files, which use the above PDB ID to identify the structure contained in each file. These flat files are distributed by the members of the World Wide Protein Data Bank.

Over the years numerous derived databases have been developed to integrate and classify the PDB in terms of protein structure, protein function and protein evolution. These databases are used by bioinformaticians all around the world for research into the structures of biological macromolecules and their relationships to sequence, function, and disease. All these derived databases contain at most one entry for each PDB entry. These entries are normally also stored in flat files identified by the PDB ID of the corresponding PDB entry.

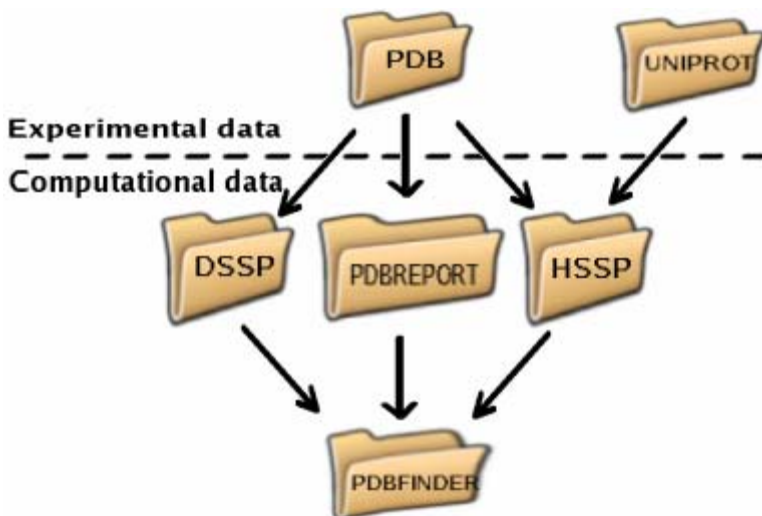


Figure 1: A selection of PDB related and derived databases and their correlation.

Figure 1 shows a selection of PDB related and derived databases and their correlation. A brief description for each of these databases:

- **PDB**
The Protein Data Bank is the single worldwide repository for the storage and distribution of experimentally derived 3D structure data of large molecules like proteins and nucleic acids.
- **UniProt²**
The Universal Protein database is a central repository of protein sequence data. Storing expansive and meaningful protein annotation (such as the description of the function of a protein, its domains structure, post-translational modifications, variants, etc.)
- **DSSP³**
The DSSP contains secondary structure assignments to the amino acids of a protein, based on the atomic-resolution coordinates of the protein.
- **HSSP⁴**
The HSSP database is a database of multiple sequence alignments of UniProt on PDB.
- **PDBFinder⁵**
For each PDB entry the PDBFinder database contains a search-engine-friendly-formatted entry containing information found in the PDB, DSSP, HSSP and PDBReport databases.
- **PDBReport⁶**
A collection of reports describing structural problems in PDB entries.

Both the PDB and UniProt contain experimental data, whereas the DSSP, HSSP, PDBReport and PDBFinder databases contain computational data based on entries in the PDB and UniProt.

New PDB entries are added and distributed by the members of the Worldwide Protein Data Bank on a weekly basis. Because entries for the derived databases are created in a separate process that runs after each weekly update, it is inevitable that the creation of derived database entries lags behind. In such cases the PDB entry is present, but the derived database entries can not be found yet.

Due to differing content and quality of PDB entries something occasionally goes wrong when trying to create a derived database entry for a given PDB entry. For instance: the DSSP algorithm is not able to determine the secondary structure of nucleic acids. If the DSSP algorithm determines that a PDB entry only contains nucleic acids, the computation terminates and the DSSP entry is not created. In such cases the PDB entry is present, but the derived database entry can not be found. Similar problems can occur for a variety of reasons and algorithms.

Occasionally a PDB entry is removed from the PDB. When a PDB entry is removed, the derived database entries related to this PDB entry should also be removed. Similar to the creation of derived database entries, the derived database entries are also removed in a separate process that runs after each weekly update. This causes the removal of derived database entries to lag behind. In such cases the PDB entry is no longer present, but the derived database entries can still be found.

In all the above three cases problems can occur when external applications refer to a non-existing database entry. When a non-existing file is referred, the user receives an error saying the database entry can not be found. Without any system in place to account for missing files, users turn to the database maintainer(s) to ask why an entry can not be found. For each such request a database maintainer has to determine why an entry can not be found and report this information back to the user. As there is no form of quality control or accounting for missing database entries, these rather simple requests take up valuable time.

Solution

The WhyNot project was started to solve the above problems associated with missing entries and other database inconsistencies. The project aims to set up a system that introduces both quality control and accountability for missing entries to the PDB-related databases by checking cross-database integrity and storing the reason why an entry is missing. This information should then be made available to end-users. Providing the above functionality requires executing the following steps for each of the PDB-related databases:

1. Locate the database (mirror).
A system administrator will have to provide a locally readable mirror of the database.
2. Index all entries in the database.
A subsystem will index and store information about all the entries in the local mirror.
3. Check cross-database consistency.
Based upon dependencies between the PDB and PDB-related databases a subsystem will evaluate which entries are obsolete and which are missing.
4. Consult database expert about inconsistencies.
A database expert can determine the reason for database inconsistencies and either store the reason or in some cases resolve the database inconsistencies.
5. Iterate steps 3 and 4.
Repeat until there are either no more database inconsistencies or until a reason has been provided for all existing database inconsistencies.
6. Provide a graphical interface to end-user.
An internet accessible graphical interface allows end-users to view which entries are missing and the reason why a given entry is missing.

With the above system in place end-users, external applications and web services can refer to this system when confronted with a non-existing database entry. This eliminates the need to contact a database expert, saving valuable time and effort while increasing response time.

Implementation

In the solution we outlined the steps that have to be taken to provide the proposed functionality. The WhyNot system was custom-built to implement this functionality. This chapter discusses the choices made in designing WhyNot, the subsystems that make up WhyNot and how these subsystems combine to offer the above described functionality.

Aside from the functional requirements outlined in the solution the WhyNot system also would have to meet a number of requirements related to presentation, extendibility and maintainability. These requirements are the direct result of the expressed wish to alter and extend the system at a later moment in time. As such WhyNot would have to be:

- Highly flexible in both the accepted input and the presentation of output
- Easily extendible to support additional databases and functionality
- Easy to maintain and extend by someone other than the original author

Figure 2 shows the various subsystems that together make up WhyNot. The arrows indicate the flow of information between the PDB-related databases, the database experts, the end-users and WhyNot, as well as the flow of information among the subsystems.

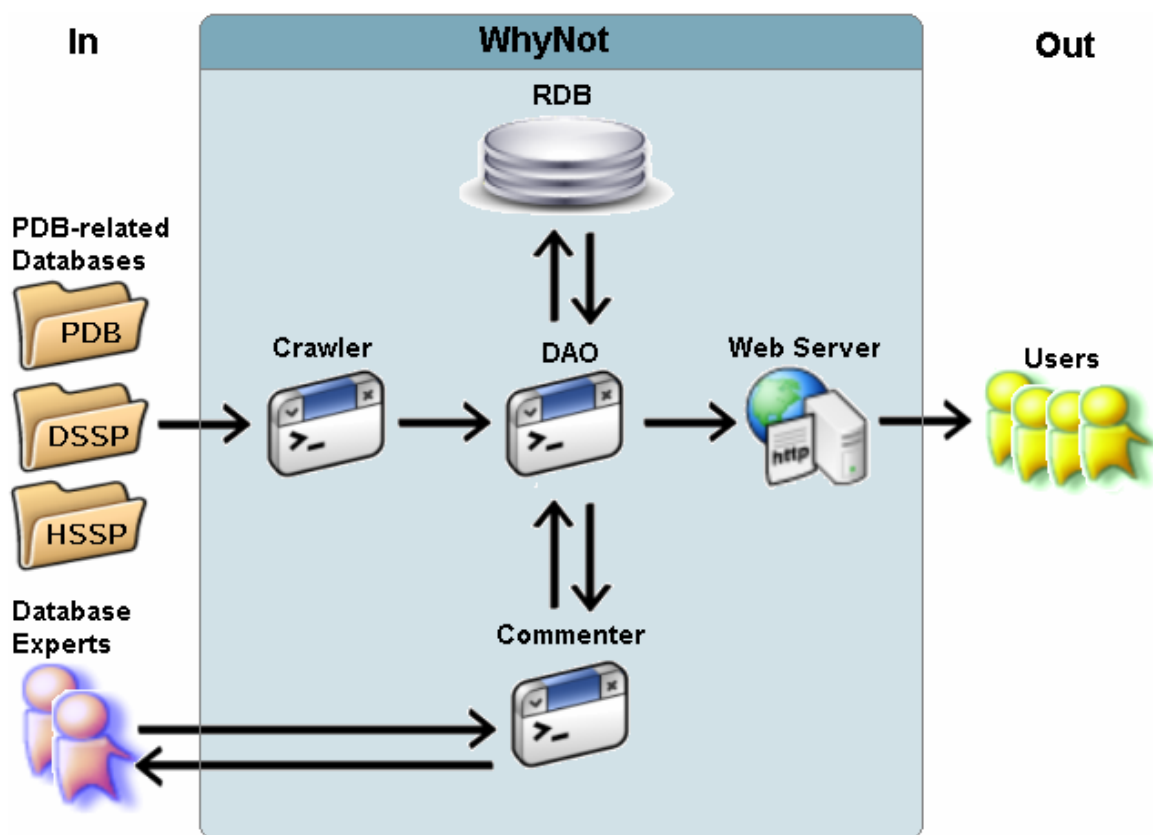


Figure 2: An illustration of the flow of information among the various WhyNot subsystems.

A brief description of all of the subsystems in the above figure:

- **Relational Database (RDB)**
The RDB stores all the information gathered and used by the various subsystems.
- **Data Access Object (DAO)**
The DAO is the only part of the system that directly interacts with the RDB.
- **Crawler**
The Crawler indexes all the entries in the local PDB-related database mirror(s) and asks the DAO to store this information in the RDB.

- Commenter
The commenter parses the reason(s) for database inconsistencies provided by the database experts and asks the DAO to store this information in the RDB.
- Web Server
The web server asks the DAO for either information related to a single PDBID or for information about collections of PDBIDs and presents this information on a webpage.

We'll discuss each of these subsystems in more detail throughout the rest of this chapter.

Relational database

The design of the system started with analyzing the data the system will store and choosing a method to store this information. The data analysis indicated the following information will have to be stored:

- Information about every entry in the PDB and the PDB-related databases.
Both the PDB and PDB-related databases store all entries in flat files; one for each entry. The only exception to this rule is the PDBFinder database which stores all entries in a single file. All entries related to the same PDB entry share the PDB entry's PDBID for identification. For each entry the PDBID, the corresponding database name, path to the flat file and flat file's timestamp will be stored.
- When needed: the reason for database inconsistencies.
A database expert can specify a reason for database inconsistencies, for instance when a PDB-related database entry is missing. The reason will be stored along with details on who submitted the reason and when the reason was submitted. There is no ontology for these reasons.

The relational nature of the above data quickly pointed toward using a relational database to store this information. Although the nature of the problem would allow using a spreadsheet to store the same information, using a relational database introduces certain benefits. These benefits include high performance and a high level of flexibility in accessing the data. Normalizing the manner in which the data is stored in the RDB can also eliminate data redundancy. Using a RDB also greatly increases the ease with which the system can be extended, especially when compared to using a spreadsheet. An evaluation of these benefits ultimately resulted in choosing a relational database to store the above data.

Having made the decision to use a relational database, the next step in designing the system was designing the database. For this we used the modeling technique Fully Communication Oriented Information Modeling⁷. This resulted in the database scheme described in appendix A. A simplified graphical representation of relevant database characteristics can be seen in Figure 3. Here we'll describe some of the relevant database characteristics to explain how the information is stored.

Both the PDB and PDB- databases use the PDBID to identify database entries. All the entries related to a single PDB entry share the PDBID of that entry. Therefore it made sense to utilize this PDBID to uniquely identify entries within the PDB and the PDB-related databases.

At this point we could have used multiple tables to store the entries for each of the databases separately. However, due to the large similarities in the stored data and the desire to create a flexible and extensible system, this setup would not have been optimal, as it would require an additional table for each database added to the WhyNot system.

Seeing how each entry within a database can be uniquely identified using a PDBID, and that each database can be uniquely identified using the database name, the combination of these two can uniquely describe any database entry in the system. It is this exact combination that is used throughout the system to identify database entries, as demonstrated in figure 3.

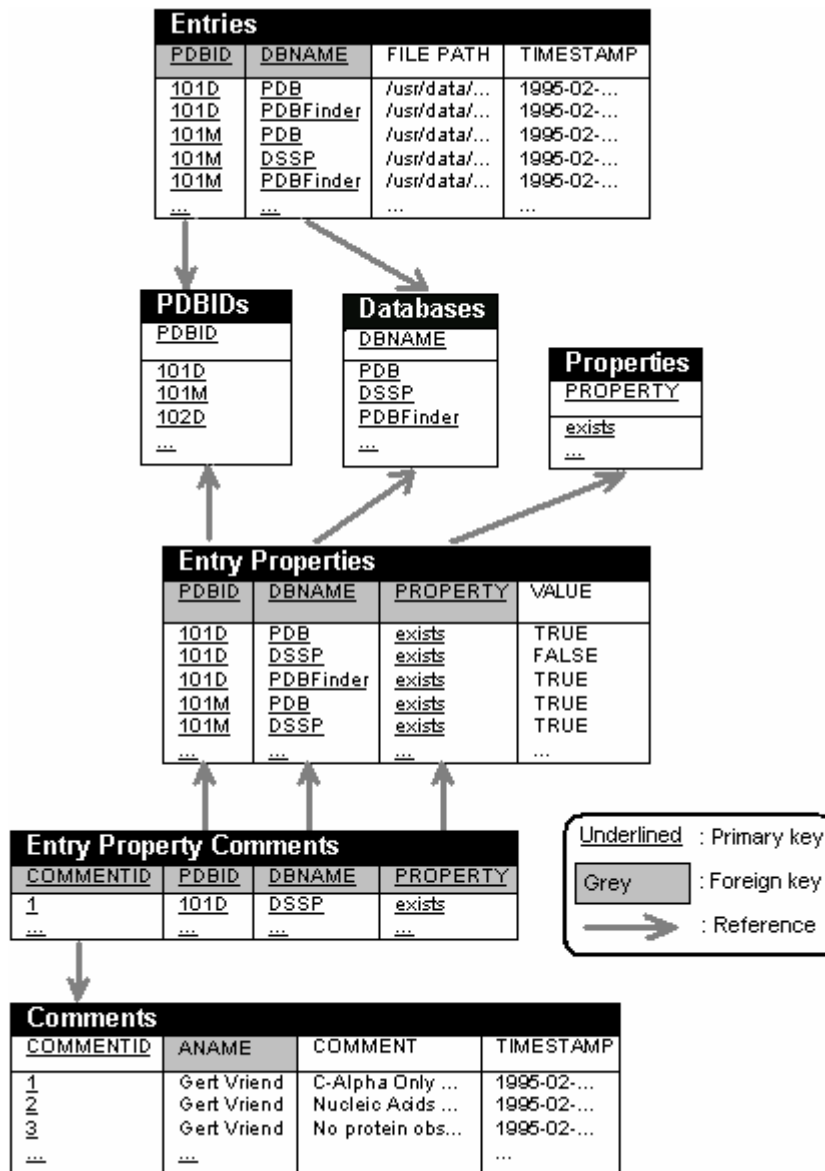


Figure 3: A simplified representation of relevant database characteristics.

Two separate tables “*PDBIDs*” and “*Databases*” are used to store the PDBIDs and the database names. The cross table “*Entries*” between these two is used to store the information about the files associated with database entries. For identification this table uses a combination of foreign key references to the “*PDBIDs*” and “*Databases*” tables. The additional fields store the path to the file and the timestamp.

Identifying PDB-related database entries by using a combination of PDBID and Database name rather than the flat file associated with the entry allows us to link additional information about an entry to this combination, even when there is no file associated with this entry.

We already established that the reason for database inconsistencies would also have to be stored in the RDB. We could have stored this reason by adding a “*comment*” field to the “*Entries*” table and making the “*file path*” and “*timestamp*” fields optional. Any remarks about database inconsistencies could then be stored in this field. However, this approach would have harmed the database’s flexibility and extensibility, which were two key requirements to begin with. Therefore

we chose an alternative means of storing the reason for database inconsistencies, as explained in the remainder of this section.

We first set out to store database inconsistencies related to individual PDB-related database entries. For this we added two tables; one called “Properties” and another called “Entry Properties”.

The “*Properties*” table stores the inconsistency as a property an entry either does or does not have. For instance: the database inconsistency “entry missing” is formulated as that entry not having the property “*Exists*”. This approach allows other information that can be formulated as a binary property of an entry to be stored in a similar fashion.

The “*Entry Properties*” table is a cross table between the tables “*PDBIDs*”, “*Databases*” and “*Properties*”. For selected combinations of these three this table can store whether or not the described entry has the property by setting the boolean in the “*value*” field to true or false.

Having first stored the database inconsistencies we then set out to store the reason for these database inconsistencies separately. For this we used an approach similar to the one used to store the database inconsistencies. This required another two tables called “*Comments*” and “*Entry Property Comments*”.

The “*Comments*” table stores the reason for database inconsistencies as flat text. As there is no formal ontology for the reason for database inconsistencies, this approach proved sufficiently flexible. When the same reason is given to explain multiple database inconsistencies, the same record in the “*Comments*” table is utilized. As such database experts are advised to reuse existing comments and terminology.

The “*Entry Property Comments*” table is a cross table between the tables “*Comments*”, “*PDBIDs*”, “*Databases*” and “*Properties*”. The first field references a record in the comment table. The last three fields reference a record in the “*Entry Properties*” table. This approach allows storing multiple reasons for a single database inconsistency related to an entry while preventing duplicate entries.

We’ve chosen to implement the Relational Database using PostgreSQL as our Relational Database Management Server. The choice for PostgreSQL was based on past experiences with PostgreSQL and the fact that this RDBMS is available free of cost while supporting all the required functionality.

Data Access Object

All subsystems in the WhyNot system will have to enter, alter or access information in the RDB. To access this information a subsystem would have to be familiar with how the data is stored internally, i.e.: know the relevant table names, fields, foreign key references, primary keys, etc. Implementing this functionality into each of the various subsystems can create problems when some of these details change, as all the programs using these details will then have to be altered.

To prevent all subsystems from having to know how the data is stored in the relational database, we've set up a Data Access Object to handle all the interaction with the RDB. The subsystems can interact with the DAO via a predefined set of methods to access, enter and alter the information stored in the database, as shown in figure 2.

The benefit of this approach is that only one application needs to know how the data is stored internally. This means that when the manner in which the information is stored changes, only one application will have to be altered. Having only one application access the database also allows us to easily add additional checks on the validity of the entered data; increasing data integrity.

One additional feature of the DAO is that it automatically sets the binary property "*Exists*" for all possible PDB or PDB-related database entries. This behavior manifests itself when adding a new database, adding a new PDBID, adding an entry or removing an existing entry.

When a new database is added to the RDB a new record is added to the "*Entry Properties*" table for each combination of a PDBID, the database's name and the property "*Exists*". The "*value*" boolean for all these records has the initial value of *false*.

When a new PDBID is added to the RDB a new record is added to the "*Entry Properties*" table for each combination of this PDBID, any database's name and the property "*Exists*". The "*value*" boolean for all these records has the initial value of *false*.

When an entry is added to the RDB the "*value*" boolean of the record in the "*Entry Properties*" table for this entry's PDBID, database name and the property "*Exists*" is set to *true*.

When an entry is removed from the RDB the "*value*" boolean of the record in the "*Entry Properties*" table for this entry's PDBID, database name and the property "*Exists*" is set to *false*.

The DAO connects to the RDB through a Java Database Connection. The settings the DAO should use in connecting to the RDB can be defined when invoking the DAO. Both the crawler and the commenter utilize the "*jdbc.xml*" configuration file to read these settings. An example of such a file can be found in Appendix B.

Crawler

The Crawler is the part of WhyNot that indexes all the entries in the PDB and PDB-related databases. A system administrator will have to undertake a number of steps to index all the flat files associated with the entries in a database:

1. Provide a unique name to identify the database
2. Point the Crawler towards either the database or a locally accessible mirror
3. Provide a regular expression that only matches the flat files associated with this database
4. Specify at which character the PDBID starts in the filename of the flat files
5. Optionally: Provide a hyperlink to both the database and the entries in the database
6. Run the Crawler application either specifically for this database or for all databases

Using this input the Crawler can crawl through the directory provided in step 2 for files matching the regular expression given in step 3. Based on the number provided in step 4 the Crawler will extract the PDBID from the filename and ask the DAO to store this PDBID along with the database name, the path to the file and the file's timestamp. For diagnostic purposes the stored entries are also written to a flat file identified by the database name.

After the Crawler has indexed and stored all the entries in the database it turns its attention to the DAO and requests all the entries stored in the RDB for that database. For each of these entries the Crawler evaluates if the associated file can still be found on the local file system. If such a file can no longer be found on the local file system the Crawler asks the DAO to remove the entry from the RDB. The intention of this step is to remove entries that have been marked obsolete and have therefore been removed from the database.

The behavior of the Crawler program is largely customizable through both configuration files and command line parameters. Multiple instances of the same program can be run consecutively or concurrently using the same configuration files and RDB, for instance when multiple separate source folders need to be crawled for each type of file. For more information see Appendix C. The crawler could best be run after each weekly update of the PDB and PDB-related databases.

Commenter

The Commenter is the part of WhyNot that allows database experts to provide a reason for database inconsistencies. Through the web server the database experts can see which database inconsistencies currently exists and request to see the affected PDBIDs. External applications can then be used to determine the reason for these database inconsistencies.

The current implementation of the Commenter is a command line application that reads strictly formatted flat files. To store the reason(s) for database inconsistencies a system administrator needs to create such a formatted flat file and have the Commenter parse that file. Such a file should have the following format:

PDBID	: 1XJ9
Database	: DSSP
Property	: Exists
Boolean	: false (this line is optional)
Comment	: Not enough amino acids
//	

Table 1: Example comments file record

Multiple records can be stored in the same file by repeating the above pattern one after the other. The Commenter will read each individual record and ask the DAO to store both the database inconsistency ("*Property*") and reason ("*Comment*") in the database. The value behind "*Boolean*" in the above example (which can be either *true* or *false*) will be used to indicate if the Entry either does or does not have the specified binary property. This line is optional.

The Commenter application can be asked to either parse a single comments file or to parse all the comments files inside a given directory. Being able to parse all the comment files inside a given directory enables the system administrator to set up a folder that contains all the comment files. This directory can serve as a destination for external application to write comment files to. Parsing the same comments file a second time does not alter the data inside the RDB in anyway, so parsed files can remain in this directory to serve as a form of backup. Appendix D gives some details on using the Commenter application.

Web server

All the aforementioned subsystems serve to gather and store information about PDB-related database entries, properties and comments. The Web server intends to make this information available to a wide range of users. Implemented using the Apache Struts2 framework, the Web server interacts with the DAO to retrieve information from the RDB. The Web server has three levels of detail to display information about PDB-related database entries, properties and comments:

- For each database a report about the number of entries stored, obsolete, missing, with and without comment
- For each of the collections mentioned above the PDBIDs in this collection
- For each PDBID an overview of which entries exist and comments related to this property

We'll discuss each of these pages that display this information in more detail here, which are also accessible from <http://www.cmbi.ru.nl/whynot/>.

Reports

The focus of the WhyNot project and website is on providing a meaningful explanation for why PDB-related database entries do not exist. As such it seemed only logical to provide an overview of which PDB-related entries exist as well as for which of the non-existing entries an explanation has been provided. For each of the databases an evaluation is made of which entries are present in this database, compared to the entries present in the database this database is derived from. In almost all cases the PDB is the database the evaluated database is derived from, with the exception of the HSSP database which is derived from the DSSP database and UniProt. This evaluation divides all the PDBIDs in four distinct categories:

1. PDBIDs present in both the database the evaluated database is derived from and in the evaluated database
2. PDBIDs present in the database the evaluated database is derived from but missing in the evaluated database
3. PDBIDs missing in the database the evaluated database is derived from but present in the evaluated database
4. PDBIDs missing in both the database the evaluated database is derived from and in the evaluated database

An example of these categories is illustrated in table 2 for the database DSSP, with checks indicating correct collections and crosses indicating database inconsistencies.

DSSP		
PDB		
	Entry Present	Entry Missing
	Entry Present	Entry Missing
	1 ✓	2 ✗
	3 ✗	4 ✓

Table 2: PDBIDs categories in evaluating present and missing database entries for DSSP.

Both the collection 1 and 4 are correct and do not need any further attention. The collections 2 and 3 however show PDBIDs related to database inconsistencies that will have to either be resolved or explained by a database expert. Once a database inconsistency has been explained it is no longer requires attention from the data expert. As such we've divided collection 2 into two distinct groups: those with a comment explaining the inconsistency and those without a comment.

The reports page of the Web server is shows the number of items in collections 1, 2 and 3 for each of the databases evaluated by the WhyNot system. For collection 2 details are also given about how many PDBIDs in this collection have comments explaining why the entry does not exist. This information is divided into two columns, one for stored entries and one for missing entries. For each of these columns the total number of items is given. For stored entries this number is split up into correct entries (category 1) and obsolete entries (category 3). For missing entries this number is split up into entries with a comment and entries without a comment explaining why the entry is missing. An example output for the reports page is shown in figure 4.

Reports

Database	Stored Entries			Missing Entries		
	<i>Correct</i>	<i>Obsolete</i>	<i>Total</i>	<i>With Comment</i>	<i>Without Comment</i>	<i>Total</i>
DSSP reference	42222	0	42222	1920	58	1978
HSSP reference	41595	0	41595	0	627	627
PDB reference	44200	0	44200	0	0	0
PDBFINDER reference	44200	0	44200	0	0	0
PDB_REDO reference	18074	11	18085	1	26125	26126
PDBREPORT reference	42314	148	42462	0	1886	1886
STRUCTUREFACTORS reference	26934	63	26997	0	17266	17266

Figure 4: Example output for the reports page.

Collection

The numbers given on the reports page represent collections of PDBIDs meeting a certain set of requirements. The collection page shows the PDBIDs in this collection and is accessible by clicking on any of the numbers on the reports page.

Collection

All PDBIDs in PDB but not in DSSP without a comment.

PDBIDs (58)

```
2CD1 2CD3 2CD5 2CD6 2DP7 2DPB 2DPC 2DQ0 2DQP 2DQQ
2F1Q 2G91 2GJB 2GOT 2GV4 2GVO 2GVR 2GW0 2GWE 2GWQ
2GYX 2H0N 2HK4 2HOU 2HPX 2HRI 2HSK 2HSL 2HSR 2HSS
2HUA 2HY9 2I7E 2I7Z 2IC4 2NPW 2NQ0 2NQ1 2NQ4 2O3M
2O4F 2O4Y 2OCW 2OEY 2OF6 2OGN 2OGO 2OJ7 2OJ8 2ORF
2ORG 2ORH 2P7D 2P7E 2P7F 2P89 2PCV 2PCW
```

Figure 5: Example output for the collection page.

WhyNot

For selected PDBIDs the Web server can show which database entries are present and any comments related to the existence of the database entry. For entries that are present a hyperlink to view the file is given, as well as the timestamp on which the file was created. This page is accessible by entering a PDBID at the top of the page and pressing enter. An example output for the PDBID *101D* is given in figure 6.

Show files, properties and comments related to a single Protein Data Bank Identifier.

PDBID:

Why Not: 101D








Database	Entry file	Comments
DSSP reference	 Entry file missing. -	Nucleic acids only Administrator - 19/06/2007 10:26
HSSP reference	 Entry file missing. -	Nucleic acids only Administrator - 19/06/2007 10:26
PDB reference	 Entry file present. 28/02/1995 09:00	
PDBFINDER reference	 Entry file present. 21/06/2007 09:40	
PDBREDO reference	 Entry file present. 21/05/2007 01:25	
PDBREPORT reference	 Entry file present. 28/09/2004 03:04	
STRUCTUREFACTORS reference	 Entry file missing. -	

Figure 6: Example output for database entries present and missing for PDBID 101D.

Conclusion

The WhyNot project was started to add quality control to the PDB and PDB related databases and to solve the problems occurring when a user is referred to non-existing database entry. To accomplish this all PDB and PDB-related database entries are indexed and stored in a relational database. Based on an evaluation of database inconsistencies a database expert is consulted to either resolve or explain the database inconsistencies, the results of which are stored in the RDB. A Web server located at <http://www.cmbi.ru.nl/whynot/> makes the information gathered above available to a wide audience.

Future work

Due to the project's limited scope and time span WhyNot is currently limited to the above described functionality. There are however a number of ways in which the current functionality can be extended or improved upon. We will briefly name and explain a number of possible extensions to the WhyNot system which might be implemented at a later point in time:

1. Web service
The information in the RDB is only accessible via the WhyNot Web server. Adding a Web service would allow external applications to access this information directly rather than referring users to the WhyNot Web server.
2. Additional databases
Additional databases could be added as long as they use the PDBID for identification and are stored in a manner similar to the databases already stored in WhyNot.
3. Web administration
The current manner for database experts to add comments to WhyNot requires them to use the Commenter command line application. Adding a web interface to manage comments is highly preferential in this day and age.
4. PDBID collection refinement
Adding the option for users to self define a collection of PDBIDs and/or refine this collection based on a certain set of properties could make it easier to get information about a custom collection of PDBIDs.
5. Additional properties in Web server
Currently the RDB, DAO, Crawler and Commenter all support multiple properties per database entry. However the Web server does not yet show this additional information. Adding this to the Web server would allow us to take advantage of this functionality.
6. Store reason for failed entry creation automatically
The Commenter command line application is able to read all the comment files in a directory. By altering the external applications to write comment files to a preset directory when failing to compute a PDB-related database entry, one can automate adding the reason why a database entry could not be created rather than having to rely on a database expert to enter this information.

References

1. *The Protein Data Bank. A computer-based archival file for macromolecular structures.*
FC Bernstein, TF Koetzle, GJ Williams, EF Meyer Jr, MD Brice, JR Rodgers, O Kennard, T Shimanouchi and M Tasumi.
European Journal of Biochemistry, Vol 80, 319-324.
2. *UniProt: the Universal Protein knowledgebase*
Rolf Apweiler*, Amos Bairoch¹, Cathy H. Wu², Winona C. Barker³, Brigitte Boeckmann¹, Serenella Ferro¹, Elisabeth Gasteiger¹, Hongzhan Huang², Rodrigo Lopez, Michele Magrane, Maria J. Martin, Darren A. Natale², Claire O'Donovan, Nicole Redaschi¹ and Lai-Su L. Yeh³
Nucleic Acids Research, 2004, Vol. 32, Database issue D115-D119
3. *Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features.*
Biopolymers, (1983) Dec; 22(12):2577-637.
PMID: 6667333; UI: 84128824.
4. *Database of homology-derived protein structures.*
Sander C., Schneider R.
Proteins, (1991) 9:56-68 .
5. *The PDBFINDER database: A summary of PDB, DSSP and HSSP information with added value.*
R.W.W. Hooft, C. Sander and G. Vriend.,
CABIOS (1996) 12, 525-529.
6. *Errors in protein structures.*
R.W.W. Hooft, G. Vriend, C. Sander, E.E. Abola,
Nature (1996) 381, 272-272.
7. *Fully Communication Oriented NIAM.*
Bakema GP, Zwart JPC, Lek H van der (1994)
Conf Working papers, Albuquerque, NM USA, pp L1–35

Appendix A: Database model

The following is a SQL script that will create the users, tables, functions and foreign keys that make up the RDB used by WhyNot. This script was created to be run on PostgreSQL.

```
/* Drop database and users */
/* - might fail if non-existent */
DROP DATABASE whynot;
DROP USER whynot_owner;

/* Create users whynot_owner and whynot_viewer with */
CREATE USER whynot_owner WITH SYSID 105 PASSWORD 'password' CREATEDB NOCREATEUSER VALID
UNTIL 'infinity';

/* Create database whynot with owner whynot_owner */
CREATE DATABASE whynot WITH OWNER = whynot_owner TEMPLATE = template0 ENCODING =
'UNICODE';
REVOKE ALL ON DATABASE whynot FROM PUBLIC;
GRANT ALL ON DATABASE whynot TO whynot_owner;
GRANT ALL ON DATABASE whynot TO postgres;
GRANT ALL ON DATABASE whynot TO tbeek;

/* All SQL statements past this comment should be executed
 * - as user whynot_owner
 * - on database whynot
 */

/* Drop default schema public */
DROP SCHEMA public CASCADE;
/* Create schema public with owner whynot_owner */
CREATE SCHEMA public AUTHORIZATION whynot_owner;
REVOKE ALL ON SCHEMA public FROM PUBLIC;
GRANT ALL ON SCHEMA public TO whynot_owner;
GRANT ALL ON SCHEMA public TO postgres;
SET search_path TO public;

CREATE TABLE AUTHORS (
    NAME VARCHAR(50) NOT NULL,
    EMAIL VARCHAR(50) NOT NULL,

    PRIMARY KEY (NAME)
);

CREATE TABLE COMMENTS (
    COMID SERIAL NOT NULL,
    AUTHOR VARCHAR(50) NOT NULL,
    COMMENT TEXT NOT NULL,
    TIMESTAMP TIMESTAMP NOT NULL,

    PRIMARY KEY (COMID)
);

CREATE TABLE DATABASES (
    NAME VARCHAR(50) NOT NULL,
    REGEX VARCHAR(50) NOT NULL,
    PDBIDOFFSET INTEGER NOT NULL,
    REFERENCE TEXT NULL,
    FILELINK TEXT NULL,

    INDBALL INTEGER NULL,
    INDBCORRECT INTEGER NULL,
    INDBINCORRECT INTEGER NULL,
    NOTINDBALL INTEGER NULL,
    NOTINDBCORRECT INTEGER NULL,
    NOTINDBINCORRECT INTEGER NULL,

    PRIMARY KEY (NAME)
);

CREATE TABLE ENTRIES (
```

```

    DATABASE VARCHAR(50) NOT NULL,
    PDBID VARCHAR(4) NOT NULL,
    FILEPATH VARCHAR(200) NOT NULL,
    TIMESTAMP TIMESTAMP NOT NULL,

    PRIMARY KEY (PDBID,DATABASE)
);

CREATE TABLE ENTRYPROPERTIES (
    DATABASE VARCHAR(50) NOT NULL,
    PDBID VARCHAR(4) NOT NULL,
    PROPERTY VARCHAR(50) NOT NULL,
    BOOLEAN BOOLEAN NOT NULL,

    PRIMARY KEY (PDBID, PROPERTY, DATABASE)
);

CREATE TABLE ENTRYPROPERTYCOMMENTS (
    DATABASE VARCHAR(50) NOT NULL,
    PDBID VARCHAR(4) NOT NULL,
    PROPERTY VARCHAR(50) NOT NULL,
    COMMENT INTEGER NOT NULL,

    PRIMARY KEY (PDBID, PROPERTY, COMMENT, DATABASE)
);

CREATE TABLE PDBIDS (
    PDBID VARCHAR(4) NOT NULL,

    PRIMARY KEY (PDBID)
);

CREATE TABLE PROPERTIES (
    NAME VARCHAR(50) NOT NULL,
    EXPLANATION TEXT,

    PRIMARY KEY (NAME)
);

/* Alter all tables to add foreign keys and implicit indices */
ALTER TABLE COMMENTS
    ADD FOREIGN KEY (AUTHOR)
    REFERENCES AUTHORS (NAME);

ALTER TABLE ENTRIES
    ADD FOREIGN KEY (DATABASE)
    REFERENCES DATABASES (NAME);

ALTER TABLE ENTRIES
    ADD FOREIGN KEY (PDBID)
    REFERENCES PDBIDS (PDBID);

ALTER TABLE ENTRYPROPERTIES
    ADD FOREIGN KEY (DATABASE)
    REFERENCES DATABASES (NAME);

ALTER TABLE ENTRYPROPERTIES
    ADD FOREIGN KEY (PROPERTY)
    REFERENCES PROPERTIES (NAME);

ALTER TABLE ENTRYPROPERTIES
    ADD FOREIGN KEY (PDBID)
    REFERENCES PDBIDS (PDBID);

ALTER TABLE ENTRYPROPERTYCOMMENTS
    ADD FOREIGN KEY (DATABASE, PDBID, PROPERTY)
    REFERENCES ENTRYPROPERTIES (DATABASE, PDBID, PROPERTY);

ALTER TABLE ENTRYPROPERTYCOMMENTS
    ADD FOREIGN KEY (COMMENT)
    REFERENCES COMMENTS (COMID);

```

```

/* Create explicit alternative indices on selected tables for increased performance */
CREATE UNIQUE INDEX comments_altkey ON COMMENTS USING btree (comment);
CREATE UNIQUE INDEX entries_altkey ON ENTRIES USING btree (database, pdbid);

/* Create functions for often reused queries */
CREATE FUNCTION PDBIDsInDB(varchar(50)) RETURNS SETOF PDBIDS AS '
    SELECT pdbid
    FROM entries
    WHERE database = $1
    ORDER BY pdbid;
' LANGUAGE SQL;

CREATE FUNCTION PDBIDsNotInDB(varchar(50), varchar(50)) RETURNS SETOF PDBIDS AS '
    SELECT pdbid
    FROM pdbids
    WHERE pdbid IN (
        SELECT *
        FROM PDBIDsInDB($2)
    EXCEPT
        SELECT *
        FROM PDBIDsInDB($1))
    ORDER BY pdbid;
' LANGUAGE SQL;

CREATE FUNCTION PDBIDsNotInDBWithComment(varchar(50),varchar(50)) RETURNS SETOF PDBIDS AS '
    SELECT * FROM PDBIDsNotInDB($1,$2)
    WHERE pdbid IN (
        SELECT DISTINCT pdbid
        FROM entrypropertycomments
        WHERE database = $1
        AND property = \'Exists\')
    ORDER BY pdbid;
' LANGUAGE SQL;

CREATE FUNCTION PDBIDsNotInDBWithoutComment(varchar(50),varchar(50)) RETURNS SETOF PDBIDS
AS '
    SELECT * FROM PDBIDsNotInDB($1,$2)
    WHERE pdbid NOT IN (
        SELECT DISTINCT pdbid
        FROM entrypropertycomments
        WHERE database = $1
        AND property = \'Exists\')
    ORDER BY pdbid;
' LANGUAGE SQL;

CREATE FUNCTION PDBIDsInDBNotObsolete(varchar(50), varchar(50)) RETURNS SETOF PDBIDS AS '
    SELECT pdbid
    FROM pdbids
    WHERE pdbid IN (
        SELECT *
        FROM PDBIDsInDB($1)
    EXCEPT
        SELECT *
        FROM PDBIDsNotInDB($2,$1))
    ORDER BY pdbid;
' LANGUAGE SQL;

CREATE FUNCTION DATABASESTATS(varchar(50),varchar(50)) RETURNS INTEGER[] AS '
    SELECT ARRAY [
        (select count(*) from PDBIDsInDB($1))::INTEGER,
        (select count(*) from PDBIDsInDBNotObsolete($1,$2))::INTEGER,
        (select count(*) from PDBIDsNotInDB($2,$1))::INTEGER,
        (select count(*) from PDBIDsNotInDB($1,$2))::INTEGER,
        (select count(*) from PDBIDsNotInDBWithComment($1,$2))::INTEGER,
        (select count(*) from PDBIDsNotInDBWithoutComment($1,$2))::INTEGER];
' LANGUAGE SQL;

```

Table 3: Database model

Appendix B: JDBC.XML example

The following is an example of the “*jdbc.xml*” file that is used by both the Crawler and the Commenter to read the settings need to have the DAO connect to the RDB. These settings might have to be altered to be able to connect to any existing databases.

```
<?xml version='1.0'?>
<database>
  <connection-settings>
    <driver>org.postgresql.Driver</driver>
    <protocol>jdbc:postgresql:</protocol>
    <host>127.0.0.1</host>
    <port>5432</port>
    <name>whynot</name>
    <user>whynot_owner</user>
    <pass>password</pass>
  </connection-settings>
</database>
```

Table 4: Example jdbc.xml

Appendix C: Crawler

The following is an example “*databases.xml*” file that is used by the Crawler to read the available databases and the format in which the entries in these databases are stored. These settings might also have to be altered to fit the actual situation.

```
<?xml version='1.0'?>
<databases>
  <database>
    <name>DSSP</name>
    <reference>http://swift.cmbi.ru.nl/gv/dssp/</reference>
    <filelink>http://mrs.cmbi.ru.nl/mrs-3/entry.do?db=dssp&id=</filelink>
    <regex>.*[\d\w]{4}\.dssp</regex>
    <offset>0</offset>
  </database>
  <database>
    <name>HSSP</name>
    <reference>http://swift.cmbi.kun.nl/gv/hssp/</reference>
    <filelink>http://mrs.cmbi.ru.nl/mrs-3/entry.do?db=hssp&id=</filelink>
    <regex>.*[\d\w]{4}\.hssp</regex>
    <offset>0</offset>
  </database>
  <database>
    <name>PDB</name>
    <reference>http://www.pdb.org/</reference>
    <filelink>http://mrs.cmbi.ru.nl/mrs-3/entry.do?db=pdb&id=</filelink>
    <regex>.*pdb[\d\w]{4}\.ent(\.Z)?</regex>
    <offset>3</offset>
  </database>
  <database>
    <name>PDBFINDER</name>
    <reference>http://swift.cmbi.ru.nl/gv/pdbfinder/</reference>
    <filelink>http://mrs.cmbi.ru.nl/mrs-
3/entry.do?db=pdbfinder2&id=</filelink>
    <regex>.*PDBFIND2\..TXT</regex>
    <offset>-1</offset>
  </database>
  <database>
    <name>PDBREPORT</name>
    <reference>http://swift.cmbi.ru.nl/gv/pdbreport/</reference>
    <filelink>http://www.cmbi.ru.nl/cgi-bin/nonotes?PDBID=</filelink>
    <regex>.*pdbreport.*[\d\w]{4}</regex>
    <offset>0</offset>
  </database>
  <database>
    <name>PDBREDO</name>
    <reference>http://swift.cmbi.ru.nl/pdb_redo/</reference>
    <filelink>http://swift.cmbi.ru.nl/pdb_redo/</filelink>
    <regex>.*pdb_redo.*[\d\w]{4}</regex>
    <offset>0</offset>
  </database>
  <database>
    <name>STRUCTUREFACTORS</name>

    <reference>http://www.google.com/search?q=pdb+structure+factors</reference>
    <filelink>http://www.google.com/search?q=pdb+structure+factors+</filelink>
    <regex>.*r[\d\w]{4}sf\.ent</regex>
    <offset>1</offset>
  </database>
</databases>
```

Table 5: Example databases.xml

The following is the output given by the Crawler application when run with the argument --help.

```
$ java -jar WhyNot-crawler.jar --help
Usage: whynot-crawler [OPTION]...
Collects, Stores and Validates information about Molecular Structure Data Files
```

in a Relational Database.

Mandatory arguments to long options are mandatory for short options too.

--help	display this help and exit
--version	output version information and exit
-t, --type REGEX	regular expression describing the databases (defined in databases.xml) to index (default is .* (all databases consecutively))
-s, --source database / directory / pdbfinderfile / none	define the information source for the entries. information source can be either the RDB, a directory, a pdbfinderfile or none. (default is directory)
-p, --path DIRECTORY / FILE	path to the directory containing the entries, OR path to the file containing entries (default is . (local directory))
-nofilestorage	do not store entries in textfiles.
-nodbstorage	do not store entries in relational database.
-nodbvalidation	do not validate entries in relational database.

Report bugs to <timtebeek@gmail.com>.

Table 6: Usage instructions

The following script gives shows a number of ways in which the Crawler can be started consecutively. This script can be executed on a weekly basis.

```
cd /home/stage/tbeek/Desktop/whynot;
java -jar whynot-crawler.jar -t PDB -s directory -p /mnt/cmbi8/raw/pdb/;
java -jar whynot-crawler.jar -t DSSP -s directory -p /mnt/cmbi8/uncompressed/dssp/;
java -jar whynot-crawler.jar -t HSSP -s directory -p /mnt/cmbi8/uncompressed/hssp/;
java -jar whynot-crawler.jar -t PDBFINDER -s pdbfinderfile -p
/mnt/cmbi8/raw/pdbfinder/PDBFIND.TXT;
java -jar whynot-crawler.jar -t PDBREPORT -s directory -p /mnt/cmbi8/raw/pdbreport/;
java -jar whynot-crawler.jar -t PDBREDO -s directory -p /mnt/cmbi8/raw/pdb_redo/;
java -jar whynot-crawler.jar -t STRUCTUREFACTORS -s directory -p
/mnt/cmbi8/srs/structure_factors/;
read -p "Press any key to continue..."
```

Table 7: Example bash script

The following is an example of the output provided by the Crawler application.

```
$ java -jar whynot-crawler.jar -t PDB -s directory -p /mnt/cmbi8/raw/pdb/;
--- Starting new run [06/21/2007, 13:16:10]

### WhyNot - Crawler v1.0 #####
- Regex for database(s):      PDB
- Information Source:        directory
- Path to directory or file:  /mnt/cmbi8/raw/pdb/
- Store results to file:     true
- Store results to database: true
- Validate entries in database: true

[13:16:10] ### PDB: ###
[13:16:11] Processing files:
[13:17:11] Completed: 6618 files
[13:18:11] Completed: 26168 files
[13:19:07] Done: 44200 succeeded, 0 failed, 44200 files total.
[13:19:07] Verifying entries in database exist locally:
[13:20:00] Done: 44202 succeeded, 0 failed, 44202 files total.
[13:20:00] Cleaning up database (cache/vacuum/analyze): This may take a while...
[13:22:29] Execution completed.
```

Table 8: Example output for indexing PDB database

Appendix D: Commenter

The following is the output given by the Commenter application when run with the argument `--help`.

```
$ java -jar whynot-commenter.jar --help;
Usage: whynot-commenter [OPTION]...
Stores comments about properties of Molecular Structure Data Files in a
Relational Database.

Mandatory arguments to long options are mandatory for short options too.
  --help                display this help and exit
  --version              output version information and exit

  -p, --path DIRECTORY / FILE      path to the directory containing textfiles, OR
                                      path to the textfile containing comments
                                      (default is: ./comments/ (all textfiles))
                                      textfiles should have the following format:

PDBID          : 1XJ9

                                      Database      : DSSP
                                      Property       : Exists
                                      Boolean        : false
                                      Comment        : Not enough amino acids
                                      //

Report bugs to <timtebeek@gmail.com>.
```

Table 9: Usage instructions

The following script gives shows a way in which the Commenter application is instructed to read a directory for comment files. This script can be executed on a weekly basis.

```
cd /home/stage/tbeek/Desktop/whynot;
java -jar whynot-commenter.jar -p ./comments/;
read -p "Press any key to continue..."
```

Table 10: Example bash script

The following is an example of the output provided by the Commenter application.

```
$ java -jar whynot-commenter.jar -p ./comments/;
--- Starting new run [06/21/2007, 13:46:04]

### WhyNot - Batch Commenter v1.0 #####

- Path to file: ./comments/

[13:46:04] ### hssp-2007-april.txt: ###
[13:46:04] Processing comments:
[13:46:26] Done: 1920 succeeded, 0 failed, 1920 files total.

[13:46:26] ### dssp-2007-april.txt: ###
[13:46:26] Processing comments:
[13:46:51] Done: 1920 succeeded, 0 failed, 1920 files total.

[13:46:51] Cleaning up database (cache/vacuum/analyze): This may take a while...
[13:49:18] Execution completed.
```

Table 11: Example output for parsing all comment files in a directory