GENDB
A second generation genome annotation system

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
der Universität Bielefeld
vorgelegte
Dissertation

von

Folker Meyer 29. Oktober 2001
1. Introduction
   1.1. A shift of paradigms from analysing genes to analyzing genomes 2
   1.2. Data from genome analysis ........................ 3
       1.2.1. Sequence data .................................. 3
       1.2.2. Transcriptomics data .......................... 5
       1.2.3. Proteomics data ............................... 6
       1.2.4. Metabolomics data ............................ 7
   1.3. Genome annotation systems ............................ 8
       1.3.1. Existing genome annotation systems .......... 9
       1.3.2. Other Genome Information Systems ............ 16

2. Design of a new genome annotation system .......................... 17
   2.1. New challenges for genome annotation systems ............ 17
Contents

2.2. Software engineering design criteria ................................................. 18
2.3. The need for conservation of resources ............................................ 19
  2.3.1. Computational resources ....................................................... 19
  2.3.2. Storage resources ............................................................... 19
  2.3.3. Human resources ............................................................... 19
2.4. Other requirements ................................................................. 20
2.5. Overview of the GENDB system ..................................................... 21
  2.5.1. Data storage .............................................................. 22
  2.5.2. Frontends – The user interfaces ............................................ 27
  2.5.3. Wizards – automation of complex repetitive tasks .................... 29
  2.5.4. Scheduler – computing observations ....................................... 31
  2.5.5. The application programmer’s interface ................................... 38
  2.5.6. The flow of information in GENDB ......................................... 39
  2.5.7. Conventions used in GENDB ............................................... 42

3. Implementation of GENDB .............................................................. 43
  3.1. Implementation details ......................................................... 43
    3.1.1. Achieving modularity and extensibility .................................. 43
    3.1.2. Use of object oriented approach ......................................... 44
    3.1.3. Use of a DBMS ............................................................. 45
    3.1.4. Using objects with a RDBMS .............................................. 47
    3.1.5. Class and object methods generated by O2DBI ....................... 51
    3.1.6. Functions automatically generated by O2DBI .......................... 52
  3.2. The Application Programmers Interface ....................................... 54
    3.2.1. Autogenerated functions for all object types ......................... 54
    3.2.2. Not automatically generated methods for all object types .......... 67
## 4. Results

4.1. User interface manual ........................................ 76
   4.1.1. Uploading data and gene prediction ................. 77
   4.1.2. Viewing the Contig .................................... 78
   4.1.3. Viewing observations ................................ 81
   4.1.4. Annotation ............................................. 85
   4.1.5. Pathway viewer ....................................... 87
   4.1.6. Wizards ............................................... 88
   4.1.7. The virtual 2D-Gel ................................... 91

4.2. The web interface of GENDB .................................. 93

4.3. Analysis of genomic data with GENDB ..................... 98
   4.3.1. Analysis of the complete genome of C. glutamicum ... 98
   4.3.2. Analysis of several S. meliloti contigs ............ 100
   4.3.3. Analysis of several smaller contigs with GENDB ... 100
   4.3.4. Analysis of the B. subtilis, E. coli and M. tuberculosis genomes ................................................. 101
   4.3.5. Analysis of a mycoplasma genome .................... 101

4.4. Summary ....................................................... 104

## 5. Discussion

5.1. Availability and Future work for GENDB .................... 110
   5.1.1. Availability ............................................ 110
   5.1.2. Future work ............................................ 110

## A. Roadmaps for Installation and future development

A.1. Installation Roadmap for a GENDB system .................. 113
Contents

A.1.1. Step: 1 – Installing the necessary software . . . . . . . 113
A.1.2. Step: 2 – Installation of GENDB . . . . . . . . . . . . 114
A.2. Roadmap for future GENDB versions. . . . . . . . . . . 115

B. Source Code 117
B.1. Source code for GENDB.pl . . . . . . . . . . . . . . . . . . 117
B.2. Code generated for the contig object by O2DBI . . . . . . 127

Bibliography 138
Acknowledgements

The author wishes to thank Prof. Dr. Robert Giegerich, Prof. Dr. Alfred Pühler and Dr. Jörn Kalinowski for their scientific guidance.

Many thanks also to the following people for their work on the GENDB system: Burkhard Linke who implemented a lot of the functionality in the _add modules. Oliver Rupp who implemented the new user interface. Dipl.-Inform. Jörn Clausen who wrote the O2DBI package in the context of this project. Dipl.-Inform. Torsten Kasch, our System administrator, who kept the machines and databases working. Dipl.-Inform. Alexander Goesmann who made numerous changes to the web interface and together with Dipl.-Inform. Martin Haubrock implemented the metabolic pathway component (PathViz).
Abstract

The advent of new high throughput technologies opens the road towards a new era of genome analysis. Data from high throughput sequencers, chip based RNA expression analysis and proteome analysis systems create the need for software systems to support new kinds of analysis and data.

At the same time the focus of molecular research shifted from the analysis of single genes to the analysis of whole genomes, multiple high throughput sources of data are routinely used. Yet there is a shortage of software systems that help store, integrate and analyse the wealth of information now available.

We describe the development of a new genome annotation system (GENDB) based on a relational database system and object oriented technology that helps with the analysis of this data. GENDB significantly reduces the storage and compute overhead of existing systems, while offering more flexibility. The ability to integrate new kinds of data and new methods of analysis is one of the primary design targets for GENDB. The GENDB system has been successfully used in a number of genome projects.
CHAPTER 1

Introduction

1.1. A shift of paradigms from analysing genes to analyzing genomes

The new sequencing technologies that have become available in the last decade introduced a shift of paradigms in molecular biology. While researchers traditionally focused on single genes or gene clusters, now whole genomes or large genomic regions started to become available. With the advent of the *S. cerevisiae* genome in 1996 [CJB+97] a new area of research was opened, the analysis of large amounts of genomic data.

From the sheer volume of information in genomic data – the genomes starting to become available contained thousands of genes – the need for automation became immediately clear to researchers. In the relatively young field of bioinformatics, a number of systems for the automatic analysis of the new data were developed.
1. Introduction

1.2. Data from genome analysis

Initially, the DNA sequence data was all that was available in large quantities, but new technologies in other areas allowed them to scale up towards the analysis of whole genomes as well.

1.2.1. Sequence data

The sequencing technology introduced by Maxam and Gilbert [MG77] granted researchers access to genomic data.

The genomic data consists of the primary DNA (Desoxyribonucleic Acid) sequence of the genome and data derived from that genome using bioinformatics tools.
1. Introduction

(a) Step1: Multiple chromatogram (SCF) files as visualized by the trev [SBB98] program.

(b) Step2: Creation of a consensus sequence from the SCF files. Visualized by gap4 [SBB98].

(c) Step3: Analysis of the consensus sequence using a genome annotation system. (Picture taken from Magpie [GS96]).

Figure 1.1.: The flow of data from sequencer to genome annotation system.

Once the primary sequence is determined, the remaining information is added \textit{in silico} by a collection of bioinformatics tools and databases.

The most important piece of information is the data on sequence similarity that allows the classification of the putative genes in the new organism based on their similarity to other genes already in the sequence databases.

Figure 1.1 shows the flow of information and data when analyzing a genome. First, thousands of sequence files are obtained from the sequencing automata,
then the files are assembled in one or more contiguous stretches of DNA, which are then analyzed with a genome annotation system.

1.2.2. Transcriptomics data

With the microarray technology developed by Brown et al. [SSDB95, SDDP95, BB99] in 1995 it became possible to measure the amount of a specific mRNA in a given cell, thus adding a new kind of information to the existing data. Using a single chip or glass slide it became possible to measure the expression level of thousands of genes in a matter of hours.

![Typical microarray showing different expression levels](image)

A similar method was developed by Affymetrix [Inca]. In addition to the hybridization based methods, a sequencing based method, the serial analysis of gene expression (SAGE) developed by Velculescu et al. [VZVK97] in 1995 can be used to obtain similar data. SAGE uses a tagged DNA approach to measure the level of gene expression. Figure 1.2 shows the results of a microarray experiment.
1. Introduction

All methods can be employed to detect changes in the level of expression of individual genes after some changes in the cell. As a result the researcher can now study the changes in the level of expression for all genes in the genome instead of focussing on a single gene that is affected by some changes in the cell.

Expression data can be utilized to gather information on gene regulation and metabolic pathways, e.g. for the \textit{S. cerevisiae} genome, a systematic effort is under way to establish the rules that govern the expression of genes in the living yeast cell. Other uses of microarray technology include data mining for regulatory elements as demonstrated by Vilo and Brazma \textit{et al.} [VBRU00]. Zien \textit{et al.} [ZKZL00] have shown the usefulness of integrating microarray data with other genomic data.

1.2.3. Proteomics data

While the methods described above for measuring the level of mRNA in a cell provide a valuable tool to researchers, there are nevertheless some limitations to studying gene expression levels based on the mRNA amounts. As shown by Gygi \textit{et al.} [GRFA99], the level of protein expression cannot be predicted by the abundance of mRNA. Protein function is subject to many posttranslational modifications and the dynamic process of protein degradation has a significant influence on the amount of active protein in the cell. Therefore, in addition to measuring the abundance of mRNA in the cell, the protein expression, modification and activity in the cell have to be measured. Two techniques have become closely associated with analyzing the proteome of a cell:

- Two-Dimensional (2D) Gel Electrophoresis
  Using the 2D Gel method, proteins are separated in the first dimension according to their isoelectric points, in the second dimension according to their mass [O’F75]. Figure 1.3 on the following page shows a 2D gel from \textit{C. glutamicum}. 
1. Introduction

- Mass Spectrometry (MS)

Since the resolution of 2D gels is limited, a technique with higher resolution is required. Mass spectrometry provides sequence spectra for proteins in the low-femtomole or even attomole region [Jam01], thus providing a much higher resolution than 2D gels.

![Figure 1.3.: A 2D gel from the C. glutamicum genome project.](image)

1.2.4. Metabolomics data

Using the data shown so far, the analysis of the metabolic pathways of an organism has become feasible. Databases like KEGG [KG00] have become increasingly important in the last few years. KEGG contains pictures for the pathways known so far. The researcher can select a specific pathway and highlight a set of proteins in that pathway. The system offers a user friendly
1. Introduction

interface but unfortunately no programmer’s interface which would enable automation of queries.

Figure 1.4 shows a pathway as visualized by the KEGG [KG00] system.

![Image: The lysine biosynthesis pathway as shown by KEGG [KG].]

1.3. Genome annotation systems

Definition of a genome annotation system

A genome annotation system provides the software infrastructure necessary to compute, store and visualize genomic data for a given genome. In addition, the genome annotation system needs to accommodate user analysis and interpretations.
1. Introduction

Definitions of terms

- observations
  Information derived from the primary sequence data using various bioinformatics tools is referred to as observations. Observations need to be stored in a format that facilitates the automated analysis and quintessentially drawing of interferences from the data.

- annotations
  The interpretation of the observations is referred to as annotation.

1.3.1. Existing genome annotation systems

Starting in 1995, a number of genome annotation systems were developed by various groups.

In 1995, the Magpie [GS96] system by Gaasterland et al. was immediately followed by the Genequiz [ABL99] system developed at the EBI by Sander et al. Later, the Pedant system [MFG00] was developed at MIPS by Mewes et al. to help with the analysis of the S. cerevisiae genome.

All systems perform the task of extracting possible genes (referred to as ORFs – open reading frames)\(^1\) and try to assign meaning to each ORF by automatically comparing it to a number of sequence databases and running a number of bioinformatics tools. All the information is then presented to the user with a some kind of frontend that allows navigation and, to some extent, searching. To a varying degree, the system store results of analyses performed by the user, i.e. human interpretation of the homology data computed automatically, for individual ORFs.

Because of the immense commercial interest in this area, none of the systems are in the public domain. The author has access to results (HTML documents) from Pedant and Genequiz, less than one workday with the successor to the Genequiz system (BioScout) and about 1.5 years experience

---

\(^{1}\)The original Pedant and Genequiz systems required the ORFs to be input by the user. A task easily automated by using one of the numerous gene prediction tools.
1. Introduction

with Magpie. An elaborate comparison of Magpie, Genequiz and Pedant was done in [Scz98], reflecting the state of 1998.

The immense commercial interest in this area makes it almost impossible to perform a scientific evaluation of existing tools.

Analysis of the various genome annotation systems

The systems mentioned above have been used to analyse a large number of genomes. They provide the user with a valuable tool to assign functions to individual genes and gain insight into the genome.

The remainder of this section describes the main characteristics of the various systems and also aims at a comparison.

- Web Frontend
  All systems provide a web frontend in form of precomputed data stored in HTML [BL92] documents that are stored as flat files. Java [GA96] or Javascript [Net] applications provide information and control elements that are not available in HTML.

  All systems provide tables that offer an overview of the ORFs found for a contig. Figure 1.6 on page 12 shows the table generated by Magpie for the $pCM1$ plasmid from *C. glutamicum*. 
1. Introduction

Figure 1.5.: The contig as represented by Magpie.

Figure 1.5 shows the graphical contig representation generated by Magpie.
1. Introduction

Figure 1.6.: The table of ORFs generated for pCM1 by Magpie.

- No well defined storage component

In their original implementations, all of the systems used a hierarchy of flat files to store the various bits of information. Some systems relied on HTML documents as the sole form of information storage (Pedant and Genequiz), others (Magpie) used Sicstus Prolog [Sah91] facts files to store some information and a collection of flat files and HTML documents to store the rest.

To the authors knowledge, both the original Pedant and Genequiz directly produce HTML output, no other means of storing information is present in the system. In both Pedant and Genequiz, data is input in form of one or more sequences, a number of computations are started, and a set of HTML documents is generated.
1. Introduction

![Diagram of data flow in Pedant and Genequiz](image1)

Figure 1.7.: The flow of data in Pedant and Genequiz. There is no reasoning component.

The Magpie system uses an intermediate stage where some of the information that is computed is stored in form of Sicstus Prolog [Sah91] fact files, but some information is output directly into HTML documents.

![Diagram of data flow in Magpie](image2)

Figure 1.8.: The flow of data in Magpie. The red arrows indicate the flow of information bypassing the reasoning component.

- No API

None of the systems has a clearly defined programmer’s interface that would enable a systematic query or the extension of the system. Since all systems generate HTML output, parsing the HTML documents is usually the unfortunately the only way to extract information. In the case of Magpie, parsing the Prolog fact files is another viable option, but as shown in figure 1.8, some information is directly output into HTML documents bypassing the Prolog fact stage.
1. Introduction

- Static visualization
  Since all systems use precomputed – static – visualisations all are prone to run into problems once the original input data changes. In this event all visualisations have to be recomputed. During the possibly lengthy process of regenerating the HTML output, documents will be inconsistent, some showing up to date information, others showing obsolete information.

The successor of the Pedant system, the Pedant Pro system that is available as a commercial software package, now uses a MySql database system to store the information from which it generates HTML representations dynamically.

Figure 1.9 shows the result of changing the orientation of a contig in the Magpie system. The gene prediction algorithm is rerun if the contig data changes, resulting in ORF names changes.

![Diagram showing changes to ORF names after changing the orientation of a contig in Magpie.](image)

Since Magpie uses static HTML documents as output, a user retrieving information on the ORF a_002 (via the specific URL for this ORF,
1. Introduction

e.g. by using the web browser’s bookmark mechanism) would access a different ORF after the contig orientation is changed.

Unfortunately, a user accessing the information saved by the Magpie system for that open reading frame is likely to run into even more trouble. The additional information (i.e. Blast results, etc.) computed by the system for ORF a_002 is invalid after changes to the contig. It refers to a different part of the contig. All the information for that contig needs to be recomputed. During the lengthy recomputation, the old information is still available to the user.

- Consumption of disk space
Since raw data, numerous results of computations, precomputed HTML documents, and various intermediate formats are kept on disk, a large number of files is generated and stored for each of the systems described above. At least some of the data is redundant, most of it will almost certainly remain unused. This creates output data in excess of 1 gigabyte per megabase of genome information input into the system.

- Consumption of CPU time
In their original implementation the systems described above relied on a single computer to perform all the computations necessary. Therefore, a single – very expensive – large computer was needed to compute all the information.

For the Magpie system, annotating a contig of cosmid size (approx. 40 kilobases) on a 4 CPU, 300MHz, 2GB RAM Sun Ultra 450 Workstation takes up to 2 weeks.\(^2\) For a genome that consists of up to 60 such cosmids this results in 120 weeks of computation. This computational effort is easily multiplied by the need to recompute information due to changing data or updated sequence databases that form the basis of the annotation.

All systems are non conservative with respect to their resources usage. If a single base in a contig of 2 Megabases changes, all the information has to be recomputed.

\(^2\)Depending on the number of tools that are set up to be used for the contig in question.
1. Introduction

- Systems are commercial and non “open source”
  All existing systems described in section 1.3.1 are either commercial or not publically available for other reasons (unknown to the author).
  Since the funding of most projects does not allow to purchase one of these systems, most genome projects have to develop their own genome annotation system, thus re-inventing the wheel over and over again.

1.3.2. Other Genome Information Systems

While genome annotation systems provide the researcher with a framework with which he/she can perform the annotation and store the relevant information, a number of systems provide similar functionality, in some cases overlapping some of the functions described for the genome annotation system.

Other systems provide the infrastructure to store information on a given genome but do not integrate the computation of results or subsystems that allow user input: AceDB [WAC98] and GIMS [PKH00]. A very simple system that allows genome visualization and annotation in the form of EMBL flat files is the artemis system [RPC00].

And some systems provide access to automatically computed results but do not allow the anaylsis of the results by the user: ENSEMBL [ENS].
2.1. New challenges for genome annotation systems

The new transcriptome, proteome and metabolome data described in section 1.2.2 provide a new challenge for the existing genome annotation systems. By integrating the results of the genome annotation with these new kinds of data, a better understanding of the respective organisms can be achieved.

This integration process will enable future experiments and future analysis. The experiments done by Zien et al. [ZKZL00] combining expression data and metabolic pathways demonstrate the need for data integration.
2. Design of a new genome annotation system

2.2. Software engineering design criteria

Since integrating new kinds of data and integrating new approaches to the analysis of data requires access to the information in the genome annotation system, there is a need for modular, extensible and open systems.

The systems described in section 1.3.1 were designed for a completely different situation, their output was destined for the human reader. Nowadays, the generation of human readable tables is no longer sufficient.

The systems lack a number of important properties necessary to meet the new demands for data integration. Neither an application programmer’s interface nor a well defined internal data representation exists for the systems described above. Therefore, we chose to implement a completely new system. Another reason for re-implementing was the lack of either extensibility or source code availability for the existing systems.

Also, the fast moving research in molecular biology and other areas requires frequent changes or addons to a genome annotation systems. Therefore, it should provide support for rapid changes of the system itself and the integration of new components. Any new system should rather provide a framework for the fast integration of new data and new research questions than a static system for addressing a well defined problem.

There is a need for a new system that provides the following features:

- abstract data representation
- an application programmer’s interface
- modularity
- extensibility
- an open source implementation accessible to other researchers

The choice of open source as a licensing model should enable researchers to use the framework to address their own problems and adapt the system to their data.
2. Design of a new genome annotation system

2.3. The need for conservation of resources

2.3.1. Computational resources

The computation of the data involved in a genome annotation takes a large amount of computational resources. For a genome of 5000 genes, approximately 15000 to 20000 Blast [AGM+90] runs are computed initially and a large portion of those are recomputed once the underlying sequence databases change.

Therefore, the genome annotation system should only compute a minimum set of observations for a given problem. In the Magpie system, once a single nucleotide changes in a genome, this invalidates all the information for a genomic region, in a worst case scenario. This leads to a massive waste of computational resources.

In addition the system should support distributed computing, thus obviating the need for a single large computer system.

2.3.2. Storage resources

The amount of data stored for a megabase of primary sequence data can well exceed one gigabyte. This starts to become a problem once many genomes are analysed on a single system. A large genome of 20 megabases would require up to 310 gigabytes of storage capacity. Therefore, a genome annotation system should store only a minimal set of data and at the same time provide access to a maximum of information.

A well designed data representation layer and the use of adequate storage technology – a database system – have to be employed to achieve these goals.

2.3.3. Human resources

The data in a genome annotation system is presented to the human annotator who provides his or her interpretation of the information generated by the
2. Design of a new genome annotation system

It should provide a well designed user interface, thus minimizing the time required for any particular task. Therefore, automated assistance for recurring operations should be provided. In addition to the user interface, the system should enable the developer to rapidly implement new features that help the human annotator perform his or her task.

2.4. Other requirements

There are a number of other requirements for a genome annotation system:

- **Portability**
  The system should not be vendor specific wherever possible. Minimal or no changes should be needed to port the system to as many other platforms as possible. To achieve this aim, the choice of the underlying operating system has to be made carefully. Currently, applications for some operating systems need to include a large amount of vendor specific code (i.e. Microsoft Windows) while other platforms (i.e. UNIX) are in the process of achieving cross platform compatibility.

- **Transparency**
  The system should be transparent. It should be clear who added what information and who made which changes to the primary data for what reason.

- **Use of adequate technology**
  A new system should not re-invent the wheel but instead use existing technology and existing standards. Designing a purpose built DBMS (database management system) makes little sense when well tested, well supported, and fast systems already exist. Therefore, existing software systems should be used whenever feasible.

- **Use of dynamic visualisation**
  To avoid the pitfalls associated with the use of precomputed visualisation, the system should contain a visualisation component that generates views of the data stored in the system on demand.
2. Design of a new genome annotation system

This not only increases the flexibility for the user, who can customize his or her output.

- Minimal costs
  The system should cost as little as possible in terms of both software licenses needed to install and run it and requirements for non standard hardware. For the software side, an attempt should be made to use little or no commercial subsystems, thus making the whole system affordable.

2.5. Overview of the GENDB system

From the criteria presented above we designed the GENDB system.

Of the different types of data described earlier, currently only genomics and metabolomics data are managed by the system. The development of proteomics and transcriptomics components is an ongoing project.

A schematic overview of the GENDB system is represented in figure 2.1 on the following page.
The O2DBI layer shown in blue is the glue that connects the data backend with the other parts of the system. All access from one component to the data in the storage component is via the O2DBI layer. The O2DBI layer also provides an application programmer’s interface for external applications or extensions to the system.

### 2.5.1. Data storage

Since we want GENDB to posses a user interface that dynamically creates view for the data in the system, an internal representation of the genome data
and a mechanism for storing and retrieving that data is essential. A relational database in combination with the O2DBI tool described in section 3.1.4 on page 47 is used to implement persistent data storage.

Figure 2.2 shows the role of objects and tables in GENDB. A single object may contain information from multiple tables. Using the application programmer’s interface, the tables are hidden from the programmers of user level applications.

We chose this approach to reduce complexity and facilitate the replacement of storage backends. Only library functions (i.e. code in Perl [WS91] modules) are allowed to contain direct references to tables or real SQL code. Should a change of the storage module become necessary, no single line
2. Design of a new genome annotation system

of code in the higher-level applications will have to be modified. Instead, only the Perl modules that form the API need to be updated accordingly.

The data model

In addition to the means to store and retrieve information from a database system, an internal data representation is needed. Only by providing abstract data types can an application programmer’s interface be implemented and the data used for reasoning. A simplified model of the data entities in GENDB is shown in figure 2.3.

Since the GENDB system was designed to replace an existing system (Magpie) we chose – for the first version – to implement only the features of that system. A future version (1.10) of GENDB will allow for the complete set of EMBL features to be added to a contig, thus making the ORF a special case.

Figure 2.3.: A simplified view of the GENDB data model.
2. Design of a new genome annotation system

As shown in figure 2.3 on the preceding page, ORFs, sequences (contigs), observations a.k.a facts (i.e. information deduced by various bioinformatics tools), and annotations (information added by human annotators) are the only sequence feature objects currently represented in GENDB.

For the objects shown in figure 2.3 on the page before, the following gives a brief overview of the information contained in the objects.

- **contigs**
  The sequences that are uploaded into the system. In addition to the sequence, the name and the position of the contig in a replicon are stored. In detail that is:
  - sequence name
  - sequence
  - the identity of left and right neighbor contigs
  - the overlap with the left and right neighbor contigs

- **ORFs**
  The positions of putative open reading frames are stored along with the information generated by the gene prediction algorithm.
  The sequences of the ORFs are not stored; they are extracted on demand from the sequence object.
  The following information is stored:
  - name
  - start
  - length
  - the identity of the respective sequence object

- **observations**
  The information computed for a given ORF is referred to as observations.
  The following information is stored:
2. Design of a new genome annotation system

– the identity of the respective ORF
– the tool and parameters used to compute the observation
– the description line of a database hit
– the score for a database hit
– the database identifier of the sequence hit in the database
– the unique identifier for the database
– the regions in the ORF and in the database entry that match

Any additional information can be recreated using the information stored for the observation.

• annotations

The information added by a human annotator is referred to as annotation.

The following information is stored:

– the name of the annotator
– the identity of the ORF being annotated
– any number of identities of observations relevant to this annotation
– the gene name
– the gene product
– the EC-number
– a description line
– a comment line

There is no gene object or table in GENDB. Instead, a gene is represented by an ORF combined with an annotation and possibly a number of observations.

The complete sources for the objects and the data model used in GENDB are listed in O2DBI syntax in the appendix. See section B.1 on page 117 for a complete listing.
2. Design of a new genome annotation system

2.5.2. Frontends – The user interfaces

A genome annotation system needs to provide a comprehensive and flexible user interface. Providing an internal representation and a data storage module facilitate the creation of interfaces that are adaptable to the user demand. Instead of relying on a single way of presenting the results, a number of different “views” can be created from the internal representation of the data.

At present we chose to implement two interfaces:

- A graphical user interface (GUI) (see figure 2.4) implemented in Perl/Tk [IS, Wal99] for the manipulation of data and

![Figure 2.4.: Overview of the graphical user interface.](image)
2. Design of a new genome annotation system

- a web interface (see figure 2.5) for read only access to the data over the Internet.

![Image of a genome annotation system](image)

**Figure 2.5.: Overview of the web interface.**

The web interface will be enhanced to provide the complete functionality of the GUI in a future version (1.10 or later).
2. Design of a new genome annotation system

The frontends are the primary user visible parts of the system. Careful planning of those in close interaction with the biologists using the system creates interfaces that facilitate human interpretation of the data in the system.

While the web frontend generates dynamic views of the data, a static version (which may be written to a CD-ROM) can be created with very little changes to the code. Both frontends are further described in section 4.1 and 4.2.

2.5.3. Wizards – automation of complex repetitive tasks

A number of complex tasks (e.g. searching for frameshifts, editing of the automatic ORF prediction) recur multiple times in a project. Wizards are software agents that guide the user through these tasks. By implementing these tools, the workload for the human annotators is reduced. At the same time, changes to the data are automatically done in a transparent and comprehensive manner. Each individual edit operation is treated as an annotation and thus recorded by the data storage component.

The following wizards exist today:

- ORF editor
  This wizard allows the editing of the gene prediction computed when uploading the contig into the system.

- frameshift editor
  The frameshift wizard allows for corrections in the contig sequence while maintaining the integrity of the data objects that exist for the contig.

- data entry editor
  Allows the uploading of contigs and running of gene prediction tools.

Other wizards are in preparation at the time of writing:

- the frameshift detection wizard
  This wizard automatically detects possible frameshifts and returns a list of regions for the contig that contain potential frameshifts.
2. Design of a new genome annotation system

- the automatic annotation wizard
  This wizard uses the observations to suggest a possible annotation, which can be exported or entered as a putative annotation into GENDB. Using this wizard requires to add a non human annotator to the system. All annotations performed by this wizard are easily identified as computer generated.

- the manual annotation wizard
  This wizard lists all ORFs not previously annotated and presents a summary of the observations computed for that ORF to the human annotator.

Figure 2.6 shows the role of wizards in the GENDB system.

Figure 2.6.: The role of wizards in GENDB.
2. Design of a new genome annotation system

2.5.4. Scheduler – computing observations

One important subsystem of a genome annotation system is the mechanism used to compute, evaluate, and store observations for the genome data. The subsystem of GENDB that performs this function is referred to as the scheduler.

The scheduler was designed to compute and store the observations necessary with a minimum overhead. It implements a FIFO (first in first out) scheduling algorithm. In addition, the scheduler allows the user to trigger a recomputation once external data has changed (e.g. new sequence databases have become available) and automatically recompute observations that refer to internal data when it has changed.

Usually, a genome annotation requires thousands of runs of tools. While a single computer could handle this workload, the price for such a machine is prohibitively high. Instead of buying a single large computer, we chose to use a cluster of workstations to run the analysis tools on. Running a single job at a time on a number of low-cost workstation class computers requires a system to distribute the jobs over the cluster and to collect the results. Systems that perform this function are referred to as “batch systems”. While the GENDB system allows for the easy integration of existing batch processing systems (e.g. NQS [NQS00], Codine [Inc00a] or LSF [Inc00b]), we have developed a light weight system of our own. The main reason for developing a light weight batch system within GENDB was the complexity of installation and runtime overhead of existing systems.

The GENDB scheduler consists of three components. The first component is a job table stored in the database using the established O2DBI mechanisms. The table is used to provide synchronisation for the scheduler components. Another component is the job_submitter that enters information on jobs to run into the database table.

The last part is responsible for running the jobs (GENDB_daemon.pl). Fig-

1The following document gives a overview of existing batch processing systems: http://www.cmpharm.ucsf.edu/~srp/batch/systems.html.
2. Design of a new genome annotation system

Figure 2.7 gives an overview of the components involved.

Figure 2.7.: Overview of the GENDB scheduler.

The job_submitter

Once the job_submitter is run, a list of jobs is created from the ORFs in the database for that observations have not yet been computed. If any new ORFs are found or old observations are to be recomputed, a set of jobs is entered into the job queue. Optionally, new observations can be computed for all ORFs in the database, overwriting all existing observations.
2. Design of a new genome annotation system

**Job locking in the GENDB scheduler**

One instance of the daemon is started on each node in the compute cluster. Each of these processes reads a single job from the job queue using the `fetchnextjob` method of the job class. Then the job is locked so that no other scheduler will try to execute this particular job. Upon completion of the job execution, the job is finished, the results are parsed, and stored in the database.

Figure 2.8 illustrates this process.

![Job locking in the GENDB scheduler](image)

*Figure 2.8.: Job locking in the GENDB scheduler.*

Figure 2.9 on the following page lists the methods that are implemented for
2. Design of a new genome annotation system

the job object.

- **newid**
  Return a new unique job id, used when inserting new jobs into the job table.

- **fetchnextjob**
  This class method retrieves the next unlocked job from the database using a FIFO scheduling strategy.

- **lock**
  Lock a job prior to processing it. This method returns immediately, if the job is already locked.

- **unlock**
  Unlock a job, upon completion or in case of problems.

- **delete**
  Delete a job from the job table.

Figure 2.9.: The additional methods for the job object.

**The GENDB_daemon**

The GENDB_daemon waits for new jobs, this is implemented avoiding busy waiting. As soon as a new job is available, the daemon locks the job, sets the UNIX process environment according to the job information and starts runtool.pl to process the job. After executing the job, the corresponding job table entry is marked as completed. When no job is available, the daemon does an unconditional wait, using no CPU resources.

runtool.pl is the main stage of execution. It locks the project-internal job database (orfstate) and starts the tools using the tool-specific run_job command.

Since each tool can have several features that distinguish it from other tools, the run_job routines that perform the actual start of each tool are part of
2. Design of a new genome annotation system

the tool specific code in GENDB.

**Storing observations in GENDB**

The observations that are generated by the tools are stored in the data storage component. Only a minimal set of information is stored for the observations in GENDB. If more detail is needed, that detail can be recomputed at very low computational cost.

The following information is stored in GENDB to represent observations.

```perl
$geneproject{'observation'} = {
    members => {
        'orf_id' => 'int',
        'description' => 'text',
        'toolresult' => 'text',
        'information' => 'int',
        'dbref' => 'text',
        'orffrom' => 'int',
        'orfto' => 'int',
        'dbfrom' => 'int',
        'dbto' => 'int',
        'tool_id' => 'int'
    },
    creator => [
        'orf_id'
    ]
};
```

Figure 2.10.: The observation information stored in GENDB

The *orf_id* is used to identify the corresponding open reading frame. The *description* is used to store the description line from a database hit. *dbref* and *tool_id* are used to describe the database and the tool used to compute this observation. In addition to this information, the location of the
2. Design of a new genome annotation system

match both in the database and in the ORF are stored (orffrom, orfto, dbfrom, dbto). The toolresult and information fields are used to store the scores returned by the application computing the observation. One field (toolresult) contains the result in text form as returned by the application and the other contains the result converted into bits (as in information theory according to Shannon [Sha48]) in the field information.

Extracting information in greater detail.

As shown above (see figure 2.10 on the page before), a database reference (dbref) and a tool (tool_id) are stored with the observations.
2. Design of a new genome annotation system

```perl
$geneproject{'tool'} = {
    members => {
        'name' => 'char(20)',
        'description' => 'text',
        'input_type' => 'int',
        'user_value' => 'int',
        'number' => 'int',
        'cost' => 'int',
        'dburl' => 'text',
        'dbname' => 'text',
        'executable_name' => 'text',
        'level1' => 'int',
        'level2' => 'int',
        'level3' => 'int',
        'level4' => 'int',
        'level5' => 'int',
        'helper_package' => 'text',
    },
    creator => [ 'name' ],
    constructors => [
        [ 'name' ]
    ];
}
```

Figure 2.11.: The tool information stored in GENDB

Using the `tool_id` as a reference, the name (dbname) and the URL (dburl) of the relevant sequence database are accessible for every observation in GENDB.

If the user needs more information than that stored in the observation entry, the following algorithm is applied:
2. Design of a new genome annotation system

INPUT: observation object

1. $orf = \text{observation}\rightarrow\text{init}\_\text{name}(\text{orf}\_\text{id})$
2. $\text{dbref} = \text{observation}\rightarrow\text{dbref}$ and $\text{tool}\_\text{id} = \text{observation}\rightarrow\text{tool}\_\text{id}$
3. $\text{tool} = \text{init}\_\text{id}(\text{tool}\_\text{id})$
4. $\text{database}\_\text{sequence} = \text{srsfetch}(\text{tool}\rightarrow\text{dburl}, \text{tool}\rightarrow\text{dbname}, \text{dbref})$
5. If necessary convert the database record into a suitable format.
6. run \(\text{tool}\rightarrow\text{executable}\_\text{name}\) (\(\text{database}\_\text{sequence}, \text{orf}\_\text{sequence}\))

Figure 2.12.: The algorithm to obtain detailed observation information.

This algorithm is implemented in the object method \(\text{tool} \rightarrow \text{run}\_\text{job}(\text{observation})\). The method is polymorphic. If no observation is given as a parameter, a new observation will be computed, parsed, and the relevant information is stored in the database system. The data retrieval from the sequence database is handled by the SRS [EA93, EUA96] system developed by Thure Etzold et al. at the EBI.

The whole process described in figure 2.12 takes less than 1.5 CPU seconds on a moderate workstation. Retrieving the same information from flat file storage using the HTTP protocol requires up to 5 seconds.

2.5.5. The application programmer’s interface

Providing both an internal representation of the data and a mechanism for efficient storage and retrieval opens the way towards an application programmer’s interface. The data structures created with O2DBI together with the functions that are added manually form the GENDB application programmer’s interface (API).
2. Design of a new genome annotation system

The components (e.g. wizards, frontends, scheduler) of the GENDB system use the API. That application programmer’s interface is described in sections 3.2. Details on O2DBI can be found in section 3.1.5 on page 51.

Using the application programmer’s interface, the system can be extended.

2.5.6. The flow of information in GENDB

The GENDB system is used to process data that originates from DNA sequencing machines. Once that data is saved in SCF format [DS92], it is run through a pipeline of applications in order to perform the following steps:

- Normalize traces from different machines and different runs.
- Remove any vector sequences and low quality regions.
- Assemble the sequences into one or more (consensus) sequences, referred to as contigs.

This pipeline is implemented using a variety of existing tools:

- cap3 – An DNA sequence fragment assembly program developed by Huang [HM99].
- phrap – A DNA sequence fragment assembly program developed at the University of Washington by Greene et al. [Greb].
- phred – A DNA basecalling program developed at the University of Washington by Greene et al. [EHWG98].
- crossmatch – A sequence comparison utility specialized in vector sequence detection by Greene et al. [Grea].
- vector_db – A vector sequence database that is part of Genbank [WCL+01]
2. Design of a new genome annotation system

Figure 2.13 on the following page summarises the flow of information in a genome project and the role of GENDB.

Within the GENDB system, a gene prediction tool is called, its results are parsed and the resulting open reading frames (ORFs) are added to the database. Then a set of tools selected by the user is run for each of these ORFs. Once that is done, the user interfaces (see section 4.1 on page 76) can be employed to interpret the results and add to a manual annotation. A list of tools used in the GENDB system is available in section 3.2.2 on page 75.
2. Design of a new genome annotation system

Figure 2.13.: The GENDB data flow.
2. Design of a new genome annotation system

2.5.7. Conventions used in GENDB.

A number of conventions are used throughout the GENDB system:

- Genes are annotated ORFs
  As described in section 2.5.1 on page 24, there is no gene object as such. Each ORF, once annotated by an annotator, becomes a gene.

- Multiple names for a single object.
  Since researches like to use different names for a single entity – often this behaviour is induced by software systems requiring different naming schemes – ORF objects can have multiple names.

- Conserve ORF names
  When updating the gene prediction or the sequence, the system should preserve the old ORF names. Any new ORFs will have names not previously used in the project. Thus the ORF names do no longer contain information on the neighbourhood relationship of the ORFs, but experiments conducted using the results of a previous sequence analysis are still useful when the sequence has been updated.

- Multiple annotations for a single ORF
  While the system stores multiple annotations for a single ORF, by default, the latest annotation is used. The programmer can change this behaviour (e.g. show only annotations of a specific user) by implementing additional methods for access to the annotation objects.

- Adherence to the layer model
  Figure 2.1 shows the layer model in GENDB. As described in section 2.5.1 on page 24 in order to facilitate future changes to parts of the system, all access from frontends, wizards, or backends to the SQL storage is performed through the O2DBI object methods. Direct SQL access is strictly prohibited. Only the objects themselves can see or utilize the features of the database. This allows changing the storage backend with little or no modifications to the code.

42
CHAPTER 3

Implementation of GENDB

3.1. Implementation details

In the previous chapter we have defined a number of issues to address for a new genome annotation system. In this chapter we present a number of aspects that together form our new system.

3.1.1. Achieving modularity and extensibility

In order to achieve both a modular and an extensible system, we need to provide some internal data model and an application programmer’s interface to access data and functions in the system. The existence of an abstract data representation is the key towards achieving all of this goal. Since a large amount of information is computed at some cost, the system needs to be able to store and retrieve this information efficiently. A mechanism for
3. Implementation of GENDB

persistence is needed.

3.1.2. Use of object oriented approach

To implement the abstract data representation we have chosen an object oriented approach.

Using an object oriented programming paradigm has a number of advantages:

- “simple, readable programs” [Obj00, SWA98]
  The complexity is mostly hidden in the objects themselves, the programs usually are very short and are easy to comprehend even for the uninitiated.

- “objects provide a clearly defined application programmer’s interface” [Obj00]
  The objects and the properties and methods define the application programmer’s interface thus a new module can be implemented using the existing data.

- “the source code is almost self documenting” [Obj00]
  Since the source code is usually free of technical details, it can be read and understood by non programmers, thereby offering a bridge between the biologists using the system and the computer scientist developing it. A distribution of labor where the computer scientist develops an object and its methods and the biologist writes small programs that use these objects becomes feasible.

- “less error prone” [Obj00]
  There are fewer lines of code and the code is easier to read. This, in addition to the high degree of code reusage, leads to less error prone development.
3. Implementation of GENDB

An example for object oriented development

Figure 3.1 shows an excerpt of a Perl [WS91] program that lists all ORFs in a genome, computes the value of “startcodon”, and stores it in the corresponding object property.

```perl
# A definition for the ORF object used below
$ORF->name;
$ORF->sequence;
$ORF-> ....

# walk thru all ORFs
foreach $orf (GENDB::orf->fetchall) {

  # extract the first 3 characters of each ORF
  $startcodon=substr($orf->sequence,0,3);

  # write the startcodon property
  $orf->startcodon($startcodon);
}
```

Figure 3.1.: A simple OO program.

In the example above, the ORF objects are loaded into memory using the `fetchall` method, then, for each ORF, the substring property is computed using the Perl substring method to extract the first three characters in the ORF’s sequence and store it in the ORF object.

3.1.3. Use of a DBMS

To achieve data persistency, we have chosen to use a database management system.

From a programmers’ point of view, a database management system is a
3. Implementation of GENDB

subsystem that manages data handling. There are a number of established such systems that fall into three categories:

- relational database management systems (RDBMS)
- object oriented database management systems (OODBMS)
- a combination of these: object relational database management systems

In conjunction with the object oriented model described above, the use of an OODBMS seems clearly the way to go, but unfortunately there are no object oriented DBMS systems in the public domain [Obj00]. So the use of an OODBMS collides with our design target of using as little as possible commercial software. While commercial database management systems (e.g. Objectstore [OBJ]) exist, they are rather expensive for an academic environment.

There are a number of RDBM systems in the public domain (e.g. MySQL [Incb] or Postgres [Pos]).

This, in conjunction with the fact that relational database management systems are long established led us to choose an RDBMS.

An RDMBS also provides a simple method for making complex queries, e.g. the distribution of start codons can be easily obtained if a RDBMS is used to store the data. The Standard Query Language (SQL) [MS93] language can be used to formulate queries:
3. Implementation of GENDB

Example use of SQL: Distribution of start codons

```sql
select count(*) from orfs where startcodon='ttg';
```

returns the number of ORFs with the respective start codon.

This is a more complex query:

```sql
select count(*) from orfs where startcodon='ttg' and length>1000;
```

Figure 3.2.: Determine the distribution of start codons using SQL.

Issuing queries like those shown in figure 3.2 with one of the genome annotation systems described in section 1.3.1 would require a large amount of code since the necessary information would have to be parsed out of either HTML documents and/or Prolog fact files.

3.1.4. Using objects with a RDBMS

An RDBMS stores data in a table format, it cannot handle objects. In order to achieve persistency of objects in an RDBMS, code has to be written that handles the conversion from objects to table formats. While the process of storing objects in database tables is well defined [Amb99], it is a labor intensive one.

While systems that map objects to database tables exist for the Java programming language [Cat], for the Perl programming language no such system existed at the time the GENDB project was started. An ideal system would allow the description of objects in a Perl like syntax and automatically create the code needed to read from a database system and store objects in a database system.

In the example in figure 3.1 on page 45 the ORF object is defined using the Perl standard way of declaring object properties.
3. Implementation of GENDB

The O2DBI-- tools developed by Jörn Clausen [Cla00] in the context of the GENDB project provide automatic conversion from Perl objects to tables in an relational database. Based on the description of the objects in Perl-like syntax, the code for object persistency is automatically generated by O2DBI. For each class of objects, a Perl module is generated that allows the user to specify additional functionality in a separate file.

Figure 3.3 shows the O2DBI description of an ORF object. Section 3.1.5 on page 51 describes the functions that are automatically created.

```perl
$geneproject{"orf"} = {
    members => {
        'contig_id' => 'int',
        'name' => 'char(20)',
        'status' => 'int',
        'start' => 'int',
        'stop' => 'int',
        'molweight' => 'float',
        'isoelp' => 'float',
        'frame' => 'int',
        'ag' => 'int',
        'gc' => 'int',
        'startcodon' => 'char(3)',
        '@names' => {
            'name' => 'char(20)',
        },
        '@annotations' => {
            'annotation_id' => 'int',
        }
    },
};
```

Figure 3.3.: The O2DBI description of an ORF object.

Figure 3.4 on page 50 shows part of the code generated for the create and the fetchall methods of the ORF object. The create method establishes
3. Implementation of GENDB

a database entry and the corresponding ORF object. The parameters are then stored as object properties using a number of object methods automatically created by O2DBI. The Perl object is then returned as a result of the create method. The fetchall method creates an array of Perl objects of the type ORF. Using an SQL select statement, a list of all ORFs is returned from a database table. The content of the table is then converted into ORF objects which are stored in an array. The code returns an array reference.

The complete code generated for the ORF object is listed in the appendix see B.2 on page 127.
3. Implementation of GENDB

```perl
# create a new object and insert it into the database
sub create {
    my ($class, $contig_id, $start, $stop, $name) = @_; 
    # fetch a fresh id
    my $id = newid('orf');
    if ($id < 0) {
        return(-1);
    }
    # insert the primary key into the database
    $GENDB_DBH->do(qq {
        INSERT INTO orf (id) VALUES ($id)
    });
    if ($GENDB_DBH->err) {
        return(-1);
    }
    # create the perl object
    my $orf = { 'id' => $id,
        '_buffer' => 1 };
    bless($orf, $class);
    # fill in the remaining data
    $orf->contig_id($contig_id);
    $orf->start($start);
    $orf->stop($stop);
    $orf->name($name);
    $orf->unbuffer;
    return($orf);
}

# get all objects from the database efficiently and return an array reference
sub fetchall {
    my ($class) = @_; 
    local @orf = ();
    my $sth = $GENDB_DBH->prepare(qq {
        SELECT molweight, contig_id, startcodon, name, status, stop, ag, gc, frame, isoelp, id, start FROM orf
    });
    $sth->execute;
    while (($molweight, $contig_id, $startcodon, $name, $status, $stop, $ag, $gc, $frame, $isoelp, $id, $start) = $sth->fetchrow_array) {
        my $orf = {
            'molweight' => $molweight,
            'contig_id' => $contig_id,
            'startcodon' => $startcodon,
            'name' => $name,
            'status' => $status,
            'stop' => $stop,
            'ag' => $ag,
            'gc' => $gc,
            'frame' => $frame,
            'isoelp' => $isoelp,
            'id' => $id,
            'start' => $start
        };
        bless($orf, $class);
        push(@orf, $orf);
    }
    $sth->finish;
    return(@orf);
}
```

Figure 3.4.: The code generated for the create method of the ORF object.

50
3. Implementation of GENDB

3.1.5. Class and object methods generated by O2DBI

A number of methods that allow access to objects and their properties are automatically created by O2DBI

- **Class methods**

- **create**
  A new object is created simultaneously in memory and in the database.

- **init_name**
  An object is created in memory from the data stored in the database. The object has to be specified via its name.

- **init_id**
  An object is created in memory from the data stored on disk. The object is specified via its id.

- **fetchallby_id**
  All objects of a specified type are fetched from the database and stored in a hash. The subsequent access to the objects is based on the object id.

- **fetchallby_name**
  All objects of a specified type are fetched from the database and stored in a hash. The subsequent access to the objects is based on the object name.

- **fetchallby_SQL**
  A selection of objects of a specified type are fetched from the database and stored in memory. The selection is based on the result of an SQL statement.

- **fetchall**
  All objects of a specified type are fetched from the database and stored in an array.
3. Implementation of GENDB

- Object methods
  For each object property, a method is created allowing read and write access to the property in question.

3.1.6. Functions automatically generated by O2DBI

The following functions are provided by O2DBI and should only be used in object methods. They are not part of the GENDB application programmer’s interface.

The function getset ()

The code autogenerated by O2DBI uses the function getset () described in figure 3.5 to get or set any of the member variables.

```
sub getset {
    my ($self, $var, $val) = @_;  
    my $id = $self->id;  
    if (defined($val)) {
        if (!$self->buffered) {
            my $qval = $GENDB_DBH->quote($val);  
            $GENDB_DBH->do(qq {
                UPDATE contig SET $var=$qval WHERE id=$id
            }) || return(-1);  
        }
        $self->{$var} = $val;
    }
    return($self->{$var});
}
```

Figure 3.5.: The function getset ()

This allows very easy access to the member variables, figure 3.6 on the next page shows code using the getset () function.
3. Implementation of GENDB

```perl
# get or set the member variable 'sequence'
sub sequence {
    my ($self, $sequence) = @_;  
    return($self->[getset('sequence', $sequence)));
}
```

Figure 3.6.: The sample code using the function `getset()`.

**The function `mset()`**

The function `mset()` allows the setting (and automatically storing) of several member variables at the same time. `mset()` is called with a hash reference as parameter and stores all values listed for the variables. The hash keys are the variables to be set.

```perl
sub mset {
    my ($self, $hashref) = @_;  
    my $curbuffer = $self->[buffered;  
    $self->[buffer;  
    foreach $key (keys(%$hashref)) {
        # prevent really stupid tricks
        if ($key eq 'id') {
            return(-1);
        }  
        my $val = $hashref->[key];  
        eval $self->[key]($val);
    }
    if (!$curbuffer) {
        $self->[unbuffer;  
    }
}
```

Figure 3.7.: The `mset()` function.
3. Implementation of GENDB

3.2. The Application Programmers Interface

As described earlier (see section 2.5.5 on page 38), the application programmer’s interface plays a central role in GENDB.

The following section explains the various routines that constitute the GENDB API.

3.2.1. Autogenerated functions for all object types.

For each class specified in the GENDB.pl (listed in the appendix, see section B.1 on page 117) file, a set of methods is generated. Table 3.1 lists these methods.
### Implementation of GENDB

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init_id()</code></td>
<td>Constructor method to fetch an object from the database using the object id</td>
</tr>
<tr>
<td></td>
<td>to select the object. This method is always created.</td>
</tr>
<tr>
<td><code>init_xyz()</code></td>
<td>Additional constructor method xyz, if additional constructors are specified.</td>
</tr>
<tr>
<td><code>create()</code></td>
<td>Creator method requiring the parameters described in the object definition.</td>
</tr>
<tr>
<td><code>delete()</code></td>
<td>Destructor, deletes the object and the database entry.</td>
</tr>
<tr>
<td><code>id()</code></td>
<td>The unique database identifier for this object.</td>
</tr>
<tr>
<td><code>fetchall()</code></td>
<td>Retrieve an array containing all objects of this type.</td>
</tr>
<tr>
<td><code>fetchallby_name()</code></td>
<td>Retrieve a hash containing all objects of this type (indexed by the object</td>
</tr>
<tr>
<td></td>
<td>name). This class method returns a hash reference.</td>
</tr>
<tr>
<td><code>fetchallby_id()</code></td>
<td>Retrieve an hash containing all objects of this type (indexed by their id).</td>
</tr>
<tr>
<td><code>fetchbySQL($sql_statement)</code></td>
<td>Retrieve a list containing all objects of this type resulting from executing</td>
</tr>
<tr>
<td></td>
<td>$sql_statement.</td>
</tr>
</tbody>
</table>

Table 3.1.: The autogenerated methods by O2DBI.

See complete source code for `GENDB.pl` in the appendix (see section B.1).
3. Implementation of GENDB

The contig object

The definition for the contig object is shown in figure 3.8. In addition to the seven member variables, a creator method is defined which will allow the creation of a contig object with the name and sequence of a new contig. A constructor method is defined that will allow access to the contig object based on the contig name.

```perl
$geneproject{'contig'} = {
    members => {
        'name' => 'char(20)',
        'sequence' => 'text',
        'length' => 'int',
        'rneighbor_id' => 'int',
        'lneighbor_id' => 'int',
        'loverlap' => 'int',
        'roverlap' => 'int'
    },
    creator => [ 'name', 'sequence' ],
    constructors => [
        [ 'name' ]
    ]
};
```

Figure 3.8.: GENDB.pl – the object definition for contig.

The information on left and right neighbor (\(r|l\)neighbor_id) and their respective overlap (\(r|l\)overlap) is used to tie together individual contigs into a larger supercontig in a later version of the program. The unique object identifier is added by the O2DBI system.

The supercontig object

The supercontig object offers access to a supercontig made up of several contig objects. Two member variables are stored, a name for the supercontig and
3. Implementation of GENDB

the contig_id of the first contig object in the supercontig. Access to and creation of new supercontig objects is done via the name of the supercontig.

```
$geneproject{supercontig} = {
  members => {
    'name' => 'char(50)',
    'first_contig' => 'int',
  },
  creator => [ 'name' ],
  constructors => [
    [ 'name' ]
  ];
```

Figure 3.9.: GENDB.pl – the object definition for supercontig.

Although the object is present in the system, the functionality for supercontigs is not yet fully completed (see the roadmap in the appendix). Once completed, accessing a supercontig will be identical to accessing a contig object. Thus the software using contig objects will be able to use supercontig objects without a change. All additional functionality will be implemented in the application programmer’s interface, therefore, the user level applications will be able to use supercontigs in addition to contigs with little or no changes in code.

The orf object

The ORF object is the most complex object in GENDB so far. While some object properties are self explanatory, some are more complex or rely on definitions within GENDB. The following table contains definitions and explanations for the not self evident properties.

The notion of multiple names for a single ORF object is probably the single most important feature in the ORF description. It enables the integration of
data from multiple sources and different stages of the genome analysis. In the past, extensive efforts went into the creation of translation tables for ORF identifiers.

For the names of contigs see section 2.5.7 on page 42.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contig_id</td>
<td>The object identifier of the contig object this ORF is in.</td>
</tr>
<tr>
<td>name</td>
<td>The ORF name for this ORF object as defined by the basecaller or by GENDB.</td>
</tr>
<tr>
<td>status</td>
<td>The state of the ORF as defined in figure 3.15 on page 70.</td>
</tr>
<tr>
<td>start,stop</td>
<td>The start and stop position.</td>
</tr>
<tr>
<td>molweight,isoelp</td>
<td>The molecular weight and the isoelectric point for the ORF.</td>
</tr>
<tr>
<td>startcodon</td>
<td>To allow easy access to the startcodon, e.g. using the SQL query interface of the DBMS.</td>
</tr>
<tr>
<td>names</td>
<td>Since biologists like to rely on an early annotation to start running costly laboratory experiments, we need to store a number of names for each ORF.</td>
</tr>
<tr>
<td>annotations</td>
<td>For a single ORF any number of annotations can be stored. A list of annotation ids is stored for each ORF.</td>
</tr>
</tbody>
</table>

The inclusion of multiple annotations for a single ORF is another distinguishing feature of GENDB. The annotations are ordered according to the date they were entered. By definition (see section 2.5.7 on page 42), the latest annotation is used when generating output.

Figure 3.10 on the following page shows the complete object definition for
3. Implementation of GENDB

the ORF object.

```perl
$geneproject{‘orf’} = {
  members => {
    ‘contig_id’ => ‘int’,
    ‘name’ => ‘char(20)’,
    ‘status’ => ‘int’,
    ‘start’ => ‘int’,
    ‘stop’ => ‘int’,
    ‘molweight’ => ‘float’,
    ‘isoelp’ => ‘float’,
    ‘frame’ => ‘int’,
    ‘ag’ => ‘int’,
    ‘gc’ => ‘int’,
    ‘startcodon’ => ‘char(3)’,
    ‘@names’ => {
      ‘name’ => ‘char(20)’,
    },
    ‘@annotations’ => {
      ‘annotation_id’ => ‘int’,
    },
  },
  creator => [ ‘contig_id’, ‘start’, ‘stop’, ‘name’ ],
  constructors => [ [ ‘name’ ] ],
};
```

Figure 3.10.: GENDB.pl – the object definition for orf.

The creator method for an orf object requires a `contig_id`, the start and stop positions and a name for the ORF. A constructor is created that uses the name to access the ORF object.
3. Implementation of GENDB

The annotation object

As shown in section 2.5.7 on page 42, a gene is defined as an ORF object plus an associated annotation object. The annotation stored for an ORF is very similar to the annotation stored in the Magpie system. As detailed in the roadmap in the appendix, this will change in a future version of GENDB. Unlike the Magpie system, a number of observation_ids can be stored that support an individual annotation.

Figure 3.11 shows the complete definition for the annotation object.

```perl
$geneproject{'annotation'} = {
    members => {
        'name' => 'char(20)',
        'orf_id' => 'int',
        'product' => 'char(20)',
        'comment' => 'text',
        'date' => 'int',
        'description' => 'text',
        'ec' => 'text',
        'category' => 'int',
        'offset' => 'int',
        'annotator_id' => 'int',
        '@observations' => {
            'observation_id' => 'int',
        },
    },
    creator => [ 'name', 'orf_id' ],
};
```

Figure 3.11.: GENDB.pl – the object definition for annotation.

No additional constructor method is defined, thus access to an annotation has to be done via the database object id only. Creation of an annotation object is possible using both a name and an orf_id.
3. Implementation of GENDB

The date field enables the sorting of annotation entries according to their date of entry. Each annotation is linked to the annotator who must be listed in the table of annotators. As with ORFs and annotations, a one to many relation exists between annotations and observations, allowing for multiple observations to be added to a single annotation object.

The annotator object

The annotator object is used to store information on the persons performing annotations. Currently, only the name of the annotator is stored. The annotator object can easily be expanded to contain the person’s email address or telephone number to facilitate discussion among different annotators. Also, the inclusion of passwords to use over the web interface is feasible.

No creator method is defined, thus the annotators has to be added using the SQL monitor tool of the DBMS. Access to the annotators is performed via the annotators id and the annotators name.

```
$geneproject{'annotator'} = {
    members => {
        'name' => 'char(20)',
        'description' => 'text',
    },
    constructors => [
        ['name']
    ]
};
```

Figure 3.12.: GENDB.pl – the object definition for annotator.

Since O2DBI allows for the easy extension of the objects without the need to rewrite the user level applications, the annotator object will be extended in a later version of GENDB when the need for a more complex annotator object arises.
3. Implementation of GENDB

The observation object

As explained earlier, the need to conserve resources led us to store only a minimal set of data for a given observation. Since recreating the complete observation is feasible once the tool, the exact parameters, and the database entry that are referenced in the observation are known, this information has to be stored.

For a given observation the following information is stored:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Description from the database record.</td>
</tr>
<tr>
<td>toolresult</td>
<td>The result from the tool. e.g. E value and score from Blast.</td>
</tr>
<tr>
<td>information</td>
<td>The information content in the observation.</td>
</tr>
<tr>
<td>dbref</td>
<td>Database identifier for the hit.</td>
</tr>
<tr>
<td>dbfrom,dbto</td>
<td>Region in the database entry that matches to the ORF.</td>
</tr>
<tr>
<td>orffrom,orfto</td>
<td>Region in the ORF that matches to the database entry.</td>
</tr>
<tr>
<td>tool_id</td>
<td>The database identifier of the tool used to create this entry.</td>
</tr>
</tbody>
</table>

Other information, e.g. the original alignment, is recomputed on demand. This is a very storage efficient strategy, only a small subset of the tool results is actually stored. Most of the information is discarded and recomputed on demand. From a user perspective, recomputing the original tool result takes about the same time as retrieving the same information from storage.

For a given genome project, thousands of tool results are computed, each containing up to several hundred individual results. Of this multitude of
3. Implementation of GENDB

results, only a small percentage is ever viewed by the human annotator. This strategy has proven to be very effective for GENDB.

```perl
$geneproject{'observation'} = {  
    members => {  
        'orf_id' => 'int',  
        'description' => 'text',  
        'toolresult' => 'text',  
        'information' => 'int',  
        'dbref' => 'text',  
        'orffrom' => 'int',  
        'orfto' => 'int',  
        'dbfrom' => 'int',  
        'dbto' => 'int',  
        'tool_id' => 'int',  
    },  
    creator => [  
        'orf_id'  
    ]  
};
```

Figure 3.13.: GENDB.pl – the object definition for observation.

Figure 3.13 shows the observation (or fact) object.

No additional constructor method has been specified. The creator method uses a single argument, the ORF database identifier. The observation object is only used in conjunction with an ORF object therefore, a single creator method is sufficient.

**The tool object**

In the context of GENDB, a tool is an external program in conjunction with a sequence or sequence motif database. Tools are used to compute observations. An example for a tool is Blast using the Swissprot database.
3. Implementation of GENDB

Therefore, the tool object contains information on how to compute observations with a given tool and also information on how to present these observations.

We have decided to store only a minimal amount of data for observations and regenerate the rest dynamically to save resources. Only a fraction of the observations are viewed by a human annotator and recomputing the original results is easily possible when the tool, the parameters and the database entry that was found are known. The tool object thus includes information on how to recreate the original query result by rerunning the tool.

The following table lists the more complex properties in the tool object definition.
### 3. Implementation of GENDB

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_type</td>
<td>A tool uses either DNA or amino acid sequences to perform the query.</td>
</tr>
<tr>
<td>number</td>
<td>The user can sort the tools, thus the tools with the lowest user priority are computed first. A simple scheduling algorithm can e.g. compute observations until a level $n$ observation is found, starting from the tool with the lowest number.</td>
</tr>
<tr>
<td>user_value</td>
<td>This is intended for a future scheduling algorithm that allows the user to group tools into classes.</td>
</tr>
<tr>
<td>cost</td>
<td>A systemwide value for the computational costs of a tool, intended for future use.</td>
</tr>
<tr>
<td>dburl</td>
<td>The URL of the database, used for downloading the database sequences before recomputing the observations.</td>
</tr>
<tr>
<td>dbname</td>
<td>The name of the database.</td>
</tr>
<tr>
<td>level$\text{n}$</td>
<td>A score threshold value that is to be exceeded by the tool score to create a level $n$ observation.</td>
</tr>
<tr>
<td>helper_package</td>
<td>The parser for the tool results and other functions are implemented in the helper packages, see section 3.2.2 on page 74.</td>
</tr>
</tbody>
</table>

Figure 3.14 on the following page shows the object definition for the tool object.
3. Implementation of GENDB

```perl
$geneproject{'tool'} = {
    members => {
        'name' => 'char(20)',
        'description' => 'text',
        'input_type' => 'int',
        'user_value' => 'int',
        'number' => 'int',
        'cost' => 'int',
        'dburl' => 'text',
        'dbname' => 'text',
        'executable_name' => 'text',
        'level1' => 'int',
        'level2' => 'int',
        'level3' => 'int',
        'level4' => 'int',
        'level5' => 'int',
        'helper_package' => 'text'
    },
    creator => ['name'],
    constructors => [
        ['name']
    ]
};
```

Figure 3.14.: GENDB.pl – the object definition for tool.

A creator and a constructor method are defined both requiring a name as the only parameter. By changing the settings for the different levels the presentation of the observations can be achieved in the user level programs. This implicates that the level settings are changed on a per project basis instead of on a per user basis.

For each tool, the system stores information on how to classify the tool results into five different levels. The user can decide to trust level 1 results and write a program that automatically uses level 1 results to annotate the ORFs.

66
3. Implementation of GENDB

3.2.2. Not automatically generated methods for all object types

The code for the manually added functions is located in the *_.add.pm perl modules located in the GENDB directory.

We list the functions available in this part of the application programmer’s interface that are currently available on a file by file basis.

$GENDBROOT/lib/annotation_add.pm

fetchall ecs()

This ORF object method returns all Enzyme Classification Numbers numbers stored in the database for the ORF as a list of strings of Enzyme Classification Numbers and hash containing all annotation_ids for each EC-number.

latest_annotation_init_orf_id()

This ORF object method returns an annotation object with the latest annotation for a given ORF.

$GENDBROOT/lib/annotator_add.pm

annotator()

This object method for a annotation object returns a string containing the name of the last annotator.
3. Implementation of GENDB

$GENDBROOT/lib/contig_add.pm

fetchorfs()

This object method returns all ORFs for a given contig. A reference to a hash containing the ORFs indexed by name is returned.

fetchallby_id()

This object method returns all ORFs for a given contig. A reference to a hash containing the ORFs indexed by id is returned.

num_orfs()

This object method returns the number (an integer) of ORFs for a contig object.

num_genes()

This object method returns the number (an integer) of annotated ORFs (called genes) for a contig object.

orf_stats()

This object method returns an array with statistics about the state for a given contig object. $array[0]$ contains the number of ORFs, $array[1]$ the number of ORFs with state 0, $array[2]$ the number of ORFs with state 1, etc.

fetchOrfsInRange()

This object method, requiring the two parameters $start$ and $stop$, returns all ORFs within the range described by the integers $start$ and $stop.$
3. Implementation of GENDB

`fetchOrfsatPosition()`

This object method requiring one parameter `$position` (integer), returns all ORFs which overlap the position.

`getTranslationFrame($frame,$fill)`

This object method uses two parameters `$frame,$fill` and returns the translation of the DNA sequence contained in the contig. `$frame` denotes the reading frame to be translated. If `$fill` is not empty, the sequence of each amino acid letter followed by two “whitespace” characters.

`$GENDBROOT/lib/observation_add.pm()`

A constant ($srs_query_url`) set in this module defines the fixed host prefix of the URL used for sequence database information retrieval. Currently this is set to `http://srs.Genetik.Uni-Bielefeld.DE/cgi-bin/wgetz?`.

`SRSrecordURL()`

For a given observation object, this method returns a string containing the URL of the associated sequence database entry.

`SRSrecord()`

For a given observation object, this method returns a string containing the complete sequence database entry as returned by the SRS system.

`dbsequence()`

This object method returns only the sequence portion of a database entry associated with the observation.
3. Implementation of GENDB

$GENDBROOT/lib/orf_add.pm

The ORF states are defined in this module as follows.

$ORF_STATE_PUTATIVE = 0;
$ORF_STATE_ANNOTATED = 1;
$ORF_STATE_IGNORED = 2;
$ORF_STATE_FINISHED = 3;
$ORF_STATE_ATTENTION_NEEDED = 4;
$ORF_STATE_USER_1 = 5;
$ORF_STATE_USER_2 = 6;

Figure 3.15.: The definition of the ORF states.

The methods defined in this module are based on the idea that the first base in the start codon is the start position of an ORF, the last position (stop position) is the last base before the stop codon. Thus we count positions starting from 1 (not from 0 as it is common, e.g. in the C programming language).

**length()**

Based on the assumptions described above, this object method returns the length of the DNA of an ORF as an integer.

**aa_length()**

Based on the assumptions described above, this object method returns the length of the amino acid sequence for an ORF as an integer. The length is computed by simply dividing the DNA sequence length by three.

**sequence()**

The DNA sequence for an ORF is returned by this object method.
3. Implementation of GENDB

**aasequence()**

This object method for an ORF object returns the amino acid sequence for a given ORF.

**fetchobservations()**

For a given ORF object this object method returns a hash reference with all observations associated with the ORF. The hash is indexed by the observation id.

**best_observation()**

For a given ORF object, this object method returns the best observation. The best observation is computed by SQL sorting the observations by level within the DBMS and returning the topmost observation.

**no_observation()**

For a given ORF object this object function returns the number of observations for this ORF.

**latest_annotation()**

This object function for an ORF object returns the latest annotation as an annotation object.

**fetch_annotations()**

For a given ORF object this object function returns a hash reference with all annotations associated with the ORF. The hash is indexed by the annotation id.
3. Implementation of GENDB

fetchAllOrfsWithState()

This is not an object method, but a class method returning all ORF objects of the $state given as parameter.

nexttool()

This object function returns the id of the next analysis tool to run.

toollevel()

This object function returns the level of the last tool result that was computed for this ORF. The tools are stored in an ordered list.

order_next_job()

For a given ORF object, this object method creates the next job in line for $orf and returns the $job_id (from the orfstate table).

set_orf_alias()

This object function stores a new name for a given ORF in the database. The new name is given as a parameter.

$GENDBROOT/lib/orfstate_add.pm

To implement a simple batch processing system, a table of jobs is used. Each of the jobs is in any one of the following states

- unlocked
  No tool is currently running for this job and no tool has completed a successful run for this job.
3. Implementation of GENDB

- **Locked**
  A tool is currently running for this job.

- **Finished**
  A tool has successfully run for this job.

**lock()**

This object method locks a job. The internal $date\_done$ variable is set to 1. Before computing a job for a given ORF, the scheduler will always perform a **lock()** on the job. If **lock()** returns 1 the job is already locked and will not be computed. Only if **lock()** succeeds (returning 0) will the job be computed.

**unlock()**

If the execution of a job failed, the process calls **unlock()**. The job can then be rerun the next time the scheduler is updated. **unlock()** sets $date\_done$ to 0, the ready-to-run job state.

**finished()**

This object function is used after successfully running a job. The process uses **finish()** to indicate this state, **finish()** sets $date\_done$ to the current date.

**fetch_failed_jobs()**

Assuming the scheduler completed a run, only jobs still in state locked or unlocked are assumed to have failed. This object function returns a list of job ids that have $date\_done$ equal to 0 or 1.
3. Implementation of GENDB

$GENDBROOT/lib/supercontig_add.pm

_build_contig_chain()

This internal method is used to build a chain of contigs.

length()

This object method returns the overall length of this supercontig.

sequence($start, $stop)

This object method for the supercontig object returns a string containing the DNA sequence for the supercontig. If $start and $stop are defined, only the region from $start to $stop is returned.

$GENDBROOT/lib/tool_add.pm

This package contains a number of methods that are only used to refer to functions in the helper packages (see below).

fetch_ordered_tools()

This class method retrieves all tools from the database efficiently and returns a sorted array reference.

Stubs for the helper tools

- command_line()
  A command line is needed by the batch system to actually execute the tool run.

- run_job()
  This function uses the $command_line described above and parses
3. Implementation of GENDB

the results. Any number of tool and environment specific settings are
contained in this function.

- **score()**
  This returns the score as computed by the tool, e.g. Blast.

- **bits()**
  This returns a normalized score using bits from information theory.

- **level()**
  Depending on the user definable settings for tool levels, this function
  returns a level for an observation.

- **configure()**
  This is the most complex function of the stubs since it creates a window
  and allows the user to modify the tool settings. This function is part of
  the tool specific code in GENDB.

### Tool helper packages

The tool helper packages implement the stubs described above. Currently,
the following helper packages are implemented in the GENDB system:

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blast</td>
<td>Homology searches using blast1, blast2, psi-blast and phi-blast.</td>
</tr>
<tr>
<td>hmmpfam</td>
<td>Homology searches using Hidden Markov models.</td>
</tr>
<tr>
<td>glimmer1+2</td>
<td>Gene prediction [DHK+99].</td>
</tr>
<tr>
<td>gff</td>
<td>Import and Export data in GFF [DH00] format.</td>
</tr>
<tr>
<td>embl</td>
<td>Import and Export data in EMBL [SMS+01] format.</td>
</tr>
</tbody>
</table>
CHAPTER 4

Results

4.1. User interface manual

The user interface is one of the most critical components of a genome annotation system. Since thousands of genes have to be manually inspected and various pieces of information are available for each single gene, an intuitive user interface is a must for such a system.

The user interface is embedded in the workflow of a genome annotation with GENDB.

Figure 4.1 on the next page shows the flow of information in the GENDB system.
This chapter is the user manual for the graphical user interface, i.e. various tasks the user can perform with the GENDB system are described in detail.

4.1.1. Uploading data and gene prediction

The first task that a user usually faces with GENDB is uploading a sequence into the system.  

1 Please note that the system administrator’s tasks for setting up GENDB are listed in the appendix.
4. Results

Loading a contig

From the main dialog that GENDB starts up with, select Import Contig from the Management menu. Once this dialog has been selected, a new window appears that allows browsing for a filename for the new contig. After the filename has been added, a number of options can be set.

The file must be in the FASTA format, only a single contig per file is allowed.

The current version of GENDB only supports Glimmer as a gene prediction tool, so all options refer to the options of that tool.

The user can choose to use the longest contig in the database as a basis for a Glimmer ORF model or choose a model file to use. Should the new contig be longer than those already in the database, the option update all contigs results in a new gene prediction being computed for all contigs in the database.

Figure 4.2 shows the user interface for adding new contigs.

![Figure 4.2: Uploading a new contig and setting the preferences for gene prediction.](image)

4.1.2. Viewing the Contig

The most important element of the graphical user interface is the visualization of the contig and the ORFs contained in the contig.
4. Results

Main view

The single most prominent view in GENDB is the contig view. Figure 4.3 shows a sample contig view. The top half of the window contains a scalable graphical representation of the contig that the user can scroll through. The lower half contains a table that lists the ORFs in the contig.

![Figure 4.3: The Contig view.](image)

Further information on the ORFs is contained in the table. The graphical view is used as a basis for accessing information on observations associated with an ORF.
4. Results

Base view

In addition to the contig view, the so called base view, can be used to navigate through the sequence. The two complementary DNA strands are shown in the center of the window, the six derived amino acid sequences are shown above and below the DNA. Stop Codons are indicated by blue ticks on the center line. The ORFs are indicated by colored boxes around the amino acid sequences.

The contig view and the base view are synchronized, clicking on an ORF in the contig view automatically causes the base view to show the position of that ORF.

![BaseView of contig smbp11 e2.253](image)

Figure 4.4.: The base view showing both DNA strands and the derived six amino acid sequences.
4. Results

4.1.3. Viewing observations

For each ORF in GENDB a number of observations can be computed. These are stored in the database and are accessible through the user interface. By right-clicking on a ORF symbol in the contig view, the user can select show observations from a pop-up window. This opens up the multiple observation viewer.

The multiple observation viewer

For each ORF a graphical view of the observations associated with that ORF is computed on demand. The window is separated into five columns that contain the following information:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observation overview</td>
<td>A graphical view of the ORF showing where the database matches occured.</td>
</tr>
<tr>
<td>Score</td>
<td>The score as returned by the analysis tool.</td>
</tr>
<tr>
<td>Bits</td>
<td>The information content contained in the observation.</td>
</tr>
<tr>
<td>Tool description</td>
<td>The name of the analysis tool.</td>
</tr>
<tr>
<td>Hit description</td>
<td>The database description.</td>
</tr>
</tbody>
</table>

Both the graphical description and the database description are clickable. If the user performs a release of the right mouse button on a database description, the original database entry is shown using the SRS [EV97] system. Upon a right click on the graphical view, the original analysis results are recomputed and displayed.
4. Results

Figure 4.5.: The observations view.

The original tool results

The complete results of the analysis are not stored in GENDB but are recomputed on demand by the system. The system uses the database identifier of the sequence involved in a match and the tool used to recompute the complete analysis. Since we now know the sequence in the database, recomputing a tool result (e.g. Blast against EMBL) we do not need to compare the entry to the complete EMBL database, but can compare against the single sequence in EMBL that is known to match.

Figure 4.6 on the following page shows the alignment results of a hmmp-fam [AER\textsuperscript{+}00] analysis. Since only a single sequence is involved, the recomputation is done in a matter of seconds (usually 1-2 seconds) on a reasonably fast workstation.\footnote{From an end user perspective, this is faster or at least as fast as the alternative, storing the complete results and looking them up.}

\textsuperscript{2}From an end user perspective, this is faster or at least as fast as the alternative, storing the complete results and looking them up.
4. Results

Figure 4.6.: The original tool results.

A sample output from the SRS system is shown in figure 4.7.

**Integration of sequence database information**

Each observation in GENDB is associated with a sequence database entry. The system does not store the complete entry, but the user still has access to the complete information via the SRS system developed by Thure Etzold at the EBI [EV97].
4. Results

Figure 4.7.: The database information as viewed with SRS.
4. Results

4.1.4. Annotation

The automatic DNA sequence analysis by tools like Blast is not the end of a genome analysis, but rather the beginning. The user needs to store additional information and his or her interpretation of the observations computed by the system. Confusingly, both the automatic analysis by bioinformatics tools and the manual interpretation of results is referred to as annotation. In GENDB we use the term “automatic annotation” for the bioinformatics tools and “annotation” for the information added by a human researcher analyzing the automatic annotation.

The GENDB system allows the user to store any number of annotations for a single ORF.

Adding an annotation

By double clicking on the ORF name in the table of ORFs in the contig view, the user can view the annotation window, see figure 4.8.

![Annotation Window](image-url)

Figure 4.8.: The annotation window.
4. Results

Annotation history

For each ORF multiple annotations can coexist. The most recent annotation is shown in the contig view table (e.g. gene name) and other places, but the other annotations are still available in the system. GENDB shows clearly who did which annotation at what time, thus annotation based on old or outdated information can be easily identified. The left column in the annotation window shows the annotation history in figure 4.8.
4. Results

4.1.5. Pathway viewer

From the information contained in the annotation and a separate database of metabolic pathways, the PathFinder [GHM+01] tool computes metabolic pathways that are represented in the genome. The graphical pathway representation can be used to navigate through the genome and also to improve the annotation. Enzymes missing in the annotation can be easily identified by looking at the complete pathway. Missing enzymes are identified by color code here.

For a more detailed description of the PathViz tool, see the paper [GHM+01].

Figure 4.9.: The pathway viewer.
4. Results

4.1.6. Wizards

A number of modules in GENDB perform special functions that do not fit easily into the user interface of GENDB as described so far.

We have therefore introduced a new category for these functions, the so-called “wizards”. Wizards are software agents that help automating complex recurring tasks in genome annotation. Adding wizards is an ongoing task, once new problems are identified, new wizards will be implemented to address these specific issues.

The following paragraph outlines the wizards implemented so far.

Frame shift wizard

In almost any genome project the need for quick analysis leads the researcher to perform an automated sequence analysis once the first contig is assembled. While this early analysis is desirable for the researcher, it causes some problems with later changes in the contig data.

Most genome annotation system will require the user to reannotate the contig if changes in the contig sequence are made. GENDB actually provides an interface to alter the primary sequence and update its internal data structures to allow editing of the sequence.

Upon completion of the changes to the sequence, the gene prediction tool is called and any new, altered or obsolete ORFs are added, updated or deleted as appropriate in the GENDB system. The data for all other ORFs, observations, etc. in the system is updated accordingly.

If a frameshift is found in the middle of a contig, the positions of all ORFs downstream of the position with the frameshift are adjusted accordingly.

Figure 4.10 on the next page shows the frameshift editor of the GENDB system. The wizard allows the insertion, deletion or alteration of bases in the contig sequence. The user first selects a contig in the upper half of the window and the performs the edit operations in the lower half.
4. Results

Upon completion of the changes, the user can choose whether or not to update the GENDB database or discard the changes that were made.

![Figure 4.10.: The frameshift editor.](image)

89
4. Results

**ORF editor wizard**

Similar to the frameshift editor, the ORF editor allows editing of the ORFs created by the gene prediction tool. The user can choose to shorten or prolong the ORF either to the next or previous ATG or to the next or previous general start codon.

Upon completion of the changes, the user can choose whether or not to update the GENDB database or discard the changes made.
4. Results

4.1.7. The virtual 2D-Gel

In addition to the pathway and the contig view, the virtual 2D-gel is another navigation metaphor in GENDB.

The system provides the user with a 2D-gel computed by using the informa-
4. Results

tion on the isoelectric point ($x$ axis) and the molecular weight ($y$ axis) of each ORF. By selecting one or more spots on the gel, the user can navigate through this gel and view the observations and or the annotation for any spot in the gel.

Figure 4.12 shows the current virtual 2D gel.

Figure 4.12.: The virtual 2D gel.
4. Results

4.2. The web interface of GENDB.

In addition to the frontend described in the previous section, a web frontend exists for the GENDB system. The contigs, ORFs, observations, and annotations can be viewed using a set of CGI scripts.

The data is tied together by hyperlinks, allowing the navigation through the genome data via the web interface.

Figures 4.13-4.16 show various screenshots of the web interface corresponding to the different views that the GENDB GUI provides.

Figure 4.13 on the next page shows a list of contigs from the GENDB web user interface. This view is very similar to that generated by the Magpie [GS96] system.
4. Results

Figure 4.13.: The contig view of the web interface.

The open reading frames for a single contig are represented in figure 4.14 on the following page.
4. Results

Figure 4.14.: The ORF information in the web user interface.

The information shown for a single ORF (see figure 4.15 on the next page) is again very similar to the Magpie system.
4. Results

Figure 4.15.: The single ORF info view of the web interface.

Figure 4.16 on the following page shows another view for the ORF highlighting the ORF sequence in the complete contig sequence.
4. Results

Figure 4.16.: The ORF in context view of the web interface.

The current version of the system does not allow human annotation to be performed via the web interface, that is it provides a read-only version of the data. A later version will include this capability.
4. Results

4.3. Analysis of genomic data with GENDB

The GENDB system has been applied to several genomes or genome scale sequences. Using GENDB the average time required for manual annotation of a single ORF was reduced to between three and five minutes for most of these projects.

The next paragraph gives a short outline for several projects.

4.3.1. Analysis of the complete genome of *C. glutamicum*

The ongoing *C. glutamicum* genome project has used a number of genome annotation systems. The data from Magpie and Pedant has been imported into the GENDB system.

The capability to include multiple ORF names into a single database has greatly improved the integration of data from multiple genome annotation efforts.

The compute cluster in figure 4.18 on page 106 of the Center for Genome Research at Bielefeld University took 6 days to compute the following tool results for all ORFs in *C. glutamicum*:

- Blast against nt (EMBL/NCBI non redundant nucleotide database)
- Blast against nr (EMBL/NCBI non redundant protein database)
- hmmpfam
- Blast against Swissprot

Less than 80 percent of the 3.3 megabase genome of *C. glutamicum* have been analyzed with Magpie. Storing the sequence data and the automatic annotation created by Magpie was done via more than 450,000 files in the UNIX filesystem. Table 4.1 on the following page compares some figures about data storage for Magpie and GENDB.
4. Results

<table>
<thead>
<tr>
<th>category</th>
<th>Magpie</th>
<th>GENDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>457487</td>
<td>uses SQL database</td>
</tr>
<tr>
<td>directories</td>
<td>6522</td>
<td>uses SQL database</td>
</tr>
<tr>
<td>static images</td>
<td>≥ 50,000</td>
<td>uses dynamic visualization</td>
</tr>
<tr>
<td>Disc space</td>
<td>≥ 10 Gigabyte</td>
<td>133.614 Kilobytes</td>
</tr>
</tbody>
</table>

Table 4.1.: Comparison of storage requirements for Magpie and GENDB.

Despite the fact that GENDB was applied to the finished genome and contained at least 15% more sequence data, the genome analysis stored by GENDB requires 75 times less storage capacity compared to the Magpie system.

Figure 4.17 shows the ORF view for a region of the C. glutamicum genome. In total, the genome contains 3688 putative ORFs for which a total of 707,506 database hits have been stored.

Figure 4.17.: The ORF view for a region of C. glutamicum
Blue ORFs are annotated, green putative, and red ORFs have been marked for further analysis

Figure 4.19 on page 107 shows the virtual gel for C. glutamicum computed by GENDB.
4. Results

4.3.2. Analysis of several *S. meliloti* contigs

In the context of the international *S. meliloti* analysis project, several contigs have been uploaded to a GENDB system and analyzed. Upon completion of the annotation by an international team the results will be uploaded to a GENDB system using the EMBL import function.

Unfortunately, the creative use of the EMBL format to store annotation results makes exchanging annotations using EMBL format a very time consuming task.

4.3.3. Analysis of several smaller contigs with GENDB

Several smaller replicons have been analyzed with the GENDB system. Among them, the *pCM1* plasmid from *C. glutamicum* and the *pSB102* plasmid analyzed by Susanne Schneiker or the *pb4* plasmid analyzed by Andreas Tauch.

The *pb4* plasmid with 79371 (see figure 4.21 on the next page) bases has been analysed using GENDB, generating 12054 database hits.

Of the 123 open reading frames detected by glimmer, the user validated 78 ORFs, 44 were ignored, one ORF still needs further attention. Figure 4.20 shows the way GENDB presents this information to the user.
4. Results

Figure 4.21.: The ORF view for the pb4 plasmid
All 78 validated ORFs are shown in black, one ORF needing further analysis is shown in red.

The analysis of plasmids has become a routine task to which GENDB is applied.

4.3.4. Analysis of the B. subtilis, E. coli and M. tuberculosis genomes

When developing the pathway analysis component for their diploma thesis, Alexander Goesmann and Martin Bennemann used GENDB as a basis for their software development and also used the system to automatically reannotate the respective genomes [GB00].

The genomes of B. subtilis, E. coli and M. tuberculosis were subjected to automatic annotation by GENDB.

4.3.5. Analysis of a mycoplasma genome

In a collaboration with Joakim Westberg from the swedish Royal Institute of Technology the web interface of GENDB is currently used for the analysis of a mycoplasma genome.
4. Results

Figure 4.22 on the next page shows the web interface for the mycoplasma genome. The web interface represents the database hits as shown in figure 4.23 on page 104. Clicking on the database id shows the original database entry, clicking on the overlap graph yields the underlying alignment. By clicking on the tool name, the database hit can be used as a basis for a manual annotation.
4. Results

Figure 4.22.: The web interface for the mycoplasma genome
4. Results

![Database Hits for ORF 53 of the Mycoplasma Genome](image)

Figure 4.23.: The database hits for ORF 53 of the mycoplasma genome

4.4. Summary

Today GENDB is routinely used to annotate contig data and as a point of reference for researchers. The ability to include multiple ORF names has proven to be valuable. Various GENDB installations outside of Bielefeld show the portability of the system (e.g. at the German MPI bioinformatics server at Garching, at MWG near Munich).

As outlined in the roadmap (see on page 115), future work for the GENDB system includes its application to EST data, eucaryotic sequence data, the inclusion of expression data as well as many small improvements. The ability to extend the system was demonstrated by many small enhancements to the original design. The provision of the application programmer’s interface has been crucial for extending the system.

Since only a subset of information is stored and the rest is recreated on demand, the storage needs of GENDB are small when compared to the Magpie system that was used previously.

The usefulness of the O2DBI tool has been proven by changing the DBMS system halfway into the project from PostgreSQL [Pos] to MySQL [Incb]. The fact that the transition was possible without manual changes to the GENDB code shows that the GENDB system is not specific to a single database management system.
4. Results

A most impressive proof of the concepts used in GENDB has been the very straightforward manner in which the inclusion of a component for the analysis and visualization of metabolic pathway data by Alexander Goesmann and Martin Haubrock [GB00, GHM01] took place.

Also, the EMMA project for the storage, analysis and visualization of expression data in conjunction with EST (expressed sequence tags) developed at Bielefeld University uses an approach very similar to the one chosen for GENDB.
Figure 4.18.: The compute cluster of the Center for Genome Research at Bielefeld University. 30 Sun Netra t1(105) workstations with Ultra Sparc Ile CPUs (360Mhz, 128 MB RAM)
4. Results

Figure 4.19.: The virtual 2D gel for *C. glutamicum*
While the first generation genome annotation systems had to provide the researchers with a reasonably fast solution for viewing genomic data in a very short timeframe, the GENDB system with the benefit of hindsight was designed to avoid the problems that occurred when using the older systems for today’s tasks. Unlike the older systems, we have tried to use state-of-the-art technology whenever possible and to make the system extensible and well documented, thus facilitating novel uses for the system.

While traditionally a genome annotation system offered a graphical user interface that presented the results of a number of tools to the user, there was clearly no intention for them to be used as a data repository for new analysis software. The new data described in section 1.2.2 clearly demands such an open approach.

In contrast to the first generation systems, GENDB provides the following features or benefits:
5. Discussion

- Reduced storage requirements
  Since only a minimal amount of information is stored for observations, the amount of storage required is reduced drastically. For the *C. glutamicum* genome, there was a 75 fold reduction in the storage requirements compared to Magpie.

- Reduced CPU time and computer costs
  By selective recomputation of facts, we have reduced the number of CPU cycles used by the system. We have implemented a batch processing system that utilises a cluster of inexpensive workstations, thus providing the necessary computing power in a very cost effective way.

- System accommodates changes to the sequence data
  The GENDB system allows the user to edit the sequence and invalidates only the data affected by such changes. By only recomputing the invalidated data, CPU resources are conserved.

- GENDB provides an application programmer’s interface
  The API provided by GENDB enables easy extension of the system and addition of new methods of analysis.

- Extensibility
  Using the application programmer’s interface, the system can be adapted for new data or new research questions.

- Support for the integration of new technologies
  The upcoming proteome and expression data can be easily accommodated in the GENDB framework and linked with the results of the genome analysis.

The new system – in our opinion – presents a significant progress in the area of genome annotation systems. As the roadmap in the appendix (see page 115) shows, a series of extensions and improvements to the existing system are planned.
5. Discussion

5.1. Availability and Future work for GENDB

5.1.1. Availability

While the software presented here has been available inhouse for some time, only a few beta testers outside of Bielefeld had access to the software so far. We are planning to release GENDB as open source.

The software (including the complete source code) will be made available to academics at no charge from the GENDB web site: http://gendb.Genetik.Uni-Bielefeld.DE.

While there already is a mailing list for GENDB developers, a mailing list for GENDB users and announcements will be set up by the time the initial release takes place.

5.1.2. Future work

A detailed in the roadmap for futher development (see page 115) the first public release of GENDB will be with version 1.0.5.

The next releases will contain a number of significant improvements:

- Full support for EMBL feature (v1.1)
  While top level EMBL features can be selected for each ORF in version 1.0.5, the next version will include the whole range of EMBL features.

- Inclusion of multiple feature and gene prediction tools (v1.1)
  Currently, the only supported feature prediction tool is Glimmer. For the next release an extensible subsystem that includes various tools (e.g. GeneMark/S [BJM01] by Borodovsky et al, tRNAscan [LE97] and qrna [RE01] by Eddy, CRITICA [BG99] by Badger and Olson) and uses an expert system tool (CLIPS) to analyse the different predictions.

- Generation of annotation suggestions using an expert system (v1.2)
  Using the CLIPS system mentioned above, an analysis of the data pertaining to a sequence feature is performed and a suggested annotation
5. *Discussion*

is created. This takes into account multiple database hits and the various bits of information gathered by GENDB.

- Support for EST projects (v1.3)
  A number of projects sequence ESTs and perform an annotation on those instead of sequencing whole genomes. GENDB will be extended to allow for EST input instead of sequence contigs.
APPENDIX A

Roadmaps for Installation and future development
A. Roadmaps for Installation and future development

A.1. Installation Roadmap for a GENDB system.

A.1.1. Step: 1 – Installing the necessary software

Requirements for GENDB as of 29th October 2001. The following software packages are required:

- a UNIX like operating system (Solaris preferred)
- the X Windows System (X11) with gtk-libraries
- various gnome libraries
- a Web Server (Apache 1.3 or later preferred)
- a relational database management system (MySQL [Incb] preferred)
- Perl 5.005 or later with a number of perl modules

  - Perl/Tk
  - DBI + appropriate DBD
  - GtkPerl
  - Bio::Perl (version $1.6$
  - ...

The GENDB installation includes a CPAN bundle file that automatically uploads and installs the required modules into a perl installation.

- Blast with the respective databases and the formatdb utility
- the PFAM database and hmmpfam [AER^00]
- the Glimmer [DHK^99] (>= 2.0.1) gene prediction software.
A. Roadmaps for Installation and future development

A.1.2. Step: 2 – Installation of GENDB

After installing the necessary modules unpack the GENDB distribution by using the command `gzcat <archive> | tar xf -`.

run `setup.pl` and follow the instructions.
A. Roadmaps for Installation and future development

A.2. Roadmap for future GENDB versions.

**v1.0.5: (first public release)**

- Replacing ORFs by EMBL Features (partial).
- Integration of a Web User Interface.
- Replacing of the Perl/Tk User Interface for new Interface implemented in Perl/Gtk [MKM] offering more flexibility.
- Integration of KeggMapper by Alexander Goesmann for projecting GENDB annotations on the well known KEGG [KG00] pathway maps.
- Inclusion of a new interface to additional gene prediction tools.

**v1.1:**

- Replacing ORFs by EMBL Features (complete).
- Automatic analysis and comparison of the results of gene and feature prediction tools.
- Allow annotation via the Web interface

**v1.2:**

- Automatic analysis of observations (former facts) and features using an expert system (CLIPS).
- Supercontigs – tying together contigs and projecting EMBL features onto the supercontig.
- Adding features for use with eucaryotes (limited support).
A. Roadmaps for Installation and future development

**v1.3:**

- EST project support, replacing contigs and EMBL features with the notion of ESTs.

**v2.0:**

- Support for the integration of micro- and macroarray data.
- Support for the integration of proteomics data.
#!/vol/perl/bin/perl
use O2DBI;
%geneproject = ();
=head1 NAME
GENDB:: gendb -- A perl sql database interface for genome analysis
=head1 DESCRIPTION
Using an SQL database persistent perl objects are created that store
sequences, orfs, facts and other data for genome annotation.

Using the O2DBI tool a set of perl modules is generated from a perl
object description. Persistence of these modules is achieved via
SQL using Perl::DBI. The object modelled in GENDB.pl are used for
storing data from a genome project.

Several classes exist:
=over 4
B. Source Code

- item * contig
  A sequence of any length (max. $10^{32}$ characters).

- item * orf
  An open reading frame is a potential gene.

- item * annotation
  Further description of an orf. I<orf + annotation = gene>. Only a member of C<annotator> can create an C<annotation> object.

- item * fact
  The results from different bioinformatics analysis tools are parsed and facts are extracted into fact objects. Only relevant information is contained in a fact object, the whole tool output is recreated on demand.

- item * annotator
  Persons or software tools allowed to create annotation records. Currently no automatic annotation tools exist.

- item * orfstate
  The orfstate objects contain information on the analysis tools run started or completed for each orf.

- item * tool
  A description of the bioinformatics tool used for collecting evidence in the form of facts. The information to display the complete hit is also contained in each tool object. To recreated the output of a tool SRS is used to retrieve the relevant data sets from the databases. Since we know exactly what database record we want to look at, we extract only this record and rerun the analysis for on query object and a single database record. This works in E<lt> 2 seconds on a modest workstation (C<Ultra Sparc I>).

=back

=head2 Several methods are automagically created for each class.

=over 4

- item * init_name

- item * init_id

- item * add INFO HERE !!!!

=back

=head2 automatically created member methods

For every individual field a method is created, that allows read and write access to the database and the perl object. I<e.g.> C<$orf->C<status('ignore');>

=head1 TODO::

118
B. Source Code

The methods in _add.pm need documentation as well.

=head1 SYNOPSIS
use GENDB::contig;
use GENDB::orf;
use GENDB::annotation;
use GENDB::tool;
use GENDB::annotator;
use GENDB::orfstate;
Sample use:
C<use GENDB::orf;>
C<$orf::GENDB::orf->init_id(0131043);>
C<if ($orf<0) {
C<die "can’t read orf from database";>
C<}>
C<print $orf->length;>

=head1 Module descriptions
=head2 contig object
=over 4
=item * constructor methods
constructors => [[ ‘name’ ]]
=item * creator methods
creator => [ ‘name’, ‘sequence’ ]
=item * name
the name of the sequence object in $contig
=item * sequence
a string of characters over the alphabet {G,A,T,C} (up to 10^32 chars)
=item * [rl]neighbor
the id of the [right|left] neighbor of $contig in the genome
=item * length
the length of $contig->sequence in bases
B. Source Code

(std. sql does not permit asking for the length of any field)

=item * [rl]overlap
the overlap with the [right|left] neighbor in bases
=back
=cut

$geneproject{'contig'} = {
  members => {
    'name' => 'char(20)',
    'sequence' => 'text',
    'length' => 'int',
    'rneighbor_id' => 'int',
    'lneighbor_id' => 'int',
    'loverlap' => 'int',
    'roverlap' => 'int',
  },
  creator => [ 'name', 'sequence' ],
  constructors => [ ['name'] ];
}

$geneproject{'supercontig'} = {
  members => {
    'name' => 'char(50)',
    'first_contig' => 'int',
  },
  creator => [ 'name' ],
  constructors => [ ['name'] ];
}

=head2 orf object
=over 4
=item * name
name of the orf
=item * status
state of work already done

status may be one of the following

0 = ignore (white or not displayed),
1 = putative (green),
2 = annotated (blue),
3 = finished (dark blue) and
4 = attention needed (red)

=item * start
the start of the orf on the contig
**B. Source Code**

```perl
=item * stop
stop of the orf, excluding the stop codon
=item * contig_id
the db_id of the contig this orf is in
=item * molweight
the molecular weight
=item * isoelp
the isoelectric point
=item * frame
the reading frame can be any of (1,2,3,-1,-2,-3)
=item * startcodon
for quick access we store the first three characters of the
orf
=item * names
a list of alternative names for a single orf object
=back
=cut

$geneproject{'orf'} = {
   members => {
       'contig_id' => 'int',
       'name' => 'char(20)',
       'status' => 'int',
       'start' => 'int',
       'stop' => 'int',
       'molweight' => 'float',
       'isoelp' => 'float',
       'frame' => 'int',
       'agf' => 'int',
       'gc' => 'int',
       'startcodon' => 'char(3)',
       '@names' => {
           'name' => 'char(20)',
       },
       '@annotations' => {
           'annotation_id' => 'int',
       },
   },
   creator => ['contig_id', 'start', 'stop', 'name'],
   constructors => ['name']
};

=head2 orfstate object
```
B. Source Code

Information on the analysis tools run for $orf.

For each tool that is run a orfstate object is created every time the tool is run for $orf.

Both the id of the tool and the orf are stored together with dates for the ordering a specific analysis and the finishing date for the analysis.

$orfstate object can be used to check what analyses have been performed already

constructor methods
none

creator methods
creator => [ 'orf_id', 'tool_id' ]

(annotation object)

Using fact and orf data, annotations are constructed that turn orfs into genes.

creator methods
creator => [ 'name', 'orf_id' ]

name
the gene name
orf_id
the id of the orf object this annotation refers to
B. Source Code

=item * product
the gene product

=item * comment
a comment by the annotator

=item * date
the date and time when the annotation was added to the database

=item * description
a textual description of the gene

=item * ec
the Enzyme Classification Number

=item * category
a pointer to a member in the list of categories (derived from Monika Reilly)

=item * annotator_id
a pointer into a list of annotators, indentifying the owner of the current annotation object

=item * facts
a list of fact_ids that lead to $orf being annotated as $orf->name

=back
=cut
$geneproject{'annotation'} = {
    members => {
        'name' => 'char(20)',
        'orf_id' => 'int',
        'product' => 'char(20)',
        'comment' => 'text',
        'date' => 'int',
        'description' => 'text',
        'ec' => 'text',
        'category' => 'int',
        'offset' => 'int',
        'annotator_id' => 'int',
        '@facts' => {
            'fact_id' => 'int',
        },
    },
    creator => [ 'name', 'orf_id' ],
};
=export annotator object
Stores a name and a description for the annotator.

=over 4
B. Source Code

```plaintext
=item * creator methods
none

=item * constructor methods

constructors => [ [ 'name' ] ]

=head2 tool object

A tool object contains the information for computing analysis and displaying the resulting facts. The command line for running the tool is generated by the helper package. It uses the executable_name and dbname fields to construct the command line.

=over 4

=item * name

a clear text short name for the tool. e.g. blast2n_EMBL

=item * description

a clear text description for the tool

=item * input_type

the type of data DNA=0, AA=1

=item * cost

(not used currently) the computational cost in minutes for a single analysis

=item * dburl

a url to fetch database records from, required for retrieval of info on facts

=item * dbname

the name of the database, required for srs retrieval of information on facts

=item * executable_name

the name of the executable file (maybe including path to file)

=item * level[12345]
```
B. Source Code

A cut off value in bits that allows ordering the facts into different levels a fact level1<gt>level2<gt>level3<gt>level4<gt>level5. The values are read by the visualization and further analysis components.

```perl
$item * number
```

The number in the ordered list of tools. This is in effect the tool level referred to in by $orf->toollevel.

```perl
$item
```

```perl
$geneproject{'tool'} = {
    members => {
        'name' => 'char(20)',
        'description' => 'text',
        'input_type' => 'int',
        'user_value' => 'int',
        'number' => 'int',
        'cost' => 'int',
        'dburl' => 'text',
        'dbname' => 'text',
        'executable_name' => 'text',
        'level1' => 'int',
        'level2' => 'int',
        'level3' => 'int',
        'level4' => 'int',
        'level5' => 'int',
        'helper_package' => 'text',
    },
    creator => [ 'name' ],
    constructors => [ [ 'name' ] ]
};
```

=head2 fact object

A database object to store abstract information on a single hit returned by a bioinformatics analysis tool for a given $orf. A blast2 hit (HSP) e.g. is one fact.

```perl
$item * toolresult
```

the output of the tool e.g. C<e:10^-20, s:1000> for a blast2 fact

```perl
$item * information
```

the information returned by the analysis tool in bits

```perl
$item * dbref
```

The database reference to the hit. e.g. X10236 for a Swissprot entry
B. Source Code

=item * orf[from|to]
region in $orf that matches the database entry

=item * db[from|to]
region in the db hit that $orf matches to

=item * tool_id
a pointer to the list of tools

=item * constructor methods
none

=item * creator methods
creator => [‘orf_id’ ]

=back

cut

$geneproject{’fact’} = {
  members => {
    ‘orf_id’ => ‘int’,
    ‘description’ => ‘text’,
    ‘toolresult’ => ‘text’,
    ‘information’ => ‘int’,
    ‘dbref’ => ‘text’,
    ‘orffrom’ => ‘int’,
    ‘orfto’ => ‘int’,
    ‘dbfrom’ => ‘int’,
    ‘dbto’ => ‘int’,
    ‘tool_id’ => ‘int’
  },
  creator => [‘orf_id’]
};

O2DBI->deploy(‘$geneproject’, ‘GENDB’, ‘mysql’);

=head1 See also
O2DBI by Joern Clausen
Perl::DBI
SRS

cut
B. Source Code

B.2. Code generated for the contig object by O2DBI

```perl
package GENDB::contig;

use GENDB::DBMS;

1;

sub create {
    my ($class, $name, $sequence) = @_; # constructor and destructor methods for contig
    # create a new object and insert it into the database
    my ($id, $_buffer) = 0, 1;
    # fetch a fresh id
    my $id = newid('contig');
    if ($id < 0) {
        return(-1);
    }
    # insert the primary key into the database
    $GENDB_DBH->do(qq {
        INSERT INTO contig (id) VALUES ($id)
    });
    if ($GENDB_DBH->err) {
        return(-1);
    }
    # create the perl object
    my $contig = { 'id' => $id,
                  '_buffer' => 1 ];
    bless($contig, $class);
    # fill in the remaining data
    $contig->name($name);
    $contig->sequence($sequence);
    $contig->unbuffer;
    return($contig);
}

sub init_id {
    my ($class, $req_id) = @_; # create an object for already existing data
    my $sth = $GENDB_DBH->prepare(qq {
        SELECT name, length, loverlap, lneighbor_id, sequence,
             roverlap, rneighbor_id, id FROM contig
        WHERE id='$req_id'
    });
    if ($sth->err) {
        return(-1);
    }
    my ($name, $length, $loverlap, $lneighbor_id,
        $sequence, $roverlap, $rneighbor_id, $id) = $sth->fetchrow_hash;
    $contig = { 'id' => $id,
                 'name' => $name,
                 'sequence' => $sequence,
                 'loverbund' => 1,
                 'lneighbor_id' => $lneighbor_id,
                 'roverbund' => 1,
                 'rneighbor_id' => $rneighbor_id; # Additional methods can be defined in the file GENDB::contig_add.pm.
    }
    bless($contig, $class);
    return($contig);
}
```
B. Source Code

```perl
$sth->execute;
my ($name, $length, $loverlap, $lneighbor_id, $sequence,
$roverlap, $rneighbor_id, $id) = $sth->fetchrow_array;
$sth->finish;
# if successful, return an appropriate object
if (!defined($id)) {
  return(-1);
} else {
  my $contig = {
    'name' => $name,
    'length' => $length,
    'loverlap' => $loverlap,
    'lneighbor_id' => $lneighbor_id,
    'sequence' => $sequence,
    'roverlap' => $roverlap,
    'rneighbor_id' => $rneighbor_id,
    'id' => $id
 );
  bless($contig, $class);
  return($contig);
}

# get all objects from the database efficiently and return a hash reference
sub fetchallby_id {
  my ($class) = @_;  
  local %contig = ();
  my $sth = $GENDB_DBH->prepare(qq{
    SELECT name, length, loverlap, lneighbor_id, sequence,
    roverlap, rneighbor_id, id FROM contig
  });
  $sth->execute;
  while (($name, $length, $loverlap, $lneighbor_id, $sequence,
  $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array) {
    my $contig = {
      'name' => $name,
      'length' => $length,
      'loverlap' => $loverlap,
      'lneighbor_id' => $lneighbor_id,
      'sequence' => $sequence,
      'roverlap' => $roverlap,
      'rneighbor_id' => $rneighbor_id,
      'id' => $id
    };
    bless($contig, $class);
    $contig{$id} = $contig;
  }
  $sth->finish;
  return(%contig);
}

# get all those objects from the database efficiently that conform to the
# given WHERE clause and return an array reference
sub fetchbySQL {
  my ($class, $statement) = @_;  
  local @contig = ();
  my $sth = $GENDB_DBH->prepare(qq{
    SELECT name, length, loverlap, lneighbor_id, sequence,
    roverlap, rneighbor_id, id FROM contig WHERE $statement
  });
  $sth->execute;
  while (($name, $length, $loverlap, $lneighbor_id, $sequence,
  $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array) {
    my $contig = {
      'name' => $name,
      'length' => $length,
      'loverlap' => $loverlap,
      'lneighbor_id' => $lneighbor_id,
      'sequence' => $sequence,
      'roverlap' => $roverlap,
      'rneighbor_id' => $rneighbor_id,
      'id' => $id
    };
    bless($contig, $class);
    $contig{$id} = $contig;
  }
  $sth->finish;
  return(@contig);
}
```

128
B. Source Code

```perl
$roverlap, $rneighbor_id, $id) = $sth->fetchrow_array){
    my $contig = {
        'name' => $name,
        'length' => $length,
        'loverlap' => $loverlap,
        'lneighbor_id' => $lneighbor_id,
        'sequence' => $sequence,
        'roverlap' => $roverlap,
        'rneighbor_id' => $rneighbor_id,
        'id' => $id
    );
    bless($contig, $class);
    push(@contig, $contig);
}

$sth->finish;
return(@contig);
}

# create an object for already existing data
sub init_name {
    my ($class, $req_name) = @_; 
    # fetch the data from the database
    my $sth = $GENDB_DBH->prepare(qq {
        SELECT name, length, loverlap, lneighbor_id, sequence,
        roverlap, rneighbor_id, id FROM contig
        WHERE name='$req_name'
    });
    $sth->execute;
    my ($name, $length, $loverlap, $lneighbor_id, $sequence,
    $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array;
    $sth->finish;
    # if successful, return an appropriate object
    if (!defined($id)) {
        return(-1);
    } else {
        my $contig = {
            'name' => $name,
            'length' => $length,
            'loverlap' => $loverlap,
            'lneighbor_id' => $lneighbor_id,
            'sequence' => $sequence,
            'roverlap' => $roverlap,
            'rneighbor_id' => $rneighbor_id,
            'id' => $id
        );
        bless($contig, $class);
        return($contig);
    }
}

# get all objects from the database efficiently and return a hash reference
sub fetchallby_name {
    my ($class) = @_; 
    local %contig = ();
    my $sth = $GENDB_DBH->prepare(qq {
        SELECT name, length, loverlap, lneighbor_id, sequence,
        roverlap, rneighbor_id, id FROM contig
    });
    $sth->execute;
    while (($name, $length, $loverlap, $lneighbor_id, $sequence,
    $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array) {
        my $contig = {
...}
```

129
B. Source Code

```
'name' => $name,
'length' => $length,
'loverlap' => $loverlap,
'lnneighbor_id' => $lnneighbor_id,
'sequence' => $sequence,
'roverlap' => $roverlap,
'rneighbor_id' => $rneighbor_id,
'id' => $id
};
bless($contig, $class);
$contig{$name} = $contig;
}$sth->finish;
return(%contig);
}

# get all those objects from the database efficiently that conform to the
# given WHERE clause and return an array reference
sub fetchbySQL {  
  my ($class, $statement) = @_;  
  local @contig = ();  
  my $sth = $GENDB_DBH->prepare(qq {
    SELECT name, length, loverlap, lnneighbor_id, sequence,  
    roverlap, rneighbor_id, id FROM contig WHERE $statement  
  });  
  $sth->execute;
  while (($name, $length, $loverlap, $lnneighbor_id, $sequence,  
  $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array) {  
    my $contig = {
      'name' => $name,
      'length' => $length,
      'loverlap' => $loverlap,
      'lnneighbor_id' => $lnneighbor_id,
      'sequence' => $sequence,
      'roverlap' => $roverlap,
      'rneighbor_id' => $rneighbor_id,
      'id' => $id
    };  
    bless($contig, $class);
    push(@contig, $contig);
  }  
  $sth->finish;
  return(@contig);
}

# get all objects from the database efficiently and return an array reference
sub fetchall {  
  my ($class) = @_;  
  local @contig = ();  
  my $sth = $GENDB_DBH->prepare(qq {
    SELECT name, length, loverlap, lnneighbor_id, sequence,  
    roverlap, rneighbor_id, id FROM contig  
  });  
  $sth->execute;
  while (($name, $length, $loverlap, $lnneighbor_id, $sequence,  
  $roverlap, $rneighbor_id, $id) = $sth->fetchrow_array) {  
    my $contig = {
      'name' => $name,
      'length' => $length,
      'loverlap' => $loverlap,
      'lnneighbor_id' => $lnneighbor_id,
      'sequence' => $sequence,
    }
  }
```
B. Source Code

```perl
‘roverlap’ => $roverlap,
‘rneighbor_id’ => $rneighbor_id,
‘id’ => $id
);
bless($contig, $class);
push(@contig, $contig);
} $sth->finish;
return($@contig);
}

# delete an object completely from the database
sub delete {
  my ($self) = @_;;
  my $id = $self->id;
  $GENDB_DBH->do(qq {
    DELETE FROM contig WHERE id=$id
  }) || return(-1);
  undef($self);
}

########################################################################
#
# methods to access the member variables
#
########################################################################

# get or set the member variable ‘name’
sub name {
  my ($self, $name) = @_;;
  return($self->getset('name', $name));
}

# get or set the member variable ‘length’
sub length {
  my ($self, $length) = @_;;
  return($self->getset('length', $length));
}

# get or set the member variable ‘loverlap’
sub loverlap {
  my ($self, $loverlap) = @_;;
  return($self->getset('loverlap', $loverlap));
}

# get or set the member variable ‘lneighbor_id’
sub lneighbor_id {
  my ($self, $lneighbor_id) = @_;;
  return($self->getset('lneighbor_id', $lneighbor_id));
}

# get or set the member variable ‘sequence’
sub sequence {
  my ($self, $sequence) = @_;;
  return($self->getset('sequence', $sequence));
}

# get or set the member variable ‘roverlap’
sub roverlap {
  my ($self, $roverlap) = @_;;
  return($self->getset('roverlap', $roverlap));
}
```
# get or set the member variable ‘rneighbor_id’

```perl
sub rneighbor_id {
    my ($self, $rneighbor_id) = @_;  
    return($self->getset('rneighbor_id', $rneighbor_id));
}
```

# get the member variable ‘id’

```perl
sub id {
    my ($self) = @_;  
    return($self->{'id'});
}
```

# set several member variables at the same time

```perl
sub mset {
    my ($self, $hashref) = @_;  
    my $curbuffer = $self->buffered;
    $self->buffer;
    foreach $key (keys(%$hashref)) {
        # prevent really stupid tricks
        if ($key eq 'id') {
            return(-1);
        }
        my $val = $hashref->{$key};
        eval $self->$key($val);
    }
    if (!$curbuffer) {
        $self->unbuffer;
    }
}
```

# load additional methods from self made module

```perl
require GENDB::contig_add;
```

# private functions used inside this module

```perl
# test if the data is buffered or passed to the DBMS immediately
sub buffered {
    my ($self) = @_;  
    return($self->{'_buffer'});
}
```

# make the data buffered, i.e. don’t write to the database

```perl
sub buffer {
    my ($self) = @_;  
    $self->'_buffer' = 1;
}
```

# write the current contents to the database and declare the object unbuffered

```perl
sub unbuffer {
    my ($self) = @_;  
    if ($self->buffered) {
        my @sql = ();
```
B. Source Code

```perl
foreach $key (qw(name length loverlap lneighbor_id sequence roverlap rneighbor_id id)) {
    push(@sql, "$key=\".\$GENDB_DBH->quote($self->{\$key})\";
}
my $id = $self-&gt;id;
my $sql = "UPDATE contig SET \join\(\', \' ', \$sql\)\ WHERE id=$id";
$GENDB_DBH-&gt;do($sql) || return(-1);

$self-&gt;{\'_buffer\'} = 0;
}

# get or set one of the member variables
sub getset {
    my ($self, $var, $val) = @_; #
    my $id = $self-&gt;id;
    if (defined($val)) {
        if (!$self-&gt;buffered) {
            my $qval = $GENDB_DBH-&gt;quote($val);
            $GENDB_DBH-&gt;do(qq {UPDATE contig SET $var=$qval WHERE id=$id
            }) || return(-1);
        }
        $self-&gt;{$var} = $val;
    }
    return($self-&gt;{\$var});
}
```

133
List of Figures

1.1. The flow of data from sequencer to genome annotation system. 4
1.2. A typical microarray showing different expression levels (from Brown et al. [Bro00]). 5
1.3. A 2D gel from the C. glutamicum genome project. 7
1.4. The lysine biosynthesis pathway as shown by KEGG [KG]. 8
1.5. The contig as represented by Magpie. 11
1.6. The table of ORFs generated for pCM1 by Magpie. 12
1.7. The flow of data in Pedant and Genequiz. 13
1.8. The flow of data in Magpie. 13
1.9. Changes to ORF names after changing the orientation of a contig in Magpie. 14

2.1. GENDB schematic view 22
2.2. The ORF object and the corresponding database tables in GENDB. 23
List of Figures

2.3. A simplified view of the GENDB data model. .................. 24
2.4. Overview of the graphical user interface. ..................... 27
2.5. Overview of the web interface. .............................. 28
2.6. The role of wizards in GENDB. ............................... 30
2.7. Overview of the GENDB scheduler. ........................... 32
2.8. Job locking in the of the GENDB scheduler. ................. 33
2.9. The additional methods for the job object. .................. 34
2.10. The observation information stored in GENDB ............... 35
2.11. The tool information stored in GENDB ....................... 37
2.12. The algorithm to obtain detailed observation information. . 38
2.13. The GENDB data flow. ...................................... 41

3.1. A simple OO program. ........................................ 45
3.2. Determine the distribution of start codons using SQL. ....... 47
3.3. The O2DBI description of an ORF object. .................... 48
3.4. The code generated for the create method of the ORF object. 50
3.5. The function getset(). ..................................... 52
3.6. The sample code using the function getset(). ............... 53
3.7. The mset() function. ....................................... 53
3.8. GENDB.pl – the object definition for contig. ............... 56
3.9. GENDB.pl – the object definition for supercontig. .......... 57
3.10. GENDB.pl – the object definition for orf. .................. 59
3.11. GENDB.pl – the object definition for annotation. ........ 60
3.12. GENDB.pl – the object definition for annotator. ........... 61
3.13. GENDB.pl – the object definition for observation. ......... 63
3.14. GENDB.pl – the object definition for tool. ................ 66
List of Figures

3.15. The definition of the ORF states. .................................. 70

4.1. The flow of information in the GENDB system. .............. 77
4.2. Uploading a new contig and setting the preferences for gene prediction. ................................................. 78
4.3. The Contig view. ..................................................... 79
4.4. The base view showing both DNA strands and the derived six amino acid sequences. ................................. 80
4.5. The observations view. .............................................. 82
4.6. The original tool results. ........................................... 83
4.7. The database information as viewed with SRS. ............... 84
4.8. The annotation window. ............................................ 85
4.9. The pathway viewer. ................................................ 87
4.10. The frameshift editor. .............................................. 89
4.11. The ORF editor. ...................................................... 91
4.12. The virtual 2D gel. ................................................ 92
4.13. The contig view of the web interface. ......................... 94
4.14. The ORF information in the web user interface. ............... 95
4.15. The single ORF info view of the web interface. ............. 96
4.16. The ORF in context view of the web interface. ............... 97
4.17. The ORF view for a region of C. glutamicum ................. 99
4.20. The Contig information for the pb4 plasmid ................. 100
4.21. The ORF view for the pb4 plasmid .............................. 101
4.22. The web interface for the mycoplasma genome ............. 103
4.23. The database hits for ORF 53 of the mycoplasma genome . 104
List of Figures

4.18. The compute cluster of the Center for Genome Research at Bielefeld University. ........................................... 106
4.19. The virtual 2D gel for C. glutamicum ........................................... 107


Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


[WCL+01] David L. Wheeler, Deanna M. Church, Alex E. Lash, Detlef D. Leipe, Thomas L. Madden, Joan U. Pontius, Gregory D. Schuler,
Bibliography

