Biological database design and implementation

Ewan Birney and Michele Clamp

Date received (in revised form): 19th December 2003

Ewan Birney
joined the EBI as one of the founding investigators for the Ensembl project. He is now a senior scientist at the EBI and runs both research and database projects.

Michele Clamp
joined the Sanger Institute as one of the founding investigators for the Ensembl project. She now works at the Broad Institute, Cambridge, MA.

Abstract

We present our experience of building biological databases. Such databases have most aspects in common with other complex databases in other fields. We do not believe that biological data are that different from complex data in other fields. Our experience has led us to emphasise simplicity and conservative technology choices when building these databases. This is a short paper of advice that we hope is useful to people designing their own biological database.

Keywords: *database design, software engineering, source code control*

Ewan Birney,
EBI,
Wellcome Trust Genome Campus,
Hinxton,
Cambridge CB10 1SD,
UK

Tel: +44 (0)1223 494420
Fax: +44 (0)1223 494468
E-mail: birney@ebi.ac.uk

INTRODUCTION

Building a biological database is, in theory, no different from building a database for an investment bank, government agency, business or another scientific endeavour. One needs to understand the information the database is going to store and present, translate that understanding into a rigid framework which one can imagine implementing using programmatic tools, write and debug such programs and then finally actually run and use the database, usually with additional 'on-site' development and debugging. Of course, such a simple plan is often complicated by tough decisions about resources, the need to show a working system (in particular for the funding source) and people's goals and understanding of the system changing over time. All such annoyances are also present in more 'traditional' (ie business) database settings. In the main the real challenge for any database system are the myriad details of understanding, programming logic and presentation which must all be in synchrony to deliver an effective solution. From this perspective all one can write about effective databases is that it is the details that one has to get right, and the details are always specific to a particular database.

Even though it is predominantly about details, we believe that there are aspects of building a successful biological database that deserve specific attention. In addition there are occasions both when biologists do not appreciate aspects of running databases, and IT professionals are naive about the challenge they face in implementing a successful biological database. All three aspects form the focus of this paper, with Ensembl being our main experience, but a number of other biological databases, which we have either participated in or observed, also contribute. The precise technical details of Ensembl are laid out in a number of papers.[1,2] Most of what we say in this paper is either 'common knowledge' or 'common sense' but our own experience is that translating such objective truisms is hard in practice. We hope that this paper helps other people avoid some of the problems we have had in developing databases.

UNIQUE PROBLEMS FOR BIOLOGICAL DATABASES

There are two problems often encountered with biological databases which are (in our limited understanding of the broad field) rare in other database implementations. The first, and by far the most interesting, is that the 'true' biological interpretation of data stored in a database not only can change over time, but discovering new relationships between aspects of the data is part of the motivation for storing information in a

Changing data models can be an aim for a biological database

database in the first place. Changing one's view of the data is not an annoyance but an outright aim in some cases. For example, it was a common belief before 1997 that the vast majority of 'genes' encoded protein structures, which implied a series of data structures to store aspects of protein coding function, eg the start codon. Although many structural RNA genes were known, they were often either not modelled in a database or had specialised descriptions. However, it has now become clear that there are a considerable number of RNA genes with a number of different roles, and the storage and display of these genes are of at least as much interest as protein coding genes. This change in understanding has clear knock-on effects for the data modelling.

Often one will have aspects of a particular database that are considered to be 'invariable' experiment-based data (such as sequence information, or read-outs from microarray chips), but it is worth noting that in general this is a deliberate decision about where to assume 'data' and 'interpretation' starts and ends; the sequence in Ensembl is composed of specific trace information that is (to different degrees of confidence) believed to be true; microarray data are heavily manipulated before being stored; gene structures in model organisms as often assumed to be true for the analysis of other features. Often storing the interpretation of 'low-level' data is the *raison d'être* of a database:in the case of Ensembl we developed a highly effective automatic gene-building system, and the main feature of the system was the calculation, storage and display of this interpretation. Interpreting data sets, usually by attempting to reconcile different 'low-level' data sets into one biological synthesis, is the essence of complex bioinformatics data sets and directly provides insight into molecular biology. This 'problem' is the exciting puzzle to solve.

The other problem is far more mundane: there is still a dearth of people who appreciate both the biological problem and the programmatic problem. This means that most groups have a combination of people with either background, with overheads in communication (for example, the meaning of the word 'vector' meaning alternatively a set of numbers used in mathematics, a plasmid system used to propagate DNA in cells in the laboratory or an organism involved in host–parasite transmission) and, far more importantly, a sense of how complete a particular understanding of molecular biology is. Both sides can dramatically over- or underestimate the problems and effective solutions to aspects of data implementation. The inability to rarely have a full understanding of both biology and programming affects all levels of building databases, from programmers to funders and from users to reviewers. Thankfully there are now far more people with honest dual skill sets graduating, and there are a number of specific educational courses that blend computer science and biology. This problem dominates many aspects of working biological databases, and the only solution is to find and hire good people who have at least an appreciation of the other field, even if they do not have truly ambidextrous skills.

GENERAL DATABASE IMPLEMENTATION POINTS

These points are either 'common sense' or 'common knowledge' but yet we still see biological databases that seemingly wilfully want to invent problems for themselves. We have both made the same errors ourselves over time (and indeed ignore the very same advice we are giving here, to our cost). Perhaps the only way to learn these lessons is to make your own mistakes. But we hope some of this advice will affect people's attitude to new databases.

Always use source code control

Source code control is a system where text files can be managed by multiple

Source code control is vital for any successful project

individuals over time, and can be branched to provide 'stable' or 'development' branches. CVS is a decent, open source and free source code control system, so there is no excuse not to use one. Source code control is useful for any size of group, from one programmer onwards, and is essential once you get to two people. Effective projects usually have everything tracked inside a common source code control system, which is accessible by the whole group. This gives a common sense of ownership of the project and the capability of different individuals to flexibly contribute and fix aspects of the system.

Do not use text files in directories as a poor-person's database

It is not worth with the current availability of sensible database solutions to go down this route, however 'simple' your data are. Both MySQL and Postgres are easy to install, free and simple to learn and use. One week's investment into learning one of these systems will save months of otherwise more and more baroque systems. The only exception to this is storing inherently 'opaque' data that have to be read by specialised tools, eg images. Sequence files are also a special case because of the large number of third party tools that are tied to specific formats.

We have not had any success ourselves at using more 'pure' object-based databases or extensible mark-up language (XML)-oriented databases; in all the cases we have encountered, 'straightforward' relational systems work better (at least for us). We might just be dinosaurs in this regard, and such solutions might work better for specific cases. (In the interests of full disclosure, both of us have previously made databases via files in directory structures and although they did work, ultimately they did not scale and had a far higher software cost of writing specific directory traversing routines.)

Only use cutting edge technology if you really have to

Even then think twice about it. There are two major problems generated by using cutting edge technology. Firstly, it is inherently more buggy because bugs are only found and removed by the software being used in different scenarios. Secondly, it is likely that there will be only a small number of people in the world (and generally only one of them in the group) who will understand the precise details of how things work, making debugging a bottleneck on that individual and the entire enterprise reliant on a single individual.

By using broadly understood technology one lowers the barriers for new people to come into the group and for other people to use the system directly (assume one is developing an open system). The practical consequence of this is that in terms of programming languages, try to stick to C, C++, Java or Perl, potentially Python and, at a longer stretch, Lisp. In terms of databases, use a relational database that supports ANSI SQL and use specific extensions with caution.

Try not to directly mix production database development with either computer science research or algorithmical 'bioinformatics' research

If you are serious about developing a new database with valuable biological data, then it must be done using concepts and technology that people are truly confident in. We are amazed when people try to take on both tasks at the same time – one is easily enough! This often seems to be a problem with more computer science oriented people, who want to make an advance in computer science using biological data as an exemplar. This is fine to do, but must never be confused with building a biological database where the focus is the storage, manipulation and inference from the data.

Try to focus on doing a few things well, rather than requiring the whole universe of biological data to be integrated locally

Ideally, make sure there is no one or only a few other groups providing a similar resource. This often seems to be a problem with people with a more biological background, who immediately sketch out some grandiose scheme of incorporating nearly every aspect of biology from a topic of interest. Instead try to find the real missing pieces not done well elsewhere, and then focus on delivering good data in this area.

MORE DETAILED ADVICE

There are many styles for managing database projects – we present ours

We have a particular style of designing and running databases, but we recognise that this is just one style; other people run other databases differently with equal or better results. We expect readers might be interested in our 'best practice' rules, but this advice is more 'to taste' for each individual.

A standard database project in our minds would have the following features:

- A CVS repository (potentially shared with other projects).
- A closed mailing list for all members of group, which is typically not publicly archived. This is where the day-to-day business of the group is discussed, and sometimes hard decisions have to be taken.
- A mailing list for the CVS commit messages, which all programming developers should be on.
- At least one public mailing list where users and/or other developers can provide input and discuss aspects of the system. This usually is publicly archived.
- A well-backed-up production database instance.
- A relatively isolated web-viewed database instance, with appropriate web code. It is rare in our experience to run the web views directly off the production database, but this is probably because we have been involved in more analysis-oriented databases. When databases are used directly in a production work-flow (eg closer to wet experiments or with many curators), often the web instance has to be the same as the production instance. In this case, have a good data replication and archive strategy, such that at regular intervals (for example, daily) one has a complete copy of the database archived, usually first on disk and then certain weekly copies are then moved to a longer-term medium (eg tape).
- Development areas where developers can create and destroy development databases and have enough space to develop new code.

In terms of people, it is of course variable in terms of both the scope of the project and the precise mix of skills required. Generally one needs:

- A bioinformatician who understands what the biological data are and why they are being stored, and can spend the majority of his or her time on this project.
- A web developer who understands how to generate broadly browser compatible HTML dynamically and has a sense of how to make user-friendly web pages.
- A programmer who understands the precise database and software implementation that actually writes the database and supporting software.
- A specific contact person in the systems group, if not a dedicated systems person who is fully part of the computer administration team and who can forewarn people of systems upgrades and provide support during specific systems problems. Out of all the positions, this one is often the most overlooked.

Occasionally one person can take on

more than one of these roles, but it is rare for a single person to do three or more.

For larger databases, these often become separate groups. Our experience is to try to keep these groups although identifiable still as one broader 'team', with, for example, all the code base accessible to all groups. Globally accessible code is important not only so that any developer can in theory have a complete local system, but also so that developers can insert debugging statements into other parts of the code base while tracking down bugs. When there is a bug in a piece of software, people's natural tendency is to suggest it is not in their piece of code, and only by being able to track a problem up and down all the layers of the code can bugs be found and removed.

CODE AND SCHEMA DESIGN

Only have one master data model format

One has to decide on a principal data model description as a master of what a database stores. Examples would be relational database Data Description Language (DDL) definition, XML DTD or XML Schema descriptions or unified modelling language (UML) files. No matter which method is used, use just one of them as the master data model description. Our preference is now firmly in DDL definitions, but this is a 'to taste' piece of advice. Every biological data model at some point starts to mushroom towards modelling 'all of biology'. Resist this. Try to focus on the core aspects of data or analysis being generated in house and then use 'links' to other well-structured biological databases. Investment into data model design is usually time well spent, but beyond a month's worth of discussion and sketching out the returns drop off dramatically. This is mainly because no one can fully understand the task at hand and consequences of the design until one tries it out. Secondly one can expect the task to change in the time-scale of a year, so dramatic investment in data modelling becomes a futile cycle.

Sequence reagents are often crucial for inter-linking data items

An important aspect immediately grasped by computational programmers but sometimes hard to explain to biological researchers is that one needs anonymous tracking identifiers for all data items. In our experience one needs two types of tracking identifiers – internal database identifiers that are used only in database joins and programmatic bindings of code to database, and then 'published' but still anonymous identifiers which can be tracked by other databases (in Ensembl these would be an ENSG number, in Swiss-Prot/SPTrEMBL, these are accession numbers, etc). The need for externally accessible but still anonymous tracking identifiers is that if 'meaningful' names are used to track something, there is very strong likelihood that someone will wish to change the meaningful name at some point in the future. Although one might naively think that using 'update' pointers might solve this, it is a clumsy solution; far better is to have an textual identifier whose sole use is to track a particular biological object over time.

One consequence of focusing on a limited scope of biology is that one often needs to provide integration with other databases managing a related but not core aspect. It is here that the key roles of (a) well-managed tracking identifiers and (b) clear linkage of tracking identifiers to experimental results such that data sets can be combined are important. An often key experimental aspect to track are the sequenced-based reagents (eg the polymerase chain reaction (PCR) primers, actual microarray oligos or clone sequences for proteomic expression) which are used in the data generation. Often it is by using these sequence-based reagents that one can associate tracking identifiers between two related databases. Notice that this requires these sequence-based reagents to be captured in the data-gathering step for a database. Once tracking identifiers can be associated between two databases, then one often has to programmatically integrate data either to enhance your own database or to investigate some aspect of biology. There

are a number of ways to do this; from replication of an external database in house through to internet-accessible programmatic access through protocols such as CORBA or SOAP. In our experience the technical aspects are generally easily solved when all the data from each database are fully open, and so each side of the integration can see all aspects of the other system.

It almost goes without saying that we advocate a straightforward 'object-oriented' approach to code. Object orientation is a programming style that considers the unit of code to be a combination of a data structure and functions that provide access and manipulation of that structure. A crucial aspect in object orientation is the ability of differently specialising objects to share common functional interfaces. For example, both a 'gene' and a 'marker location' might be specialisations of 'sequence feature' as there is the common aspect of position on a genome in both cases. Wherever sensible, people should isolate code paths from each other to allow the easy understanding of the code and also the potential for easy debugging and replacement of code. Interestingly it is actually quite easy to develop object-oriented code that does not isolate components (eg by having a large, complex base class) or does not help debugging (eg with extreme multiple inheritance and delegation layers for a function call). We could probably write a long paper on many bad designs we have either designed ourselves, or directly been part of. Our only advice is to aim primarily for simplicity and then for simultaneous isolation of different components and reuse of code – this is easy to say and hard to do well.

There are always tough decisions to balance responsiveness vs re-engineering

Software testing is another 'common-sense' part of programming. Like many software developers, we are not immune from the belief that our code is right simply because it compiles and runs on one example. If we are honest with ourselves, this is just not true. For code bases involving more than two modules, independent testing is crucial, and over time, software testing becomes the only way to have confidence that changes have not compromised another piece of functionality. We make heavy use the Perl test harness system and Java unit tests. In a database system, an additional problem is that one needs a small 'test instance' of data for the code to interact with to run tests. We have solved this by having compressed copies of test instance data and written (in the case of Perl) our own test harness system that initialises, loads data, runs tests and then destroys the data.

As a project progresses, aspects of the problem will either become clearer or indeed change, and the scope will almost certainly shift. This happens to every database project (whether biological or not) and is simply part of database life. Parts of the system will become inappropriate or just accumulate a series of short-term fixes around less optimal aspects. There are generally tough decisions that must balance the delivery of the system to the actual users against the stability and potential of the system to adjust to change in the future. There are no clear guidelines we know of to make these decisions, but both extremes are wrong. Always rushing a short-term fix on top of a previous pressurised fix condemns the project to an eventual dead-end where the system cannot be sensibly moved to accommodate any change. In some worse-case scenarios, even an OS version upgrade may cause bizarre, unexplained behaviour. Conversely, always re-engineering cleanly as soon as a problem is presented nearly always leads to delivery delays and in the worse case prevents the database ever being delivered.

As with any team, communication and trust are crucial. The type of communication and trust across a team of programmers generally has to be deeper than a normal molecular biology laboratory where post-docs and students are relatively independent. A software team has to have trust in other people's

code, design decisions and debugging; it must treat all bugs as common problems and sense when one person is overloaded and move tasks off them to relieve pressure. Such communication and trust have to be developed and maintained actively, and the leaders of the project have to take responsibility for ensuring that this occurs.

KEEPING A BIOLOGICAL FOCUS

A final common problem to biological databases is to lose the biological focus of the database; symptoms can include having web pages that have prominent displays of the database-oriented information (such as tracking identifiers) or deploying technology to access databases which requires considerable programming skill to simply install. As people become more confident in their programming skills, this route becomes increasingly tempting as it seems 'natural' to enhance the database in this way.

Keeping a good eye on the usage of the database and the needs of people using it is the only way to stay grounded. One easy way to achieve this is to hire in the 'bioinformatician' role people with directly the biological experience of the area one is working in (eg a geneticist for a genetics database, a cell biologist for a cell movement database), but even then these individuals can 'go native' and become entranced by aspects of the technology. It is also often hard to find someone who has specific experience of an area of molecular biology and some programming skills, and we would not recommend hiring molecular biologists, however appropriate their wet biology is, if they have no or 'just starting' programming skills. Another route is to embed the whole group in an institute that will use this database. This usually has a very positive effect of ensuring that the database remains focused on providing useful information, but there are down-sides. Firstly, the database group can become captured by the local research groups and effectively change the database into a glorified LIMS (Laboratory Information Management System) for these groups. There is nothing wrong with running a good LIMS group, but it should not be confused with running a useful scientific database that presents biological understanding to a broad group of researchers. Secondly, in most departments or research institutes dedicated to wet biology work it is often hard to get the right sort of IT infrastructure support, with good machine room handling, server administration and network support.

In any situation it also useful to have regular (for example, yearly or six monthly) focused meetings with biological users, ideally with some spread of geography and types of user. Such 'focus group' meetings can be very productive about understanding new research emerging and the needs of the users. However, be aware that if one simply asks a set of biologists what they would like, often the 'wish list' goes out of scope of the database and can easily become unfeasible to execute – remember that they rarely understand which things are easy and which hard to execute. Considerable feedback between biologists and database programmers is often required, and this usually means that the early part of the day is spent learning each other's language such that people can actually describe the details of aspects of work. By doing this regularly with some continuity of the invited biologists, one can actually build up a very productive relationship. These focus group meetings, along with good usage statistics, are also often precisely what funding agencies want to see in reviews.

CONCLUSIONS

Designing, implementing and running databases are predominantly a series of decisions about intricate details. Experience is the best teacher about how to make these decisions, but you can learn

from other people's mistakes. We hope this paper goes some way to passing on our experience. If you are a database neophyte and want to gain experience, there is nothing better than both investing your own personal time into examining an existing database project and, ideally, embedding yourself inside a well-managed database group. There are now a number of these groups worldwide, many of them represented in this journal. We hope you are successful in your database endeavour.

References

1. Clamp, M., Andrews, D., Barker, D. *et al.* (2003), 'Ensembl 2002: Accommodating comparative genomics', *Nucleic Acids Res.*, Vol. 31(1), pp. 38–42.
2. Birney, E., Andrews, D., Barker, D. *et al.* (2004), 'An overview of Ensembl', *Genome Res.*, accepted for publication.