PhyloInformatics 7: 1-66 - 2005

Relational Database Design and Implementation for Biodiversity Informatics

Paul J. Morris

The Academy of Natural Sciences

1900 Ben Franklin Parkway, Philadelphia, PA 19103 USA

Received: 28 October 2004 - Accepted: 19 January 2005

Abstract

The complexity of natural history collection information and similar information within the scope of biodiversity informatics poses significant challenges for effective long term stewardship of that information in electronic form. This paper discusses the principles of good relational database design, how to apply those principles in the practical implementation of databases, and examines how good database design is essential for long term stewardship of biodiversity information. Good design and implementation principles are illustrated with examples from the realm of biodiversity information, including an examination of the costs and benefits of different ways of storing hierarchical information in relational databases. This paper also discusses typical problems present in legacy data, how they are characteristic of efforts to handle complex information in simple databases, and methods for handling those data during data migration.

Introduction

The data associated with natural history collection materials are inherently complex. Management of these data in paper form has produced a variety of documents such as catalogs, specimen labels, accession books, stations books, map files, field note files, and card indices. The simple appearance of the data found in any one of these documents (such as the columns for identification, collection locality, date collected, and donor in a handwritten catalog ledger book) mask the inherent complexity of the information. The appearance of simplicity overlying highly complex information provides significant challenges for the management of natural history collection information (and other systematic and biodiversity information) in electronic form. These challenges include management of legacy data produced during the history of capture of natural history collection information into database management systems of increasing sophistication and complexity.

In this document, I discuss some of the issues involved in handling complex biodiversity information, approaches to the stewardship of such information in electronic form, and some of the tradeoffs between different approaches. I focus on the very well understood concepts of relational database design and implementation. Relational[1] databases have a strong (mathematical) theoretical foundation

[1] Object theory offers the possibility of handling much of the complexity of biodiversity information in object oriented databases in a much more effective manner than in relational databases, but object oriented and object-relational database software is much less mature and much less standard than relational database software. Data stored in a relational DBMS are currently much less likely to become trapped in a dead end with no possibility of support than data in an object oriented DBMS.

(Codd, 1970; Chen, 1976), and a wide range of database software products available for implementing relational databases.

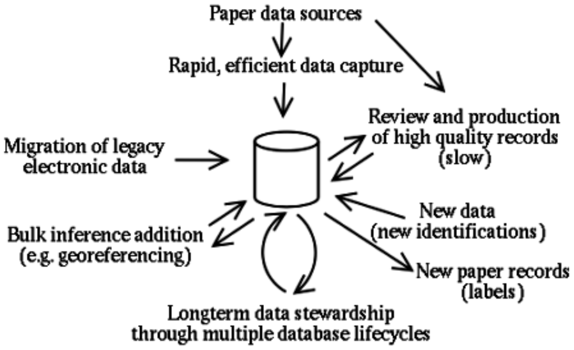

Paper data sources
Rapid, efficient data capture
Review and production of high quality records (slow)
Migration of legacy electronic data
New data (new identifications)
Bulk inference addition (e.g. georeferencing)
New paper records (labels)
Longterm data stewardship through multiple database lifecycles

Figure 1. Typical paths followed by biodiversity information. The cylinder represents storage of information in electronic form in a database.

The effective management of biodiversity information involves many competing priorities (Figure 1). The most important priorities include long term data stewardship, efficient data capture (e.g. Beccaloni et al., 2003), creating high quality information, and effective use of limited resources. Biodiversity information storage systems are usually created and maintained in a setting of limited resources. The most appropriate design for a database to support long term stewardship of biodiversity information may not be a complex highly normalized database well fitted to the complexity of the information, but rather may be a simpler design that focuses on the most important information. This is not to say that database design is not important. Good database design is vitally important for stewardship of biodiversity information. In the context of limited resources, good design includes a careful focus on what information is most important, allowing programming and database administration to best support that information.

Database Life Cycle

As natural history collections data have been captured from paper sources (such as century old handwritten ledgers) and have accumulated in electronic databases, the natural history museum community has observed that electronic data need much more upkeep than paper records (e.g. National Research Council, 2002 p.62-63). Every few years we find that we need to move our electronic data to some new database system. These migrations are usually driven by changes imposed upon us by the rapidly changing landscape of operating systems and software. Maintaining a long obsolete computer running a long unsupported operating system as the only means we have to work with data that reside in a long unsupported database program with a custom front end written in a language that nobody writes code for anymore is not a desirable situation. Rewriting an entire collections database system from scratch every few years is also not a desirable situation. The computer science folks who think about databases have developed a conceptual approach to avoiding getting stuck in such unpleasant situations – the database life cycle (Elmasri and Navathe, 1994). The database life cycle recognizes that database management systems change over time and that accumulated data and user interfaces for accessing those data need to be migrated into new systems over time. Inherent in the database life cycle is the insight that steps taken in the process of developing a database substantially impact the ease of future migrations.

A textbook list (e.g. Connoly et al., 1996) of stages in the database life cycle runs something like this: Plan, design, implement, load legacy data, test, operational maintenance, repeat. In slightly more detail, these steps are:

1. Plan (planning, analysis, requirements collection).
2. Design (Conceptual database design, leading to information model, physical database design [including system architecture], user interface design).
3. Implement (Database implementation, user interface implementation).
4. Load legacy data (Clean legacy data, transform legacy data, load legacy data).
5. Test (test implementation).
6. Put the database into production use and perform operational maintenance.
7. Repeat this cycle (probably every ten years or so).

Being a visual animal, I have drawn a diagram to represent the database life cycle (Figure 2). Our expectation of databases should not be that we capture a large quantity of data and are done, but rather that we will need to cycle those data through

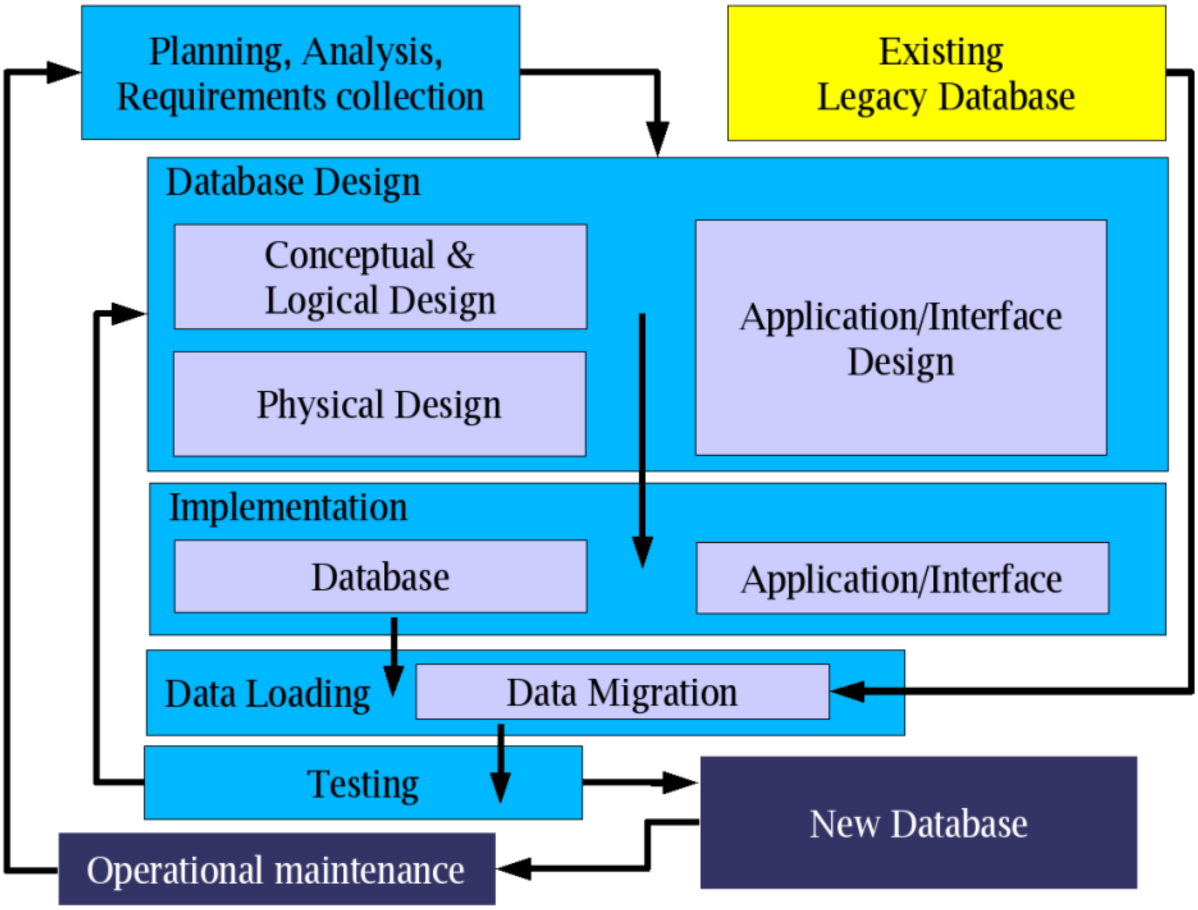


Figure 2. The Database Life Cycle

the stages of the database life cycle many times.

In this paper, I will focus on a few parts of the database life cycle: the conceptual and logical design of a database, physical design, implementation of the database design, implementation of the user interface for the database, and some issues for the migration of data from an existing legacy database to a new design. I will provide examples from the context of natural history collections information. Plan ahead. Good design involves not just solving the task at hand, but planning for long term stewardship of your data.

Levels and architecture

A requirements analysis for a database system often considers the network architecture of the system. The difference between software that runs on a single workstation and software that runs on a server and is accessed by clients across a network is a familiar concept to most users of collections information. In some cases, a database for a collection running on a single workstation accessed by a single user provides a perfectly adequate solution for the needs of a collection, provided that the workstation is treated as a server with an uninterruptible power supply, backup devices and other means to maintain the integrity of the database. Any computer running a database should be treated as a server, with all the supporting infrastructure not needed for the average workstation. In other cases, multiple users are capturing and retrieving data at once (either locally or globally), and a database system capable of running on a server and being accessed by multiple clients over a network is necessary to support the needs of a collection or project.

It is, however, more helpful for an understanding of database design to think about the software architecture. That is, to think of the functional layers involved in a database system. At the bottom level is the DBMS (database management system [see glossary, p.64]), the software that runs the database and stores the data (layered below this is the operating system and its filesystem, but we can ignore these for now). Layered above the DBMS is your actual database table or schema layer. Above this may be various code and network transport layers, and finally, at the top, the user interface through which people enter and retrieve data (Figure 29). Some database software packages allow easy separation of these layers, others are monolithic, containing database, code, and front end into a single file. A database system that can be separated into layers can have advantages, such as multiple user interfaces in multiple languages over a single data source. Even for monolithic database systems, however, it is helpful to think conceptually of the table structures you will use to store the data, code that you will use to help maintain the integrity of the data (or to enforce business rules), and the user interface as distinct components, distinct components that have their own places in the design and implementation phases of the database life cycle.

Relational Database Design

Why spend time on design? The answer is simple:

Poor Design + Time = Garbage

As more and more data are entered into a poorly designed database over time, and as existing data are edited, more and more errors and inconsistencies will accumulate in the database. This may result in both entirely false and misleading data accumulating in the database, or it may result in the accumulation of vast numbers of inconsistencies that will need to be cleaned up before the data can be usefully migrated into another database or linked to other datasets. A single extremely careful user working with a dataset for just a few years may be capable of maintaining clean data, but as soon as multiple users or more than a couple of years are involved, errors and inconsistencies will begin to creep into a poorly designed database.

Thinking about database design is useful for both building better database systems and for understanding some of the problems that exist in legacy data, especially those entered into older database systems. Museum databases that began development in the 1970s and early 1980s prior to the proliferation of effective software for building relational databases were often written with single table (flat file) designs. These legacy databases retain artifacts of several characteristic field structures that were the result of careful design efforts to both reduce the storage space needed by the database and to handle one to many relationships between collection objects and concepts such as identifications.

Information modeling

The heart of conceptual database design is information modeling. Information modeling has its basis in set algebra, and can be approached in an extremely complex and mathematical fashion. Underlying this complexity, however, are two core concepts: atomization and reduction of redundant information. Atomization means placing only one instance of a single concept in a single field in the database. Reduction of redundant information means organizing a database so that a single text string representing a single piece of information (such as the place name Democratic Republic of the Congo) occurs in only a single row of the database. This one row is then related to other information (such as localities within the DRC) rather than each row containing a redundant copy of the country name.

As information modeling has a firm basis in set theory and a rich technical literature, it is usually introduced using technical terms. This technical vocabulary include terms that describe how well a database design applies the core concepts of atomization and reduction of redundant information (first normal form, second normal form, third normal form, etc.) I agree with Hernandez (2003) that this vocabulary does not make the best introduction to information modeling[2] and, for the beginner, masks the important underlying concepts. I will thus

[2] I do, however, disagree with Hernandez' entirely free form approach to database design.

describe some of this vocabulary only after examining the underlying principles.

Atomization

1) Place only one concept in each field.

Legacy data often contain a single field for taxon name, sometimes with the author and year also included in this field. Consider the taxon name *Palaeozygopleura hamiltoniae* (HALL, 1868). If this name is placed as a string in a single field “Palaeozygopleura hamiltoniae (Hall, 1868)”, it becomes extremely difficult to pull the components of the name apart to, say, display the species name in italics and the author in small caps in an html document: <em>Palaeozygopleura hamiltoniae</em> (H<font size=-2>ALL</font>, 1868), or to associate them with the appropriate tags in an XML document. It likewise is much harder to match the search criteria Genus=Loxonema and Trivial=hamiltoniae to this string than if the components of the name are separated into different fields. A taxon name table containing fields for Generic name, Subgeneric name, Trivial Epithet, Authorship, Publication year, and Parentheses is capable of handling most identifications better than a single text field. However, there are lots more complexities – subspecies, varieties, forms, cf., near, questionable generic placements, questionable identifications, hybrids, and so forth, each of which may need its own field to effectively handle the wide range of different variations of taxon names that can be used as identifications of collection objects. If a primary purpose of the data set is nomenclatural, then substantial thought needs to be placed into this complexity. If the primary purpose of the data set is to record information associated with collection objects, then recording the name used and indicators of uncertainty of identification are the most important concepts.

2) Avoid lists of items in a field.

Legacy data often contain lists of items in a single field. For example, a remarks field may contain multiple remarks made at different times by different people, or a geographic distribution field may contain a list of geographic place names. For example, a geographic distribution field might contain the list of values “New York; New Jersey; Virginia; North Carolina”. If only one person has maintained the data set for only a few years, and they have been very careful, the delimiter “;” will separate all instances of geographic regions in each string. However, you are quite likely to find that variant delimiters such as “,” or “ ” or “:” or “'” or “|” have crept into the data.

Lists of data in a single field are a common legacy solution to the basic information modeling concept that one instance of one sort of data (say a species name) can be related to many other instances of another sort of data. A species can be distributed in many geographic regions, or a collection object can have many identifications, or a locality can have many collections made from it. If the system you have for storing data is restricted to a single table (as in many early database systems used in the Natural History Museum community), then you have two options for capturing such information. You can repeat fields in the table (a field for current identification and another field for previous identification), or you can list repeated values in a single field (hopefully separated by a consistent delimiter).

Reducing Redundant Information

The most serious enemy of clean data in long -lived database systems is redundant copies of information. Consider a locality table containing fields for country, primary division (province/state), secondary division (county/parish), and named place (municipality/city). The table will contain multiple rows with the same value for each of these fields, since multiple localities can occur in the vicinity of one named place. The problem is that multiple different text strings represent the same concept and different strings may be entered in different rows to record the same information. For example, Philadelphia, Phil., City of Philadelphia, Philladelphia, and Philly are all variations on the name of a particular named place. Each makes sense when written on a specimen label in the context of other information (such as country and state), as when viewed as a single locality record. However, finding all the specimens that come from this place in a database that contains all of these variations is not an easy task. The Academy ichthyology collection uses a legacy Muse database with this structure (a single table for locality information), and it contains some 16 different forms of "Philadelphia, PA, USA" stored in atomized named place, state, and country fields. It is not a trivial task to search this database on locality information and be sure you have located all relevant records. Likewise, migration of these data into a more normal database requires extensive cleanup of the data and is not simply a matter of moving the data into new tables and fields.

The core problem is that simple flat tables can easily have more than one row containing the same value. The goal of normalization is to design tables that enable users to link to an existing row rather than to enter a new row containing a duplicate of information already in the database.

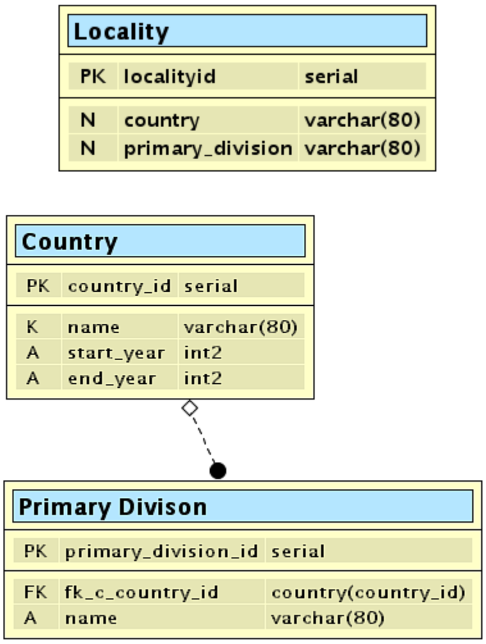


Figure 3. Design of a flat locality table (top) with fields for country and primary division compared with a pair of related tables that are able to link multiple states to one country without creating redundant entries for the name of that country. The notation and concepts involved in these Entity-Relationship diagrams are explained below.

Contemplate two designs (Figure 3) for holding a country and a primary division (a state, province, or other immediate subdivision of a country): one holding country and primary division fields (with redundant information in a single locality table), the other normalizing them into country and primary division tables and creating a relationship between countries and states.

Rows in the single flat table, given time, will accumulate discrepancies between the name of a country used in one row and a different text string used to represent the same country in other rows. The problem arises from the redundant entry of the Country name when users are unaware of existing values when they enter data and are freely able to enter any text string in the relevant field. Data in a flat file locality table might look something like those in Table 1:

Table 1. A flat locality table.

Locality id	*Country*	*Primary Division*
300	USA	Montana
301	USA	Pennsylvania
302	USA	New York
303	United States	Massachusetts

Examination of the values in individual rows, such as, "USA, Montana", or "United States, Massachusetts" makes sense and is easily intelligible. Trying to ask questions of this table, however, is a problem. How many states are there in the "USA"? The table can't provide a correct answer to this question unless we know that "USA" and "United States" both occur in the table and that they both mean the same thing.

The same information stored cleanly in two related tables might look something like those in Table 2:

Table 2. Separating Table 1 into two related tables, one for country, the other for primary division (state/province/etc.).

Country id	*Name*
300	USA
301	Uganda

Primary Division id	*fk_c_country_id*	*Primary Division*
300	300	Montana
301	300	Pennsylvania
302	300	New York
303	300	Massachusetts

Here there is a table for countries that holds one row for USA, together with a numeric Country_id, which is a behind the scenes database way for us to find the row in the table containing "USA' (a surrogate numeric primary key, of which I will say more later). The database can follow the country_id field over to a primary division table, where it is recorded in the fk_c_country_id field (a foreign key, of which I will also say more later). To find the primary divisions within USA, the database can look at the Country_id for USA (300), and then find all the rows in the primary division table that have a fk_c_country_id of 300. Likewise, the database can follow these keys in the opposite direction, and find the country for Massachusetts by looking up its fk_c_country_id in the country_id field in the country table.

Moving country out to a separate table also allows storage of a just one copy of other pieces of information associated with a country (its northernmost and southernmost bounds or its start and end dates, for example). Countries have attributes (names, dates, geographic areas, etc) that shouldn't need to be repeated each time a country is mentioned. This is a central idea in relational database design – avoid repeating the same information in more than one row of a table.

It is possible to code a variety of user interfaces over either of these designs, including, for example, one with a picklist for country and a text box for state (as in Figure 4). Over either design it is possible to enforce, in the user interface, a rule that data entry personnel may only pick an existing country from the list. It is possible to use code in the user interface to enforce a rule that prevents users from entering Pennsylvania as a state in the USA and then separately entering Pennsylvania as a state in the United States. Likewise, with either design it is possible to code a user interface to enforce other rules such as constraining primary divisions to those known to be subdivisions of the selected country (so that Pennsylvania is not recorded as a subdivision of Albania).

By designing the database with two related tables, it is possible to enforce these rules at the database level. Normal data entry personnel may be granted (at the database level) rights to select information from the country table, but not to change it. Higher level curatorial personnel may be granted rights to alter the list of countries in the country table. By separating out the country into a separate table and restricting access rights to that table in the database, the structure of the database can be used to turn the country table into an authority file and enforce a controlled vocabulary for entry of country names. Regardless of the user interface, normal data entry personnel may only link Pennsylvania as a state in USA. Note that there is nothing inherent in the normalized country/primary division tables themselves that prevents users who are able to edit the controlled vocabulary in the Country Table from entering redundant rows such as those below in Table 3. Fundamentally, the users of a database are responsible for the quality of the data in that database. Good design can only assist them in maintaining data quality. Good design alone cannot ensure data quality.

It is possible to enforce the rules above at the user interface level in a flat file. This enforcement could use existing values in the country field to populate a pick list of country names from which the normal data entry user may only select a value and may not enter new values. Since this rule is only enforced by the programing in the user interface it could be circumvented by users. More importantly, such a business rule embedded in the user interface alone can easily be forgotten and omitted when data are migrated from one database system to another.

Table 3. Country and primary division tables showing a pair of redundant Country values.

Country id	*Name*
500	USA
501	United States

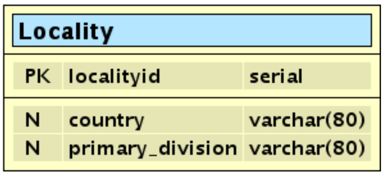
Primary Division id	*fk_c_country_id*	*Primary Division*
300	500	Montana
301	500	Pennsylvania
302	500	New York
303	501	Massachusetts

Normalized tables allow you to more easily embed rules in the database (such as restricting access to the country table to highly competent users with a large stake in the quality of the data) that make it harder for users to degrade the quality of the data over time. While poor design ensures low quality data, good design alone does not ensure high quality data.

Good design thus involves careful consideration of conceptual and logical design, physical implementation of that conceptual design in a database, and good user interface design, with all else following from good conceptual design.

Entity-Relationship modeling

Understanding the concepts to be stored in the database is at the heart of good database design (Teorey, 1994; Elmasri and Navathe, 1994). The conceptual design phase of the database life cycle should produce a result known as an information model (Bruce, 1992). An information model consists of written documentation of concepts to be stored in the database, their relationships to each other, and a diagram showing those concepts and their relationships (an Entity-Relationship or E-R diagram,). A number of information models for the biodiversity informatics community exist (e.g. Blum, 1996a; 1996b; Berendsohn et al., 1999; Morris, 2000; Pyle 2004), most are derived at least in part from the concepts in ASC model (ASC, 1992). Information models define entities, list attributes for those entities, and relate entities to each other. Entities and attributes can be loosely thought of as tables and fields. Figure 5 is a diagram of a locality entity with attributes for a mysterious localityid, and attributes for country and primary division. As in the example above, this entity can be implemented as a table with localityid, country, and primary division fields (Table 4).



Locality

PK	localityid	serial
N	country	varchar(80)
N	primary_division	varchar(80)

Figure 5. Part of a flat locality entity. An implementation with example data is shown below in Table 4.

Table 4. Example locality data.

Locality id	Country	Primary Division
300	USA	Montana
301	USA	Pennsylvania

Entity-relationship diagrams come in a variety of flavors (e.g. Teorey, 1994).   The Chen (1976) format for drawing E-R diagrams uses little rectangles for entities and hangs oval balloons off of them for attributes. This format (as in the distribution region entity shown on the right in Figure 6 below) is very useful for scribbling out drafts of E-R diagrams on paper or blackboard. Most CASE (Computer Aided Software Engineering) tools for working with databases, however, use variants of the IDEF1X format, as in the locality entity above (produced with the open source tool Druid [Carboni et al, 2004]) and the collection object entity on the left in Figure 6 (produced with the proprietary tool xCase [Resolution Ltd., 1998]), or the relationship diagram tool in MS Access.   Variants of the IDEF1X format (see Bruce, 1992) draw entities as rectangles and list attributes for the entity within the rectangle.

Not all attributes are created equal. The diagrams in Figures 5 and 6 list attributes that have “ID” appended to the end of their names (localityid, countryid, collection _objectid, intDistributionRegionID). These are primary keys. The form of this notation varyies from one E-R diagram format to another, being the letters PK, or an underline, or bold font for the name of the primary key attribute.   A primary key can be thought of as a field that contains unique values that let you identify a particular row in a table. A country name field could be the primary key for a country table, or, as in the examples here, a surrogate numeric field could be used as the primary key.

To give one more example of the relationship between entities as abstract concepts in an E-R model and tables in a database, the tblDistributionRegion entity shown in Chen notation in Figure 6 could be implemented as a table, as in Table 5, with a field for its primary key attribute, intDistributionRegionID, and a second field for the region name attribute vchrRegionName. This example is a portion of the structure of the table that holds geographic distribution area names in a BioLink database (additional fields hold the relationship between regions, allowing Pennsylvania to be nested as a geographic region within the United States nested within North America, and so on).

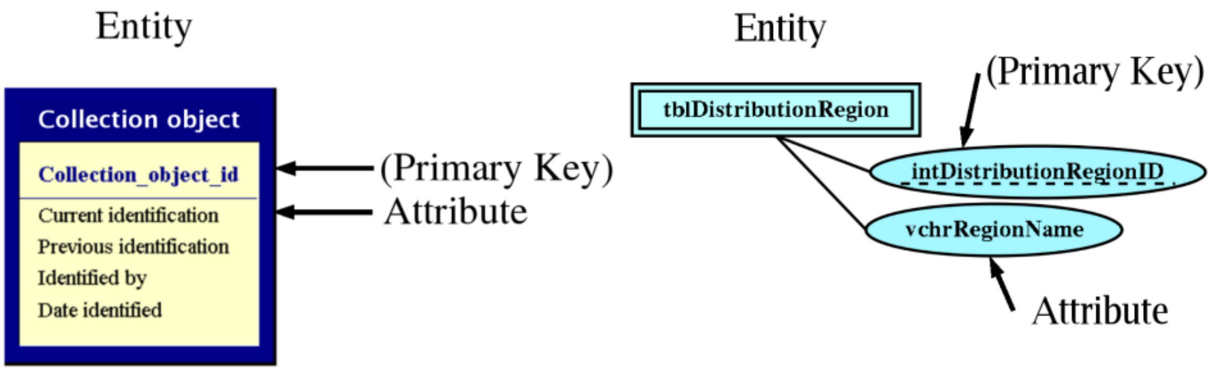


Figure 6. Comparison between entity and attributes as depicted in a typical CASE tool E-R diagram in a variant of the IDEF1X format (left) and in the Chen format (right, which is more useful for pencil and paper modeling). The E-R diagrams found in this paper have variously been drawn with the CASE tools xCase and Druid or the diagram editor DiA.

Table 5. A portion of a BioLink (CSIRO, 2001) tblDistributionRegion table.

intDistributionRegionID	*vchrRegionName*
15	Australia
16	Queensland
17	Uganda
18	Pennsylvania

The key point to think about when designing databases is that things in the real world can be thought of in general terms as entities with attributes, and that information about these concepts can be stored in the tables and fields of a relational database. In a further step, things in the real world can be thought of as objects with properties that can do things (methods), and these concepts can be mapped in an object model (using an object modeling framework such as UML) that can be implemented with an object oriented language such as Java. If you are programing an interface to a relational database in an object oriented language, you will need to think about how the concepts stored in your database relate to the objects manipulated in your code. Entity-Relationship modeling produces the critical documentation needed to understand the concepts that a particular relational database was designed to store.

Primary key

Primary keys are the means by which we locate a single row in a table. The value for a primary key must be unique to each row. The primary key in one row must have a different value from the primary key of every other row in the table. This property of uniqueness is best enforced by the database applying a unique index to the primary key.

A primary key need not be a single attribute. A primary key can be a single attribute containing real data (generic name), a group of several attributes (generic name, trivial epithet, authorship), or a single attribute containing a surrogate key (name_id). In general, I recommend the use of surrogate numeric primary keys for biodiversity informatics information, because we are too seldom able to be certain that other potential primary keys (candidate keys) will actually have unique values in real data.

A surrogate numeric primary key is an attribute that takes as values numbers that have no meaning outside the database. Each row contains a unique number that lets us identify that particular row. A table of species names could have generic epithet and trivial epithet fields that together make a primary key, or a single species_id field could be used as the key to the table with each row having a different arbitrary number stored in the species_id field. The values for species_id have no meaning outside the database, and indeed should be hidden from the users of the database by the user interface. A typical way of implementing a surrogate key is as a field containing an automatically incrementing integer that takes only unique values, doesn't take null values, and doesn't take blank values. It is also possible to use a character field containing a globally unique identifier or a cryptographic hash that has a high probability of being globally unique as a surrogate key, potentially increasing the ease with which different data sets can be combined.

The purpose of a surrogate key is to provide a unique identifier for a row in a table, a unique identifier that has meaning only internally within the database. Exposing a surrogate key to the users of the database may result in their mistakenly assigning a meaning to that key outside of the database. The ANSP malacology and invertebrate paleontology collections were for a while printing a primary key of their master collection object table (a field called serial number) on specimen labels along with the catalog number of the specimen, and some of these serial numbers have been copied by scientists using the collection and have even made it into print under the rational but mistaken belief that they were catalog numbers. For example, Petuch (1989, p.94) cites the number **ANSP 1133** for the paratype of *Malea springi*, which actually has the catalog number **ANSP 54004**, but has both this catalog number and the serial number 00001133 printed on a computer generated label. Another place where surrogate numeric keys are easily exposed to users and have the potential of taking on a broader meaning is in Internet databases. An Internet request for a record in a database is quite likely to request that record through its primary key. An URL with a http get request that contains the value for a surrogate key directly exposes the surrogate key to the world . For example, the URL: http://erato.acnatsci.org/wasp/search.php?species=12563 uses the value of a surrogate key in a manner that users can copy from their web browsers and email to each other, or that can be crawled and stored by search engines, broadening its scope far beyond simply being an arbitrary row identifier within the database.

Surrogate keys come with risks, most notably that, without other rules being enforced, they will allow duplicate rows, identical in all attributes except the surrogate primary key, to enter the table (country 284, USA; country 526, USA). A real attribute used as a primary key will force all rows in the table to contain unique values (USA). Consider catalog numbers. If a table contains information about collection objects within one catalog number series, catalog number would seem a logical choice for a primary key. A single catalog number series should, in theory, contain only one catalog number per collection object. Real collections data, however, do not usually conform to theory. It is not unusual to find that 1% or more of the catalog numbers in an older catalog series are duplicates. That is, real duplicates, where the same catalog number was assigned to two or more different collection objects, not simply transcription errors in data capture. Before the catalog number can be used as the primary key for a table, or a unique index can be applied to a catalog number field, duplicate values need to be identified and resolved. Resolving duplicate catalog numbers is a non-trivial task that involves locating and handling the specimens involved. It is even possible for a collection to contain real immutable duplicate catalog numbers if the same catalog number was assigned to two different type specimens and these duplicate numbers have been published. Real collections data, having accumulated over the last couple hundred years, often contain these sorts of unexpected inconsistencies. It is these sorts of problematic data and the limits on our resources to fully clean data to fit theoretical expectations that make me recommend the use of surrogate keys as primary keys in most tables in collections databases.

Taxon names are another case where a surrogate key is important. At first glance, a table holding species names could use the generic name, trivial epithet, and authorship fields as a primary key. The problem is, there are homonyms and other such historical oddities to be found in lists of taxon names. Indeed, as Gary Rosenberg has been saying for some years, you need to know the original genus, species epithet, subspecies epithet, varietal epithet (or trivial epithet and rank of creation), authorship, year of publication, page, plate and figure to uniquely distinguish names of Mollusks (there being homonyms described by the same author in the same publication in different figures).

Normalize appropriately for your problem and resources

When building an information model, it is very easy to get carried away and expand the model to cover in great elaboration each tiny facet of every piece of information that might be related to the concept at hand. In some situations (e.g. the POSC model or the ABCD schema) where the goal is to elaborate all of the details of a complex set of concepts, this is very appropriate. However, when the goal is to produce a functional database constructed by a single individual or a small programming team, the model can easily become so elaborate as to hinder the production of the software needed to reach the desired goal. This is the real art of database design (and object modeling); knowing when to stop. Normalization is very important, but you must remember that the ultimate goal is a usable system for the storage and retrieval of information.

In the database design process, the information model is a tool to help the design and programming team understand the nature of the information to be stored in the database, not an end in itself. Information models assist in communication between the people who are specifying what the database needs to do (people who talk in the language of systematics and collections management) and the programmers and database developers who are building the database (and who speak wholly different languages). Information models are also vital documentation when it comes time to migrate the data and user interface years later in the life cycle of the database.

Example: Identifications of Collection Objects

Consider the issue of handling identifications that have been applied to collection objects. The simplest way of handling this information is to place a single identification field (or set of atomized genus_&_higher, species, authorship, year, and parentheses fields) into a collection object table. This approach can handle only a single identification per collection object, unless each collection object is allowed more than one entry in the collection object table (producing duplicate catalog numbers in the table for each collection object with more than one identification). In many sorts of collections, a collection object tends to accumulate many identifications over time. A structure capable of holding only one identification per collection object poses a problem.

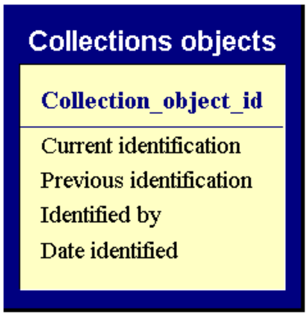


Figure 7. A non-normal collection object entity.

A standard early approach to the problem of more than one identification to a single collection object was a single table with current and previous identification fields. The collection objects table shown in Figure 7 is a fragment of a typical legacy non-normal table containing one field for current identification and one for previous identification. This example also includes a surrogate numeric key and fields to hold one identifier and one date identified.

One table with fields for current and previous identification allows rules that restrict each collection object to one record in the collection object table (such as a unique index on catalog number), but only allows for two identifications per collection object. In some collections this is not a huge problem, whereas in others this structure would force a significant information loss[3]. A tray of fossils or a herbarium sheet may each contain a long history of annotations and changes in identification produced by different people at different times. The table with one set of fields for current identification, another for previous identification and one field each for identifier and date identified suffers another problem – there is no necessary link

[3] I chose such a flat structure, with 6 fields for current identification and 6 fields for original identification for a database for data capture on the entomology collections at ANSP. It allowed construction of a more efficient data entry interface than a better normalized structure. Insect type specimens seem to very seldom have the complex identification histories typical of other sorts of collections.

between the identifications, the identifier, and the date identified. The database is agnostic as to whether the identifier was the person who made the current identification, the previous identification, or some other identification. It is also agnostic as to whether the date identified is connected to the identifier. Without carefully enforced rules in the user interface, the date identified could reflect the date of some random previous identification, the identifier could be the person who made the current identification, and the previous identification could be the oldest identification of the collection object, or these fields could hold some other arbitrary combination of information, with no way for the user to tell. We clearly need a better structure.

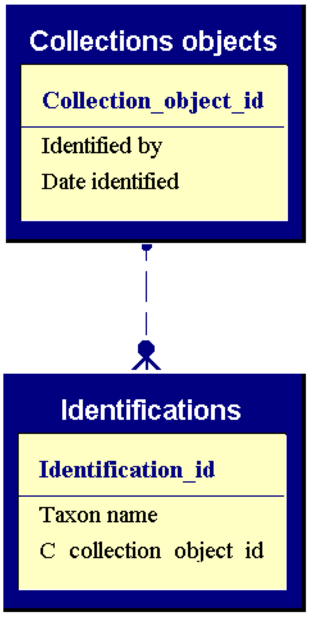


Figure 8. Moving identifications to a related entity.

We can allow multiple identifications for each collection object by adding a second table to hold identifications and linking that table to the collection object table (Figure 8). These two tables for collection object and identification can hold multiple identifications for each collection object if we include a field in the identification table that contains values from the primary key of the collection object table. This foreign key is used to link collection object records with identification records (shown by the “Crow's Foot” symbol in the figure). One naming convention for foreign keys uses the name of the primary key that is being referenced (collection_object_id) and prefixes it with c_ (for copy, thus c_collection_object_id for the foreign key). If, as in Figure 8, the identification table holds a foreign key pointing to collection objects, and a set of fields to hold a taxon name, then each collection object can have many identifications.

This pair of tables (Collection objects and Identifications, Figure 8) still has lots of problems. We don't have any way of knowing which identification is the most recent one. In addition, the taxon name fields will contain multiple duplicate values, so, for example, correcting a misspelling in a taxon name will require updating every row in the identification table holding that taxon name. Conceptually, each collection object can have multiple identifications, but each taxon name used in an identification can be applied to many collection objects. What we really want is a many to many relationship between taxon names and collection objects (Figure 9). Relational databases can not handle many to many relationships directly, but they can by interpolating a table into the middle of the relationship – an associative entity. The concepts collection object – identification – taxon name are good example of an associative entity (identification) breaking up a many to many relationship (between collection objects and taxon names). Each collection object can have many taxon names applied to it, each taxon name can be applied to many collection objects, and these applications of taxon names to collection objects occur through an identification.

In Figure 9, the identification entity is an associative entity that breaks up the many to many relationship between species names and collection objects. The identification entity contains foreign keys pointing to both the collection object and species name entities. Each collection object can have many identifications, each identification involves one and only one species name. Each species name can be used in many identifications, and each identification applies to one and only one collection object.

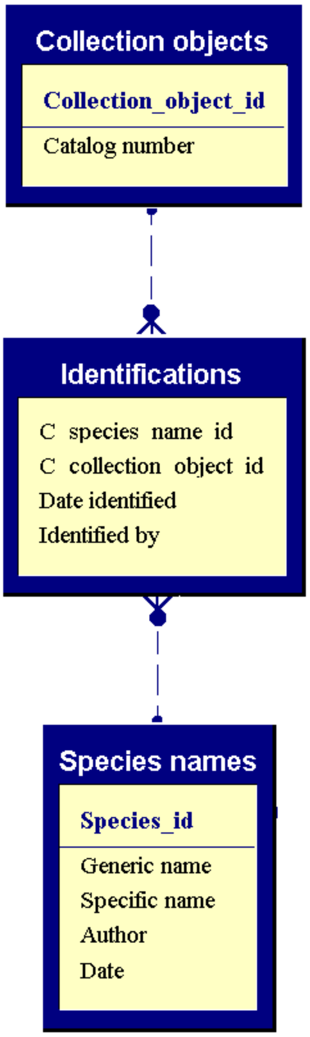


Figure 9. Using an associative entity (identifications) to link taxon names to collection objects, splitting the many to many relationship between collection objects and identifications.

This set of entities (taxon name, identification [the associative entity], and collection object) also allows us to easily track the most recent identification by adding a date identified field to the identification table. In many cases with legacy data, it may not be possible to determine the date on which an identification was made, so adding a field to flag the current identification out of a set of identifications for a specimen may be necessary as well. Note that adding a flag to track the current identification requires business rules that will need to be implemented in the code associated with the database. These business rules may specify that only one identification for a single collection object is allowed to be the current identification, and that the identification flagged as the current identification must have either no date or must have the most recent date for any identification of that collection object. An alternative, suggested by an anonymous reviewer, is to include a link to the sole current identification in the collection object table. (That is, to include a foreign key fk_current_identification_id in collection_objects, which is thus able to link a collection object to one and only one current identification. This is a very appropriate structure, and lets business rules focus on making sure that this current identification is indeed the current identification).

This identification associative entity sitting between taxon names and collection objects contains an attribute to hold the name of the person who made the identification. This field will contain many duplicate values as some people make many identifications within a collection. The proper way to bring this concept to third normal form is to move identifiers off to a generalized person table, and to make the identification entity a ternary associative entity linking species names, collection objects, and identifiers (Figure 10). People may play multiple roles in the database (and may be a subtype of a generalized agent entity), so a convention for indicating the role of the person in the identification is to add the role name to the end of the foreign key. Thus, the foreign key linking people to identifications could be called c_person_id_identifier. In another entity, say handling the concept of preparations, a foreign key linking to the people entity might be called c_person_id_preparator.

The set of concepts Taxon Name, identification (as three way associative entity), identifier, and collection object describes a way of handing the identifications of collection objects in third normal form. Person names, collection objects, and taxon names are all capable of being stored without redundant repetition of information. Placing identifiers in a separate People entity, however, requires further thought in the context of natural history collections. Legacy data will contain multiple similar entries (G. Rosenberg; Rosenberg, G.; G Rosenberg; Rosenberg; G.D. Rosenberg), all of which may or may not refer to the same person. Combining all

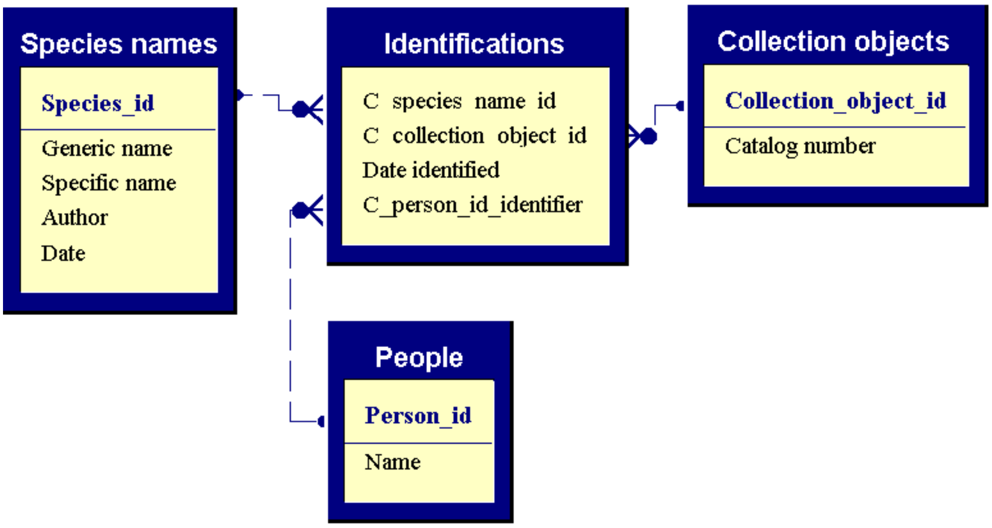


Figure 10. Normalized handling of identifications and identifiers. Identifications is an associative entity relating Collection objects, species names and people.

of these legacy entries into a normalized person table risks introducing errors of interpretation into the data. In addition, adding a generic people table and linking it to identifiers adds additional complexity and coding overhead to the database. People is one area of the database where you need to think very carefully about the costs and benefits of a highly normalized design Figure 11. Cleaning legacy data, the additional interface complexity, and the additional code required to implement a generic person as an identifier, along with the risk of propagation of incorrect inferences, may well outweigh the benefits of being able to handle identifiers in a generic people entity. Good, well normalized design is critical to be able to properly handle the existence of multiple identifications for a collection object, but normalizing the names of identifiers may lie outside the scope of the critical core information that a natural history museum has the resources to properly care for, or be beyond the scope of the critical information needed to complete a grant funded project. Knowing when to stop elaborating the information model is an important aspect of good database design.

Example extended: questionable identifications

How does one handle data such as the identification "*Palaeozygopleura hamiltoniae* (HALL, 1868) ?" that contains an indication of uncertainty as to the accuracy of the determination? If the question mark is stored as part of the taxon name (either in a single taxon name string field, or as an atomized field in a taxon name table), then you can expect your list of distinct taxon names to include duplicate entries for "*Palaeozygopleura hamiltoniae* (HALL, 1868)" and for "*Palaeozygopleura hamiltoniae* (HALL, 1868) ?". This is clearly an undesirable duplication of information.

Thinking through the nature of the uncertainty in this case, the uncertainty is an

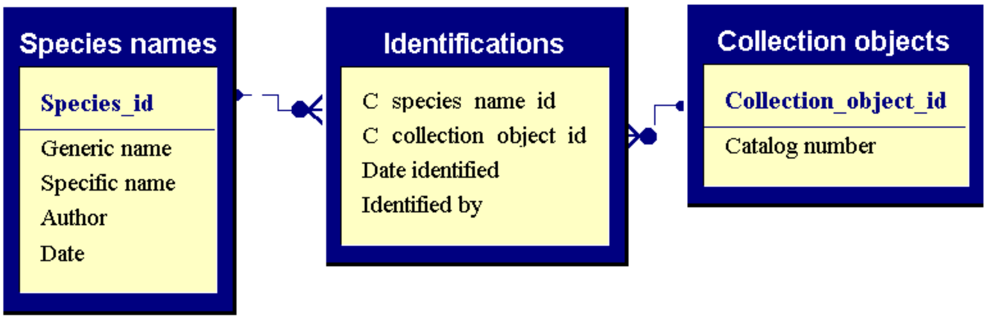


Figure 11. Normalized handling of identifications with denormalized handling of the people who perfommed the identifications (allowing multiple entries in identification containing the name of a single identifier).

Table 6. A table not in first normal form.

Catalog_number	Identification	Previous identification	Preparations
ANSP 641455	Lunatia pilla	Natica clausa	Shell, alcohol
ANSP 815325	Velutina nana	Velutina velutina	Shell

Table 7. Catalog number and identification fields from Table 6 atomized so that each field now contains only one concept.

Repository	Catalog_no	Id_genus	Id_sp	P_id_gen	P_id_sp	Preparations
ANSP	641455	Lunatia	pilla	Natica	clausa	Shell, alcohol
ANSP	815325	Velutina	nana	Velutina	velutina	Shell

attribute of a particular identification (this specimen may be a member of this species), rather than an attribute of a taxon name (though a species name can incorporate uncertain generic placement: e.g. *Loxonema? hamiltoniae* with this generic uncertainty being an attribute of at least some worker's use of the name). But, since uncertainty in identification is a concept belonging to an identification, it is best included as an attribute in an identification associative entity (Figure 11).

Vocabulary

Information modeling has a widely used technical terminology to describe the extent to which data conform to the mathematical ideals of normalization. One commonly encountered part of this vocabulary is the phrase “normal form”. The term first normal form means, in essence, that a database has only one concept placed in each field and no repeating information within one row, that is, no repeating fields and no repeating values in a field. Fields containing the value “1863, 1865, 1885” (repeating values) or the value “Paleozygopleura hamiltoniae Hall” (more than one concept), or the fields Current_identification and Previous_identification (repeating fields) are example violations of first normal form. In second normal form, primary keys do not contain redundant information, but other fields may. That is two different rows of a table may not contain the same values in their primary key fields in second normal form. For example, a collection object table containing a field for catalog number serving as primary key would not be able to contain more than one row for a single catalog number for the table to be in second normal form. We do not expect a table of collection objects to contain information about the same collection object in two different rows. Second normal form is necessary for rational function of a relational database. For catalog number to be the primary key of the collection object table, a unique index would be required to force each row in the table to have a unique value for catalog number. In third normal form, there is no redundant information in any fields except for foreign keys. A third normal treatment of geographic names would produce one and only one row containing the value “Philadelphia”, and one and only one row containing the value “Pennsylvania”.

To make normal forms a little clearer, let's work through some examples. Table 6 is a fragment of a hypothetical flat file database. Table 6 is not in first normal form. It contains three different kinds of problems that prevent it from being in first normal form (as well as other problems related to higher normal forms). First, the Catalog_number and identification fields are not atomic. Each contains more than one concept. Catalog_number contains the acronym of a repository and a catalog number. The identification fields both contain a species name, rather than separate fields for components of that name (generic name, specific epithet, etc...). Second, identification and previous identification are repeating fields. Each of these contains the same concept (an identification). Third, preparations contains a series of repeating values.

So, what transformations of the data do we need to do to bring Table 6 into first normal form? First, we must atomize, that is, split up fields until one and only one concept is contained in each field. In Table 7, Catalog_number has been split into repository and catalog_no, identification and previous identification have been split into generic name and specific epithet fields. Note that this splitting is easy to do in the design phase of a novel database but may require substantial work if existing data need to be parsed into new fields.

Table 7 still isn't in in first normal form. The previous and current identifications are held in repeating fields. To bring the table to first normal form we need to remove these repeating fields to a separate table. To link a row in our table out to rows that we remove to another table we need to identify the primary key for our table. In this case, Repository and Catalog_no together form the primary key. That is, we need to know both Repository and Catalog number in order to find a particular row. We can now build an identification table containing genus and trivial name fields, a field to identify if an identification is previous or current, and the repository and catalog_no as foreign keys to point back to our original table. We could, as an alternative, add a surrogate numeric primary key to our original table and carry this field as a foreign key to our identifications table. With an identification table, we can normalize the repeating identification fields from our original table as shown in Table 8. Our data still aren't in first normal form as the preparations field containing a list (repeating information) of preparation types.

Table 8. Current and previous identification fields from Tables 6 and 7 split out into a separate table. This pair of tables allows any number of previous identifications for a particular collections object. Note that Repository and Catalog_no together form the primary key of the first table (they could be replaced by a single surrogate numeric key).

Repository (PK)	Catalog_no (PK)	Preparations
ANSP	641455	Shell, alcohol
ANSP	815325	Shell

Repository	Catalog_no	Id_genus	Id_sp	ID_order
ANSP	641455	Lunatia	pilla	Current
ANSP	641455	Natica	clausa	Previous
ANSP	815325	Velutina	nana	Current
ANSP	815325	Velutina	velutina	Previous

Much as we did with the repeating identification fields, we can split the repeating information in the preparations field out into a separate table, bringing with it the key fields from our original table. Splitting data out of a repeating field into another table is more complicated than splitting out a pair of repeating fields if you are working with legacy data (rather than thinking about a design from scratch). To split out data from a field that hold repeating values you will need to identify the delimiter used to split values in the repeating field (a comma in this example), write a parser to walk through each row in the table, split the values found in the repeating field on their delimiters, and then write these values into the new table. Repeating values that have been entered by hand are seldom clean. Different delimiters may be used in different rows (comma or semicolon), delimiters may be missing (shell alcohol), spacing around delimiters may vary (shell,alcohol, frozen), the delimiter might be a data value in some rows(alcohol, formalin fixed; frozen, unfixed), and so on. Parsing a field containing repeating values therefore can't be done blindly. You will need to assess the results and fix exceptions (probably by hand). Once this parsing is complete, Table 9, we have a set of three tables (collection object, identification, preparation) in first normal form.

Table 9. Information in Table 6 brought into first normal form by splitting it into three tables.

Repository	Catalog_no
ANSP	641455
ANSP	815325

Repository	Catalog_no	Id_genus	Id_sp	ID_order
ANSP	641455	Lunatia	pilla	Current
ANSP	641455	Natica	clausa	Previous
ANSP	815325	Velutina	nana	Current
ANSP	815325	Velutina	velutina	Previous

Repository	Catalog_no	Preparations
ANSP	641455	Shell
ANSP	641455	Alcohol

Non-atomic data and problems with first normal form are relatively common in legacy biodiversity and collections data (handling of these issues is discussed in the data migration section below). Problems with second normal form are not particularly common in legacy data, probably because unique key values are necessary for a relational database to function. Second normal form can be a significant issue when designing a database from scratch and in flat file databases, especially those developed from spreadsheets. In second normal form, each row in a table holds a unique value for the primary key of that table. A collection object table that is not in second normal form can hold more than one row for a single collection object. In considering second normal form, we need to start thinking about keys. In the database design process we may consider candidate keys – fields that could potentially serve as keys to uniquely identify rows in a table. In a collections object table, what information do we need to know to find the row that contains information about a particular collection object? Consider Table 10. Table 10 is not in second normal form. It contains 4 rows with information about a particular collections object. A reasonable candidate for the primary key in a collections object table is the combination of Repository and Catalog number. In Table 10 these fields do not contain unique values. To uniquely identify a row in Table 10 we probably need to include all the fields in the table into a key.

Table 10. A collections object table with repeating rows for the candidate key Repository + Catalog_no.

Repository	Catalog_no	Id_genus	Id_sp	ID_order	Preparation
ANSP	641455	Lunatia	pilla	Current	Shell
ANSP	641455	Lunatia	pilla	Current	alcohol
ANSP	641455	Natica	clausa	Previous	Shell
ANSP	641455	Natica	clausa	Previous	alcohol

If we examine Table 10 more carefully we can see that it contains two independent pieces of information about a collections object. The information about the preparation is independent of the information about identifications. In formal terms, one key should determine all the other fields in a table. In Table 10, repository + catalog number + preparation are independent of repository + catalog number + id_genus + id species + id order. This independence gives us a hint on how to bring Table 10 into second normal form. We need to split the independent repeating information out into additional tables so that the multiple preparations per collection object and the multiple identifications per collection object are handled as relationships out to other tables rather than as repeating rows in the collections object table (Table 11).

Table 11. Bringing Table 10 into second normal form by splitting the repeating rows of preparation and identification out to separate tables.

Repository	Catalog_no
ANSP	641455

Repository	Catalog_no	Preparation
ANSP	641455	Shell
ANSP	641455	Alcohol

Repository	Catalog_no	Id_genus	Id_sp	ID_order
ANSP	641455	Lunatia	pilla	Current
ANSP	641455	Natica	clausa	Previous

By splitting the information associated with preparations out of the collection object table into a preparation table and information about identifications out to an identifications table (Table 11) we can bring the information in Table 10 into second normal form. Repository and Catalog number now uniquely determine a row in the collections object table (which in our limited example here now contains no other information.) Carrying the key fields (repository + catalog_no) as foreign keys out to the preparation and identification tables allows us to link the information about preparations and identifications back to the collections object. Table 11 is thus now holding the information from Table 10 in second normal form. Instead of using repository + catalog_no as the primary key to the collections object table, we could use a surrogate numeric primary key (coll_obj_ID in Table 12), and carry this surrogate key as a foreign key into the related tables.

Table 11 has still not brought the information into third normal form. The identification table will contain repeating values for id_genus and id_species – a particular taxon name can be applied in more than one identification. This is a straightforward matter of pulling taxon names out to a separate table to allow a many to many relationship between collections objects and taxon names through an identification associative entity (Table 12). Note that both Repository and Preparations could also be brought out to separate tables to remove redundant non-key entries. In this case, this is probably best accomplished by using the text value of Repository (and of Preparations) as the key, and letting a repository table act to control the allowed values for repository that can be entered into the collections object tables (rather than using a surrogate numeric key and having to follow that out to the repository table any time you wanted to know the repository of a collections object). Herein lies much of the art of information modeling – knowing when to stop.

Table 12. Bringing Table 11 into third normal form by splitting the repeating values of taxon names in identifications out into a separate table.

Repository	Catalog_no	Coll_obj_ID
ANSP	641455	100

Coll_obj_ID	Preparations
100	Shell
100	Alcohol

coll_obj_ID	C_taxon_ID	ID_order
100	1	Current
100	2	Previous

Taxon_ID	Id_genus	Id_sp
1	Lunatia	pilla
2	Natica	clausa

Producing an information model.

An information model is a detailed description of the concepts to be stored in a database (see, for example, Bruce, 1992). An information model should be sufficiently detailed for a programmer to use it to construct the back end data storage structures of the database and the code to support the business rules used to maintain the quality of the data. A formal information model should consist of at least three components: an Entity-Relationship diagram, a description of relationship cardinalities, and a detailed description of each entity and each attribute. The latter should include a description of the scope and nature of the data to be held in each attribute.

Relationship cardinalities are text descriptions of the relationships between entities. They consist of a list of sentences, one sentence for each of the two directions in which a relationship can be read. For example, the relationship between species names and identifications in the E-R diagram in could be documented as follows:

```
Each species name is used in zero or more
  identifications.
Each identification uses one and only one
  species name.
```

The text documentation for each entity and attribute explains to a programmer the scope of the entity and its attributes. The documentation should include particular attention to limits on valid content for the attributes and business rules that govern the allowed content of attributes, especially rules that govern related content spread among several attributes. For example, the documentation of the date attribute of the species names entity in Figure 11 above might define it as being a variable length character string of up to 5 characters holding a four digit year greater than 1757 and less than or equal to the current year. Another rule might say that if the authorship string for a newly entered record already exists in the database and the date is outside the range of the earliest or latest year present for that authorship string, then the data entry system should raise a warning message. Another rule might prohibit the use of a species name in an identification if the date on a species name is more recent than the year of a date identified. This is a rule that could be enforced either in the user interface or in a before insert trigger in the database.

Properly populated with descriptions of entities and attributes, many CASE tools are capable of generating text and diagrams to document a database as well as SQL (Structured Query Language) code to generate the table structures for the database with very little additional effort beyond that needed to design the database.

Example: PH core tables

As an example of an information model, I will describe the core components of the Academy's botanical collection, PH (Philadelphia Herbarium) type specimen database. This database was specifically designed for capturing data off of herbarium sheets of type specimens. The database itself is in MS Access and is much more complex than these core tables suggest. In particular, the database includes tables for handling geographic information in a more normalized form than is shown here.

The summary E-R diagram of core entities for the PH type database is shown in Figure 12. The core entity of the model is the Herbarium sheet, a row in the Herbarium sheet table represents a single herbarium sheet with one or more plant specimens attached to it. Herbarium sheets are being digitally imaged, and the database includes metadata about those images. Herbarium sheets have various sorts of annotations attached and written on them concerning the specimens attached to the sheets. Annotations can include original label data, subsequent identifications, and various comments by workers who have examined the sheet. Annotations can include taxon names, including discussion of the type status of a specimen. Figure 12 shows the entities (and key fields) used to represent this core information about a herbarium sheet.

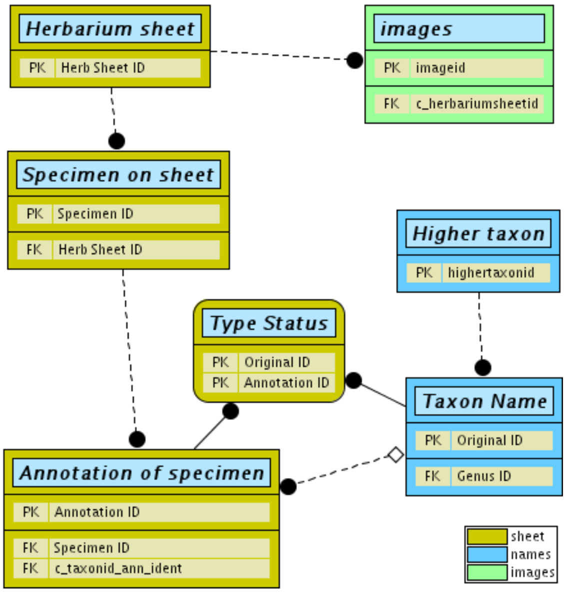


Figure 12. Core tables in the PH type database.

We can describe each of the relationships between the entities in the E-R diagram in with a pair of sentences describing the relationship cardinalities. These sentences carry the same information as the crows-foot notations on the E-R diagram, but in a more readily intelligible form. To borrow language from the object oriented programing world, they state how many instances of an entity may be related to how many instances of another entity, that is, how many rows in one table may be related to rows of another table by matching rows containing the same values for primary key (in one table) and foreign key (in the other table). The text description of relationship cardinalities can also carry additional information that a particular case tool may not include in its notation, such as a limit of an instance of one entity being related to one to three instances of another entity.

Relationship cardinalities:

Each Herbarium Sheet contains zero to many Specimens.
Each Specimen is on one and only one Herbarium sheet.

Each Specimen has zero to many Annotations.
Each Annotation applies to one and only one Specimen.

Each Herbarium sheet has zero to many Images.
Each Image is of one and only one herbarium sheet.

Each Annotation uses one and only one Taxon Name.
Each Taxon Name is used in zero to many Annotations.

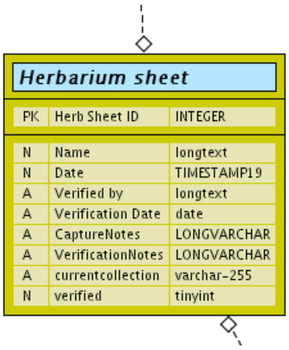
Each Annotation remarks on zero to one Type Status.
Each Type status is found in one and only one Annotation.

Each Type Status applies to one and only one Taxon Name.
Each Taxon Name has zero to many Type Status.

Each Taxon Name is the child of one and only one Higher Taxon.
Each Higher Taxon contains zero to many Taxon Names.

Each Higher Taxon is the child of zero or one Higher Taxon.
Each Higher Taxon is the parent of zero to many Higher Taxa.

The E-R diagram in describes only the core entities of the model in the briefest terms. Each entity needs to be fleshed out with a text description, attributes, and descriptions of those attributes. Figure 13 is a fragment of a larger E-R diagram with more detailed entity information for the Herbarium sheet entity. Figure 13 includes the name and data type of each attribute in the Herbarium sheet entity. The herbarium sheet entity itself contains very little information. All of the biologically interesting information about a Herbarium sheet (identifications, provenance, etc) is stored out in related tables.



Herbarium sheet

PK	Herb Sheet ID	INTEGER
N	Name	longtext
N	Date	TIMESTAMP19
A	Verified by	longtext
A	Verification Date	date
A	CaptureNotes	LONGVARCHAR
A	VerificationNotes	LONGVARCHAR
A	currentcollection	varchar-255
N	verified	tinyint

Figure 13. Fragment of PH core tables E-R diagram showing Herbarium sheet entity with all attributes listed.

Entity-relationship diagrams are still only big picture summaries of the data. The bulk of an information model lies in the entity documentation. Examine Figure 13. Herbarium sheet has an attribute called Name, and another called Date. From the E-R diagram itself, we don't know enough about what sort of information these fields might hold. As the Date field has a data type of timestamp, we could guess that it represents a timestamp generated when a row is entered into the herbarium sheet entity, but without further documentation, we can't know whether this is correct or not. The names of the attributes Name and Date are legacies of an earlier phase in the design of this database, better names for these attributes would be “Created by” and “Date created”. Entity documentation is needed to explain what these attributes are, what sort of information they should hold, and what business rules should be applied to maintain the integrity and validity of that information. Entity documentation for one entity in this model, the Herbarium sheet, follows (in Appendix A) as an example of a suitable level of detail for entity documentation. A definition, the domain of valid values, business rules, and example values all help describe the nature of the information intended to go into a table that implements this entity and can assist in physical design of the database, design of the user interface, and in future migrations of the data (Figure 1).

Physical design

An information model is a conceptual design for a database. It describes the concepts to be stored in the database. Implementation of a database from an information model involves converting that conceptual design into a physical design, into a plan for actually implementing the database in code. Large portions of the information model translate very easily into instructions for building tables. Other portions of an information model require more thought, for example, should a particular business rule be implemented as a trigger, as a stored procedure, or as code in the user interface.

The vast majority of relational database software developed since the mid 1990s uses some variant of the language SQL as the primary means for manipulating the database and the information stored within the database (the clearest introduction I have encountered to SQL is Celko, 1995b). Database server software packages (e.g. MS SQLServer, PostgreSQL, MySQL) allow direct entry of SQL statements through a command line client. However, most database software also provides for some form of graphical front end that can hide the SQL from the user (such as MS Access over the MS Jet engine or PGAccess over PostgreSQL, or OpenOffice.org, Rekall, Gnome-db, or Knoda over PostgreSQL or MySQL). Other database software, notably Filemaker, does not natively use SQL (this is no longer true in Filemaker7, which has a script step for running SQL queries). Likewise, CASE tools allow users to design, implement, modify, and reverse engineer databases through a graphical user interface, without the need to write SQL code. While SQL is the language of relational databases, it is quite possible to design, implement, and use relational databases without writing SQL code by hand.

Even if you aren't going to write SQL yourself to manipulating data, it is very helpful to think in terms of SQL. When you want to ask a question of your data, consider what query would you write to answer that question, then think about how to implement that query in your database software. This should help lead you to the desired result set. Note that phrase: result set. Set is an important word. SQL is a set based language. Tables with their rows and columns may look like a spreadsheet. SQL, however, operates not on individual rows but on sets. Set thinking is the key to working with relational databases.

Basic SQL syntax

SQL queries serve two distinctly different purposes. Data definition queries allow you to create structures for holding your data. Data definition queries define tables, fields, indices, stored procedures, and triggers. On the other hand, data manipulation queries allow you to add, edit, and view data. In particular, SELECT queries retrieve data from the database.

Data definition queries can be used to create new tables and alter existing tables. A CREATE TABLE statement simply provides the information needed to create a table, such as a table name, a list of field names, types for each field, constraints to apply to each field, and fields to index. Queries to create a very simple collection object table and to add an index to its catalog number field are shown below (in MySQL syntax, see DuBois, 2003; DuBois et al, 2004). Here I have followed a good form for readability, placing SQL commands in upper case, user supplied names for database elements in lowercase, spacing the statements out over several lines, and indenting lines to improve clarity.

```
CREATE TABLE collection_object (
    collection_object_id INT NOT NULL
       PRIMARY KEY AUTO_INCREMENT,
    acronym CHAR(4) NOT NULL
       DEFAULT “ANSP”,
     catalog_number CHAR(10) NOT NULL
);

CREATE INDEX catalog_number
  ON collection_object(catalog_number);
```

The create table query above will create a table for the collection object entity shown in Figure 14 and the create index query that follows it will index the catalog number field. SQL has a very English-like syntax. SQL uses a small set of commands such as Create, Select, Update, and Delete. These commands have a simple, easily understood syntax yet can be extremely flexible, powerful, and complex.

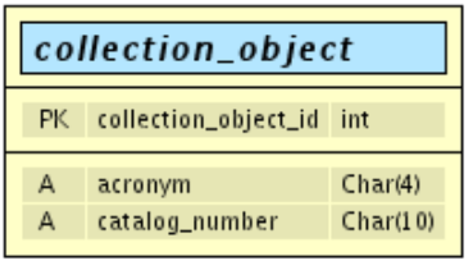


Figure 14. A collection object entity with a few attributes.

Data placed in a table based on the entity in Figure 14 might look like those in Table 13:

Table 13. Rows in a collection object table

collection_object_id	*acronym*	*catalog_ number*
300	ANSP	34000
301	ANSP	34001
302	ANSP	28342
303	ANSP	100382

SQL comes in a series of subtly different dialects. There are standards for SQL [ANSI X3.135-1986, was the first, most vendors support some subset of SQL-92 or SQL-99, while SQL:2003 is the latest standard (ISO/IEC, 2003; Eisenberg et al, 2003)], and most implementations are quite similar. However, each DBMS implements a subtly different set of features and their own extensions of the standard. A SQL statement in the PostgreSQL dialect to create a table based on the collection object entity in Figure 14 is similar, but not quite identical to the SQL in the MySQL dialect above:

```
CREATE TABLE collection_object (
  collection_object_id SERIAL NOT NULL
     UNIQUE PRIMARY KEY,
  acronym VARCHAR(4) NOT NULL
     DEFAULT 'ANSP',
  catalog_number VARCHAR(10) NOT NULL,
);
CREATE INDEX catalog_number
 ON collection_object(catalog_number);
```

Most of the time, you will not actually write data definition queries. In DBMS systems like MS Access and Filemaker there are handy graphical tools for creating and editing table structures. SQL server databases such as MySQL, Postgresql, and MS SQLServer have command line interfaces that let you issue data definition queries, but they also have graphical tools that allow creation and editing of table structures without worrying about data definition query syntax. For complex databases, it is best to create and maintain the database design in a separate CASE tool (such as xCase, or Druid, both used to produce E-R diagrams shown herein, or any of a wide range of other commercial and open source CASE tools). Database CASE tools typically have a graphical user interface for design, tools for checking the integrity of the design, and the ability to convert the design to a set of data definition queries. Using a CASE tool, one designs the database, then connects to a data source, and then has the CASE tool issue the data definition queries to build the database. Documentation of the database design can be printed from the CASE tool. Subsequent changes to the database design can be made in the CASE tool and then applied to the database itself.

The workhorse for most database applications is data retrieval. In SQL this is accomplished using the SELECT statement. Select statements can specify the desired fields and the criteria to limit the results returned by a query. MS Access has a very useful graphical query designer. The familiar queries you build with this designer by dragging fields from tables onto the query design and then adding criteria to limit the result sets are just SELECT queries (indeed it is possible to change the query designer over to SQL view and see the sql statement you have built with the designer). For those from the Filemaker world, SELECT queries are like designing a layout with the desired fields on it, then changing over to find view, adding criteria to limit the find, and then running the find to show your result set. Here is a simple select statement to list the species in the genus *Chicoreus* present in a taxonomic dictionary file:

```
SELECT generic_epithet, trivial_epithet
FROM taxon_name
WHERE generic_epithet = “Chicoreus”;
```

This SQL query will return a result set of information – all of the generic and trivial names present in the taxon_name table where the generic name is *Chicoreus.* Remember that the important word here is “set” (Figure 15). SQL is a set based language. You should think of this query returning a single set of information rather than an iterated list of rows from the source table. Set based thinking is quite different from the iterative thinking common to most programing languages . Behind the scenes, the DBMS may be walking through rows in the table, looking up values in indexes, and all sorts of interesting creative programming features that are generally of no concern to the user of the database. SQL provides a standard interface on top of the details of exactly how the DBMS is extracting data that allows you to easily think about sets of information, rather than worrying about how to get that information out of its storage structures.

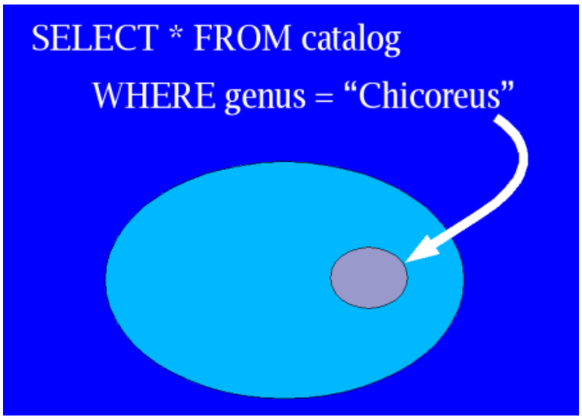


Figure 15. Selecting a set.

SELECT queries can ask sophisticated questions about aggregates of data. The simplest form of these is a query that returns all the distinct values in a field. This sort of query is extremely useful for examining messy legacy data.

The query below will return a list of the unique values for country and primary_division (state/province) from a locality table, sorted in alphabetic order.

```
SELECT DISTINCT country, primary_division
FROM locality_table;
ORDER BY country, primary_division;
```

In legacy data, a query like this will usually return an interesting list of variations on the spelling and abbreviation of both country names and states. In the MS Access query designer, a property of the query will let you convert a SELECT query into a SELECT DISTINCT query, or you can switch the query designer to SQL view and add the word DISTINCT to the sql statement. Filemaker allows you to limit options in a picklist to distinct values from a field, but doesn't (as of version 6.1) have a facility for selecting and displaying distinct values in a field other than in a picklist.

Working through an example: Extracting identifications.

SELECT queries are not limited to a single table. You can ask questions of data across multiple tables at once. The usual way of doing this is to follow a relationship joining one table to another. Thus, in our information model for an identification that has a table for taxon names, another for collection objects, and an associative entity to relate the two in identifications (Figure 11), we can create a query that starts in the collection object table and joins the identification table to it by following the primary key to foreign key based relationship. The query then follows another relationship out to the taxon name table. This join from collections objects to identifications to taxon names provides a list of the identifications for each collection object. Given a catalog number, we can obtain a list of related identifications.

```
SELECT generic_higher, trivial, author,
  year, parentheses, questionable,
  identifier, date_identified,catalog_number
FROM collections_object
  LEFT JOIN identification
    ON collection_object_id =
       c_collection_object_id
  LEFT JOIN taxon_name
    ON c_taxon_id = taxon_id
WHERE catalog_number = “34000”;
```

Because SQL is a set based language, if there is one collection object with the catalog number 34000 (Table 14) which has three identifications (Table 15,Table 16), this query will return a result set with three rows(Table 17):

Table 14. A collection_object table.

collection_object_id	catalog_number
55253325	34000

Table 15. An identification table.

c_collection_ object_id	c_taxonid	date_identified
55253325	23131	1902/--/--
55253325	13144	1986/--/--
55253325	43441	1998/05/--

Table 16. A taxon_name table

taxon_id	Generic_higher	trivial
23131	Murex	sp.
13144	Murex	ramosus
43441	Murex	bicornis

Table 17. Selected result set of joined rows from collection_object, identification, and taxon_name.

Generic_ higher	trivial	....	date_identified	catalog_ number
Murex	sp.		1902/--/--	34000
Murex	ramosus		1986/--/--	34000
Murex	bicornis		1998/05/--	34000

The collection object table contains only one row with a catalog number of 34000, but the set produced by joining identifications to collection objects contains three rows with the catalog number 34000. SQL is returning sets of information, not rows from tables in the database.

We could order this result set by the date that the collection object was identified, or by a current identification flag, or both (assuming the format of the date_identified field allows for easy sorting in chronological order):

```
SELECT generic_higher, trivial, author,
   year, parentheses,
   questionable, identifier,
   date_identified, catalog_number
FROM collections_object
   LEFT JOIN identification
      ON collection_object_id =
         c_collection_object_id
   LEFT JOIN taxon_name
      ON c_taxon_id = taxon_id
WHERE catalog_number = “34000”
ORDER BY current_identification,
         date_identified;
```

Entity-Relationship diagrams show relationships connecting entities. These relationships are implemented in a database as joins between tables. Joins can be much more fluid than implied by an E-R diagram.

```
SELECT DISTINCT
   collections_object.catalog_number
FROM taxon
  LEFT JOIN identification
    ON taxonid = c_taxon id
  LEFT JOIN collection object
    ON c_collections_objectid =
       collections_objectid
WHERE
  taxon.taxon_name = “Chicoreus ramosus”;
```

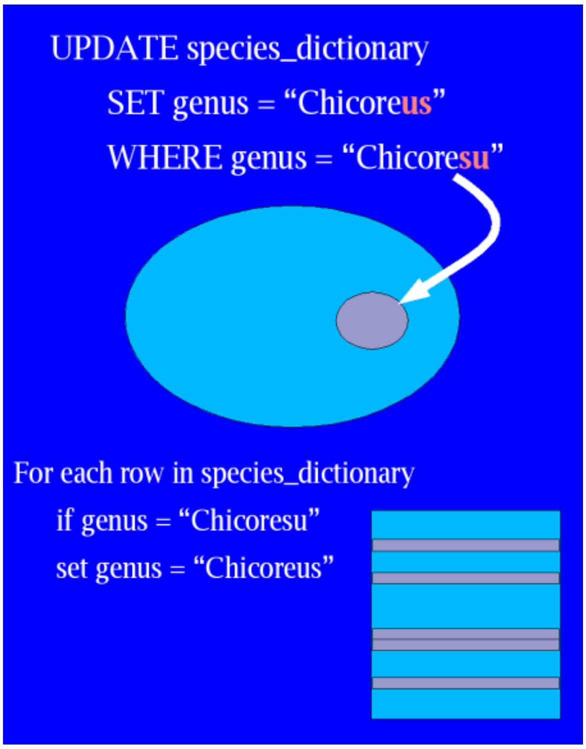
The query above is straightforward, it returns one row for each catalog number where the object has an identification of *Chicoreus ramosus*. We can also write a query to follow the same join in the opposite direction. Starting with the criterion set on the taxon table, the query below follows the joins back to the collections_object table to see a selected set of catalog numbers.

```
SELECT collections_object.catalog_number,
       taxon.taxon_name
FROM collections_object
  LEFT JOIN identification
    ON collections_objectid =
       c_collections_objectid
  LEFT JOIN taxon
    ON c_taxonid = taxon id;
```

Following a relationship like this from the many side to the one side takes a little more thinking about. The query above will return a result set with one row for each taxon name that is used in an identification, and, if a collection object has more than one identification, its catalog number will appear in more than one row. This is the normal behavior of a query across a join that represents a many to one relationship. The result set will be inflated to include one row for each selected row on the many side of the relationship, with duplicate values for the selected columns on the other side of the relationship. This also is why the previous query was a Select Distinct query. If it had simply been a select query and there were specimens with more than one identification of “*Chicoreus ramosus*”, the catalog numbers for those specimens would be duplicated in the result set. Think of queries as returning result sets rather than rows from database tables.

Thinking in sets rather than rows is evident when you perform update queries to alter data already in the database. In a programming language, you would think of iterating through each row in a table, checking to see if that row matched the criteria for an update and then applying an update to that row if it did. You can think of an SQL update query as simply selecting the set of records that match your criteria and applying the update to that set as a whole (Figure 16, top).

```
UPDATE species_dictionary
SET genus = “Chicoreus”
WHERE genus = “Chicoresu”;
```



```
UPDATE species_dictionary
    SET genus = “Chicoreus”
    WHERE genus = “Chicoresu”
```

```
For each row in species_dictionary
    if genus = “Chicoresu”
    set genus = “Chicoreus”
```

Figure 16. An SQL update statement should be thought of as acting on an entire result set at once (top), rather than walking through each row in the table, as might be implemented in an iterative programing language (bottom).

Nulls and tri-valued logic

Boolean logic with its operations on true and false is at least vaguely familiar to most of us. SQL throws in an added twist. It uses tri-valued logic. SQL expressions may be true, false, or null. A field may contain a null value. A null is different from an empty string or a zero. A character field intended to hold generic names could potentially contain “Silurus”, or “Chicoreus”, or “Palaeozygopleura”, or “” (an empty string), or NULL as valid values. An integer field could hold 1, or 5, or 1024, or -1, or 0, or NULL. Nulls make the most sense in the context of numeric fields or date fields. Suppose you want to use an real number field to hold a measurement of a specimen, say maximum shell height in a gastropod. Storing the number in a real number field will make it easy for you to calculate sums, means, and perform other mathematical operations on this field. You are left with a problem, however, when you don't know what value to put in that field. Suppose the specimen in front of you is a slug (with no shell to measure). What value do you place in the shell height field? Zero might make sense, but won't produce sensible results for some sorts of calculations. A negative number, or more broadly a number outside the range of expected valid values (such as 99 for year in a two digit date field in a database designed in the 1960s) that you could use to exclude out of range values before performing your calculation? Your perception of the scope of valid values might not match that of users of the system (as when the 1960s data survived to 1999). In our example of values for shell height, if someone decides that hyperstrophic gastropods should have negative values of shell height as they coil up the axis of coiling instead of down it like normal orthostrophic gastropods the values -1 and 0 would no longer fall outside the scope of valid shell heights. Null is the SQL solution to this problem. Nulls don't behave as numbers. Nulls allow you to flag records for which there is no sensible in range value to place in a field. Nulls make slightly less sense in character fields where you can allow explicit values such as “Not Applicable”, “Unknown”, or “Not examined” that let you explicitly record the reason that a value was not entered in the field. The difficulty in this case is in maintaining the same value for the same concept over time, preventing “Not Applicable” from being entered by some users and “N/A” by others and “n/a” and “” by others. Code to help users consistently enter “Not Applicable”, or “Unknown” can be embedded in the user interface, but fundamentally, ensuring consistent data entry in this form is a matter of careful user training, quality control procedures, and detailed documentation.

Nulls make for interesting complications when it comes time to query the database. We normally think of expressions in programs as following some set of rules to evaluate as either true or false. Most programing languages have some construct that lets us take an action if some condition is met; **`IF some expression is true THEN do something.`** The expression **`(left(genus,4) <> “Silu”)`** would sensibly seem to evaluate to true for all cases where the first four characters of the genus field are not “Silu”. Not so in an SQL database. Nulls propagate. If an expression contains a null, the null will propagate to make result of the whole expression null. If the value of genus in some row is null, the expression left(NULL,4) <> “Silu” will evaluate to null, not to true or false. Thus the statement **`select generic,  trivial from taxon_name where (left(generic,4) <> “silu”)`** will not return the expected result set (it will not include rows where generic=NULL. Nulls are handled with a function, such as IsNull(), which can take a null and return a true or false result. Our query needs to add a term: **`select generic,  trivial from taxon_name where (left((generic,4) <> “silu”) or IsNull(generic))`**.

Maintaining integrity

In a spreadsheet or a flat file database, deleting a record is a simple matter of removing a single row. In a relational database, removing records and changing the links between records in related tables becomes much more complex. A relational database needs to maintain database integrity. An important part of maintaining integrity is knowing what do you do with related records when you delete a record on one side of a join. Consider a scenario: You are cataloging a collection object and you enter data about it into a database (identification, locality, catalog number, kind of object, etc...). You then realize that you entered the data for this object yesterday, and you are creating a duplicate record that you want to delete. How far does the delete go? You no doubt want to get rid of the duplicate record in the collection object table and the identifications attached to this record, but you don't want to keep following the links out to the authority file for taxon names and delete the names of any taxa used in identifications. If you delete a collections object you do not want to leave orphan identifications floating around in the database unlinked to any collections object. These identifications (carrying a foreign key for a collections object that doesn't exist) can show up in subsequent queries and have the potential to become linked to new collections objects (silently adding incorrect identifications to them as they are created). Such orphan records, which retain links to no longer existent records in other tables, violate the relational integrity of the database.

When you delete a record, you may or may not want to follow joins (relationships) out to related tables to delete related records in those tables. Descriptions of relationships themselves do not provide clear guidance on how far deletions should propagate through the database and how they should be handled to maintain relational integrity. If a collection object is deleted, it makes sense to delete the identifications attached to that object, but not the taxon names used in those identifications as they are probably used to identify other collection objects. If, in the other direction, a taxon name is deleted the existence of any identifications that use that taxon name almost certainly mean that the delete should fail and the name should be retained. An operation such as merging a record containing a correctly spelled taxon name with a record containing an incorrectly spelled copy of the same name should correct any links to the incorrect spelling prior to deleting that record.

Relational integrity is best enforced with constraints, triggers, and code enforcing rules at the database level (supported by error handling in the user interface). Some database languages support foreign key constraints. It is possible to join two tables by including a column in one table that contains values that match the values in the primary key of another table. It is also possible to explicitly enforce foreign key constraints on this column. Including a foreign key constraint in a table definition will require that values entered in the foreign key column match values found in the related primary key. Foreign key constraints can also include cascading deletes. Deleting a row in one table can cascade out to related tables with foreign key constraints linked to the first table. A foreign key constraint on the c_collections_object_id field of an identification table could cause deletes from the related collections object table to cascade out and remove related rows from the identification table. Support for such deletion of related rows varies between database systems.

Triggers are blocks of code in a database that are executed when particular actions are performed. An on delete trigger is a block of code tied to a table in a database that can fire when a record is deleted from that table. An on delete trigger for a collections object could, like a foreign key constraint, delete related records in an identification table. Triggers, unlike constraints, can contain complex logic and can do more than simply affect related rows. An on delete trigger for a taxon name table could check for related records in an identification table and cause the delete operation to fail if any related records exist. An on insert or on update trigger can include complex format checking and business rule checking code, and we will see later, triggers can be very helpful in maintaining the integrity of hierarchical information (trees) stored in a database.

Triggers, foreign keys, and other operations executed on the database server do have a downside: they involve the processing of code, and thus reduce the speed of database operations. In many cases (where you are concerned about the integrity of the data), you will want to support these operations somewhere – either in user interface code, in a middle layer of business logic code, or as code embedded in the database. Embedding rules to support the integrity of the data in the database (through triggers and constraints) can be an effective way of ensuring that all users and clients that attach to the database have to follow the same business rules. Triggers can also simplify client development by reducing the number of operations the client must perform to maintain integrity of the data.

User rights & Security

Another important element to maintaining data quality is control over who has access to a database. Limits on who is able to add data and who is able to alter data are essential. Unrestricted database access to all and sundry is an invitation to unusable data. At a minimum, guests should have select only access to public parts of the database, data entry personnel should have limited select and update (and perhaps delete) rights to parts of the database, a limited set of skilled users may be granted update access to tables housing controlled vocabularies, and only system administrators should have rights to add users or alter user privileges. Particular business functions (such as collection managers filling loans, curators approving loans, or a registrar approving accessions) may also require restrictions to limit these operations on the database to only the correct users. User rights are best implemented at the database level. Database systems include native methods for restricting user rights. You should implement rights at this level, rather than trying to implement a separate privilege system in the user interface. You will probably want to mirror the database privileges in the front end (for example, hiding administrative menu options from lower level users), but you should not rely on code in the front end of a database to restrict the ability of users to carry out particular operations. If a database front end can connect a user to a database backend with a high privilege level, the potential exists for users to skip the front end and connect directly to the database with a high privilege level (see Morris 2001 for an example of a server wide security risk introduced by a design that implemented user access control in the client).

Implementing as joins & Implementing as views

In many database systems, a set of joins can be stored as a view of a database. A view can be treated much like a table. Users can query a view and get a result set back. Views can have access rights granted by the access privilege system. Some views will accept update queries and alter the data in the tables that lie behind them. Views are particularly valuable as a tool for restricting a class of users to a subset of possible actions on a subset of the database and enforcing these restrictions at the database level. A user can be granted no rights at all to a set of tables, but given select access to a view that shows a subset of information from those tables. An account that updates a web database by querying a master database might be granted select only access to a view that limits it to just the information needed to update the web dataset (such as a flat view of Darwin Core [Schwartz, 2003; Blum and Wieczorek, 2004] information). Given the complex joins and very complex structure of biodiversity information, views are probably not practical ways to restrict data entry privileges for most biodiversity databases. Views may, however, be an appropriate means of limiting guest access to a read only view of the data.

Interface design

Simultaneously with the conceptual and physical design of the back end of a database, you should be creating a design for the user interface to access the data. Existing user interface screens for a legacy database, paper and pencil designs of new screens, and mockups in database systems with easy form design tools such as Filemaker and MS Access are of use in interface design. I feel that the most important aspect of interface design for databases is to fit the interface to the workflow, abstracting the user interface away from the underlying complex data structures and fitting it to the tasks that users perform with the data. A typical user interface problem is to place the user interface too close to the data by creating one data entry screen for each table in the database. In anything other than a very simple database, having the interface too close to the data ends up in a bewildering profusion of pop up windows that leave users entirely confused about where they are in data entry and how the current open window relates to the task at hand.

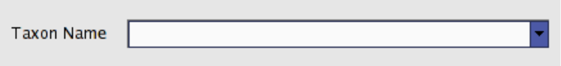
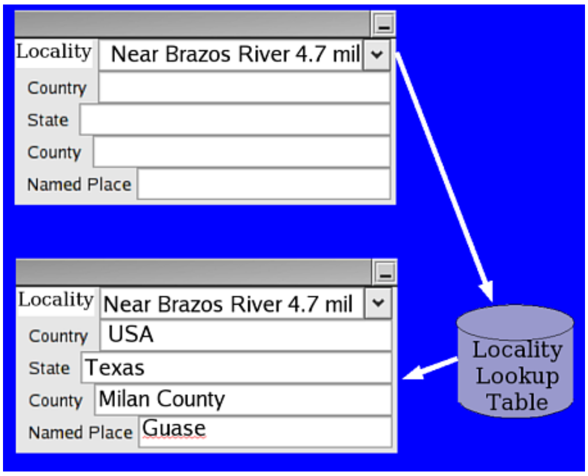


Figure 17. A picklist control for entering taxon names.

Consider the control in Figure 17. It allows a user to select a taxon name (say to provide an identification of a collection object) off of a picklist. This control would probably allow the user to start typing the taxon name in the control to jump to the relevant part of a very long picklist. A picklist like this is a very seductive form element in many situations. It can allow a data entry person to make fewer keystrokes and mouse gestures to enter a particular item of information than by filling in a set of fields. It can mask substantial complexity in the underlying database (the taxon name might be built from 12 fields or so and the control might be bound to a field holding a surrogate numeric key representing a particular combination). By having users pick values off of a list you can enforce a controlled vocabulary and can avoid the entry of misspelled taxon names and other complex vocabulary. Picklists, however have a serious danger. If a data entry person selects the wrong taxon name when entering an identification from the picklist above there is no way for anyone to find that a mistake has been made without having someone return to the original source for the information and compare the record against that source (Figure 18). In contrast, a misspelled taxon name is usually easy to locate (by comparison with a controlled list of taxon names). If data is entered as text, simple misspellings can be found, identified, and fixed. Avoid picklists as sole sources of information.



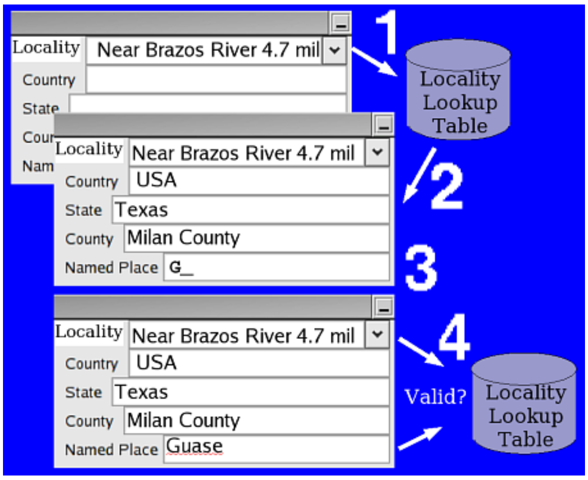
Locality Near Brazos River 4.7 mil
Country
State
County
Named Place

Locality Near Brazos River 4.7 mil
Country USA
State Texas
County Milan County
Named Place Guase

Figure 18. A picklist being used as the sole source of locality information.

One option to avoid the risk of unfindable errors is to entirely avoid the use of picklists in data entry. Simply exchanging picklists for text entry controls on forms will result in the loss of the advantages of picklist controls; more rapid data entry and, more importantly, a controlled vocabulary. It is possible to maintain authority control and use text controls by writing code behind a text entry control that will enforce a controlled vocabulary by querying an authority file using the information entered in the text control and throwing an error (and presenting the user with an error message) if no match is found in the controlled vocabulary in the authority file. This alternative can work well for single word entries such as generic names, where it is faster to simply type a name than it is to open a long picklist, move to the correct location on the list, and select a value. Replacing a picklist with a controlled text box, however, is not a good choice for complex formated information such as locality descriptions.

Another option to avoid the risk of unfindable errors is to couple a picklist with a text control (Figure 19). A collecting event could be linked to a locality through a picklist of localities, coupled with a redundant text field to enter a named place. The data entry person needs to make more than one mistake to create an unfindable error. To make an unfindable error, the data entry person needs to select the wrong value from the picklist, enter the wrong value in the text box, and have these incorrect text box value match the incorrect choice from the picklist (an error that is still quite conceivable, for example if the data entry person looks at the label for one specimen when they are typing in information about another specimen). The text box can hold a terminal piece of information that can be correlated with the information in the picklist, or a redundant piece of information that must match a value on the pick list. A picklist of species names and a text box for the trivial epithet allow an error to be raised whenever the trivial epithet in the text box does not match the species name selected on the picklist. Note that the value in the text box need not be stored as a field in the database if the



Locality Near Brazos River 4.7 mil
Country
State

Locality Near Brazos River 4.7 mil
Country USA
State Texas
County Milan County
Named Place G_

Locality Near Brazos River 4.7 mil
Country USA
State Texas
County Milan County
Named Place Guase

Figure 19. A picklist and a text box used in combination to capture and check locality information. Step 1, the user selects a locality from the picklist. Step 2, the database looks up higher level geographic information. Step 3, the user enters the place name associated with the locality. Step 4, the database checks that the named place entered by the user is the correct named place for the locality they selected off the picklist.

quality control rules embedded in the database require it to match the picklist. Alternately the values can be stored and used to flag records for later review in the quality control process.

Design your forms to function without the need for lifting hands off the keyboard. Data entry should not require the user to touch the mouse. Moving to the next control, pressing a button, moving to the next record, opening a picklist, and duplicating information from the previous record, are all operations that can be done from the keyboard. Human interface design is a discipline in its own right, and I won't say more about it here.

Practical Implementation

Be Pragmatic

Most natural history collections operate in an environment of highly limited resources. Properly planning, designing, and implementing a database system following all of the details of some of the information models that have been produced for the community (e.g. Morris 2000) is a task beyond the resources of most collections. A reasonable estimate for a 50 to 100 table database system includes about 500-1000 stored procedures, more than 100,000 lines of user interface code, one year of design, two or more years of programming, a development team including a database programmer, database administrator, user interface programmer, user interface designer, quality control specialist, and a technical writer, all running to some $1,000,000 in costs. Clearly natural history collections that are developing their own database systems (rather than using external vendors or adopting community based tools such as BioLink [CSIRO, 2001] or Specify) must make compromises. These compromises should involve selecting the most important elements of their collections information, spending the most design, data cleanup, and programing effort on those pieces of information, and then omitting the least critical information or storing it in less than third normal form data structures.

A possible candidate for storage in less than ideal form is the generalized Agent concept that can hold persons and institutions that can be linked to publications as authors, linked to collection objects as preparators, collectors, identifiers, and annotators, and linked to transactions as donors, recipients, packers, authorizers, shippers, and so forth. For example, given the focus on collection objects, using Agents as authors of publications (through an authorship list associative entity) may introduce substantial complications in user interface design, code to maintain data integrity, and the handling of existing legacy data that produce costs far in excess of the utility gained from proper third normal form handling of the concept of authors. Conversely, a database system designed specifically to handle bibliographic information requires very clean handling of the concept of Authors in order to be able to produce bibliographic citations in multiple different formats (at a minimum, the author last name and initials need to be atomized in an author table and they need to be related to publications through an authorship list associative entity). Abandoning third normal form (or higher) in parts of the database is not a bad thing for natural history collections, so long as the decisions to use lower normal forms are clearly thought out and restricted to the least important parts of the data.

I chose the example of Agents as a possible target for reduced database complexity deliberately. Some institutions and users will immediately object that a generalized Agent related to transactions and collection objects is of critical importance to their data. Perfect. This is precisely the approach I am advocating. Identify the most important parts of your data, and put your time, effort, design, programing, and data manipulation into making sure that your database system is capable of cleanly handling those most critical areas. Identify the concepts that are not of critical importance and minimize the design complexity you allow them to introduce into your database (recognizing that problems will accumulate in the quality of these data). In a setting of limited resources, we are seldom in a situation where we can build systems to store all of the highly complex information associated with collections in optimum form. This fact does not, however, excuse us from identifying the most important information and applying the best solutions we can to the stewardship of that information.

Approaches to management of date information

Dates in collection data are generally problematic as they are often known only to a level of precision less than a single day. Dates may be known to the day, or in some cases to the time of day, but often they are known only to the month, or to the year, or to the decade. In some cases, dates are known to be prior to a particular date (e.g. the date donated may be known but the date collected may not other than that it is sometime prior to the date donated). In other cases dates are known to be within a range (e.g. between 1932-June-12 and 1932-July-15[4]); in yet others they are known to be within a set of ranges (e.g. collected in the summers of 1852 to 1855). Designing database structures to be able to effectively store, retrieve, sort, and validate this range of possible forms for dates is not simple (Table 18).

Using a single field with a native date data type to hold collections date information is generally a poor idea as date data types require each date to be specified to the precision of one day (or finer). Simply storing dates as arbitrary text strings is flexible enough to encompass the wide variety of formats that may be encountered, but storing dates as arbitrary strings does not ensure that the values added are valid dates, in a consistent format, are sortable, or even searchable.

Storage of dates effectively requires the implementation of an indeterminate or arbitrary precision date range data type supported by code. An arbitrary precision date data type can be implemented most simply by using a text field and enforcing a format on the data allowed into that field (by binding a picture statement or format expression to the control used for data entry into that field or to the validation rules for the field). A format like "9999-Aaa-99 TO 9999-Aaa-99" can force data to be entered in a fixed standard order and form. Similar format checks can be imposed with regular expressions. Regular expressions are an extremely powerful tool for recognizing patterns found in an expanding number of languages (perl, PHP, and MySQL all include support for regular expressions). A regular expression for the date format above looks like this: `/^[0-9]{4}-[A-Z]{1}[a-z]{2}-[0-9]{2}( TO [0-9]{4}-[A-Z]{1}[a-z]{2}-[0-9]{2})+$/.` A regular expression for an ISO date looks like this: `/^[0-9]{2,4}(-[0-9]{2}(-[0-9]{2})+)+(-[0-9]{4}(-[0-9]{2}(-[0-9]{2})+)+)+$/.` Note that simple patterns still do not test to see if the dates entered are valid.

Another date storage possibility is to use a set of fields to hold start year, end year, start month, end month, start day, and end day. A set of such numeric fields can be sorted and searched more easily than a text date range field but needs careful planning of what values are placed in the day fields for dates for which only the month is known and other handling of indeterminate precision.

From a purely theoretical standpoint, using a pair of native date data type fields to hold start day and end day is the best way to hold indeterminate date and date range information (as 1860 translates to the range 1860-01-01 to 1860-12-31). Native date data types have native recognition of valid and invalid values, sorting functions, and search functions. Implementing dates with a native date data type avoids the need to write code to support validation, sorting, and other things that are needed for dates to work. Practical implementation of dates using a native date data type, however, would not work well as just a pair of date fields exposed for text entry on the user interface. Rather than simply typing "1860" the data entry person would need to stop, think, type 1860-01-01, move to the end date field, then hopefully remember the last day of the year correctly and enter it. Efficient and accurate data entry would require a user interface with code capable of accepting "1860" as a valid entry and storing it as an appropriate date range. Printing is also an issue – the output we would expect

[4] There is an international standard date and time format, ISO 8601, which specifies standard numeric representations for dates, date ranges, repeating intervals and durations. ISO 8601 dates include notations like 19 for an indeterminate date within a century, 1925-03 for a month, 1860-11-5 for a day, and 1932-06-12/1932-07-15 for a range of dates.

Table 18. Comparison of some ways to store date information

Fields	data type	Issues
Single date field	date	Native sorting, searching, and validation. Unable to store date ranges, will introduce false precision into data.
Single date field	character	Can sort on start date, can handle single dates and date ranges easily. Needs minimum of pattern or format applied to entry data, requires code to test date validity.
Start date and end date fields	two character fields, 6 character fields, or 6 integer fields.	Able to handle date ranges and arbitrary precision dates. Straightforward to search and sort. Requires some code for validation.
Start date and end date fields	two date fields	Native sorting and validation. Straightforward to search. Able to handle date ranges and arbitrary precision. Requires carefully designed user interface with supporting code for efficient data entry.
Date table containing date field and start/end field.	date field, numeric fields, character field, or character fields.	Handles single dates, multiple non-continuous dates, and date ranges. Needs complex user interface and supporting code.

on a label would be “1860” not “1860-01-01 to 1860-12-31” for cases where the date was only known to a year, with a range only printing when the range was the originally known value. An option to handle this problem is to use a pair of date fields for searching and a text field for verbatim data and printing, though this solution introduces redundancy and possible accumulation of errors in the data.

Multiple start and end points (such as summers of several years) are probably rare enough values to hold in a separate text date qualifier field. A free text date qualifier field, separate from a means of storing a date range, as a means for handling these exceptional records would preserve the data, but introduces a reduction of precision in searches (as effective searches could only operate on the range end points, not the free text qualifier). Properly handing events that occur within multiple date ranges requires a separate entity to hold date information. The added code and interface complexity needed to support such an entity is probably an unnecessary burden to add for most collections data.

Handling hierarchical information

Hierarchies are pervasive in biodiversity informatics. Taxon names, geopolitical entities, chronostratigraphic units and collection objects are all inherently hierarchical concepts – they can all be represented as trees of information. The taxonomic hierarchy is very familiar (e.g. a Family contains Subfamilies which contain Genera). Collection objects are intuitively hierarchical in some disciplines. For example, in invertebrate paleontology a bulk sample may be cataloged and later split, with part of the split being sorted into lots. Individual specimens may be split from these lots. These specimens may be composed of parts (paired valves of bivalves), which can have thin sections made from them. In addition, derived objects (such as casts) can be made from specimens or their parts. All of these different collection objects may be assigned their own catalog numbers but are still related by a series of lot splits and preparation steps to the original bulk sample. A bird skin is not so obviously hierarchical, but skin, skeleton, and frozen tissue samples from the same bird may be stored separately in a collection.

Some types of database systems are better at handling hierarchical information than others. Relational databases do not have easy and intuitive ways of storing hierarchies in a form that is both easy to access and maintains the integrity of the data. Older hierarchical database systems were designed around business hierarchies and natively understood hierarchies. Object oriented databases can apply some of the basic properties of extensible objects to easily handle hierarchical information. Object extensions are available for some relational database management systems

Table 19. Higher taxa in a denormalized flat file table

Class	T_Order	Family	Sub Family	Genus
Gastropoda	Caenogastropoda	Muricidae	Muricinae	Murex
Gastropoda	Caenogastropoda	Muricidae	Muricinae	Chicoreus
Gastropoda	Caenogastropoda	Muricidae	Muricinae	Hexaplex

Table 20. Examples of problems with a hierarchy placed in a single flat file.

Class	T_Order	Family	Sub Family	Genus
Gastropoda	Caenogastropoda	Muricidae	Muricinae	Murex
Gastropod	Caenogastropoda	Muricidae	Muricinae	Chicoreus
Gastropoda	Neogastropoda	Muricinae	Muricidae	Hexaplex

and can be used to store hierarchical information more readily than in relational systems with only a standard set of SQL data types.

There are several different ways to store hierarchical information in a relational database. None of these are ideal for all situations. I will discuss the costs and benefits of three different structures for holding hierarchical information in a relational database: flattening a tree into a denormalized table, edge representation of trees, and a tree visitation algorithm.

Denormalized table

A typical legacy structure for the storage of higher taxonomy is a single table containing separate fields for Genus, Family, Order and other higher ranks (Table 19). (Note that order is a reserved word in SQL [as in the ORDER BY clause of a SELECT statement] and most SQL dialects will not allow reserved words to be used as field or table names. We thus need to use some variant such as T_Order as the actual field name). Placing a hierarchy in a single flat table suffers from all the risks associated with non normal structures (Table 20).

Placing higher taxa in a flat table like Table 19 does allow for very easy extraction of the higher classification of a taxon in a single query as in the following examples. A flat file is often the best structure to use for a read only copy of a database used to power a searchable website. Asking for the family to which a particular genus belongs is very simple:

```
SELECT family FROM higher_taxonomy
  FROM higher_taxonomy
  WHERE genus = “Chicoreus”;
```

Likewise, asking for the higher classification of a particular species is very straightforward:

```
SELECT class, t_order, family
  FROM higher_taxonomy
    LEFT JOIN taxon_name
    ON higher_taxonomy.genus =
       taxon_name.genus
  WHERE taxon_name_id = 352;
```

Edge Representation

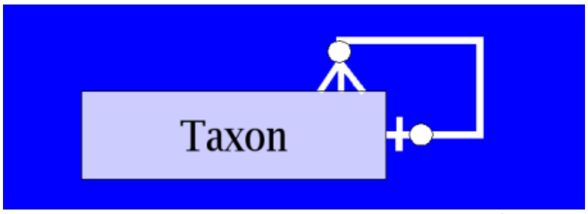


Figure 20. A taxon entity with a join to itself. Each taxon has zero or one higher taxon. Each taxon has zero to one lower taxa.

Heirarchical information is typically described in an information model using an entity with a one to many link to itself (Figure 20). For example, a taxon entity with a relationship where a taxon can be the child of zero or one higher taxon and a parent taxon can have zero to many child taxa. (Parent and child are used here in the sense of computer science description of trees, where a parent node in the tree can be linked to several child nodes, rather than in any genealogical or evolutionary sense).

Taxonomic hierarchies are nested sets and can readily be stored in tree data structures. Thinking of the classification of animals as a tree, Kingdom Animalia is the root node of the tree. The immediate child nodes under the root might be the thirty some phyla of animals, each with their own subphylum, superclass, or class children. Animalia could thus be the parent node of the phylum Mollusca. Following the branches of the tree down to lower taxa, the terminal nodes or leaves of the tree would be species and subspecies. Taxonomic hierarchies readily translate into trees and trees are very familiar data structures in computer science.

Storage of a higher classification as a tree is typically implemented using a table structure that holds an edge representation of the tree hierarchy. In an edge representation, a row in the table has a field for the current node in the tree and another field that contains a pointer to the current node's parent. The simplest case is a table with two fields, say a higher taxon table containing a field for taxon name and another field for parent taxon name.

```
CREATE TABLE higher_taxon (
   taxon_name char(40) not null
primary_key,
   parent_taxon char(40) not null);
```

Table 21. An edge representation of a tree

Taxon_name (PK)	*Higher_taxon*
Gastropoda	[root]
Caenogastropoda	Gastropoda
Muricidae	Caenogastropoda
Chicoreus	Muricidae
Murex	Muricidae

In this implementation (Table 21) you can follow the parent – child links recursively to find the higher classification for a genus, or to find the genera placed within an order. However, this implementation requires recursive queries to move beyond the immediate parent or child of a node. Given a genus, you can easily find its immediate parent and its immediate parent's parent.

```
SELECT t1.taxon_name, t2.taxon_name,
       t2.higher_taxon
FROM higher_taxon as t1
  LEFT JOIN higher_taxon as t2
    ON t1.higher_taxon = t2.taxon_name
WHERE t1.taxon_name = “Chicoreus”;
```

The query above will return a result set “Chicoreus”, ”Muricidae”, ”Caenogastropoda” from the data in Table 21. Unfortunately, unless the tree is constrained to always have the a fixed number of ranks between every genus and the root of the tree (and the entry point for a query is always a generic name), it is not possible to set up a single query that will always return the higher classification for a given generic name. The only way to effectively query this table is with program code that recursively issues a series of sql statements to walk up (or down) the tree by following the higher_taxon to taxon_name links, that is, by recursively traversing the edge representation of the tree. Such code could be either implemented as a stored procedure in the database or higher up within the user interface.

By using the taxon_name as the primary key, we impose a constraint that will help maintain the integrity of the tree, that is, each taxon name is allowed to occur at only one place in the tree. We can't place the genus *Chicoreus* into the family Muricidae and also place it in the family Turridae. Forcing each taxon name to be a unique entry prevents the placement of anastomoses in the tree. More than just this constraint is needed, however, to maintain a clean representation of a taxonomic hierarchy in this edge representation. It is possible to store infinite loops by linking the higher_taxon of one row to a taxon name that links back to it. For example (Table 22), if the genus *Murex* is in the Muricidae, and the higher taxon for Muricidae is set to *Murex*, an infinite loop is established where *Murex* becomes a higher taxon of itself, and *Murex* is not linked to the root of the tree.

Table 22. An error in the hierarchy.

Taxon_name (PK)	*Higher_taxon*
Gastropoda	[root]
Caenogastropoda	Gastropoda
Muricidae	Murex
Chicoreus	Muricidae
Murex	Muricidae

Maintenance of a table containing an edge representation of a large tree is difficult. It is easy for program freezing errors such as the infinite loop in Table 22 to be inadvertently created unless there is code testing each change to the table for validity.

The simple taxon_name, higher_taxon structure has another problem: How do you print the family to which a specimen belongs on its label? A solution is to add a rank column to the table (Table 23).

Selecting the family for a genus then becomes a case of recursively following the taxon_name – higher_taxon links back to a taxon name that has a rank of Family. The pseudocode below (Code Example 1) illustrates (in an overly simplistic way) how ranks could work by embedding the logic in a code layer sitting above the database. It is also possible to implement rank handling at the database level in a database system that supports stored procedures.

Table 23. Edge representation with ranks.

Taxon_name (PK)	*Higher_taxon*	*Rank*
Gastropoda	[root]	Class
Caenogastropoda	Gastropoda	Order
Muricidae	Caenogastropoda	Family
Chicoreus	Muricidae	Genus
Murex	Muricidae	Genus

An edge representation of a tree can be stored as in the examples above using a taxon name as a primary key. It is also possible to use a numeric surrogate key and to store the recursive foreign key for the parent as a number (e.g. c_taxon_id_parent). If you are updating a hierarchical taxonomic dictionary through a user interface with a code layer between the user and the table structures, using the surrogate key values in the recursive foreign key to identify the parent taxon of the current taxon is probably a good choice. If the hierarchical taxonomic dictionary is being maintained directly by someone who is editing the file directly, then using the taxon name as the foreign key value is probably a better choice. It is much easier for a knowledgeable person to check a list of names for consistency than a list of numeric links. In either case, a table containing an edge representation of a tree will require supporting code, either in the form of a consistency check that can be run after direct editing of the table or in the form of a user interface that allows only legal changes to the table. Without supporting code, the database itself is not able to ensure that all rows are placed somewhere in the tree (rather than being unlinked nodes or members of infinite loops), and that the tree is a consistent branching tree without anastomoses. If a field for rank is included, code can also check for rank placement errors such as the inclusion of a class within a genus.

Code Example 1.

```
// Pseudocode to illustrate repeated queries on edge
// representation table to walk up tree from current
// node to a higher taxon node at the rank of family.
// Note: handles some but not all error conditions.
$root_marker = “[Root]”;            // value of higher taxon of root node
$max_traverse = 1000;               // larger than any leaf to root path
$rank = “”;                         // holds rank of parent nodes
$targetName = $initialTarget;       // start with some taxon
$counter = 0;                       // counter to trap for infinite loops
while ($rank <> “Family”)           // desired result
  and ($counter < $max_traverse)    // errors: infinite loop/no parent
  and ($targetname <> $root_marker) // error: reached root
{
   $counter++;                      // increment counter
   $sql = “SELECT taxon_name, rank, higher_taxon
           FROM higher_taxon
           WHERE t1.taxon_name = '$targetName'”;
   $results = getresults($connection,$sql);
   if (numberofrows($results)==0)
   {
      // error condition: no parent found for current node
      $counter = $max_traverse;     // exploit infinite loop trap
   } else {
      // a parent exists for the current node
      $row = getrow($results);
      $currentname = getcolumn($row, ”taxon_name”);
      $rank = getcolumn($row,”rank”);
      $targetName = getcolumn($row”higher_taxon”);
   } // endif
} // wend
```

Table 24. Storing the path from the current node to the root of the tree redundantly in a parentage field, example shows a subset of fields from BioLink's tblBiota (CSIRO, 2001).

intBiotaID (PK)	vchrFullName	intParentID	vchrParentage
1	Gastropoda		/1
2	Caenogastropoda	1	/1/2
3	Muricidae	2	/1/2/3
4	Chicoreus	3	/1/2/3/4
5	Murex	3	/1/2/3/5

If you wish to know the family placement for a species, code in this form will be able to look it up but will require multiple queries to do so. If you wish to look up the entire higher classification for a taxon, code in this form will require as many queries to find the classification as there are steps in that classification from the current taxon to the root of the tree.

To solve the unknown number of multiple queries problem in an edge representation of a tree it is possible to store the path from the current node to the root of the tree redundantly in a parentage field (Table 24). Adding this redundancy requires supporting code, otherwise the data are certain to become inconsistent. A typical example of a redundant parentage string can be found in BioLink's taxonomic hierarchy(CSIRO 2001). BioLink stores core taxon name information in a table called tblBiota. This table has a surrogate primary key intBiotaID, a field to hold the full taxon name vchrFullName, a recursive foreign key field containing a pointer to another row in tblBiota intParentID, and a field that contains a backslash delimited list of all the intParentID values from the current node up to the root node for the tree vchrParentage. BioLink also contains other fields and sets of related tables to store additional information about the current taxon name.

Storing the path from the current node to the root of the tree (top of the hierarchy) in a parentage field allows straightforward selection of the taxonomic hierarchy of any particular taxon (Code Example 2, below). One query will extract the taxon name and its parentage string, the parentage string can then be split on its delimiter, and this list of primary key values can be assembled in a query (e.g. where intBiotaID = 1 or intBiotaID = 2 or ... order by vchrParentage) that will return the higher classification of the taxon. Note that if the field used to store the parentage is defined as a string field (with the ability to add an index and rapidly sort values by parentage), it will have a length limit in most database systems (usually around 255 characters) and in some systems may only sort on the first fifty or so characters in the string. Thus some database systems will impose a limit on how deep a tree (how many nodes in a path to root) can be stored in a rapidly sortable parentage string.

Tree Visitation

A different method for storing hierarchical structures in a database is the tree visitation algorithm (Celko, 1995a). This method works by traversing the tree and storing the step at which each node is entered in one field and in another field, the step at which each node is last exited. A table that holds a tree structure includes two fields (t_left and t_right [note that “left” and “right” are usually reserved words in SQL and can't be used as field names in many database systems]). To store a tree using these two fields, set a counter to zero, then traverse the tree, starting from the root. Each time you enter a node, increment the counter by one and store the counter value in the t_left field. Each time you leave a node for the last time, increment the counter by one and store the counter value in the t_right field. You traverse the tree in inorder[5], visiting a node, then the first of its children, then the first of its children, passing down until you reach a leaf node, then go back to the leaf's parent and visit its next child.

Table 25 holds the classification for 6 taxa. The tree traversal used to set the left and right values is shown below in Figure 21. The value of the counter on entry into a node is shown in yellow; this is the value stored in the left field for that node. The value of the counter on the last exit from a node is shown in red, this is the value stored

[5] As opposed to a preorder or postorder tree traversal.

Code Example 2.

```
// Pseudocode to illustrate query on edge representation table containing parentage
// string.  Uses field names found in BioLink table tblBiota.
// Note: does not handle any error conditions.
// Note: does not include code to place higher taxa in correct order.
$rank = “”;
$targetName = $initialTarget;  // Name of the taxon who's parentage you want to find
$sql = “SELECT vchrParentage
        FROM tblBiota
        WHERE vchrFullName = '$targetName'”;  // get parentage string
$results = getresults($connection,$sql);      // run query on database connection
$row = getrow($results);                      // get the result set from the query
$parentage = getcolumn($row,”rank”);          // get the parentage string
// the parentage string is a list of ids separated by backslashes
@array_parentage = split($parentage,”\”);     // split the parentage string into ids
$sql = “SELECT vchrFullName FROM tblBiota WHERE “;
$first=TRUE;
for ($x=1;$x<rowsin(@array_parentage);$x++) {
    // for each id in parentage, build query get the name of the taxon
    if ($first==FALSE) { $sql = $sql + “ and “; }
    // Note: enclosing integer values in quotes in sql queries is usually not
    // required, but is a good programing practice to help prevent sql injection
    // attacks
    $sql = $sql
           + “ intBiotaID = '“
           + @array_parentage[$x] + “'”;
    $first=FALSE;
}
$results = getresults($connection,$sql); // now run assembled query to get names
for ($x=1;$x<rowsin($results);$x++) {
   $row=getrows(results)
   @array_taxa[$x]=getcolumn($row,”vchrFullName”);
}
```

in the right field for that node.

We start at the root of the tree, Gastropoda. The counter is incremented to one, and the left value for Gastropoda is set to 1. Gastropoda has one child, Caenogastropoda, we increment the counter, enter that node and set its left value to two. We proceed down the tree in this manner until we reach a leaf, Murex. On entering Murex we increment the counter, set Murex's left to 5. Since Murex has no children, we leave it, incrementing the counter and setting its right value to 6. We move back to the parent node for Murex, Muricidae. Muricidae has a child we haven't visited yet, so we move down into it, Chicoreus. Entering Chicoreus we increment the counter and set Chicoreus' left to 7. Since Chicoreus has no more children, we leave it, incrementing its counter and setting its right value to 8. We step back to Chicoreus' parent, Muricnae. Since Muricinae has no more children, we leave it for the last time, incrementing the

Table 25. A tree stored in a table using a left right tree visitation algorithm.

TaxonID (PK)	*Taxon Name*	*Left*	*Right*
1	Gastropoda	1	12
2	Caenogastropoda	2	11
3	Muricidae	3	10
4	Muricinae	4	9
5	Murex	5	6
6	Chicoreus	7	8

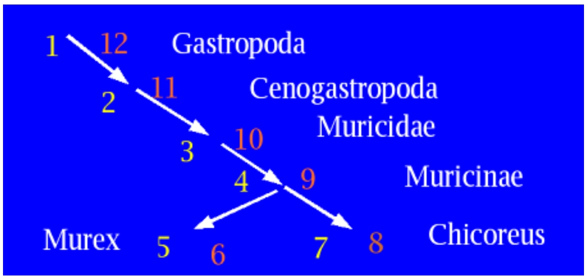


Figure 21. Representing the classification of Murex and Chicoreus as a tree and using a tree visitation algorithm to map the nodes in the tree. Values placed in the left field in yellow, values placed in the right field in red. Data as in Table 25.

counter and storing its value to Muricinae's right. We keep walking back up the tree, finally getting back to the root after having visited each node in the tree.

In comparison to an edge representation, the tree visitation algorithm has some very useful properties for selecting data from the database. The query Select taxon_name where t_left = 1 will return the root of the tree. The query Select t_right/2 where t_left = 1 will return the number of nodes in the tree. Selecting the higher classification for a particular node is also fast and easy:

```
SELECT taxon_name FROM treetable
WHERE t_left < 7 and t_right > 8
ORDER by t_left;
```

The query above will return the higher classification for *Chicoreus* in the example above. We don't need to know the number of steps to the root, make recursive queries, store redundant information or have to worry about any of the other problematic features of denormalized or edge visitation implementations of hierarchies. Selecting all of the children of a particular node, or all of the leaf nodes below a particular node are also very easy queries.

```
SELECT taxon_name FROM treetable
WHERE t_left > 3 and t_right < 10
  and t_right – t_left = 1
ORDER by t_left;
```

As grand as the tree visitation algorithm is for extracting data from a database, it has its own problems. Except for a very small tree, it will be extremely hard to create and maintain the left/right values by hand. You will probably need code to construct the left/right values by traversing through an edge representation of the tree. You will also need a user interface supported by code to insert nodes into the tree, delete nodes, and move nodes. This code base probably won't be much more complex than that needed for robust support of an edge representation of a tree, but you will need to have such code, while you might be able to get away without writing it for an edge representation.

The biggest difficulty with the tree visitation algorithm is editing the tree. If you wish to add a single node (say the genus *Hexaplex* within the Muricidae in the example above), you will need to change left and right values for many, most, or even all of the rows in the database. In order to obtain a fast retrieval speed on queries that include the left and right field in select criteria, you will probably need to index those fields. An update to most rows in the table will essentially mean rebuilding the index for each of these columns, even if you only changed one node in the tree. In a single user database, this will make edits and inserts on the tree very slow. The situation will be even worse in a multi-user database. Updating many rows in a table that contains a tree will effectively lock the entire table from access by other users while the update is proceeding. If the tree is large and frequently edited table locks will make a the database effectively unusable.

Unless a table contains very few rows, it will need to have indexes for columns that are used to find rows in the table. Without indexes, retrieval of data can be quite slow. Efficient extraction of information from a table that uses tree visitation will mean adding indexes to the left and right fields.

```
CREATE UNIQUE INDEX t_left
   ON treetable(t_left);
CREATE UNIQUE INDEX t_right
   ON treetable(t_right);
```

When a row is added to the table, the values in the left and right fields will now be indexed (with the details of how the index is created, stored, and used depending on the DBMS). In general, indexes will substantially increase the storage requirements for a database. Creating an index will take some amount of time, and an index will increase the time needed to insert a row into a table (Figure 22). Appropriate indexes dramatically reduce the time required for a select query to return a result set.

Indexing

An index allows at DBMS to very rapidly look up rows from table. An appropriate choice of fields to index is a very important aspect of database implementation, and careful tuning of indexes is an important database administration task for a large database. Being able to reduce the time for a database operation from 10 seconds to 2

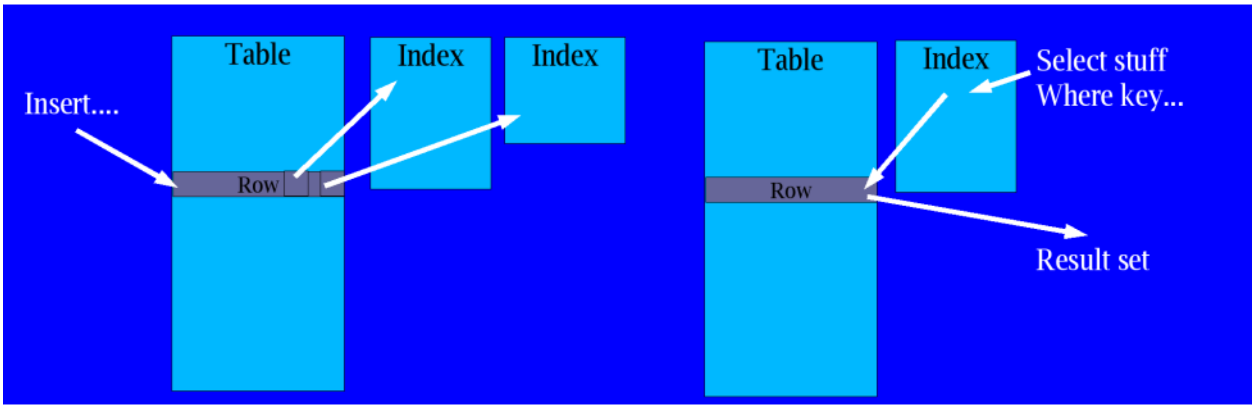


Figure 22. An index can greatly increase the speed at which data is retrieved from a database, but can slow the rate of data insertion as both the table and its indexes need to be updated.

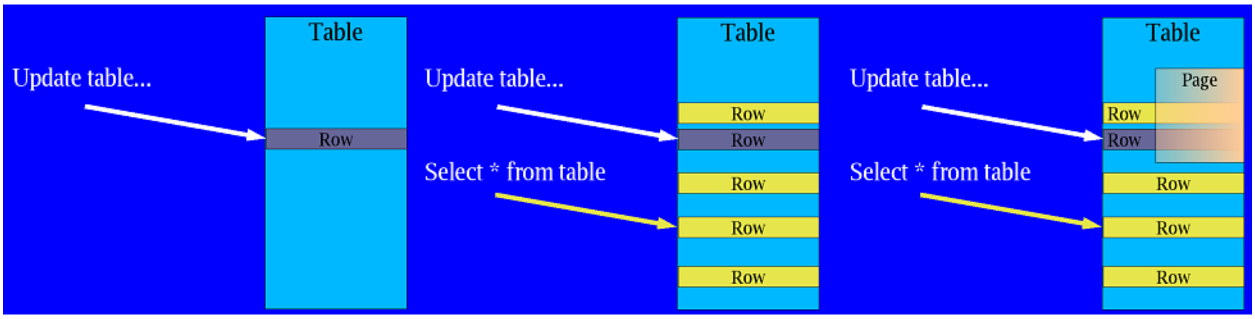


Figure 23. Page locks. Many database systems store data internally in pages, and an operation that locks a row in a page may also lock all other queries trying to access any other row on the same page.

seconds can change reduce the person years of effort in a large data capture operation and make the difference between completion of a project being feasible or infeasible. Small improvements in performance, such as reducing the time for a query to complete from 2 seconds to less than one second, will result in a large increase in efficiency in large scale (hundreds of thousands of records) data capture.

The details of how a multi-user DBMS prevents one user from altering the data that is being edited by another user, and how it prevents one user from seeing inconsistent data from a partially completed update by another user, are also an important performance concern for database implementation and administration.

An update to a single row will not interfere with other queries that are trying to select data from other rows if the table uses row level locking. If, however, the table applies locking at the level of database pages (chunks of data that the database is storing together as units behind the scenes), then an update that affects one row may lock out queries that are trying to access data from other rows (Figure 23). Some databases apply table level locking and will lock out all users until an update has completed. In some cases, locks can produce deadlocks where two users both lock each other out of rows that they need to access to complete a query.

Because of these locking issues, a tree visitation algorithm is a poor choice for storing large trees that are frequently edited in a multi-user database. It is, however, a good choice for some situations, such as storage of many small trees (using tree_number, t_left, and t_right fields) as might be used to track collection objects or for searchable web databases that are not edited by their users and are infrequently updated from a master data source. The tree visitation algorithm is particularly useful for searchable web databases, as it eliminates the need for multiple recursive queries to find data as in edge representations (a query on joined tables in MySQL in the example below will find the children of all taxa that start “Mure”).

```
SELECT a.taxon_name, b.taxon_name, b.rank
FROM treetable as a
  LEFT JOIN treetable as b
    ON a.taxonid = b.taxonid
WHERE b.t_left > a.t_left
  and b.t_right < a.t_right
  and b.taxon_name like “Mure%”
ORDER by a.taxon_name, t_left;
```

Because there are serious performance and code complexity tradeoffs between various ways of implementing the storage of hierarchical information in a relational database, choosing an appropriate tree representation for the situation at hand is quite important. For some situations, a tree visitation algorithm is an ideal way to handle hierarchical information, whereas in others it is a very poor choice. Since single point alterations to the tree result in large portions of the table holding the hierarchy being changed, tree visitation is a very poor choice for holding large taxonomic hierarchies that are frequently edited. However, its fast efficient extraction of data makes it a very good choice for situations, such as web databases, where users are seldom or never altering data, or where many small distinct trees are being held in one table.

The storage of trees is a very logical place to consider supporting data integrity with stored procedures. A table holding an edge representation of a tree is not easy to maintain by hand. Even skilled users can introduce records that create infinite loops, and unlinked subtrees (anastomoses, except those produced by misspellings, can be prevented by enforcing a unique index on child names). Murex could be linked to Murcinae which could be linked to Muricidae which could be linked by mistake back to Murex to create an infinite loop. Murex could be linked to Muricinae, which might by mistake not be linked to a higher taxon creating an unlinked subtree. An edge representation could be supported by an on insert/update trigger. This trigger could check to see if each newly inserted or updated parent has a match to an existing child (`select count(*) from taxon where child = %newparent`) – preventing the insertion of unlinked subtrees. If an on insert or on update trigger finds that a business rule would be violated by the requested change, it can fail and return an error message to the calling program. In most circumstances, such an error message should propagate back through the user interface to the user in a form that tells them what corrective action to take (“You can't insert Muricinae:Murex yet, you have to link Muricinae to the correct higher taxon first”) and in a form that will tell a programmer exactly where the origin of the error was (“snaildb_trigger_error_257”).

An on insert and on update trigger on a table holding an edge representation of a tree could also check that the parent of each newly inserted or updated row links to root without encountering child – preventing infinite loops. Checking each inserted or changed record for a parent that is linked up to the root of the tree would have a higher cost, as it would require a recursive series of queries to trace an unknown number of parent-child links back to the root of the tree. Decisions on how to store trees and what code to place where to support the storage of those trees can have a significant impact on the performance of a database. Slow updates may be tolerable in a taxonomic dictionary that serves as a rarely edited authority file for a specimen data set, but be intolerable for regular edits to a database that is compiling a tree of taxonomic names and their synonymies.

Triggers are also a very good place to maintain redundant data that has been cached in the database to reduce calculation or lookup time. On insert and on update triggers can, for example, maintain parentage strings in tables that store trees (where they will need to recurse the children of a node when that node is linked to a new parent, as this change will need to propagate down through all the parentage strings of the child nodes).

Hierarchical information is widespread in collection data. Therefore, in the process of beginning the design phase of a database life cycle, it is worth considering a wider range of DBMS options than just relational databases. Some other database systems are better than relational databases at native handling of trees and hierarchical information. Object oriented database systems should be considered, as they can include data types and data structures that are ideal for the storage and retrieval of hierarchical information. However, at the present time, using an object DBMS rather than a standard relational DBMS has costs, particularly poor portability and reduced availability of support. Relational databases have a solid well understood mathematical foundation (Codd, 1970), have widespread industry support for sql, are not hard to move data in and out of, and have a wide range of implementations by many vendors, including the open source community. Object databases (and hybrid object support in relational databases) are still none of these (thus this paper deals almost entirely with relational databases).

Data Stewardship

We often state to groups on tours through natural history collections that "our specimens are of no value without their associated data". We are well used to thinking of stewardship of our collection objects and their associated paper records, but we must also think carefully about the stewardship of electronic data associated with those specimens. I will divide data stewardship into two concepts: 1) Data security, that is maintaining the data in an environment where it is safe from malicious damage and accidental loss Data security encompasses network security, administration, pre-planning and risk mitigation. 2) Data quality control, that is procedures to assist the users of the database in maintaining clean and meaningful data. Data quality control is of particular importance during data migration, as we are often performing large scale transformations of collections records, and thus need to carefully plan to prevent both the loss of information and the addition of new erroneous information. Data stewardship involves the short term data security tasks of system administration and quality control and the long term perspective of planning for movement of the data through multiple database lifecycles.

Data Security

Providing carefully planned multiple layers of security has become an essential part of maintaining the integrity of electronic data. Any computer connected to the Internet is a target for attack, and any computer is a target for theft. Simply placing a computer behind a firewall is not, by any means, an adequate defense for it or its data. Security requires paranoia, but not just any old paranoia. Security requires a broad thinking paranoia. Computer data security, as in the design of cryptographic systems, depends on the strength of the weakest link in the system. Ferguson and Schneier (2003), discussing cryptographic system design, provide a very good visual analogy. A secure system is like the stockade walls of a fort, but, in the virtual world of computers and data, it is very easy to become focused on a single post in that stockade, trying to build that single post to an infinite height while not bothering to build any of the rest of the stockade. An attacker will, of course, just walk around this single tall post. Another good analogy for computer security is that of defense in depth (e.g. Bechtel, 2003). Think of security as involving many layers, all working together to prevent any single point of entry from giving easy access to all resources in the system.

In the collection management community there is a widespread awareness of the importance of preplanning for disaster response. Computer security is little different. Consider a plausible scenario: what do you do if you discover that your web database server is also hosting a child pornography ftp server? If you are part of a large institution, you may have an incident response team you can call on to deal with the situation. In a small institution, you might be the most skilled and trained person available. Preplanning and establishing an incident response capability (e.g. a team composed of people from around the institution with computing and legal skills who can plan appropriate responses, compile contact lists, assess the security of the institution's computing resources, and respond to computer incidents) is becoming a standard practice in information technology.

One of the routine threats for any computer connected to the Internet is attack from automated software seeking to exploit known vulnerabilities in the operating system, server services, or applications. It is therefore important to be aware of newly discovered vulnerabilities in the software that you are using in your network and to apply security patches as appropriate (and not necessarily immediately on release of a patch, Beatte et al., 2002). Vulnerabilities can also be exploited by malicious hackers who are interested in attacking you in particular, but it is much more likely that your computers will be targets simply because they are potential resources, without any particular selection of you as a target. Maintaining defenses against such attacks is a necessary component of systems administration for any computer connected to the Internet

Threat analysis

Computer and network security covers a vast array of complex issues. Just as in other areas of collection management, assessing the risks to biodiversity and collections information is a necessary precursor to effective allocation of resources to address those risks. Threat analysis is a comprehensive review and ranking of the risks associated with computer infrastructure and electronic data. A threat analysis of natural history collection data housed within an institution will probably suggest that the highest risks are as follows. Internal and external security threats exist for biodiversity information and its supporting infrastructure. The two greatest threats are probably equipment theft and non-targeted hacking attacks that seek to use machines as resources. An additional severe risk is silent data corruption (or malicious change) creeping into databases and not being noticed for years. Risks also exist for certain rare species. The release of collecting locality information for rare and endangered species may be a threat to those species in the wild. Public distribution of information about your institution's holdings of valuable specimens may make you a target for theft. In addition, for some collections, access to some locality information might be restricted by collecting agreements with landowners and have contractual limits on its distribution. A threat analysis should suggest to you where resources should be focused to maintain the security of biodiversity data.

Michael Wojcik put it nicely in a post to Bugtraq. “What you need is a weighted threat model, so you can address threats in an appropriate order. (The exact metric is debatable, but it should probably combine attack probability, likely degree of damage, and at a lesser weight the effort of implementing defense. And, of course, where it's trivial to protect against a threat, it's worth adding that protection even if the threat is unlikely.)” (Wojcik, 2004)

Implementing Security

If you go to discuss database security with your information technology support people, and they tell you not to worry because the machine is behind the firewall, you should worry. Modern network security, like navigation, should never rely on any single method. As the warning on marine charts goes “The prudent navigator will not rely solely on any single aid to navigation”, the core idea in modern computer network security is defense in depth. Security for your data should include a UPS, regular backup, offsite backup, testing backup integrity and the ability to restore from backup, physical access control, need limited access to resources on the network, appropriate network topology (including firewalls), encryption of sensitive network traffic (e.g. use ssh rather than telnet), applying current security patches to software, running only necessary services on all machines, and having an incident response capability and disaster recovery plan in place.

Other than thinking of defense in depth, the most important part of network security is implementing a rational plan given the available resources that is based upon a threat analysis. It is effectively impossible to secure a computer network against attacks from a government or a major corporation. All that network security is able to do is to raise the barrier of how difficult it will be to penetrate your network (either electronically or physically) and increase the resources that an intruder would need to obtain or alter your data. Deciding what resources to expend and where to put effort to secure your information should be based on an analysis of the threats to the data and your computer infrastructure.

While electronic penetration risks do exist for collections data (such as extraction of endangered species locality data by illicit collectors, or a malicious intruder placing a child pornography ftp site on your web server), the greatest and most immediate risks probably relate to physical security and local access control. Theft of computers and computer components such as hard drives is a very real risk. The spread of inhumane “human resources” practices from the corporate world creates a risk that the very people most involved in the integrity of collections data may seek to damage those data (in practice this is approached much the same way as the risks posed by mistakes such as accidentally deleting files by applying access control, backup, and data integrity checking schemes). After a careful threat analysis, three priorities will probably stand out: control on physical access to computers (especially servers), applying access control so that people only have read/write/delete access to the electronic resources they need, and by a well designed backup scheme (including backup verification and testing of data restoration).

Hardware

Any machine on which a database runs should be treated as a server. If that machine loses power and shuts down, the database may be left in an inconsistent state and become corrupt. DBMS software usually stores some portion of the changes being made to a database in memory. If a machine suddenly fails, only portions of the information needed to make a consistent change to the data may have been written to disk. Power failures can easily cause data corruption. A minimum requirement for a machine hosting a database (but not a machine that is simply connecting to a database server as a client) is an uninterruptible power supply connected to the server with a power monitoring card and with software that is capable of shutting down services (such as the DBMS) in the event of a prolonged power failure. In a brief power outage, the battery in the UPS provides power to keep the server running. As the battery starts to run down, it can signal the server that time is running out, and software on the server can shut down the applications on the server and shut down the server itself. Another level of protection that may be added to a server is to use a set of hard disks configured as a redundant RAID array (e.g. level 5 RAID). RAID arrays are capable of storing data redundantly across several hard disks. In the event that one hard disk in the array fails, copies of the data stored on it are also held on other disks in the array, and, when the failed drive is replaced with a new one, the redundant data are simply copied back into place.

Backup

Any threat analysis of biodiversity data will end up placing backups at or near the top of the risk mitigation list. A well planned backup scheme is of vital importance for a collection database. A good plan arises directly out of a threat analysis. The rate of change of information in a database and the acceptable level of loss of that information will strongly influence the backup plan. Each database has different rates of change, different backup needs and thus requires a plan appropriate for its needs. Regardless of how good a backup plan is in place, no records related to collection objects data should be held solely in electronic form. Paper copies of the data (preferably labels, plus printed catalog pages) are an essential measure to ensure the long term security of the data. In the event of a catastrophic loss of electronic data, paper records provide an insurance that the information associated with collection objects will not be lost.

Backups of electronic data should include regular on-site backups, regular off-site transport of backups, making copies on durable media, and remote off-site exchanges. The frequency and details of the backup scheme will depend on the rate of data entry and the threat analysis assessment of how much data you are willing to re-enter in the event of failure of the database. A large international project with a high rate of change might mandate the ability to restore up to the moment of failure. Such a project might use a backup scheme using daily full backups, hourly backups of incremental changes since the last full backup, and a transaction log that allows restoration of data from the last backup to the most recent entry in the transaction log, requiring the use of a SQL server DBMS capable of this form of robust backup. (A transaction log records every update insert and delete made to the database and allows recovery from the last incremental backup to the moment of failure). Such backups might be written directly on to a tape device by the DBMS, or they might be written to the hard disk of another server. This scheme might be coupled with monthly burns of the backups to cd, or monthly archiving of a digital backup tape, and monthly shipment of copies of these backups to remote sites. At the other end of the spectrum, a rarely changed collection database might get one annual burn to cd along with distribution of backup copies to an offsite backup store and perhaps a remote site.

Database backups are different from normal filesystem backups. In a filesystem backup, a file on disk is simply copied to another location (such as onto a backup tape). A database that is left open and running, however, needs to have backup copies made from within the database software itself. If a DBMS has a database open during a filesystem backup, a server backup process that attempts to create backup copies of the database files by copying them from disk to another location will most likely only be able to produce a corrupt inconsistent copy of the database. Database management software typically keeps recent changes to the data in memory, and does not place the database files on disk into a consistent state until the database is closed and the software is shutdown. In some circumstances, it is appropriate to shutdown the DBMS and make filesystem backups of the closed database files. Backups of data held on a database server that is running all the time, however, need to be planned and implemented from both within the database management software itself (e.g. storing backup copies of the database files to disk) and from whatever process is managing backups on the server (e.g. copying those backup files to tape archives). It is also important not to blindly trust the backup process. Data verification steps should be included in the backup scheme to make sure that valid accurate copies of the data are being backed up (minor errors in copying database files can result in corrupt and unusable backup copies).

Offsite storage of backup copies allows recovery in the case of local disaster, such as a fire destroying a server room and damaging both servers and backup tapes. Remote storage of backup copies (e.g. two museums on different continents making arrangements for annual exchange of backup copies of data) could be valuable documentation of lost holdings to the scientific community in the event of a large regional disaster.

In developing and maintaining a backup scheme it is essential to go through the process of restoring those backups. Testing and practicing restoration ensures that when you do have to restore from backup you know what to do and know that your recovery plan includes all of the pieces needed to get you back to a functional restored database. “Unfortunately servers do fail, and many an administrator has experienced problems during the restore process because he or she had not practiced restoring databases or did not fully understand the restore process” (Linsenbardt and Stigler, 1999 p.272).

With many database systems both the data in the database and the system or master database are required to restore a database. In MySQL, user access rights are stored in the MySQL database, and a filesystem backup of one database on a mysql server will not include user rights. MS SQL Server likewise suggests creating a backup of the system database after each major change to included databases, as well as making regular backups of each database. The PostgreSQL documentation suggests making a filesystem backup of only the entire cluster of databases on a server and not trying to identify the files needed to make a complete filesystem backup of a single database. “This will not work because the information contained in these files contains only half the truth. The other half is in the commit log files pg_clog/*, which contain the commit status of all transactions. A table file is only usable with this information.” (PostgreSQL Global Development Group, 2003). Database backups can be quite complex, and testing the recovery process is important to ensure that all the components needed to restore a functional database are included in the backup scheme.

Access Control

Any person with high level access to a database can do substantial damage, either accidentally or on purpose. Anyone with high level access to a database can do substantial damage with a simple command such as DROP DATABASE, or more subtly through writing a script that gradually degrades the data in a database. No purely electronic measures can protect data from the actions of highly privileged uses. Database users should be granted only the minimum privileges they need to do their work. This principle of minimum privilege is a central concept in computer security (Hoglund & McGraw, 2004). Users of a biodiversity database may include guests, data entry personnel, curators, programmers, and system administrators. Guests should be granted only read (select) only access, and that only to portions of the database. Low level data entry personnel need to be able to enter data, but should be unable to edit controlled vocabularies (such as lists of valid generic names), and probably should not be able to create or view transactions involving collection objects such as acquisitions and loans. Higher level users may need rights to alter controlled vocabularies, but only system administrators should have the ability to grant access rights or create new users. Database management systems include, to varying degrees of granularity, the ability to grant users rights to particular operations on particular objects in a database. Many support some form of the SQL command GRANT rights TO user ON resource. Most access control is best implemented by simply using the access control measures present in the database system, rather than coding access control as part of the business rules of a user interface to the database. Restriction of access to single records in the database (row level access control), however, usually needs to be implemented in higher layers.

Physical access control is also important. If a database server is placed in some readily accessible space, a passerby might shut it down improperly causing database corruption, unplug it during a disk write operation causing physical failure of a hard disk, or simply steal it. Servers are best maintained in spaces with access limited to knowledgeable IT personnel, preferably a server room with climate control, computer safe fire suppression systems, tightly restricted access, and video monitoring.

System Administration

Creating new user accounts, deactivating old user accounts and other such maintenance of user access rights are tasks that fall to a database administrator. A few years ago, such maintenance of rights on the local machine, managing backups and managing server loads were the principle tasks of a system administrator. Today, defense of the local network has become a absolutely essential system administration task. A key part of that defense is maintaining software with current security patches. Security vulnerabilities are continuously brought to light by the computer security research community. Long experience with commercial vendors unresponsive to anything other than public disclosure has led to a widespread adoption of an ethical standard practice in the security community. An ethical security researcher is expected to notify software vendors of newly discovered vulnerabilities, then provide a 30 day grace period for the vendor to fix the vulnerability, followed by public release of details of the vulnerability. Immediate public release of vulnerabilities is considered unethical as it does not provide software vendors with any opportunity to protect people using their software. Waiting an indefinite period of time for a vendor to patch their software is also considered unethical, as this leaves the entire user community vulnerable without knowing it (it being the height of hubris for a security researcher to assume that they are the only person capable of finding a particular vulnerability). A 30 day window from vendor notification to public release is thus considered a reasonable ethical compromise that best protects the interests of the software user community. Some families of vulnerabilities (such as C strcpy buffer overflows, and SQL injection) have proven to be very widespread and relatively easy to locate in both open and closed source software. A general perception among security researchers (e.g. Hoglund and McGraw, 2004 p.9) is that many exploits are known in closed circles among malicious and criminal programmers (leaving all computer systems open to attack by skilled individuals), and that public disclosure of vulnerabilities by security researchers leads to vendors closing security holes and a general increase in overall internet and computer security. These standard practices of the security community mean that it is vitally important for you to keep the software on any computer connected to the Internet up to date with current patches from the vendor of each software package on the system.

Installing a patch, however, may break some existing function in your system. In an ideal setting, patches are first installed and tested on a separate testing server (or local network test bed) and then rolled out to production systems. In most limited resource settings (which also tend to lack the requirement of continuous availability), patching involves a balance between the risks of leaving your system unpatched for several days and the risks of a patch taking down key components of your system.

Other now essential components of system administration include network activity monitoring and local network design. Network activity monitoring is important for evaluating external threats, identifying compromised machines in the internal network, and identifying internal users who are misusing the network, as well as the classical role of simply managing normal traffic flow on the network. Possible network components (Figure 24) can include a border router with firewall limiting traffic into and out of the network, a network address translation router sitting between internal network using private ip address space and the Internet, firewall software running on each server limiting accessible ports on that machine, and a honeypot connected to border router.

Honeypots are an interesting technique for system activity monitoring. A honeypot is a machine used only as bait for attackers. Any request a honeypot receives is interpreted as either a scan for vulnerabilities or a direct attack. Such requests can be logged and used to evaluate external probes of the network and identify compromised machines on the internal network. Requests logged by a honeypot can also used to update the border router's rules to exclude any network access from portions of the Internet. Honeypots can also be set up as machines left open with known vulnerabilities in order to study hacker behavior, but such use may raise ethical and legal issues in ways that using a honeypot simply to identify potential attackers does not. Monitoring system activity is, as noted above, a system administration task that is an important component of the defense in depth of a network.

Example: SQL Injection Attacks

One particular serious risk for a database that is made available for search over the Internet is sql injection attacks (Anley 2002; Smith 2002). If a database is providing a back end for a web search interface, it is possible for any person on the Internet to inject malicious queries to the database using the web interface. In some cases (such as execution of MS SQLServer stored procedures that execute shell commands), it is possible for an attacker to not only alter data in the database, but to also execute arbitrary commands on the server with the privileges of the database software. That is, it may be possible for an attacker to take complete control of a server that is providing a database for search on the web.

Most web searchable databases are set up by setting up the database in a sql server (such as MySQL, MS SQLServer, or PostgreSQL), and writing code to produce the html for the web search and results pages in a scripting language (such as PHP, ASP, or CGI scripts in PERL). A user will fill in a form with search criteria. This form will then be submitted to a program in the scripting language that will take the criteria provided in the form submission, create an sql query out of them, submit that query to the underlying database, receive a result set back, format that result set as a web page, and return the resulting html document to the user. If this system has not been set up with security in mind, a malicious user may be able to alter data in the database or even take control of the server on which the database is running. The attack is very simple. An attacker can fill in a field on a form (such as a genus field) with criteria such as `'; drop database;` which could be assembled into a query by the scripting language as “`SELECT genus, trivial`

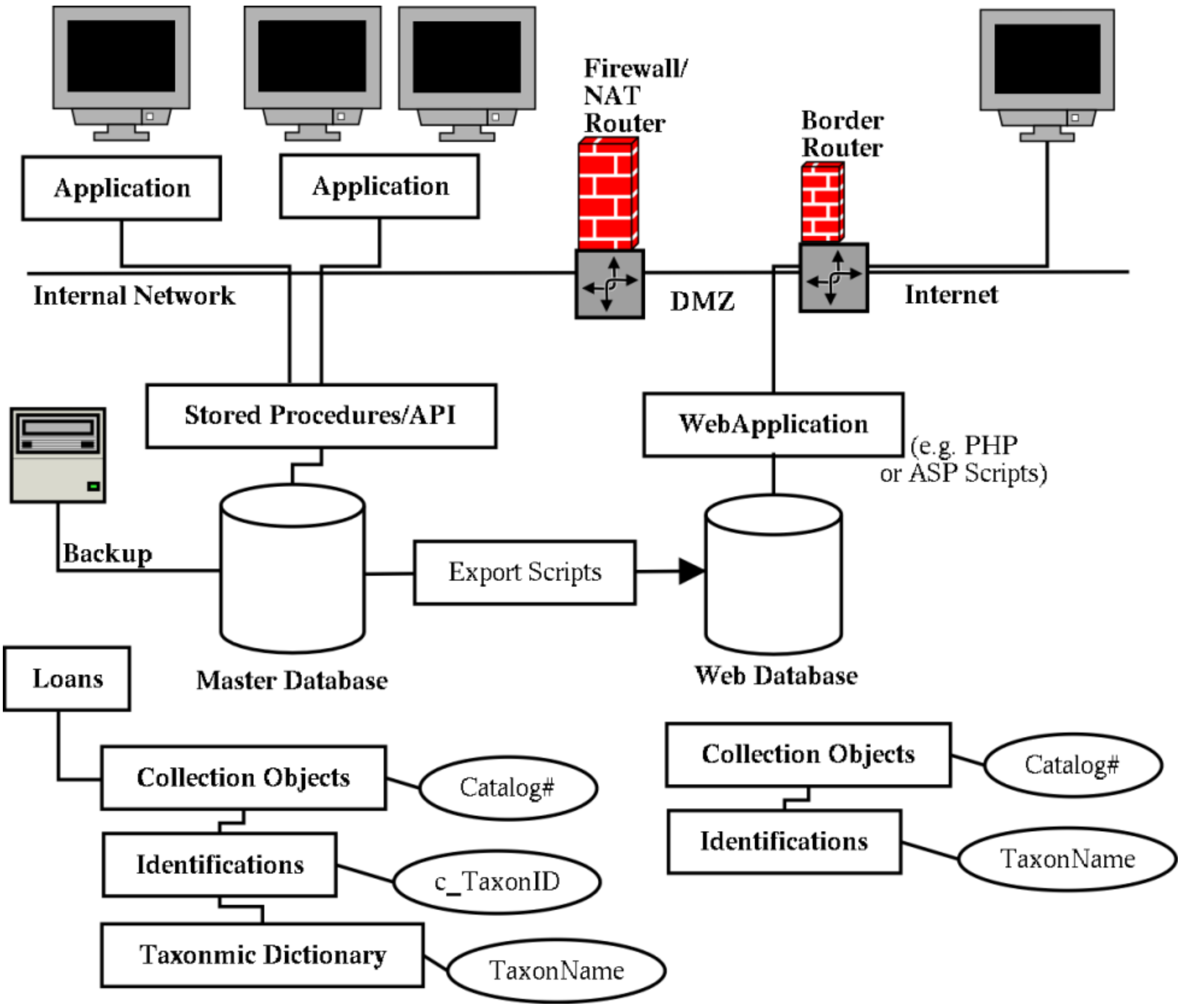


Figure 24. A possible network topology with a web accessible database readily accessible to the Internet, and a master database behind more layers of security. The web accessible copy of a database can be denormalized and heavily indexed relative to the master database.

`FROM taxon  WHERE genus like ' '; drop database; ' `", a series of three sql queries that it will then pass on to the database. The database might interpret the first command as a valid select statement, return a result set, see the semicolon as a separator for the next query, execute that by dropping (that is, deleting) the database, see the second semicolon as another separator, and return an error message for the third query made up of just a single quotation mark.

Defense against sql injection attacks involves following two basic principles of network security. First, never trust any information provided to you by users, especially users over the Internet. Second, allow users only the minimum access rights they need. All code that runs behind a web interface should sanitize anything that might be provided to it by a user (including hidden values added to a form, as the source html of the form is available to the user, and they are free to alter it to make the submission say anything they desire). This sanitization should take the form of disallowing all characters except those known to be valid (rather than disallowing a set of known attacks). The code for the example above might include a sanitize routine that contains a command that strips everything except the letters A to Z (in upper and lower case) from whatever the user provided as the criteria for genus. A regular expression pattern for everything outside this valid range of characters is: **`/[^A-Za-z]/`**, that is match anything not (^) in the range ([ ]) A-Z or a-z. An example (in the web scripting language PHP) of a function that uses such a regular expression to sanitize the content of variables holding information provided over the web that might contain generic and trivial names is shown below.

```
function sanitize() {
 global $genus,$trivial;
 $genus ~=
   preg_replace(“/[^A-Za-z] ”,””,$genus);
 $trivial ~=
   preg_replace(“/[^a-z]/”,””,$trivial);
}
```

This function uses a regular expression match that examines the variable $genus and replaces any characters that are not in the range A-Z or the range a-z with blank strings. Thus an attack suppling a value for $genus of “`'; drop database;`” would be sanitized to the innocuous search criterion “dropdatabase”. Note that the sanitize function is explicitly listing allowed values and removing everything else, rather than just dropping the known dangerous values of semicolon and single quote. An attack may evade “exclude the bad” filters by encoding attack characters so that they don't appear bad to the filter, but are decoded an act somewhere beyond the filter. A single quote might be url encoded as %27, and would pass unchanged through a filter that only excludes matches to the ; and ' characters. Always limit user input to known good characters, and be wary when the set of allowed values extends beyond [A-Za-z0-9]. Tight control on user provided values is substantially more difficult when unicode (multi-byte characters) and characters outside the basic ASCII character set are involved.

It is also important to limit user rights to the absolute minimum necessary. If your scripting language program has an embedded username and password to allow it to connect to your database and retrieve information, you should set up a username and password explicitly for this task alone, and grant this user only the minimum select privileges to the database. The details of this will vary substantially from one DBMS to another. In MySQL, the following commands might be appropriate:

```
GRANT SELECT
  ON webdb.webtable
  TO phpuser@localhost
    IDENTIFIED BY PASSWORD
       “plaintextpassword”
```

or to be explicit with the MySQL privilege tables[6]:

```
INSERT INTO user
  (host,  user,
   password,
   select_priv, insert_priv,
   update_priv, delete_priv )
 VALUES
  ( “localhost”,”phpuser”,
    password(“plaintextpassword”),
    “N”, ”N”, ”N”, “N” );
INSERT INTO db
  (db, user,
   select_priv, insert_priv,
   update_priv, delete_priv )
 VALUES
   (“webdb”, “phpuser”,
    “N”, ”N”, ”N”, “N” );
INSERT INTO tables_priv
   (host, db, user,
    table_name, table_priv )
 VALUES
   (“localhost”,“webdb”, “phpuser”,
    “webtable”, “Select” );
```

With privileges restricted to select only on the table you are serving up on the web, even if an attacker obtains the username and password that you are using to allow the scripting language to access the database (or if they are able to apply an sql injection attack), they will be limited in the kinds of further attacks they can perform on your system. If, however, an attacker can obtain a valid database password and username they may be able to exploit security flaws in the DBMS to escalate their privileges. Hardcoding a username / password combination in a web script, a web scripting function placed off the web tree, or in a configuration file off the web tree is generally necessary to allow web pages to access a database. Any hardcoded authentication credential provides a possible target for an internal or external attacker. Through an error in file naming or web server configuration, the source code of a web script placed on the publicly accessible web tree may become visible to outside users, exposing any

[6] You should empty out the history files for MySQL or your shell if you issue a command that embeds a password in it in either a MySQL query or a shell command, as history files are potentially accessible by other users.

authentication credentials hard coded in that file. Users on the local file system may be able to read files placed outside of the web tree or be able to escalate their privileges to read configuration files with embedded passwords. It is thus very important to limit to the absolute minimum needed privileges the user rights of any usernames that are hardcoded outside of the database. In a DBMS that includes stored procedures, you should explicitly deny the web user the rights to run stored procedures (especially in the case of MS SQLServer, which comes with built in stored procedures that can execute shell commands, meaning that any user who is able to execute stored procedures can access anything on the server visible to the user account that the SQLServer is running under, which ought to be an account with privileges restricted to the minimum needed for the operation of the server). Alternately, run all web operations through stored procedures, and grant the web user rights for those stored procedures and nothing else (tight checking of variables passed to stored procedures and denial of rights to do anything other than execute a small set of stored procedures can assist in preventing sql injection attacks).

Even if the details of the discussion above on sql injection attacks seem obscure, as they probably will if you haven't worked with web databases and scripting languages, I hope the basic principle of defense in depth has come across. The layers of firewall (to restrict remote access to the server to port 80 [http]) , sanitizing all information submitted with a form before doing anything with it, limiting the webuser's privileges on the sql server, and limiting the sql server's own privileges all work together to reduce the ability of attackers to penetrate your system.

Computer and network security is a constantly changing and evolving field. It is, moreover, an important field for all of us who are running client-server databases, web serving information out of databases, or, indeed, simply using computers connected to the Internet. Anyone with responsibility over data stored on a computer connected to the Internet should be at least working to learn something about network security practices and issues. Monitoring network security lists (such as securityfocus.com's bugtraq list or US-CERT's technical cyber security alerts) provides awareness of current issues, pointers to papers on security and incident response, and opportunities to learn about how software can be exploited.

Maintaining Data Quality

The quality of your data are important. Incorrect data may become permanently associated with a specimen and might in the future be used draw incorrect biological conclusions. While good database design and good user interface design assist in maintaining the quality of your data over time, they are not enough. The day to day processes of entering and modifying data must also include quality control procedures that include roles for everyone involved with the database.

Quality Control

Quality control on data entry covers a series of questions: Are literal data captured correctly? Are inferences correctly made from the literal data? Are data correctly captured into the database, with information being entered into the correct fields in the correct form? Are links to resources being made correctly (e.g. image files of specimens)? Are the data and inferences actually correct? Some of these questions can be answered by a proofreader, others by queries run by a database administrator, whereas others require the expert knowledge of a taxonomist. Tools for quality control include both components built into the database and procedures for people to follow in interacting with the database.

At the database level controls can be added to prevent some kinds of data entry errors. At the most basic level, field types in a database can limit the scope of valid entries. Integer fields will throw an error if a data entry person tries to enter a string. Date fields require a valid date. Most data in biodiversity databases, however, goes into string fields that are very loosely constrained. Sometimes fields holding string data can be tightly constrained on the database level, as in fields that are declared as enumerations (limited in their field definition to a small set of values). In some cases, atomization can help with control. A string field for specimen count might be split into an integer field to hold the count, an enumerated field to hold the kind of objects being counted, and a qualifier field. Constraints in the database can test the content of one field in a record against others. A constraint could, for example, force users to supply a source of a previous number if they provide a previous number. Code in triggers that fire on inserts and updates can force the data entered in a field to conform to a defined pattern, such as "####-##-##" for dates in ISO format. Similar constraints on the scope of data that will be accepted as valid can be applied at the user interface level. While these constraints will trap some data entry errors (some typographic errors and some data entry into incorrect fields), the scope of valid input for a database field will always be larger than the scope of correct input. Mistakes will occur on data entry. Incorrect data will be entered into the database regardless of the controls placed on data entry. To have any assurance that the data entered into a biodiversity database is accurate, quality control measures beyond simply controlling data entry and trusting data entry personnel are necessary.

The database administrator can conduct periodic review of large numbers of records, or tools can be built into the database to allow privileged users to easily review large blocks of records. The key to such bulk review of records is to look for outliers. An easy way to find outliers is to sort columns and look at top and bottom records to find obvious out of range values such dates entered in text fields.

```
SELECT TOP 10 named_place
   FROM collecting_event
   ORDER BY named_place;

SELECT TOP 10 named_place
   FROM collecting_event
   ORDER BY named_place DESC;
```

The idea of a review of the top and bottom of alphabetized lists can be extended to a broader statistical review of field content for identification and examination of outliers. Correlation of information between fields is a rich source of tests to examine for errors in data entry. Author, year combinations where the year is far different from other years with the same authorship string are good candidates for review, as are outlying collector – collecting event date combinations. Comparison of a table of bounding latitudes and longitudes for countries and primary divisions with the content of country, primary division, latitude, and longitude fields can be used either as a control on data entry or as a tool for examination of outliers in subsequent review.

Ultimately quality control rests on review of individual records by knowledgeable persons. This review is an inherent part of the use of biological collections by systematists who examine specimens, make new identifications, comment on locality information, and create new labels or annotations. In the context of biological collections, these annotations are a living part of an active working collection. In other contexts, knowledge of the quality of records is important for assessing the suitability of the data for some analysis.

The simplest tool for recording the status of a record is a field holding the status of a record. A slightly more complex scheme holds the current state of the record, who placed the record into that state, and a time stamp for that action. Any simple scheme like this (who last updated, date last updated, etc), which records status information in the same table as the data, suffers all the problems of any sort of flat file handling of relational data. A database record can have many actions taken on it by many different people. A more general solution to activity tracking holds the activity information in a separate table, which can hold records of what actions were taken by what agents at what times. If a project needs to credit the actions taken by participants in the project (how many species occurrence records has each participant created, for example), then a separate activity tracking table is essential. In some parts of some data sets (e.g. identifications in a collection), credit for actions is inherent in the data, in others it is not. A good general starting place for recording the status of records is to time stamp everything. Record who entered a record when, who reviewed it when, who modified it how when. Knowing who did what when can have much more utility than simply crediting work. In some data entry tasks, an error can easily be repeated over a set of adjacent records entered by one data entry person, analogous to an incorrect catalog number being copied to the top of the catalog number column on a new page in a handwritten ledger. If records are creator and creation time stamped, records that were entered immediately before and after a suspect record can be easily examined for similar or identical errors. Time stamping also allows for ready calculation of data entry and review rates and presentation of rate and review quality statistics to data entry personnel.

Maintenance of controlled vocabularies should be tightly regulated. If regular data entry personnel can edit controlled vocabularies, they will cease to be controlled. Dictionary information (used to generate picklists or to check for valid input) can be stored in tables to which data entry personnel are granted select only access, while a restricted class of higher level users are granted update, insert, and delete access. This control can be mirrored in the user interface, with the higher level users given additional options for dictionary editing. If dictionary tables are hard to correctly maintain (such as tables holding edge representations of trees through parent-child links), even advanced and highly skilled users will make mistakes, so integrity checks (e.g. triggers that check that the parent value of a record correctly links it to the rest of the tree) should be built into the back end of the database (as skilled users may well bypass the user interface).

Quality control is a process that involves multiple people. A well designed data entry interface cannot ensure the quality of data. Well trained data entry personnel cannot ensure the quality of data. Bulk overview of unusual records by a database administrator cannot ensure the quality of data. Random review (or statistical sampling) of newly created records by the supervisors of data entry personnel cannot ensure data quality. Review of records by specialists cannot ensure data quality. Only bringing all of these people together into a quality control process provides an effective means of quality control.

Separation of original data and inferences

Natural history collection databases contain information about several sorts of things. First, they contain data about specimens: where they were collected, their identifications, and who identified them. Second, they contain inferences about these data – inferences such as geographic coordinates added to georeference data that did not originally include coordinates. Third, they contain metadata about these subsequent inferences, and fourth, they contain transaction data concerning the source, ownership, permitting, movement, and disposition of specimens.

Efforts to capture text information about collection objects into databases have often involved making inferences about the original data and adding information (such as current country names) to aid in the retrieval of data. Clearly, distinguishing the original data from subsequent inferences is usually straightforward for a specialist examining a collection object. Herbarium sheets usually contain many generations of annotations, mammal and bird skins usually have multiple attached tags, molluscan dry collections and fossils usually have multiple sets of labels within a tray. Examining these paper records of the annotation history of a specimen usually makes it clear what data are original and what are subsequent inferences. In contrast, database records, unless they include carefully planned metadata, often present only a single view of the information associated with a specimen – not clearly indicating which portions of that data are original and which are subsequent inferences.

It is important wherever possible to store an invariant copy of the original data associated with a collection object and to include metadata fields in a database to indicate the presence and nature of inferences. For example, a project to capture verbatim records from a handwritten catalog can use these data to produce a normalized database of the collection but should also store a copy of the original verbatim records in a simple flat table. These verbatim records are thus readily available for examination by someone wondering about an apparent problem in the data related to a collection object. Likewise, it is important to include some field structure to store the source of coordinates produced in a georeferencing project – and to allow for the storage and identification of original coordinates. Most important, I feel, is that any inferences that are printed onto paper records associated with specimens need to be marked as inferences. As older paper records degrade over time, newer paper records have the potential of being considered copies of the data on those original records unless subsequent inferences included upon them are clearly marked as inferences (in the simplest case, by enclosing inferences in square brackets).

Error amplification

Large complex data sets that are used for analyses by many users, who often lack a clear understanding of data quality, are susceptible to error amplification. Natural history collection data sets have historically been used by specialists examining small numbers of records (and their related specimens). These specialists are usually highly aware of the variability in quality of specimen data and routinely supply corrections to identifications and provenance data. Consider, in contrast, the potential for large data sets of collection information to be linked to produce, for example, range maps for species distributions based on catalog data. Consider such a range map incorporating erroneous data points that expand the apparent range of a species outside of its true range. Now consider a worker using these aggregate data as an aid to identifying a specimen of another species, taken from outside the range of the first species, then mistakenly identifying their specimen as a member of the first species, and then depositing the specimen in a museum (which catalogs it and adds its incorrect data to the aggregate view of the range of the first species). Data sets that incorporate errors (or inadequate metadata) from which inferences can be made, and to which new data can be added based on those inferences are subject to error amplification. Error amplification is a known issue in molecular sequence databases such as genbank (Pennisi, 1999; Jeong and Chen, 2001) where incorrect annotation of one sequence can lead to propagation of the error as new sequences are interpreted based on the incorrect annotation and propagate the error as they are added to the database. Error amplification is an issue that we must be extremely careful with as we begin large scale analysis of linked collections databases.

Data Migration and Cleanup

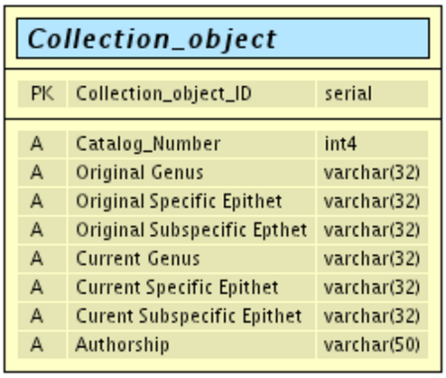
Legacy data

Biodiversity informatics data sets often include legacy data. Many natural history collections began data entry into early database systems in the 1970s and have gone through multiple cycles through different database systems. Biological data sets that we encounter were often compiled with old database tools that didn't effectively allow the construction of complex relational databases. Individual scientists with no training in database design often construct flat data sets in spreadsheets to manage results from their research and information related to their research program. Thus, legacy data often include a typical set of inventive approaches to handling relational information in flat files. Understanding the design problems that the authors of these datasets were trying to solve can be a great help in understanding what appear to be peculiarities in legacy data. Legacy data often contain non-atomic data, such as single taxon name fields, and non-normal data, such as lists of values in a single field.

In broad terms, legacy data present several classes of challenges. At one level, simply porting the data to a new DBMS may not be trivial. Older DBMS systems may not have an option to simply export all fields as text, and newer databases may not have import filters for those older systems. Consequently, you may need to learn the details of data export or data storage in the older system to extract data in a usable form. In other cases, export to a newer DBMS format may be trivial. At another level, legacy data often contain complex, poorly documented codings. Understanding the data that you are porting is very important as information can be lost or altered if codings are not properly interpreted during data migration. At another level, data contain errors. Any data set can contain data entry errors. Flat file legacy data sets tend to contain errors that make export to normal data structures difficult without extensive cleanup of the data. Non atomic fields can contain complex variant combinations of their components, making parsing difficult. Fields that contain repeated information (e.g. Country, State, PrimaryDivision) will contain many misspellings and variant forms of what should be identical rows. Cleanup of large datasets containing these sorts of issues may be assisted by several sorts of tools. A mapping table with one field containing a list of all the unique values found in one field in the legacy database and another field containing corrected values can be used to create a cleaned intermediate table, which can then be transformed into a more normal structure. Scripting languages with pattern matching capabilities can be used to compare data in a mapping table with dictionary files and flag exact matches, soundex matches, and similarities for human examination. Pattern matching can also be used to parse out data in a non-atomic field, with some fields parsing cleanly with one pattern, others with another, others with yet another, leaving some small set of exceptions to be fixed by a human. Data cleanup is a complex task that ultimately needs input from highly knowledgeable specialists in the data, but, with some thought to data structures and algorithms, much of the process can be automated. Examples of problems in data arising from non-normal storage of information and techniques for approaching their migration are given below. A final issue in data migration is coding a new front end to replace the old user interface. To most users of the database, this will be seen as the migration – replacing one database with another.

Documentation Problems

A common legacy problem during data migration is poor documentation of the original database design. Consider a database containing a set of three fields for original genus, original specific epithet, and original subspecies epithet, plus another three fields for current genus, current specific epithet, and current subspecies epithet, plus a single field for author and year of publication (Figure 25).



Collection_object

PK	Collection_object_ID	serial
A	Catalog_Number	int4
A	Original Genus	varchar(32)
A	Original Specific Epithet	varchar(32)
A	Original Subspecific Epthet	varchar(32)
A	Current Genus	varchar(32)
A	Current Specific Epithet	varchar(32)
A	Curent Subspecific Epithet	varchar(32)
A	Authorship	varchar(50)

Figure 25. An example legacy collection object table illustrating typical issues in legacy data.

Unless this database was designed and compiled by a single individual who is still available to explain the data structures, the only way to tell whether author and year apply to the current identification or the original identification will be to sample a large number of rows and look up authorships of the species names in a reliable source. In a case such as this, you might well find that the author field had been put to different purposes by different data entry people.

A similar problem exists in another typical design of fields for genus, specific epithet, subspecific epithet, author, and year. The problem here is more subtle – values in the author field for subspecies might apply to the species name or to the subspecies name, and untangling errors will require lots of work or queries against authoritative nomenclators. Such a problem might also have produced incorrect printed labels, say if in this example, the genus, specific epithet, and author were always printed on labels, but sometimes the author was filled in with the author of the subspecies epithet. These sorts of issues arise not from problems in the design of the underlying database, but in its documentation, its user interface, and in the documentation on data entry practices and training provided for data entry personnel. In particular, changes in the user interface over time, secondary to weak documentation that did not explain the business rules of the database clearly enough to the programmers of replacement interfaces, can result in inconsistent data. Given the complexity of nomenclatural and collections associated information, these problems might be more likely to be present in complex databases designed and coded by computer programmers (with strong knowledge of database design) than in simpler, less well designed databases produced by systematists (with a very detailed knowledge of the complexity of biological information).

I will now discuss specific examples of handling issues in legacy data. I have selected typical issues that can be found in legacy data related to the design of the legacy database. These problems may not reflect poor design in the legacy database. The data structures may have been constrained by the limitations of an earlier DBMS, the data may have been captured before database design principles were understood, or the database may have been designed for a different purpose.

Data entered in wrong field

A common problem in legacy data is values that are clearly outside the range of appropriate values for a field, such as "03-12-1860" as a value for donor. Values such as this commonly creep into databases from accidental entry into the wrong field on a data entry interface, such as the date donated being typed into the donor field.

A very useful approach to identifying values that were entered into the wrong field is to sort all the distinct values in each field with a case sensitive alphabetic sort. Values that are far out of range will often appear at the top or bottom of such lists (values beginning with numbers appearing before values beginning with letters, and values beginning with lower case letters appearing before those beginning with upper case letters). The database can then be searched for records that contain the strange out of range values, and these records will often contain several fields that have had their values displaced. This displacement may reflect the organization of fields on the current data entry layout, or it may reflect some older data entry screen.

Records from databases that date back into the 1970s or earlier should be examined very carefully when values that appear to have been entered into the wrong field are found. These errors may reflect blind edits applied to the database by row number, or other early data editing methods that required knowledge of the actual space occupied by a particular value in a field. Some early database editing methods had the potential to cause an edit to overwrite nearby fields. Thus, when checking values in very old data sets, examine all fields in a suspect record, as well as records that may have been stored in proximity to that record.

Regular expressions can also be used to search for out of range values existing in fields. These include simple patterns such as /^[0-9]{4}$/ which will match a four digit year, or more elaborate patterns such as /^(1[789]/20)[0-9]{2}$/ which will match four digit years from 1700-2099.

A rather more subtle problem can occur when two text fields that can contain similar values sit next to each other on the data entry interface. Species (specific epithet) and subspecies (subspecific epithet) fields both contain very similar information and, indeed, can contain identical values.

Atomization problems

Sometimes legacy datasets include multiple concepts placed together in a single field. A common offender is the use of two fields to hold author, year of publication, and parentheses. Rows make plain sense for combinations that use the original genus (and don't parenthesize the author), but end up with odd looking data for changed combinations (Table 26).

Table 26. Parentheses included in author.

Generic name	Trivial	Autho	Year of
	epithet	r	Publication
Palaeozygopleura	hamiltoniae	(Hall	1868)

The set of fields in Table 26 works perfectly well in a report that always prints taxon name + author + , + year, and can be relatively straightforwardly scripted to produce reports that print taxon name + author + closing parenthesis if author starts with parenthesis, but make for lots of complications for other operations (such as producing a list of unique authors) or asking questions of the database. Splitting out parentheses from these fields and storing them in a separate field is straightforward, as shown in the following 5 SQL statements (which add a column to hold a flag to indicate whether parentheses should be applied, populate this column, update the author and year fields to remove parentheses, and check for exceptions).

```
ALTER TABLE names ADD COLUMN paren
BOOLEAN DEFAULT FALSE;
UPDATE names
   SET paren = TRUE
   WHERE LEFT(author,1)= ”(“;
UPDATE names
   SET author =
       RIGHT(author,LENGTH(author)-1)
   WHERE paren = TRUE;
UPDATE names
   SET year = LEFT(year,LENGTH(year)-1)
   WHERE
    paren = TRUE and RIGHT(year,1)= ”)”;
SELECT names_id, author, year
 FROM names
 WHERE paren=FALSE
    AND ( INSTR(author,”(”)>0
       OR INSTR(author,”)”)> 0
       OR INSTR(year,”(”)>0
       OR INSTR(year,”)”)> 0);
```

Another common offender in legacy data is a field holding non-atomized taxon names (Table 27) including author and year.

Table 27. A non-atomic taxon name field holding a species name with its author, year of publication, and parentheses indicating that it is a changed combination.

Taxon Name
Palaeozygopleura hamiltoniae (Hall, 1868)

In manipulating such non-atomic fields you will need to parse the string content of the single field into multiple fields (Table 28). If the names are all simple binomials without author and year, then splitting such a species name into generic name and specific epithet is simple. Otherwise, the problem can be very complex.

Table 28. Name from Table 27 atomized into Generic name, specific epithet, author, year of publication fields and a boolean field (Paren) used to indicate whether parentheses should be applied to the author or author and year.

Generic name	*Specific_ epithet*	*Author*	*Year*	*Paren*
Palaeozygopleura	hamiltoniae	Hall	1868	TRUE

In some cases you will just need to perform a particular simple operation once and won't need to write any code. In other cases you will want to write code to repeat the operation several times, make the operation feasable, or use the code to document the changes you applied to the data.

One approach to splitting a non-atomic field is to write a parser that loops through each row in the dataset, extracts the string content of the non-atomic field of interest from one row, splits the string into components (based on a separator such as a space, or loops through the string one character at a time), and examines the content of each part in an effort to decide which parts of the string map onto which concept. Pseudocode for such a parser can be expressed as:

```
run query(SELECT TaxonName
          FROM sourceTable)
for each row in result set
    whole bunch of code to
    try to identify and split
    parts of name
    run query(INSERT INTO parsedTable
             (genus, subgenus...))
    log errors in splitting row
next row
```

```
Follow this with manual examination of
the results for problems, then fix any
bugs in the parser, and then run the
parser again....  Eventually a point of
diminishing returns will be reached and
you will need to handle some rows
individually by hand.
```

Writing a parser in this form, especially for complex data such as species names can be a very complex and frustrating task. There are, however, approaches other than brute force parsing to handing non-atomic fields. For long term projects, where data from many different data sources might need to be parsed, a machine learning algorithm of some form might be appropriate. For one shot data migration problems, exploiting some of the interesting and useful tools for pattern recognition in the programmer's toolkit might be of help. There are, for example, regular expressions. Regular expressions are a tool for recognizing patterns found in an expanding number of languages (perl, PHP, and MySQL all include support for regular expressions).

An algorithm and pseudocode for a parser that exploits regular expressions to match and split known taxon name patterns might look something like the following:

```
1) Compile by hand a list of patterns
that match different forms of taxon
names.
   /^[A-Z]{1}[a-z]*$/
   maps to field:  Generic
   /^[A-Z]{1}[a-z]* [a-z]?$/
   maps to fields: Generic, Trivial
     split on " "
2) Store these patterns in a list

3) for each pattern in list
   run query
      (SELECT taxonName
       WHERE taxonName matches pattern)
   for each row in results
      split name into parts
          following known pattern
      run query
         (INSERT INTO parsedTable
           (genus, subgenus...))
   next row
next pattern

4) By hand, examine rows that didn't have
matching patterns, write new patterns and
run parser again

5) Enter last few unmatched taxon names
by hand
```

Normalization Problems: Multiple values in one field

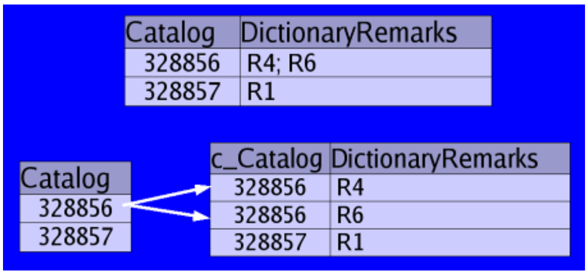
Legacy datasets very often contain fields that hold multiple repeated pieces of the same sort of information. These fields are usually remains of efforts to store one to many relationships in the flat file format allowed by early database systems. The goal in parsing these fields is to split them into their components and to place these components into separate rows in a new table, which is joined to the first in a many to one relationship.

If users have been consistent in the delimiter used to separate multiple repeated components within the field (e.g. semicolon as a separator in "R1; R2"), then writing a parser is quite straightforward. Twenty five to thirty year old data, however, can often contain inconsistencies, and you will need to carefully examine the content of these fields to make sure that the content is consistent. This is especially true if the text content is complex and some of the text could contain the character used as the delimiter.

Parsing consistent data in this form is usually a simple exercise in splitting the string on the delimiter (very easy in languages such as perl that have a split command, but requiring a little more coding in others). Pseudocode for such a parser may look something like this:

```
run query
  (SELECT Catalog, DictionaryRemarks
     FROM originalTable)
for each row in result set
    split DictionaryRemarks on the
   delimiter ";" into a list of remarks
    for each split remark in list
        run query
          (INSERT INTO parsedTable
                 (Catalog,Remark)
               VALUES ... )
    next split
    log errors found when splitting row
next row
```

The parser above could split a field into rows in a new table as shown in Figure 26.



Catalog	DictionaryRemarks
328856	R4; R6
328857	R1

Catalog
328856
328857

c_Catalog	DictionaryRemarks
328856	R4
328856	R6
328857	R1

Figure 26. Splitting a non-atomic Dictionary Remarks field containing multiple instances of DictionaryRemarks separated by a semicolon delimiter into multiple rows in a separate normalized table.

Normalization Problems: Duplicate values with misspellings

If the structures that data are stored in are not in at least third normal form, they will allow a field to contain multiple rows containing duplicate values. A locality table that contains fields for country and state will contain duplicate values for country and state for as many localities as exist in a single state. Over time, new values entered by different people, imports from external data sources, and changes to the database will result in these repeated values not quite matching each other. Pennsylvania, PA, and Penn. might all be entries in a State/Province/Primary Division field. Likewise, these repeated values can often include misspellings.

In some cases, these subtle differences and misspellings pass unnoticed, such as when a database is used primarily for printing specimen labels and retrieving information about single specimens. At other times, these not quite duplicate values create serious problems. An example of this comes from our attempt at ANSP to migrate the Ichthyology department database from Muse into BioLink. Muse uses a set of tables to house information about specimens, localities, and transactions. It includes a locality table in second normal form (the key field for locality number contains unique values) that has fields for country, for state or province, and for named place (which typically contains duplicate values). In the ANSP fish data, there were several different spellings of Pennsylvania and Philadelphia, with at least 12 different combinations of variants of United States, Pennsylvania, and Philadelphia. Since BioLink uses a single geographic hierarchy and uses a graphical tree browser to navigate through this hierarchy, it was not possible to import the ANSP fish data into BioLink and simply use it. Extensive cleanup of the data would have been required before the data would have been usable within BioLink.

Cleanup of misspellings, differences in punctuation, and other such near duplicate values is simple but time consuming for large data sets. The key components are: first, maintain a copy of the existing legacy data; second, select a set of distinct values for a field or several fields into a separate table; third, map each distinct value onto the correct value to use in its place; and fourth, build a new table containing the corrected values and links to the original literal values.

```
SELECT DISTICT country, primary_division
FROM locality
ORDER BY country, primary_division;
```

or, to select and place in a new table:

```
CREATE TABLE country_primary_map_table
  SELECT DISTICT
    country, primary_division,
    “” as map_country,
    “” as map_primary
  FROM locality
  ORDER BY country, primary_division;
```

Data in the mapping table produced by the query above might look like those in Table 29. Country and primary division contain original data, map_country and map_primary_division need to be filled in.

Table 29. A table for mapping geographic names found in legacy data to correct values.

country	primary _division	map_ country	map_primary _division
USA	PA		
USA	Pennsylvania		
US	Pennsylvana		

The process of filling in the mapping values in this table can involve both queries and direct inspection of each row by someone. This examination will probably involve one person to examine each row, and others to help resolve problematic rows. It may also involve queries to make changes to lots of rows at once, especially in cases where two fields (such as country and primary division) are being examined at once. An example of a query to set the map values for multiple different variants of a country name is shown below. Changes to the country mapping field could also be applied by selecting the distinct set of values for country alone, compiling mappings for them, and using an intermediate table to compile a list of unique primary divisions and countries. Indeed, you will probably want to work iteratively down the fields of a hierarchy.

```
UPDATE country_primary_map_table
  SET map_country = “United States”
  WHERE country = “USA” or country = “US”
     or country = “United States”
     or country =
            “United States of America”;
```

Ultimately, each row in the mapping table will need to be examined by a person and the mapping values will need to be filled in, as in Table 30.

Table 30. A table for mapping geographic names found in legacy data to correct values.

country	primary_division	Map_country	map_primary_division
USA	PA	United States	Pennsylvania
USA	Pennsylvania	United States	Pennsylvania
US	Pennsylvana	United States	Pennsylvania

Planning for future migrations

The database life cycle is a clear reminder that data in your database will need to be migrated to a new database some time in the future. Planning ahead in database design can help ease these future migrations. The two most important aspects of database design that will aid in future migration are good relational design and clear documentation of the database. Clear documentation of the database is unambiguously valuable. In ten years there will be questions about the nature of the data in a database, and the people who created the database will either be unavailable or will simply not remember all the details. Good relational design has tradeoffs.

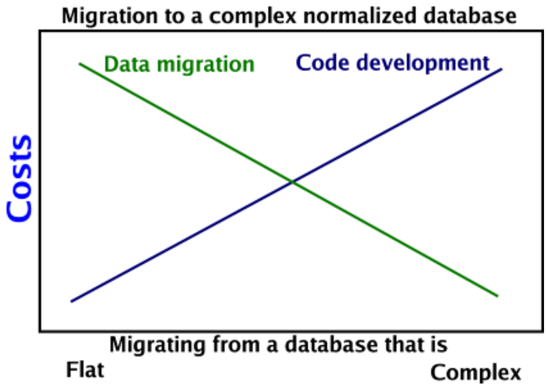


Figure 27. Migration costs from databases of various complexity into a complex normalized database. Migration into a flat or simple database will have relatively low costs, but will have the risk of substantial quality loss in the data over the long term as data are added and edited.

More complex database designs require more complex user interface code than simple database designs. Migrating a simple database (with all its contained garbage) into a new simple database is quite straightforward. Migrating a complex database will require some substantive work to replace the user interface (and any code embedded in the backend) with a working user interface in the new system. Migrating a simple database to a new complex database will require substantial time cleaning up problems in the data as well as writing code for a new user interface. This tradeoff is illustrated in Figure 27.

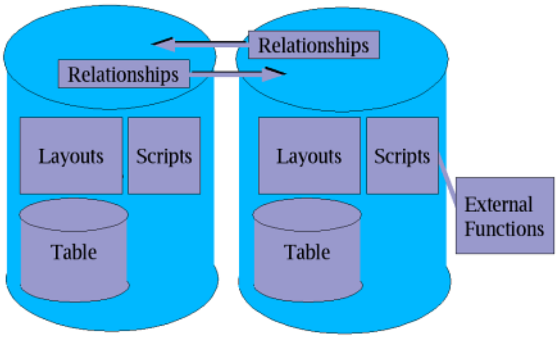


Figure 28. Architecture of a monolithic database management system.

The architecture of your code can affect the ease of migration. Some database management systems are monolithic (Figure 28). Data, code, and user interface are all integral parts of the same database. Other database systems can be decomposed into distinct layers – the user interface can be separated from the data, and the data can easily be accessed through other means.

A monolithic database system offers very limited options for either code reuse or migration. Migration requires either upgrading to a newer version of the same DBMS (and then resolving any bugs introduced into the code by changes made by the DBMS vendor), or exporting the data and importing it into an entirely new database with an entirely new user interface. In contrast, a layered architecture can allow the DBMS used to store the data from user interface components, meaning that it may be possible to upgrade a DBMS or migrate to a new DBMS without having to rewrite the user interface from scratch, or indeed without making major changes to the user interface.

Many relational database management systems are written to allow easy separation of the code into layers (Figure 29).

A database server, responsible for maintaining the integrity of the data, can be

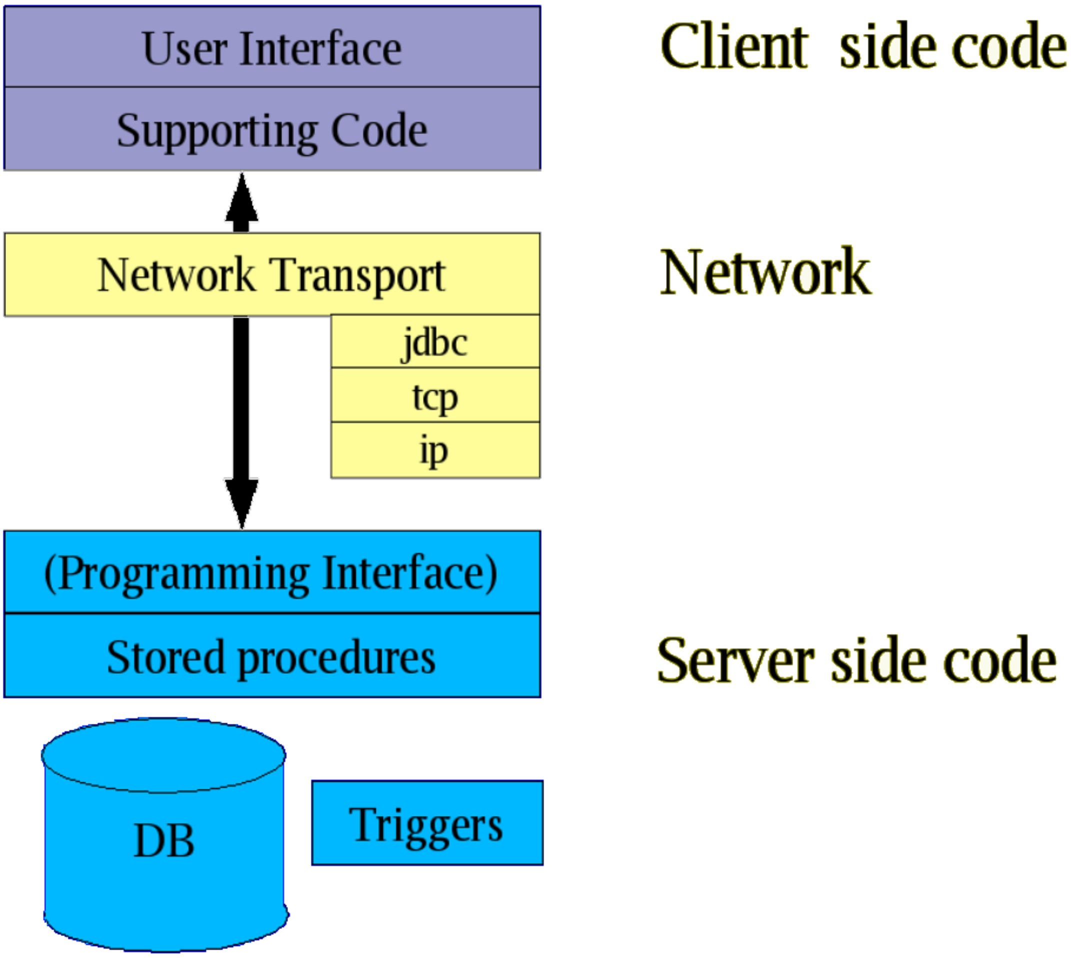
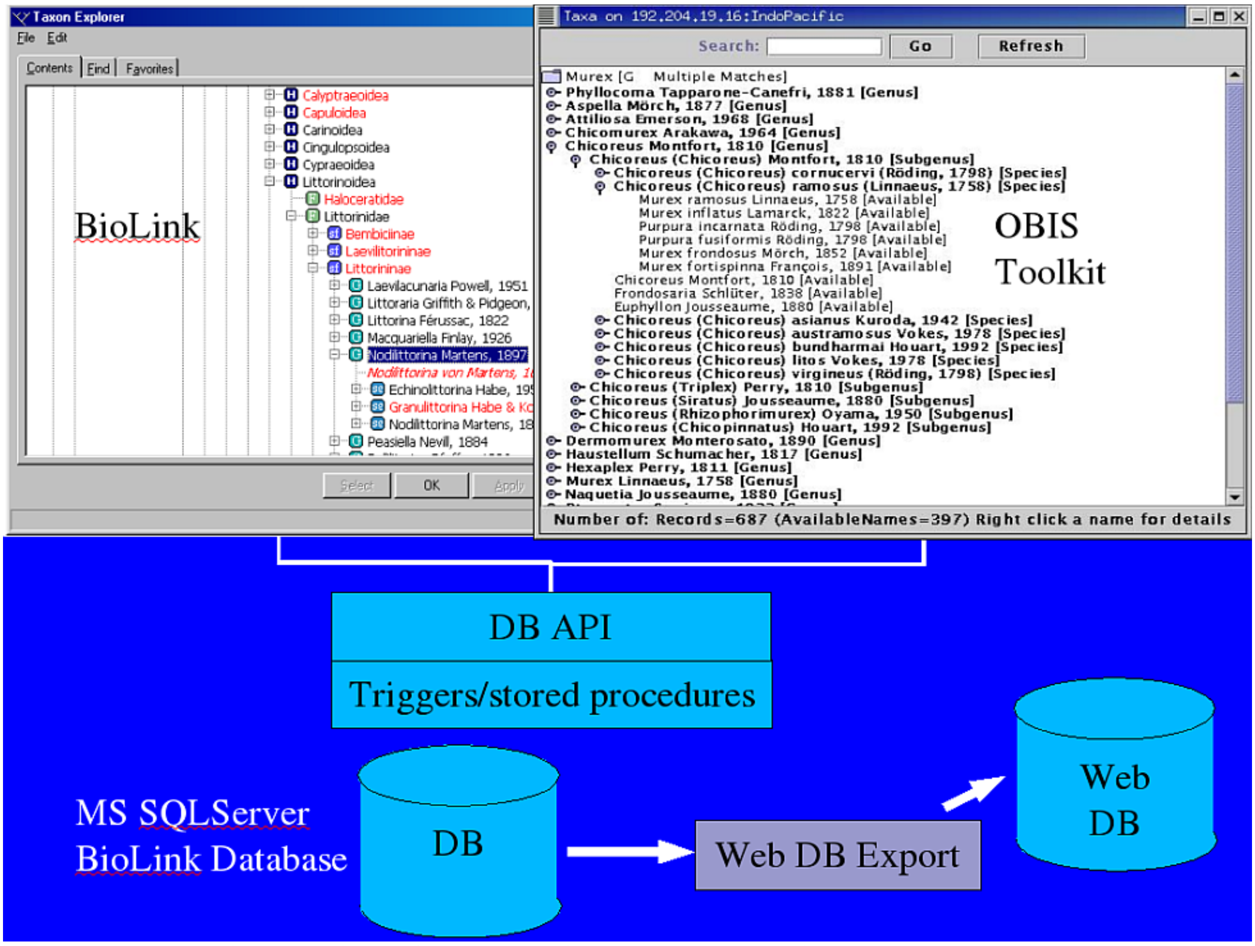


Figure 29. Layers in a database system. A database with server side code can focus on maintaining the integrity of the data, while the user interface can be separated and connect either locally to the server or remotely over a network (using various network transport layers). A stable and well defined application programming interface can allow multiple clients of different sorts to connect to the back end of the database and allow both client and server code to be altered independently.

separated from the front end clients that access the data in that server. The server stores the data in a central location, while clients access it (from the local machine or over a network) to view, add, and alter data. The server handles all of the storage of the data, clients display the data through a user interface. Multiple users spread out over the world can work on the same database at the same time, even altering and viewing the same records. Different clients can connect to the same database. Clients may be a single platform application, a cross platform application, or multiple different applications (such as a data entry client and a web database server update script). A client may be a program that a user installs on their workstation that contains network transport and business rules for data entry built into a fancy graphical user interface. A client could be a web server application that allows remote users to query the database and view result sets through their web browsers. A design that separates a database system into layers has advantages over a monolithic database. Layers can help in multi-platform support (clients can be written in a cross platform language or compiled in different versions for different operating systems), allow scaling and distribution of clients [Figure 30], and aid in migration (allowing independent migration of backend and user interface components through a common Application Programing Interface).

Although object oriented programing and object thinking is a very natural match for the complex hierarchies of biodiversity data, I have not discussed object oriented programing here. One aspect of object oriented thinking, however, can be of substantial help in informing design decisions about where in a complex database system business logic code should go. This aspect is encapsulation. Encapsulation allows an object to hide (that, is encapsulate) all of its internal storage structures and operation from the world. Encapsulation limits the abilities of the world to interactions with the information stored in an instance of an object only through a narrow window of methods. In object oriented programming, a code object that represents real world collections objects might store a catalog number somewhere inside it and allow other objects to find or change that catalog number only through invocations of methods of the object that allow the collection_object code object to apply business rules to the request and only fulfill it if it meets the requirements of those rules. Direct reading and writing to the catalog number would not be possible. The catalog number for a particular instance of a collection_object might be obtained with a call to Collection_object.get_catalog(). Conversely the catalog number for a particular instance of a a collection_ojbect might be set with a call to Collection_object.set_catalog("452685"). The set_catalog() method is a block of code that can test to see if the catalog number it has been provided is in a valid format, if the number provided is in use elsewhere, if the number provided falls within a valid range, or any other business rules that apply to the setting of the catalog number of a collection object. More complex logic could be embedded in methods such as collection_object.loan(a_loan).

We can think about the control of information going into a database as following a spectrum from uncontrolled to highly encapsulated. At the uncontrolled end of the spectrum, if raw SQL commands can be presented to a database, a



```
Calyptraeoidea
Capuloidea
Carinoidea
Cingulopsoidea
Cypraeoidea
Littorinoidea
  Haloceratidae
  Littorinidae
    Bembiciinae
    Laevilitorininae
    Littorininae
      Laevilacunaria Powell, 1951
      Littoraria Griffith & Pidgeon,
      Littorina Férussac, 1822
      Macquariella Finlay, 1926
      Nodilittorina Martens, 1897
        Nodilittorina von Martens, 1
        Echinolittorina Habe, 19
        Granulittorina Habe & Ko
        Nodilittorina Martens, 18
      Peasiella Nevill, 1884
```

```
Murex [G   Multiple Matches]
Phyllocoma Tapparone-Canefri, 1881 [Genus]
Aspella Mörch, 1877 [Genus]
Attiliosa Emerson, 1968 [Genus]
Chicomurex Arakawa, 1964 [Genus]
Chicoreus Montfort, 1810 [Genus]
  Chicoreus (Chicoreus) Montfort, 1810 [Subgenus]
    Chicoreus (Chicoreus) cornucervi (Röding, 1798) [Species]
    Chicoreus (Chicoreus) ramosus (Linnaeus, 1758) [Species]
      Murex ramosus Linnaeus, 1758 [Available]
      Murex inflatus Lamarck, 1822 [Available]
      Purpura incarnata Röding, 1798 [Available]
      Purpura fusiformis Röding, 1798 [Available]
      Murex frondosus Mörch, 1852 [Available]
      Murex fortispinna François, 1891 [Available]
    Chicoreus Montfort, 1810 [Available]
    Frondosaria Schlüter, 1838 [Available]
    Euphyllon Jousseaume, 1880 [Available]
    Chicoreus (Chicoreus) asianus Kuroda, 1942 [Species]
    Chicoreus (Chicoreus) austramosus Vokes, 1978 [Species]
    Chicoreus (Chicoreus) bundharmai Houart, 1992 [Species]
    Chicoreus (Chicoreus) litos Vokes, 1978 [Species]
    Chicoreus (Chicoreus) virgineus (Röding, 1798) [Species]
  Chicoreus (Triplex) Perry, 1810 [Subgenus]
  Chicoreus (Siratus) Jousseaume, 1880 [Subgenus]
  Chicoreus (Rhizophorimurex) Oyama, 1950 [Subgenus]
  Chicoreus (Chicopinnatus) Houart, 1992 [Subgenus]
Dermomurex Monterosato, 1890 [Genus]
Haustellum Schumacher, 1817 [Genus]
Hexaplex Perry, 1811 [Genus]
Murex Linnaeus, 1758 [Genus]
Naquetia Jousseaume, 1880 [Genus]
Number of: Records=687 (AvailableNames=397) Right click a name for details
```

Figure 30. BioLink (Windows) and OBIS Toolkit (Java, cross platform) interfaces to a BioLink database on a MS SQLServer, with a web copy of the database. The separation of user interface and database layers allowed the OBIS Indo-Pacific Mollusc project to use both BioLink's Windows client and a custom cross-platform Java application for global (US, France, Australia) distributed data entry.

collection_object table in a database could have its values changed with a command such as `UPDATE collection_object SET catalog_number = “5435416”;` a command that will probably have undesirable results if more than one record is present. At the other end of the spectrum, users might be locked out from direct access to the database tables and required to interact through stored procedures such as sp_collection_object_set_catalog(id,new_catalog) which might be invoked somewhere in the user interface code as sp_collection_object_set_catalog(“64235003”,”42005”). At an intermediate level, direct SQL query access to the tables is allowed, but on insert and on update triggers on the collection_object table apply business rules to queries that attempt to alter catalog numbers of collections objects. Encapsulating the backend of a database and forcing clients to communicate with it through a fixed Application Programming Interface of stored procedures allows development and migration of backend database and clients to follow independent paths.

Conclusion

Biological information is inherently complex. Management and long term stewardship of information related to biological diversity is a challenging task, indeed at times a daunting task. Stewardship of biodiversity information relies on good database design. The most important principles of good database design are to atomize information, to reduce redundant information, and to design for the task at hand. To atomize information, design tables so that each field contains only one concept. To reduce redundant information, design tables so that each row of each field contains a unique value. Management of database systems requires good day to day system administration. Database system administration needs to be informed by a threat analysis, and should employ means of threat mitigation, such as regular backups, highlighted by that analysis. Stewardship of biodiversity information also requires the long term perspective of the database life cycle. Careful database design and documentation of that design are important not only in maintaining data integrity during use of a database, but are also important factors in the ease and extent of data loss in future migrations (including reduction of the risk that inferences made about the data now will be taken at some future point to be original facts). Good database design is a necessary foundation, but the most important factor in maintaining the quality of data in a biodiversity database is a quality control process that involves people with a strong stake in the integrity of the data.

Acknowledgments

This document had its origins in a presentation I gave at the Use of Digital Technology in Museums workshop at SPNHC in 2003. I would like to thank Tim White for encouraging me to talk about the fundamentals of database design using examples from the domain of natural history collections information in that workshop. This paper has been substantially improved by comments from and discussions with Susan F. Morris, Kathleen Sprouffske, James Macklin, and an anonymous reviewer. Gary Rosenberg has provided many stimulating discussions over many years on the management and analysis of collections and biodiversity data. I would also like to thank the students in the Biodiversity Informatics Seminar for the REU program at ANSP in the summer of 2004 for many fruitful and pertinent discussions. The PH core tables example owes much to extensive discussions, design work and coding with James Macklin. A large and generous community of open source programmers have produced a wide variety of powerful tools for working with data including the database software MySQL, and PostgreSQL, and the CASE tool Druid.

References

Anley, C. 2002. Advanced SQL Injection in SQL Server Applications. NGSSoftware Insight Security Research [WWW PDF Document] URL http://www.nextgenss.com/papers/advanced_sql_injections.pdf

ASC [Association of Systematics Collections] 1992. An information model for biological collections. Report of the Biological Collections Data Standards Workshop, 8-24 August 1992. ASC. Washington DC. [Text available at: WWW URL http://palimpsest.stanford.edu/lex/datamodl.html]

ANSI X3.135-1986. Database Language SQL.

Beatte, S., S. Arnold, C. Cowan, P. Wagle, C. White, and A. Shostack 2002. Timing the Application of Security Patches for Optimal Uptime. *LISA XVI Proceedings* p.101-110. [WWW Document] URL http://www.usenix.org/events/lisa02/tech/full_papers/beattie/beattie_html/

Beccaloni, G.W., M.J. Scoble, G.S. Robinson, A.C. Downton, and S.M. Lucas 2003. Computerizing Unit-Level Data in Natural History Card Archives. pp. 165-176 In M.J. Scoble ed. ENHSIN: The European Natural History Specimen Information Network. The Natural History Museum, London.

Bechtel, K. 2003. Anti-Virus Defence In Depth. *InFocus* **1687**:1 [WWW document] URL http://www.securityfocus.com/infocus/1687

Berendsohn, W.G., A. Anagnostopoulos, G. Hagedorn, J. Jakupovic, P.L. Nimis, B. Valdés, A. Güntsch, R.J. Pankhurst, and R.J. White 1999. A comprehensive reference model for biological collections and surveys. *Taxon* **48**: 511-562. [Available at: WWW URL http://www.bgbm.org/biodivinf/docs/CollectionModel/]

Blum, S. D. 1996a. The MVZ Collections Information Model. Conceptual Model. University of California at Berkeley, Museum of Vertebrate Zoology. [WWW PDF Document] URL http://www.mip.berkeley.edu/mvz/cis/mvzmodel.pdf and http://www.mip.berkeley.edu/mvz/cis/ORMfigs.pdf

Blum, S. D. 1996b. The MVZ Collections Information Model. Logical Model. University of California at Berkeley, Museum of Vertebrate Zoology. [WWW PDF Document] URL http://www.mip.berkeley.edu/mvz/cis/logical.pdf

Blum, S.D., and J. Wieczorek 2004. DiGIR-bound XML Schema proposal for Darwin Core Version 2 content model. [WWW XML Schema] URL http://www.digir.net/schema/conceptual/darwin/core/2.0/darwincoreWithDiGIRv1.3.xsd

Bruce, T.A. 1992. Designing quality databases with IDEF1X information models. Dorset House, New York. 547pp.

Carboni, A. et al. 2004. Druid, the database manager. [Computer software package distributed over the Internet] http://sourceforge.net/projects/druid [version 3.5, 2004 July 4]

Celko, J. 1995a. SQL for smarties. Morgan Kaufmann, San Fransisco.

Celko, J. 1995b. Instant SQL Programing. WROX, Birmingham UK.

Chen, P.P-S. 1976. The Entity Relationship Model – Towards a unified view of data. *ACM Transactions on Database Systems*. 1(1):9-36.

Codd, E.F. 1970. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. 13(6):377-387

Connoly, T., C. Begg, and A. Strachan 1996. Database Systems: A practical approach to design, implementation and management. Addison Wesley, Harrow, UK.

CSIRO 2001. BioLink 1.0 [Computer software package distributed on CD ROM] CSIRO Publishing. [Version 2.0 distributed in 2003].

DuBois, P. 2003 [2nd ed.]. MySQL. Sams Publishing, Indiana.

DuBois, P. et al., 2004. Reference Manual for the 'MySQL Database System' [4.0.18] [info doc file, available from http://www.mysql.com] MySQL AB.

Eisenberg, A., J. Melton, K. Kulkarni, J-E Michels, and F. Zemke 2003. SQL:2003 Has been Published. *Sigmod Record* 33(1):119-126.

Elmasri, R. and S.B. Navathe 1994. Fundamentals of Database Systems. Benjamin Cummings, Redwood City, CA.

Ferguson, N. and B. Scheiner 2003. Practical Cryptography. John Wiley and Sons.

Hernandez, M.J. 2003. Database Design for Mere Mortals. Addison Wesley, Boston.

Hoglund, G. and G. McGraw 2004. Exploiting Software: How to Break Code. Addison Wesley, Boston. 512pp.

ISO/IEC 9075:2003. Information technology -- Database languages -- SQL --.

Jeong, S.S. and R. Chen 2001. Functional misassignment of genes. *Nature Biotechnology*. **19**:95.

Linsenbardt, M.A., and M.S. Stigler 1999. SQL SERVER 7 Administration. Osborne/McGraw Hill, Berkeley CA. 680pp.

National Research Council, Committee on the Preservation of Geoscience Data and Collections 2002. Geoscience Data and Collections: National resources in peril. The National Academies Press, Washington DC. 108pp.

Morris, P.J. 2000. A Data Model for Invertebrate Paleontological Collections Information. *Paleontological Society Special Publications* **10**:105-108,155-260.

Morris, P.J. 2001. Posting to taxacom@usobi.org: Security Advisory for BioLink users. Date: 2001-03-02 15:36:40 -0500 [Archived at: http://listserv.nhm.ku.edu/cgi-bin/wa.exe?A2=ind0103&L=taxacom&D=1&O=D&F=&S=&P=1910]

Pennisi, E. 1999. Keeping Genome Databases Clean and Up to Date. *Science* **286**: 447-450

Petuch, E.J. 1989. New species of *Malea* (Gastropoda Tonnidae) from the Pleistocene of Southern Florida. *The Nautilus* **103**:92-95.

PostgreSQL Global Development Group 2003. PostgreSQL 7.4.2 Documentation: 22.2. File system level backup [WWW Document] URL http://www.postgresql.org/docs/7.4/static/backup-file.html

Pyle, R.L. 2004. Taxonomer: A relational data model for handling information related to taxonomic research. *Phyloinformatics* **1**:1-54

Resolution Ltd. 1998. xCase Professional. [Computer software package distributed on CD ROM] Jerusalem, RESolution Ltd. [Version 4.0 distributed in 1998]

Schwartz, P.J. 2003. XML Schema describing the Darwin Core V2 [WWW XML Schema] URL http://digir.net/schema/conceptual/darwin/2003/1.0/darwin2.xsd

Smith, M. 2002. SQL Injection: Are your web applications vulnerable? SpiLabs SPIDynamics, Atlanta. [WWW PDF Document] URL http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf

Teorey, T.J. 1994. Database Modeling & Design. Morgan Kaufmann, San Fransisco.

Wojcik, M. 2004. Posting to bugtraq@securityfocus.com: RE: Suggestion: erase data posted to the Web Date: 2004-07-08 07:59:09 -0700. [Archived at: http://www.securityfocus.com/archive/1/368351]

Glossary (of terms as used herein).

Accession number: number or alphanumeric identifier assigned to a set of collection objects that enter the care of an institution together. Used to track ownership and donor of material held by an institution. In some disciplines and collections, accession number is used to mean the identifier of a single collection object (catalog number is used for this concept here).

Catalog number: number or alphanumeric identifier assigned to a single collection object to identify that particular collection object within a collection. Widely referred to as an accession number in some disciplines.

Collection object: generic term to refer to specimens of all sorts found in collections, refers to the standard single unit handled within a collection. A collection object can be a single specimen, a lot of several specimens all sharing the same data, or a part of a specimen of a particular preparation type. Example collection objects are a mammal skin, a tray of mollusk shells, an alcohol lot of fish, a mammal skull, a bulk sample of fossils, an alcohol lot of insects from a light trap, a fossil slab, or a slide containing a gastropod radula. Collection objects can form a nested hierarchy. An alcohol lot of insects from a light trap could have a single insect removed, cataloged separately, and then part of a wing could be removed from that specimen, prepared as an SEM stub, and used to produce a published illustration. Each of these objects (including derived objects such as the SEM negative) can be treated as a collection object. Likewise a microscope slide containing many diatoms can be a collection object that contains other collection objects in the form of identified diatoms at particular x-y coordinates on the slide.

DBMS: Database Management System. Used here to refer to the software responsible for storage and retrieval of the data. Examples include MS Access, MySQL, Postgresql, MS SQLServer, and Filemaker. A database would typically be created using the DBMS and then have a front end written in a development environment provided by the DBMS (as in MS Access or Filemaker), or written as a separate front end in a separate programming language.

Incident Response Capability: A plan for the response to computer security incidents usually involving an internal incident response team with preplanned contacts to law enforcement and external consulting sources to be activated in response to various computer incidents.

Specific epithet: The species word in a binomial species name or polynomial subspecific name. For example, palmarosae in *Murex palmarosae* and nodocarinatus in *Hesperiturris nodocarinatus crassus* are specific epithets.

Subspecific epithet: The subspecies word in the polynomial name of a subspecies or other polynomial. For example, crassus in *Hesperiturris nodocarinatus crassus* is a subspecific epithet.

Trivial epithet: The lowest rank word in a binomial species name or a polynomial subspecific name. The specific epithet palmarosae in the binomial species name *Murex palmarosae*, or the subspecific epithet crassus in the trinomial subspecies name *Hesperiturris nodocarinatus crassus* are trivial epithets.

Appendix A: An example of Entity Documentation

Table: Herbarium Sheet

Definition: A Herbarium Sheet. Normally an approximately 11 X 17 inch piece of paper with one or more plants or parts of plants and labels attached.
Comment: Core table of the Herbarium types database. Some material in herbarium collection is stored in other physical forms, e.g. envelopes, but concept of herbarium sheet with specimens that are annotated maps easily to these other forms.
See Also: Specimens, ExHerbariumSpecimenAssociation, Images

Field Summary

Field	Sql type	PrimaryKey	NotNull	Default	AutoIncrement
Herb Sheet ID	Integer	X	X		X
Name	Varchar(32)	-	X	[Current User]	-
Date	Timestamp	-	-	Now	-
Verified by	Varchar(32)	-	-	Null	-
Verification Date	Date	-	-	Null	-
CaptureNotes	Text	-	-		-
VerificationNotes	Text	-	-		-
CurrentCollection	Varchar(255)	-	-		-
Verified	Boolean	-	X	False	-

Fields

Name: Herb Sheet ID
Type: Integer
Definition: Surrogate numeric primary key for herbarium sheet.
Domain: Numeric Key
Business rules: Required, Automatic.
Example Value: 528

Name: Name
Type: Varchar(32)
Definition: Name of the person who captured the data on the herbarium sheet.
Domain: Alphabetic, Names of people in Users:Full Name
Business rules: Required, Automatic, Fill from authority list in Users:Full Name using current login to determine identity of current user when a record is created.

Name: Date
Type: Timestamp
Definition: Timestamp recording the date and time the herbarium sheet data were captured.
Domain: Timestamp
Business Rules: Required, Automatic. Auto generate when herbarium sheet record is created, i.e. default to NOW().
Example Value: 20041005:17:43:35UTC
Comment: Timestamp format is dbms dependent, and its display may be system dependent.

Name: Verified by
Type: text
Definition: Name of the person who verified the herbarium sheet data.
Domain: Alphabetic. Names of people from Users: Full Name
Business Rules: Optional. Automatic. Default to Null. On verification of record, fill with value from Users: Full Name using current login to identify the current user. Current user

must have verification rights in order to verify a herbarium sheet and to enter a value in this field.
Example Value: Macklin, James

Name: Verification Date
Type: Date
Definition: Date the herbarium sheet data were verified. Should be a valid date since the inception of the database (2003 and later).
Domain: Date. Valid date greater than the inception date of the database (2003).
Business Rules: Optional. Automatic. Default to Null. On verification of record, fill with current date. Current user must have verification rights in order to verify a herbarium sheet and to enter a value in this field. Value must be more recent than value found in Herbarium Sheet: Date (records can only be verified after they are created).
Example Value: 2004-08-12

Name: CaptureNotes
Type: Text
Definition: Notes concerning the capture of the data on the herbarium sheet made by the person capturing the data. May be questions about difficult to read text, or other reasons why the data related to this sheet should be examined by someone more experienced than the data entry person.
Domain: Free text memo.
Business rules: Manual, Optional.
Example Values: "Unable to read taxon name Q_____ ___ba", "Locality description hard to read, please check".

Name: VerificationNotes
Type: Text
Definition: Notes made by the verifier of the specimen. May be working notes by verifier prior to verification of record.
Domain: Free Text Memo.
Business Rules: Manual, Optional. May contain a value even if Verified By and Verification Date are null.
Example Values: “Taxon name is illegible”, “Fixed locality description”

Name: CurrentCollection
Type: Varchar(255)
Definition: Collection in which herbarium sheet is currently stored. Data entry is currently for the type collection, with some records being added for material in the general systematic collection that are imaged to provide virtual loans.
Domain: "PH systematic collection", "PH type collection".
Business Rules: Required, Semimanual, default to "PH type collection". Should data entry expand to routine capture of data from systematic collection, change to Manual and allow users to set their own current default value.

Name: Verified
Type: Boolean
Definition: Flag to indicate if data associated with a herbarium sheet has been verified.
Domain: True, False.
Business rules: Required, Semiautomatic. Default to False. Set to true when record is verified. Set the three fields Herbarium Sheet: Verified, Herbarium Sheet: Verified by, and Herbarium Sheet: Verification sheet together. Only a user with rights to verify material can change the value of this field. Once a record has been verified once maintain these three verification stamps and do not allow subsequent re-verifications to alter this information.