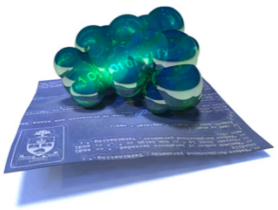
BCH441 – Bioinformatics

Data



Boris Steipe

Department of Biochemistry – Department of Molecular Genetics
University of Toronto

A systems definition

A system is a collection of collaborating genes that have more significant relationships among each other than they have with genes that are not system members.

What data will we use to define and discover biological systems?

How will we store this data, and work with it?

In the sense of the definition above, a system is both a generalization of one gene's "function" and a recipe for including and excluding components.

Data management is the fundamental task of bioinformatics.

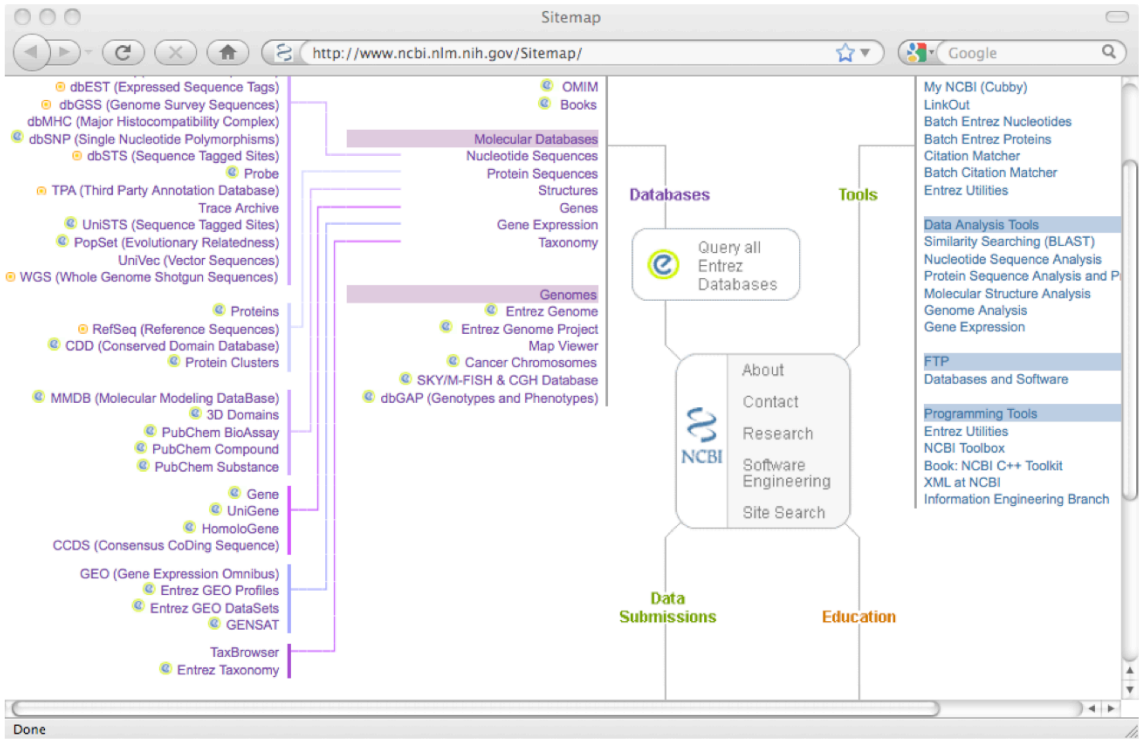
"Bioinformatics"

Starting from a biological motivation to annotate and discover systems as sets of collaborating genes, we return to *data management* as something we need to get under control for our purposes.

Don't underestimate the difficulties in getting this right. There is a lot of data, and the challenge is to figure out clearly where one wants to go with it, i.e. for what purpose the data is being collected and stored. Important examples of common data resources include ...

NCBI

http://www.ncbi.nlm.nih.gov



... the US **NCBI** (National Center for Biotechnology Information), one of the world's major centres for molecular data, especially sequence and genome data, and you can appreciate immediately that it may require some effort to figure out what to do with this.

The NCBIs databases contain a very large multitude of information items, each internally held consistent and cross-referenced to other databases, and very sophisticated tools to maintain the system and discover and retrieve data.

National Center for Biotechnology Information

National Library of Medicine

National Institutes of Health

(http://www.ncbi.nlm.nih.gov)

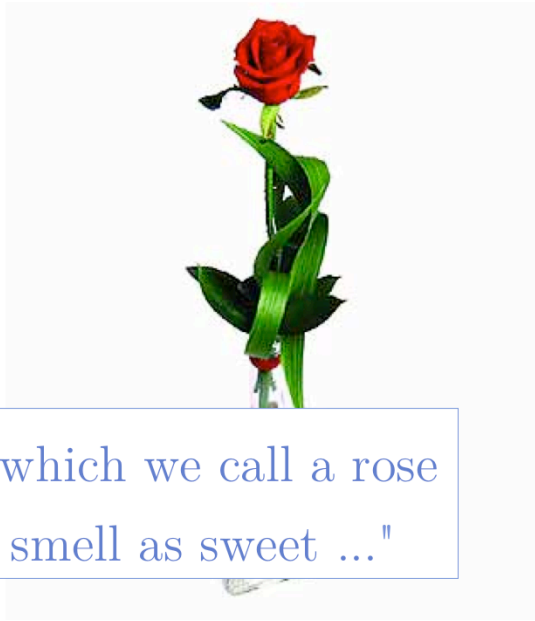
OUTLINE

Abstraction

Database systems

Data models

ABSTRACTION



"What's in a name? That which we call a rose
By any other name would smell as sweet ..."

In Shakespeare's classic tragedy of romantic love and family allegiance, Juliet encapsulates the play's central struggle in this phrase by claiming that Romeo's family name is an artificial and meaningless convention. Just like in the world of the sequence abstraction, this is only partially true: the problems are not just based in the fact that Romeo is **called** a Montague, but that he **is** in fact a member of hisfamily. Even if a *Thing* does not change when its abstract label changes, such labels rarely exist in isolation: other *Things* might be referred to by the same label and changing one changes the composition of the entire set. (Or, to remain with our example, as soon as Romeo renounces his name and thus his family, the family would be no longer the same.) Even worse - and this is something we encounter every day in bioinformatics - if identifiers are not stable over time, cross-references to that identifier fail. If you decide you'll call a rose a skunk, people would become very confused.

http://en.wikipedia.org/wiki/Romeo_and_Juliet

ABSTRACTION

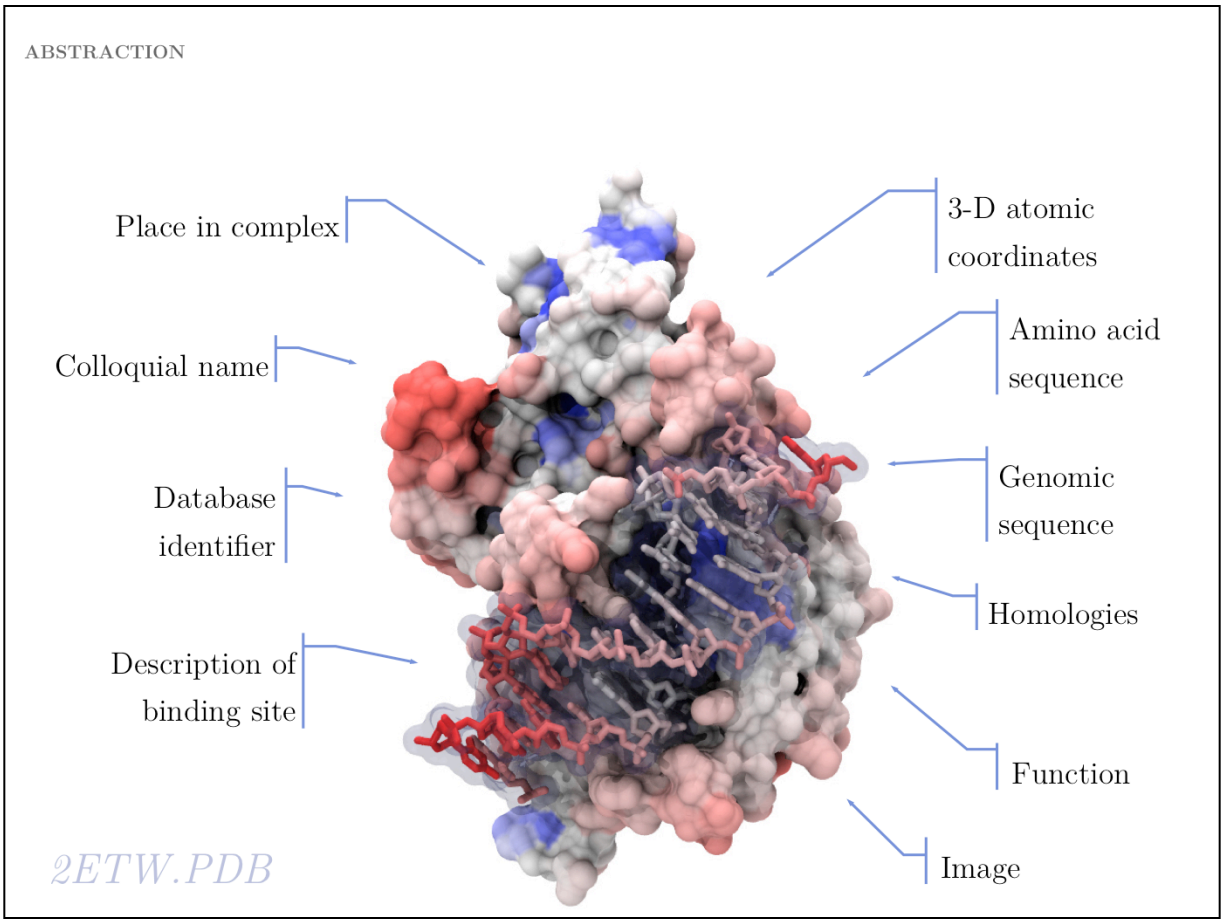
Bioinformatics models biology such that we can compute with its representations.

To compute with such models,

- Representations of biology as data ...
- Semantics of data ...
- Operations with data entities ...
- Metrics of operations ...

... need to be rigorously defined

In order to make biology computable, we have to rigorously define our system of objects and their relationships. This is useful even beyond the requirements of bioinformatics. It is an exercise in clarifying the conceptual foundations of biology itself. In many instances, definitions in current, common use are deficient, either because our current state of knowledge has gone beyond the original ideas we were trying to subsume with a term (e.g. *gene*, or *pathway*), or because an inconsistent formal and colloquial meaning of terms leads to ambiguities (e.g. *function*), or because the technical meaning of terms is poorly understood and generally misused by many (e.g. *homology*).



This image **represents** a particular biomolecule, it was derived from the coordinates of the complex of a yeast sporulation transcription factor bound to its cognate DNA sequence. **What is the best abstraction ?**

Asking about *the best* is an ill-posed question if the *purpose* is not specified. There are many possible abstractions, each serving different purposes. Even though abstractions help us model nature by focussing on particular aspects, we must be aware that real molecules have many more properties and features than any single abstraction could capture. Working with abstractions implies we are no longer manipulating the *biological entity*, but its representation. This distinction becomes crucial, when we start computing with representations to infer facts about the original entities. Inferences must be related back to biology! Common problems include (*a*) that the abstraction may not be rich enough to capture the property we are investigating (e.g. one-letter sequence codes cannot represent amino acid modifications or sequence numbers), or (*b*) that the abstraction may be ambiguous (e.g. one protein may have more than one homologue in a related organism, thus the relationship between gene IDs may be ambiguous) or (*c*) that the abstraction may not be unique (e.g. one protein may have more than one function, the same protein name may refer to unrelated proteins in different species).

That said, since the properties of biomolecules derive from their molecular structure, and structure is determined by the molecule's components, **sequence** is the most general abstraction of a gene or protein.

ABSTRACTION

Some examples of abstraction:

... representation of a molecular property
e.g. nucleotide - or amino acid sequence,
3-D coordinates

... description of a function or role
e.g. transcription factor,
checkpoint control element

... abstract label
e.g. gene name, protein name

Working with abstractions implies we are no longer manipulating the **biological entity**, but its representation.

This distinction becomes crucial, when we start computing with representations to infer facts about the original entities. Inferences must be related back to biology! Common problems include

(a) that the abstraction may not be rich enough to capture the property we are investigating (e.g. one-letter sequence codes cannot represent amino acid modifications or sequence numbers);

(b) that the abstraction may be ambiguous (e.g. one protein may have more than one homologue in a related organism, thus the relationship between gene IDs is ambiguous); or

(c) that the abstraction may not be unique (e.g. one protein may have more than one function, the same protein name may refer to unrelated proteins in different species).

COMMON ABSTRACTIONS

Biological Entity	Abstraction	Theoretical domain	Database
Polymer	20 letter amino acid code	String processing	Genbank, GenPept, RefSeq
Molecular conformation	XYZ coordinates, matrices	Floating point processing, linear algebra	PDB
Molecular Interactions	Node-Edge Graphs	Networks, Graph theory	STRING, IntAct
Function	Ontology, DAG	Networks, Graph theory	Gene Ontology
Taxonomy	Hierarchy	Database methods, Graph Theory	NCBI/EBI/DDBJ Taxon
Evolutionary relationship	Tree	Graph Theory, combinatorics	TreeBase

A selection of commonly used abstractions, the domain of computer science they relate to, and common databases that store them.

OUTLINE

Abstraction

Database systems

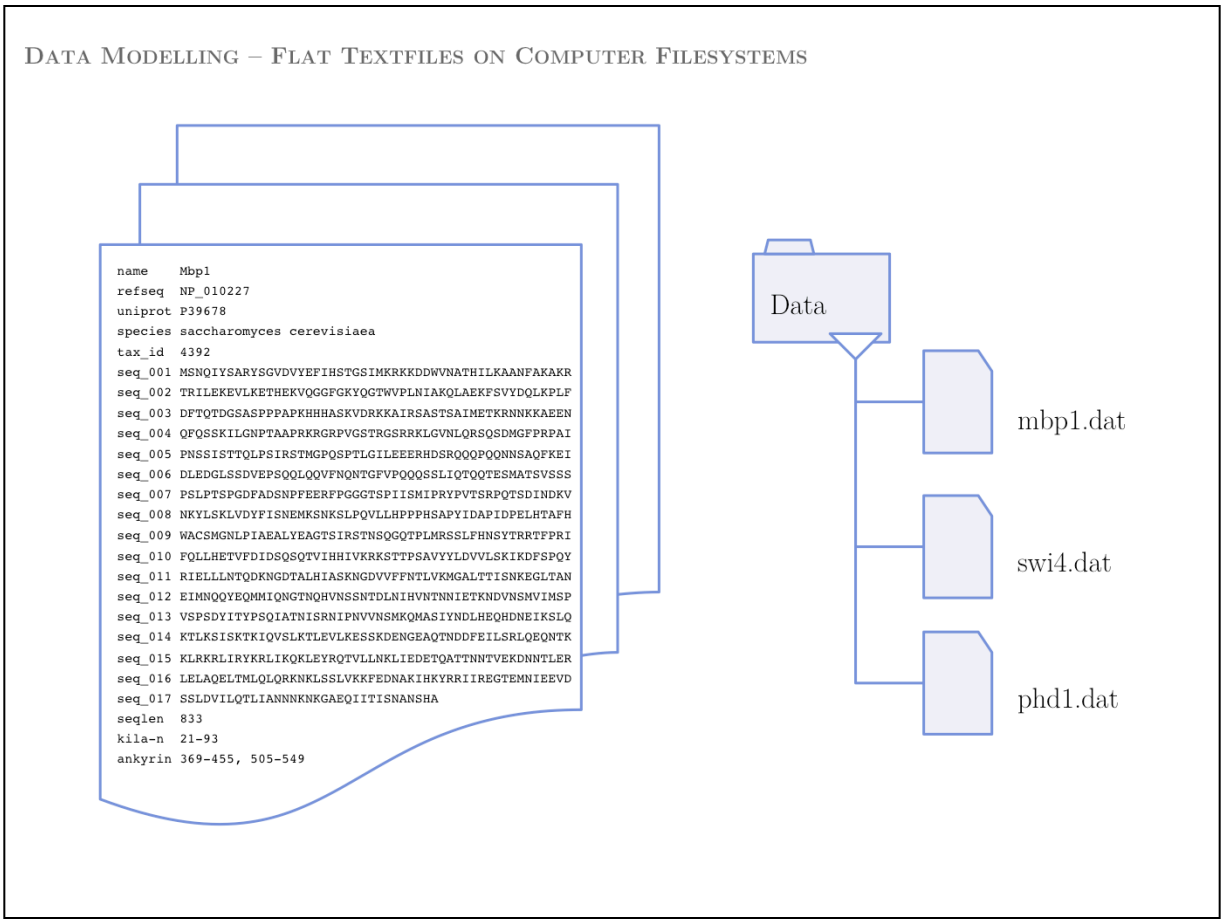
Data models

Data Modelling – Mbp1 Information

Name	`Mbp1`
RefSeq ID	`NP_010227`
UniProt ID	`P39678`
Species	`Saccharomyces cerevisiae`
Tax ID	`4932`
Sequence	`msnqiy ... nansha`
Length	`833`
KilA-N domain	`21-93`
Ankyrin repeats	`369-455, 505-549`

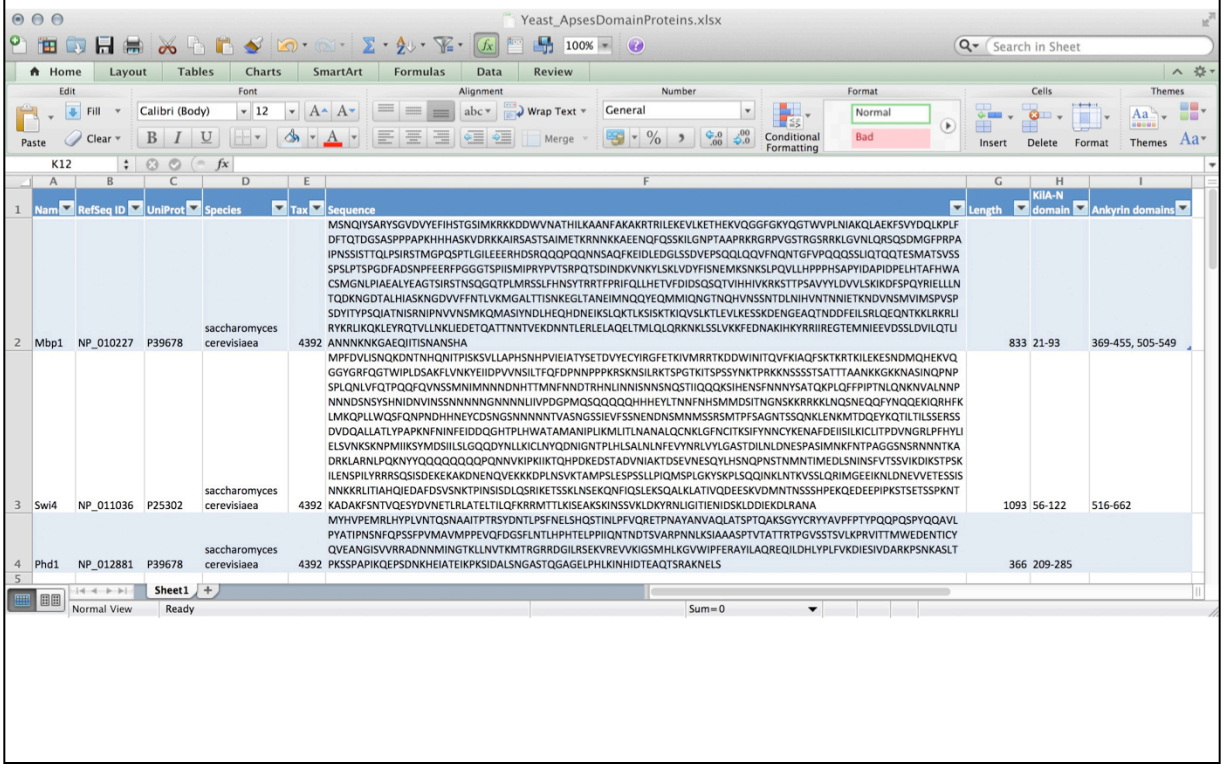
These are only *some* data items associated with the Mbp1 protein.

Let's assume we have identified the following information items that we would like to begin to store for our systems project (given here for the Mbp1 protein). How do we actually go about storing this information?



Your computer's file system is a full-featured database and we can readily use it for that purpose.

Data Modelling – Spreadsheet: Excel / OpenOffice Calc / Google Sheets



Often, simply putting data into a spreadsheet program is the right way to store it. But be careful: spreadsheets don't scale!

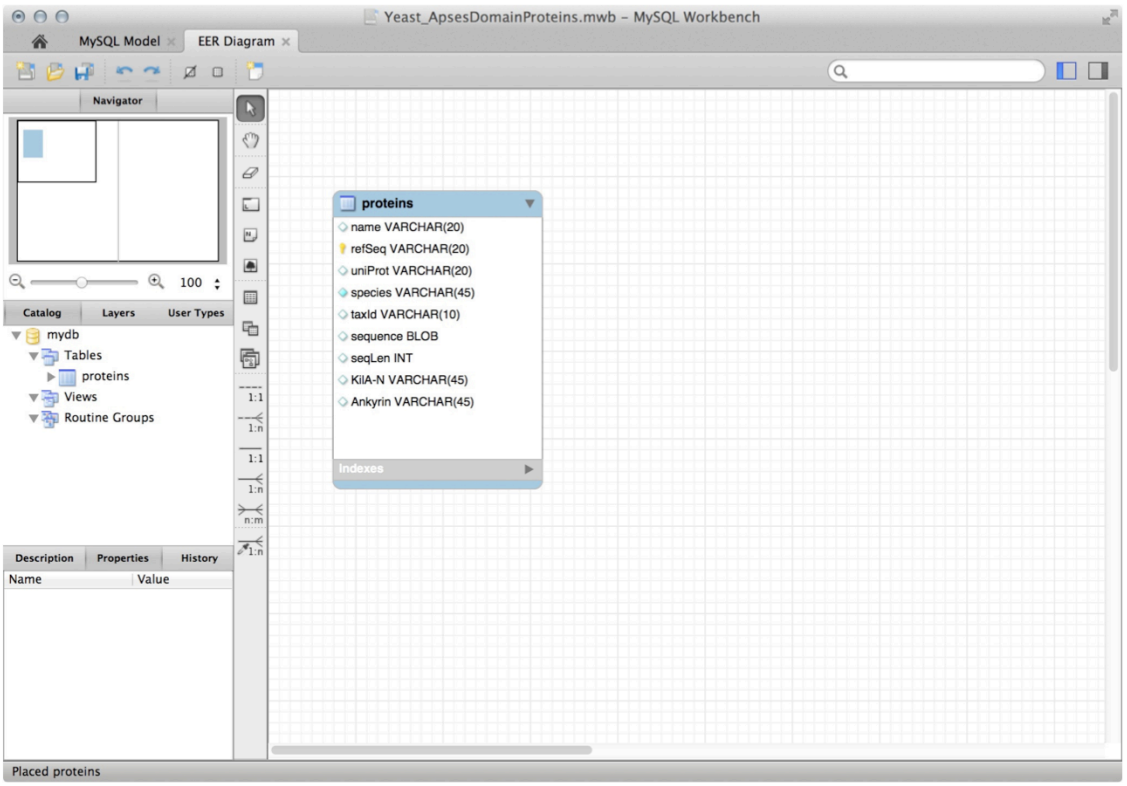
Data Modelling – R List

```
proteinData <- list(
    name     = "Mbp1",
    refSeq   = "NP_010227",
    uniProt  = "P39678",
    species  = "Saccharomyces cerevisiae",
    taxId    = "4392",
    sequence = paste(
                    "MSNQIYSARYSGVDVYEFIHSTGSIMKRKKDDWVNATHILKAANFAKAKR",
                    "TRILEKEVLKETHEKVQGGFGKYQGTWVPLNIAKQLAEKFSVYDQLKPLF",
                    "DFTQTDGSASPPPAPKHHHASKVDRKKAIRSASTSAIMETKRNNKKAEEN",
                    "QFQSSKILGNPTAAPRKRGRPVGSTRGSRRKLGVNLQRSQSDMGFPRPAI",
                    "PNSSISTTQLPSIRSTMGPQSPTLGILEEERHDSRQQQPQQNNSAQFKEI",
                    "DLEDGLSSDVEPSQQLQQVFNQNTGFVPQQQSSLIQTQQTESMATSVSSS",
                    "PSLPTSPGDFADSNPFEERFPGGGTSPIISMIPRYPVTSRPQTSDINDKV",
                    "NKYLSKLVDYFISNEMKSNKSLPQVLLHPPPHSAPYIDAPIDPELHTAFH",
                    "WACSMGNLPIAEALYEAGTSIRSTNSQGQTPLMRSSLFHNSYTRRTFPRI",
                    "FQLLHETVFDIDSQSQTVIHHIVKRKSTTPSAVYYLDVVLSKIKDFSPQY",
                    "RIELLLNTQDKNGDTALHIASKNGDVVFFNTLVKMGALTTISNKEGLTAN",
                    "EIMNQQYEQMMIQNGTNQHVNSSNTDLNIHVNTNNIETKNDVNSMVIMSP",
                    "VSPSDYITYPSQIATNISRNIPNVVNSMKQMASIYNDLHEQHDNEIKSLQ",
                    "KTLKSISKTKIQVSLKTLEVLKESSKDENGEAQTNDDFEILSRLQEQNTK",
                    "KLRKRLIRYKRLIKQKLEYRQTVLLNKLIEDETQATTNNTVEKDNNTLER",
                    "LELAQELTMLQLQRKNKLSSLVKKFEDNAKIHKYRRIIREGTEMNIEEVD",
                    "SSLDVILQTLIANNNKNKGAEQIITISNANSHA",
                    sep=""),
    seqLen   = 833,
    KilAN    = "21-93",
    Ankyrin  = "369-455, 505-549"
    )

save(proteinData, file="proteinData.Rda")
```

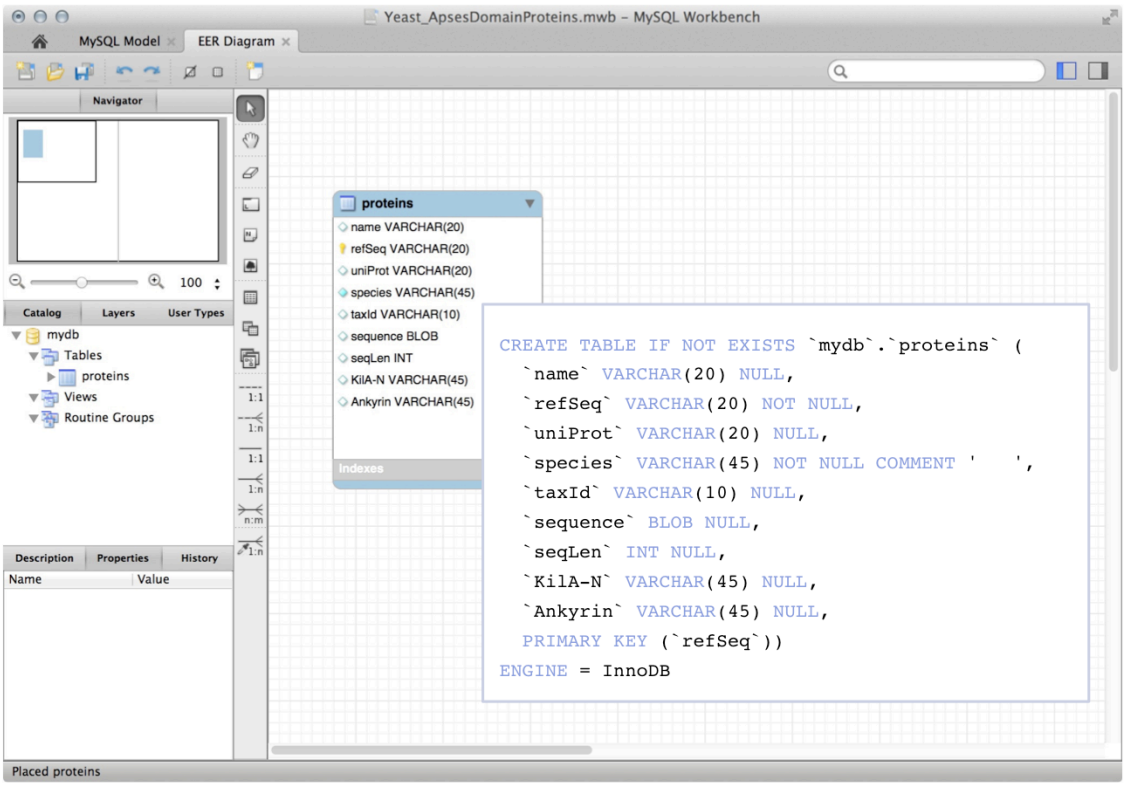
We will be using **R** lists and dataframes throughout the course to store and analyse data.

Data Modelling – Relational Database



A free, open-source, relational databse system like MySQL, Maria DB, or Postgres, provides industry strength database features and scalability – at the price of a bit of a learning curve and a bit of technical expertise to deploy the database on the computer. Not too bad though, actually and there is tons of information available about how to do this.

Data Modelling – Relational Database



This is our Mbp1 data modelled in the free MySQL Workbench application, which automatically generates the SQL syntax that will implement the model in a MySQL database. Neat.

There are many ways to store data and it is important not to be dogmatic about which database system to use. Define your objectives, evaluate alternatives, and make an informed decision.

What do you want to do with the data?

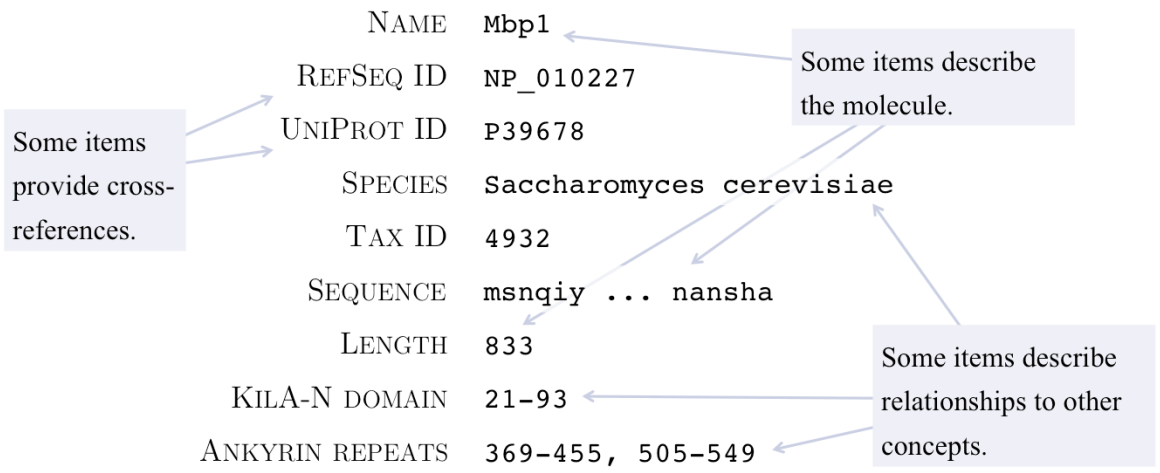
(List objectives and preferred approaches)

OUTLINE

Abstraction

Database systems

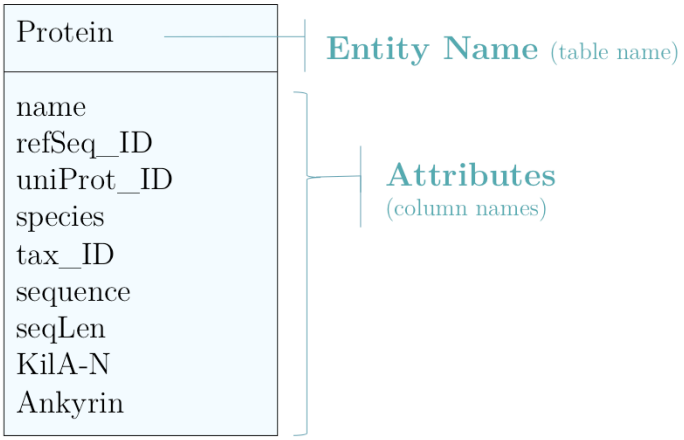
Data models



The fact that we **can** store this data does not necessarily mean we **should** store this data – at least, perhaps not in exactly this way. Before we set out to store data we need to spend some time constructing a proper data model. When the data is complex, we may in fact need to spend a lot of time building a model. But this is important: getting the data model right is the prerequisite to work with it efficiently and avoid errors.

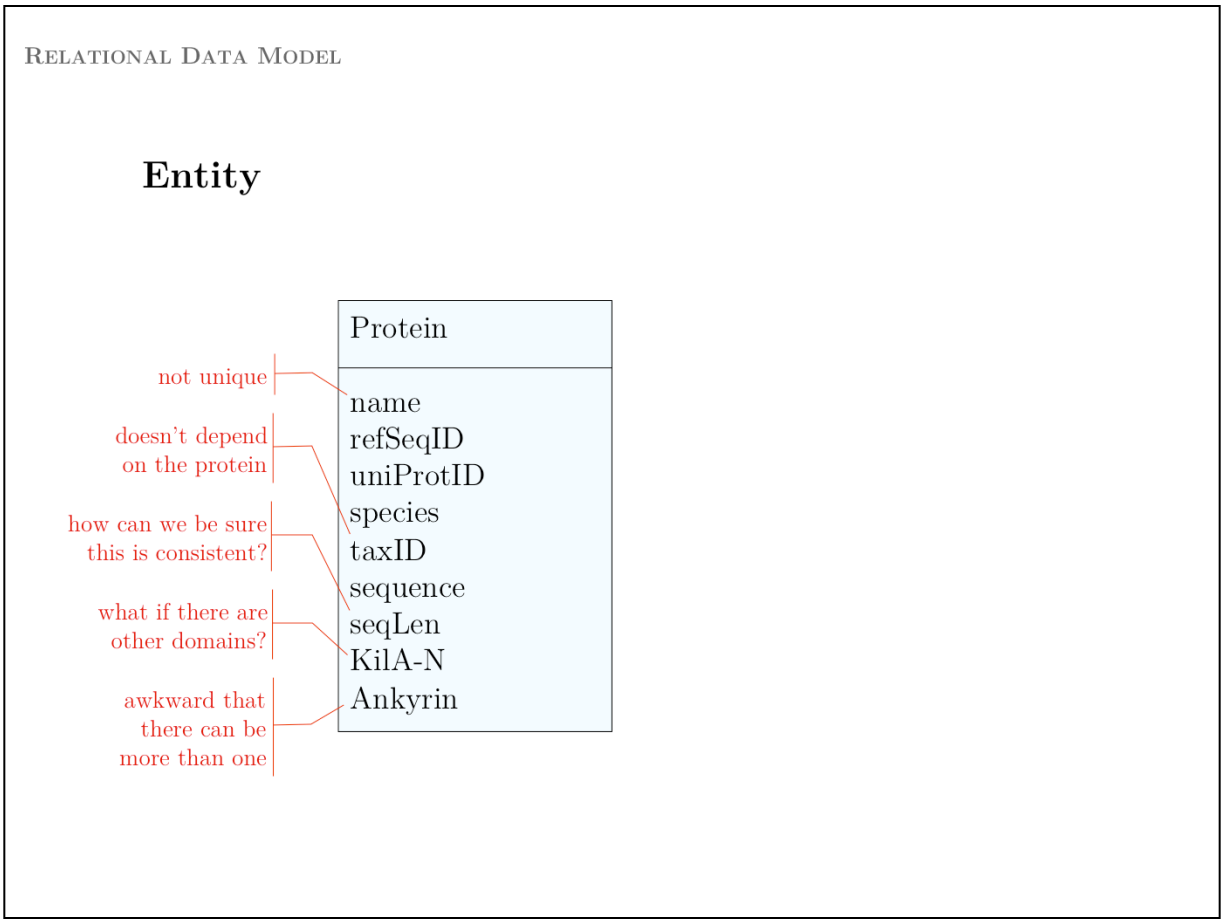
Speed only matters when you are on the right road.

Entity



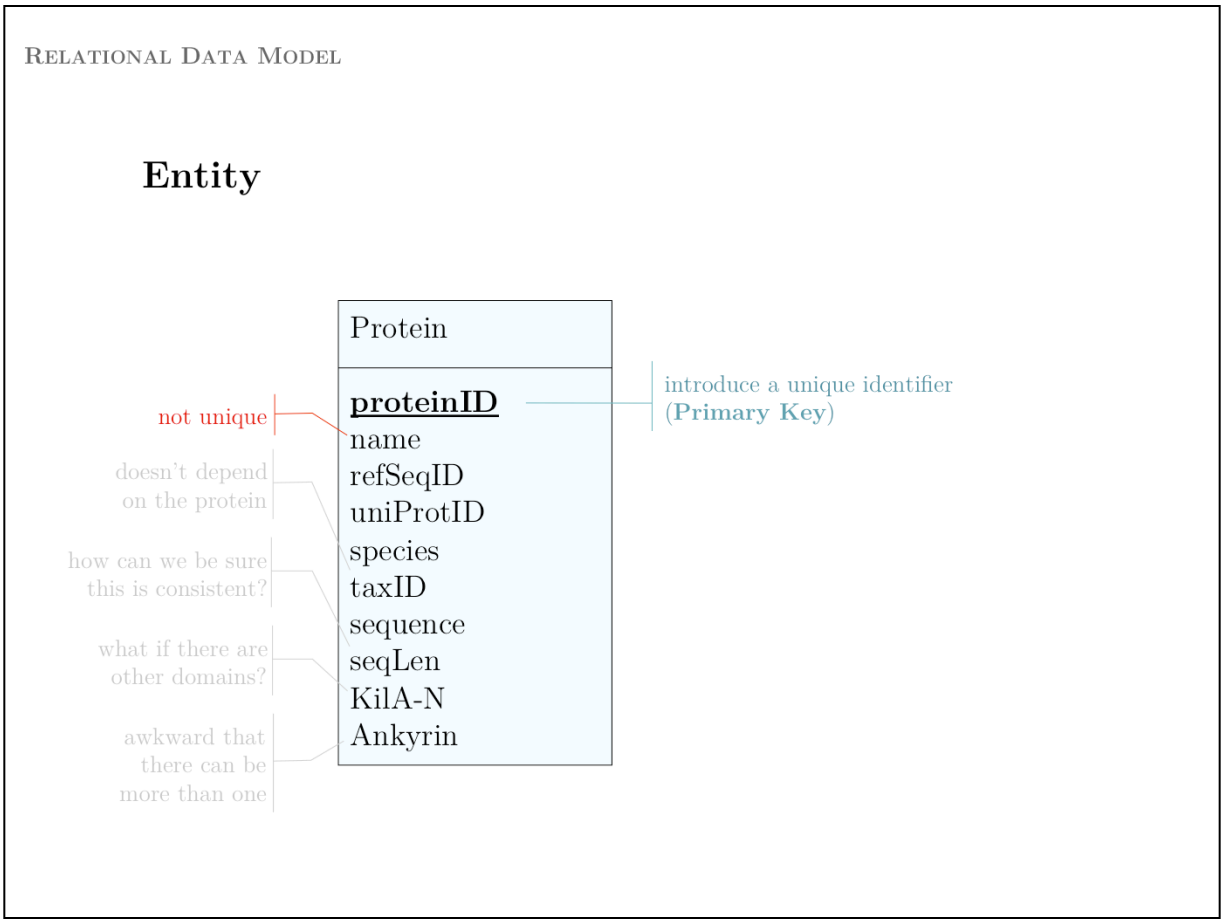
Here is a first (but naïve) implementation of a data model that captures the data we want to store. If we were to bring this into a spreadsheet format, the **Entity Name** would be the name of the spreadsheet, the **Attributes** would be the column-labels, and each row (or "record) would store the information about one protein. Fair enough – this captures what we discussed previously and it looks straightforward.

So what could possibly go wrong?



Our first try at defining the data we want to collect and store for the proteins that we want to work with has a number of problems. The structure of our *data model* is poorly conceived. This model can easily lead to inconsistent data, it will be hard to extend, and finding and cross-referencing information will be difficult.

Database theory has discussed for many years how to build data models that are structurally consistent, i.e. the structure of the model itself makes it impossible to get the data into an inconsistent state. This is called bringing a datamodel into *Normal Form*. Several types of Normal Form have been defined, For our purposes I will simply take you through the thinking we need to *normalize* our data model in practice. Learn from this example, then apply to your own ideas in the assignment.

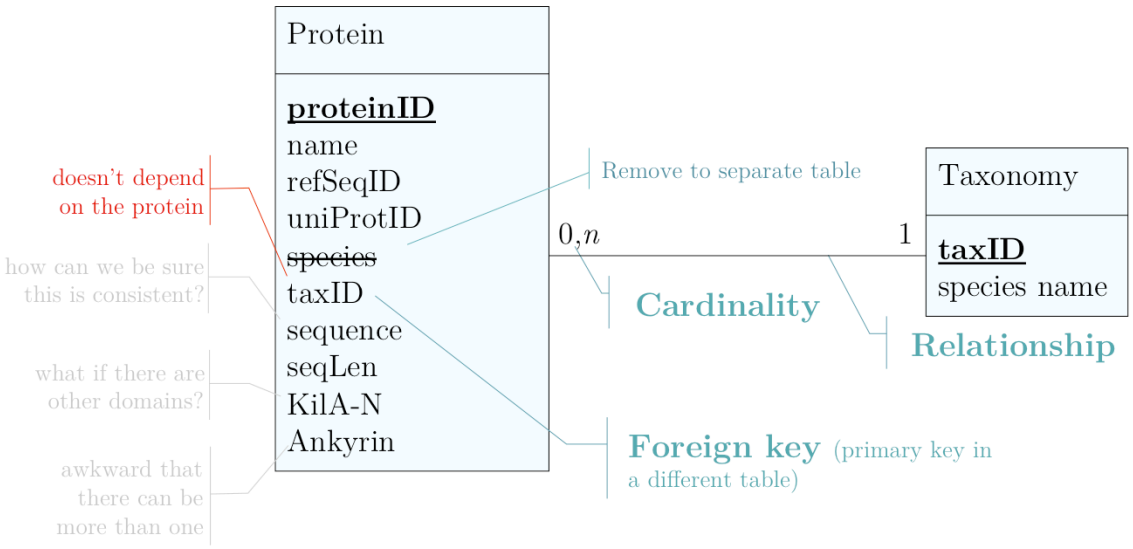


A *primary key* is a label that uniquely identifies a record in our database. This is the key that we use to find and retrieve entities, and to define relationships between records in different tables. It's convenient to simply use an integer for this ID: if the ID is simply one-larger than the largest key in the table, we can guarantee that it is unique. We shall resist the temptation to add any information into the key. The purpose of the key is not to store information, but to point to information. Appending strings, characters, keywords etc. may seem like a good idea at first, but it becomes a nightmare when the underlying information changes. In fact, such practice is an insidious example of duplicating information in different places – **and storing the same information in different places must always be avoided**. It becomes inconsistent.

That doesn't necessarily mean there are **no semantics at all** in a primary key. Take a refseq ID for example: if it reads NP_010227, it referes to an entry in the protein database. If it reads e.g. NM_001180115 it refers to an entry in the mRNA section of the nucleotide database. But note that this NM_ or NP_ prefix is information about the database, **not** about the individual record.

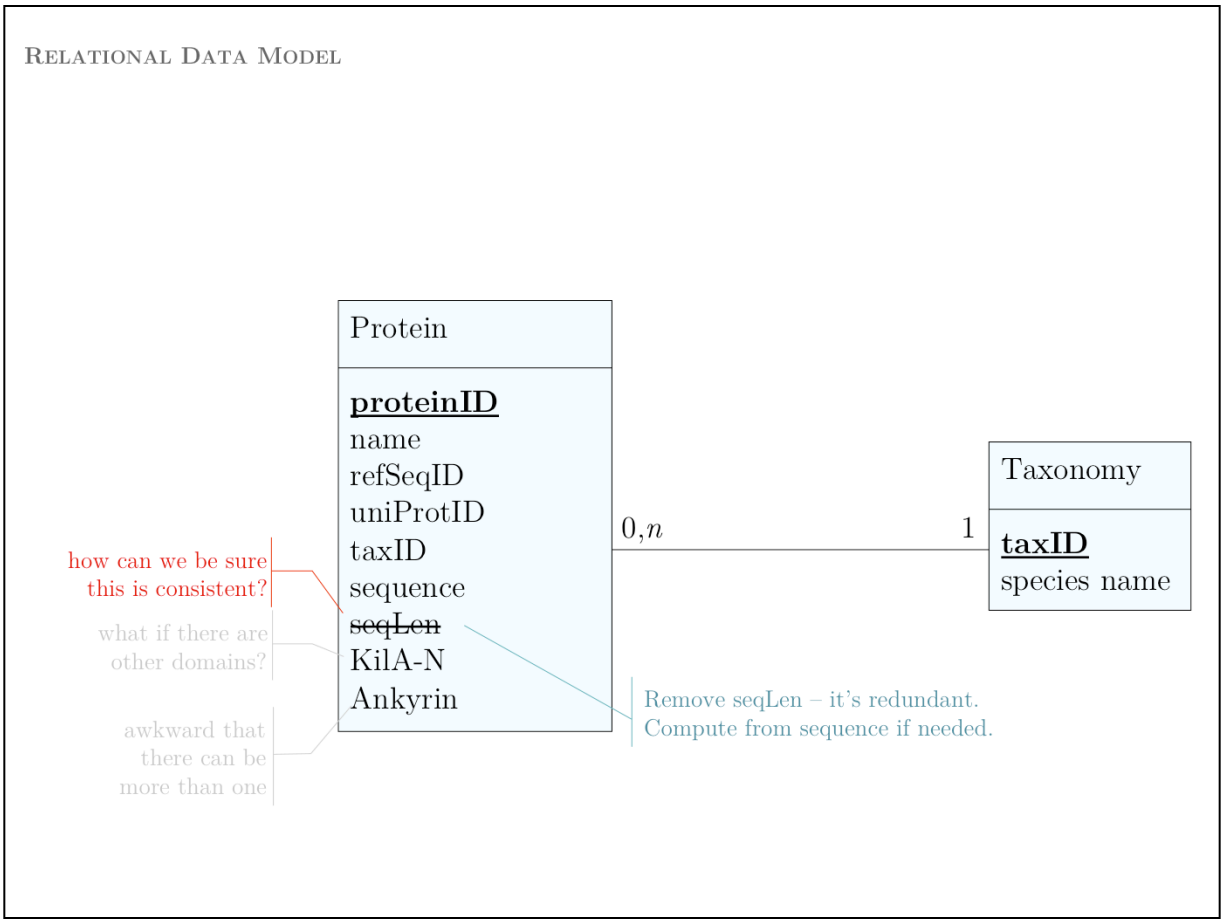
Why don't we use the refSeqID or the uniProtID of the protein as its unique, primary key? After all, what's good for the large databases should be good for us, no? We could, but these keys are not under our control. We could conceivably want to duplicate a record, and then e.g. define variant domain annotations, stemming from different algorithms. These variant records would have the same refSeqID and uniProtID, which is oK – actually intended – in our case, but then the refSeqID key is no longer unique and we can't use it as a primary key. Also, there is no promise made by the databases that there even **is** a one-to-one mapping betwen their identifiers. If we define and maintain our own key, we can extend the model

ERD: Entity Relationship Diagram

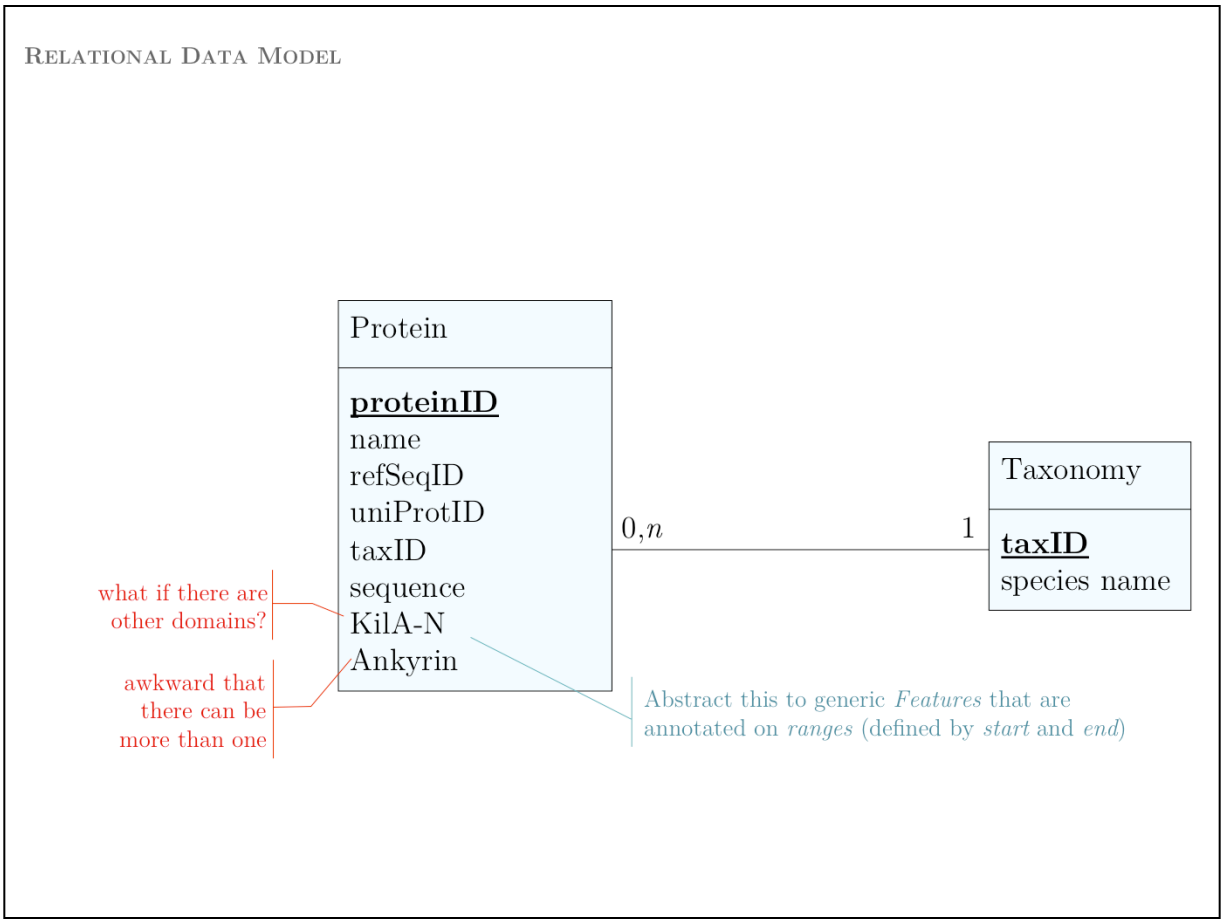


Moving the species name out of the **Protein** table requires to define a new table. The relationship between the two tables is established through two fields: the *Primary Key* of the **Taxonomy** table, and the taxID field in the **Protein** table. The relationships in data models are further described by their *Cardinality*: how often can we expect a key value to appear in a table. In our case, the relationship implies that there can be any number or even no proteins associated with a particular species (0, n), but a protein must have exactly one species associated with it. *Any number*, *exactly one* and *at least one* are the most commonly encountered cardinalities.

Thinking about the cardinalities is a good sanity check for our datamodel: the cardinalities imply constraints on the kind of record entries and updates that our database should be allowed to make. But they may also pinpoint conceptual problems. For example, if we find an n to n relationship, we can be pretty sure that our model is not really consistent. And if we have a 1 to 1 relationship, we might just as well store all of the information about one entity as an attribute of the other one – because that's what a 1 to 1 relationship means.



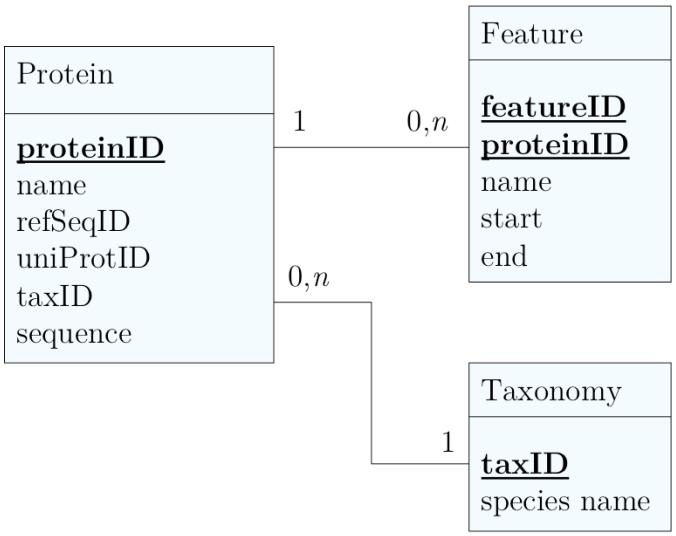
Next problem: attributes should only depend on the protein, not on each other. For example if we store both the protein sequence and its length, and if we then discover that the sequence actually contains seven more residues at the N-terminus because of an error in the gene model, will we remember to update the sequence length together with the sequence? We might forget – and that would make our record internally inconsistent. Such redundant information should not be separately stored, but simply computed on demand from the most authoritative data we have – in our case, that would be the sequence itself. We may be tempted to violate this rule in case the computation is expensive, but in that case we need to ensure the fields are automatically updated if information in one of them changes.



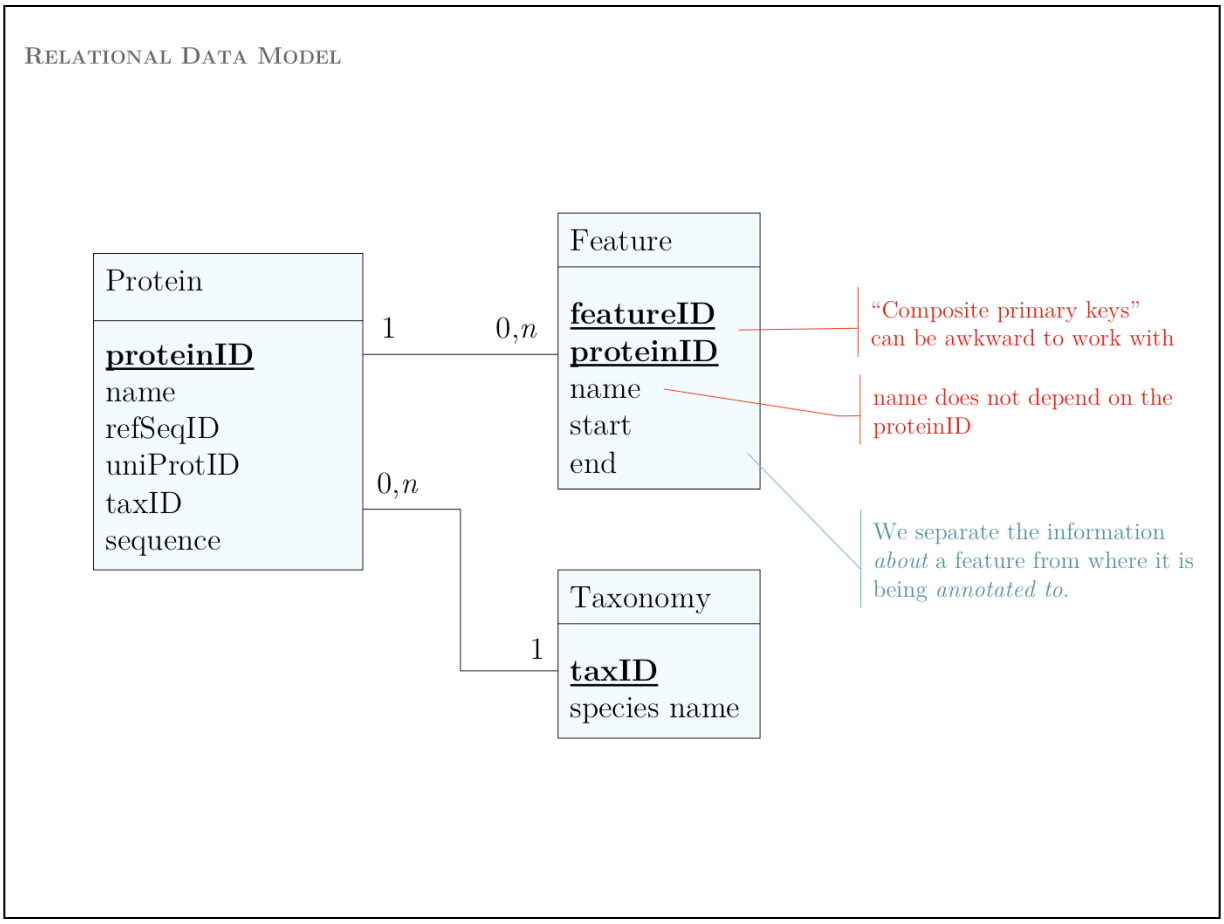
The domains we list here actually represent the worst part of our naïve datamodel. Say, we discover that a related protein contains an AT-hook motif. Should we then add a new attribute "AT-hook" to the table, for all of our proteins? And what about the other 16,306 entries in the Pfam domain database (as of June 2016)? An attribute for each? And do we really need to parse strings like "`369-455, 505-549`" and split them apart to find out how many of these domains are present and where the annotation starts and ends? A good rule of thumb says: you should build your model such that you need to parse your data only once: when you enter it into your database. From then on, it should be enough to retrieve the data in a way that you can use it directly. So: start and end should really be separate attributes of an annotation.

The answer is: it depends. Database theorists have divided opinions on what constitutes an atomic, indivisible value. E.g. we could argue that a sequence can be decomposed into its amino acids, and therefore is not really atomic. It becomes a question of context, and trying to be reasonable. In our case, I posit that this means each annotation should refer to exactly one feature (a domain, a sequence variant, a post-translational modification, a literature reference – whatever) and we should store *start* and *end* of the annotation separately, because we virtually always need to consider both values. Annotations that refer to one amino acid only (e.g. a phosphorylation site) will have the same *start* and *end*, and annotations that refer to the entire protein start at 1 and end at the last residue.

Relational Data Model

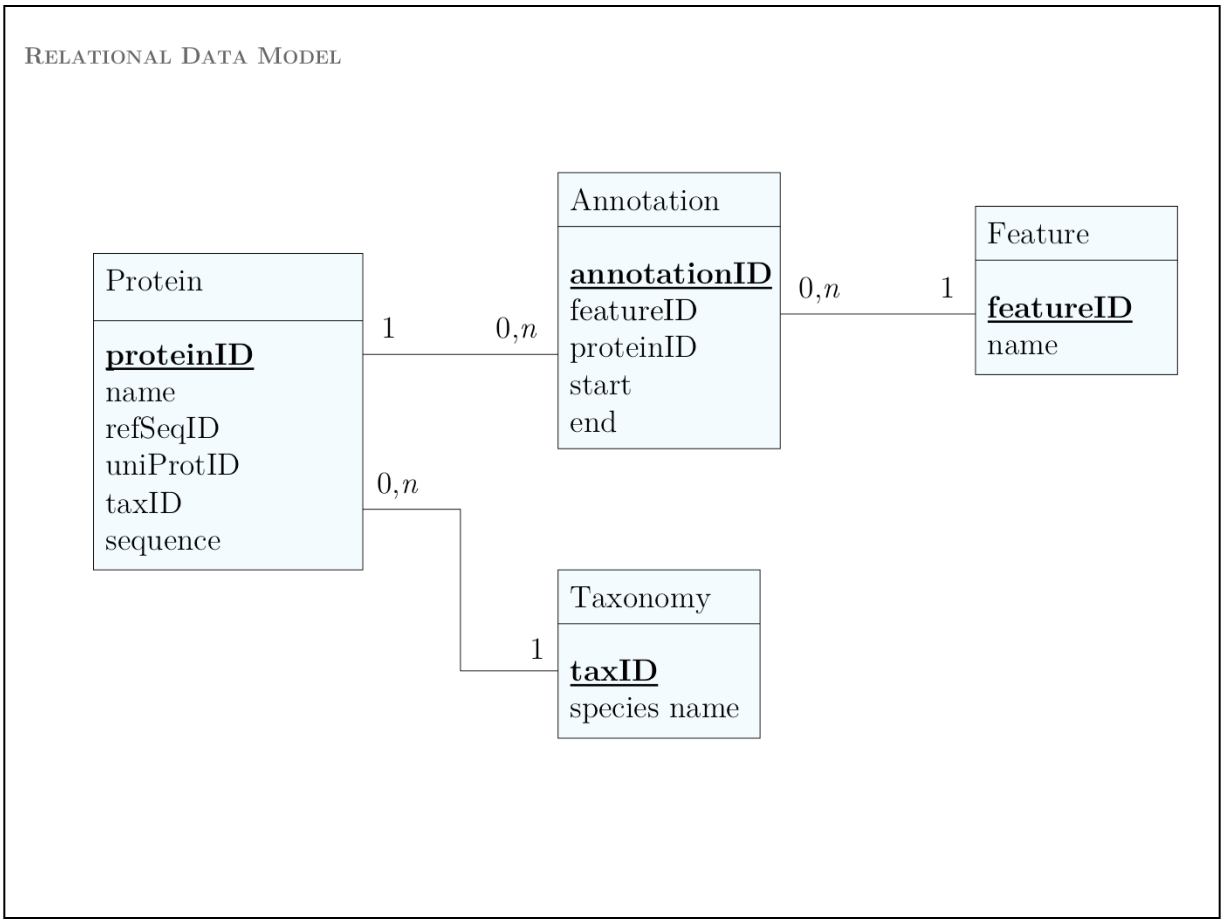


Taken together, we could put the features into a separate table like this. But this is again a bit naïve: there are problems. Can you spot some?



The new **Feature** table does not have a unique identifier, but its primary key is a composite of **featureID** and **proteinID.** It is much more work and much more error prone to program search and update procedures for composite keys. But that alone should not necessarily deter us – if there are benefits that outweigh the effort. More importantly though, the table contains a **name** attribute that depends only on part of the primary key, the **featureID**, and we are duplicating this for every actual occurrence of the feature. This is not such a big deal here, but it may become one if we decide that we really need additional information: Pfam IDs, PubMed references, notes, pointers to structure coordinates etc. etc. In general, the value of an attribute should depend on the key, the entire key, and on nothing but the key. That is not guaranteed in this model. The underlying problem is that we are actually trying to model an n to n relationship: a protein can have none or more of a particular type of feature, and a feature can be annotated to none or more proteins – in principle: in our model above, we could not even create a feature entry if we don't have at least one protein to annotate it to.

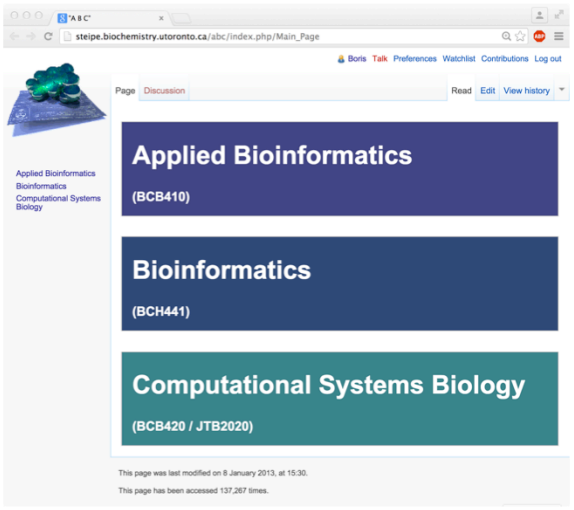
What we do instead is to employ a pattern that you will encounter very, very frequently in data models. All of the **information** about a particular entity (such as protein, feature ...) is kept in its own table. The actual **annotation** is stored in a separate table (sometimes called a "join table", a "junction table", or an "associative entity").



Here we have created **Annotation** as a join-table for proteins and their features. **proteinID** and **featureID** are foreign keys in the table, the annotation has its own ID as well, and we separate out the *start* and *end* positions to define which part of the sequence the annotation actually refers to. This solution is completely flexible and able to accomodate any kind and any number of annotations for each sequence.

This is a good beginning for a simple protein data model. In this model, every attribute *depends functionally on the primary key* – this means the information about the attribute is **specific** to each data record in the table. Each attribute is **atomic**, and all information items are **unique**, i.e. they are not duplicated anywhere.

CONTACT



http://steipe.biochemistry.utoronto.ca/abc

B O R I S . S T E I P E @ U T O R O N T O . C A

DEPARTMENT OF BIOCHEMISTRY & DEPARTMENT OF MOLECULAR GENETICS
UNIVERSITY OF TORONTO, CANADA