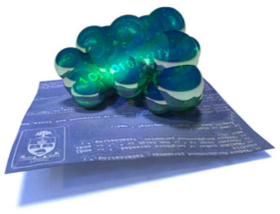
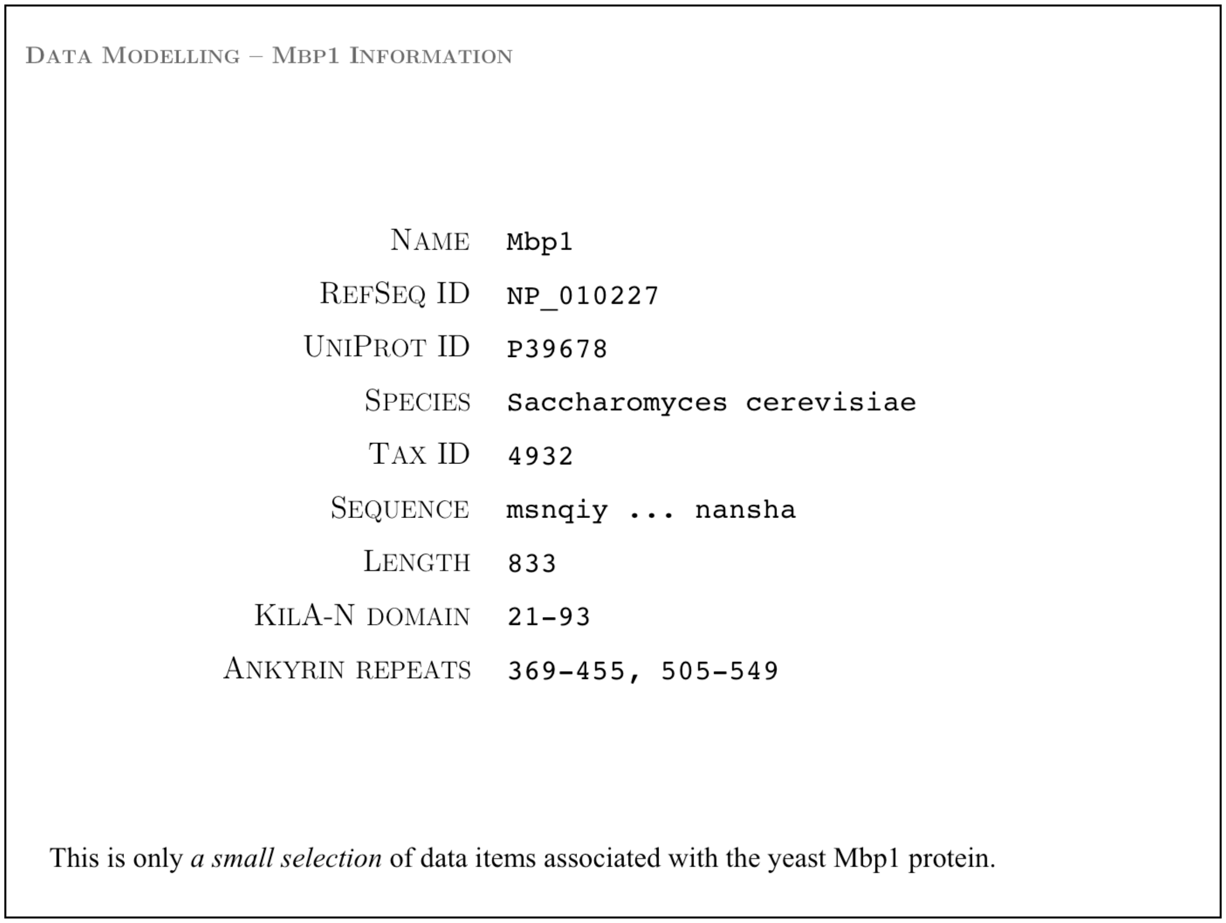
A
Bioinformatics
Course

Data Models



Boris Steipe

department of biochemistry – department of molecular genetics
University of Toronto



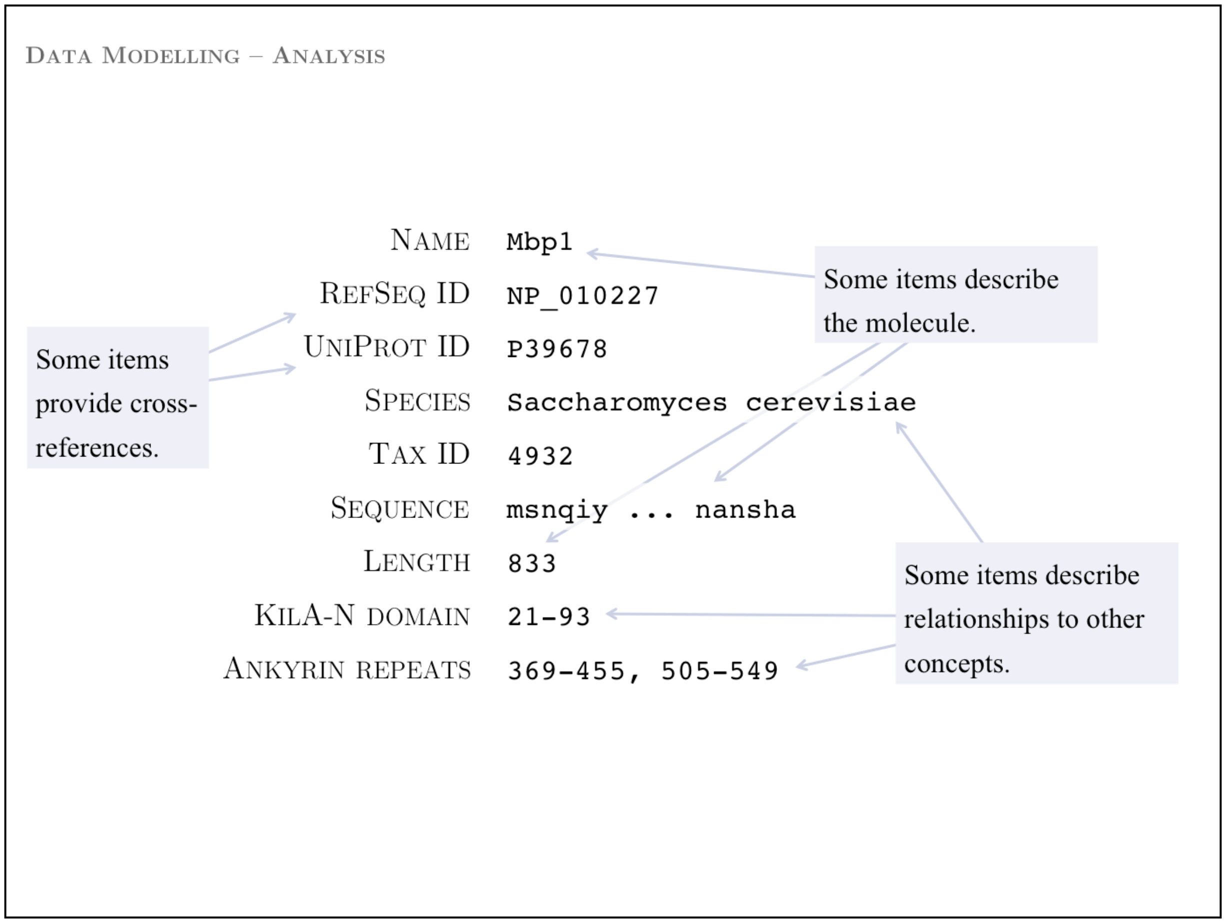
Data Modelling – Mbp1 Information

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`

This is only *a small selection* of data items associated with the yeast Mbp1 protein.

Let's assume we have identified the following information items that we would like to begin to store for a lab project. I've given it here for the Mbp1 protein, ultimately this will be relevant for proteins that you identify in other organisms. There's identifiers, there's sequence data, and there's metadata about certain features of the sequence. How do we actually go about organizing this information to make it available for computation?

Let's discuss the organizing principles first, before we talk about how to implement them in a database system, later.


DATA MODELLING – ANALYSIS

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`

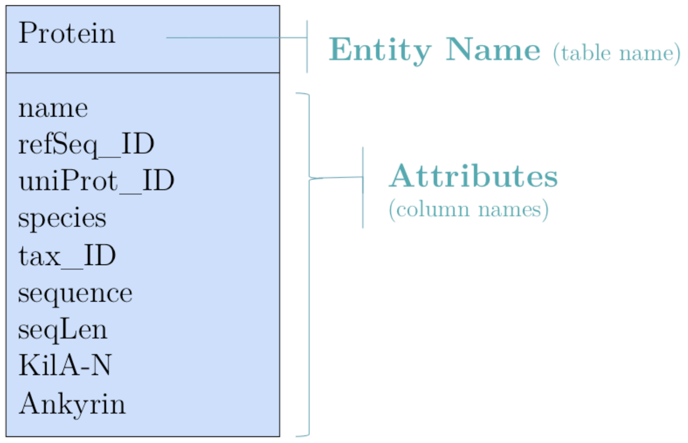
Some items provide cross-references.
Some items describe the molecule.
Some items describe relationships to other concepts.

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.

Relational Data Model

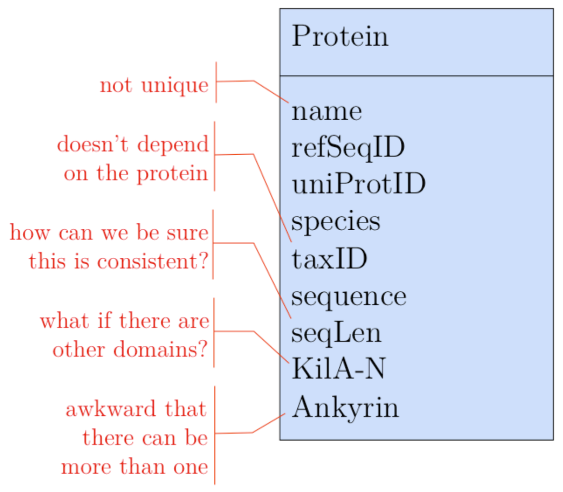
Entity



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?

Entity


Protein
name
refSeqID
uniProtID
species
taxID
sequence
seqLen
KilA-N
Ankyrin
not unique
doesn't depend on the protein
how can we be sure this is consistent?
what if there are other domains?
awkward that there can be more than one

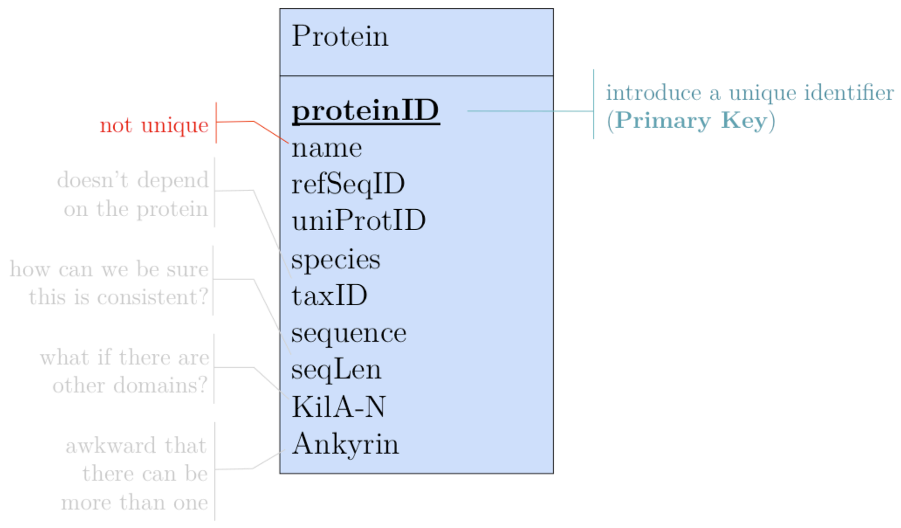
Our first try at defining the data we want to store has a number of problems. The structure of our *data model* is poorly conceived. This model can easily lead to inconsistent data, it will be hard to extend, and finding and cross-referencing information will be difficult.

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.

(You can read more about "Normal Form" here:
`https://en.wikipedia.org/wiki/Database_normalization`)

RELATIONAL DATA MODEL

Entity


Protein
proteinID
name
refSeqID
uniProtID
species
taxID
sequence
seqLen
KilA-N
Ankyrin
not unique
doesn't depend on the protein
how can we be sure this is consistent?
what if there are other domains?
awkward that there can be more than one
introduce a unique identifier (Primary Key)

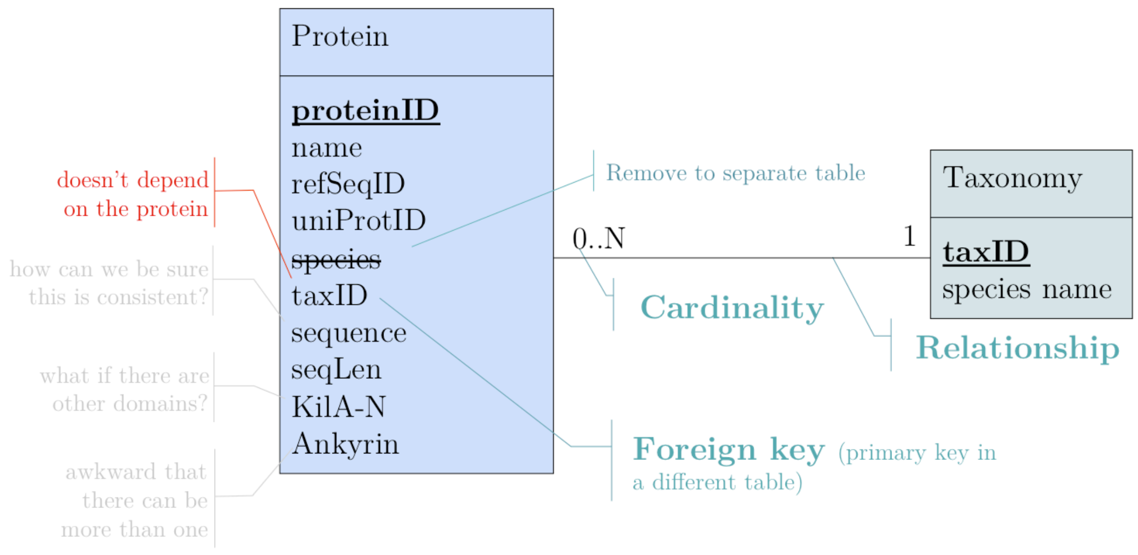
A *primary key* is a label that uniquely identifies a record in our database. This is the key that we use to find and retrieve entities, and to define relationships between records in different tables. It's convenient to simply use an integer for this ID: if a new ID is always chosen one-larger than the largest key that's already in the table, we can guarantee that the new ID 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 however we want.

RELATIONAL DATA MODEL

ERD: Entity Relationship Diagram

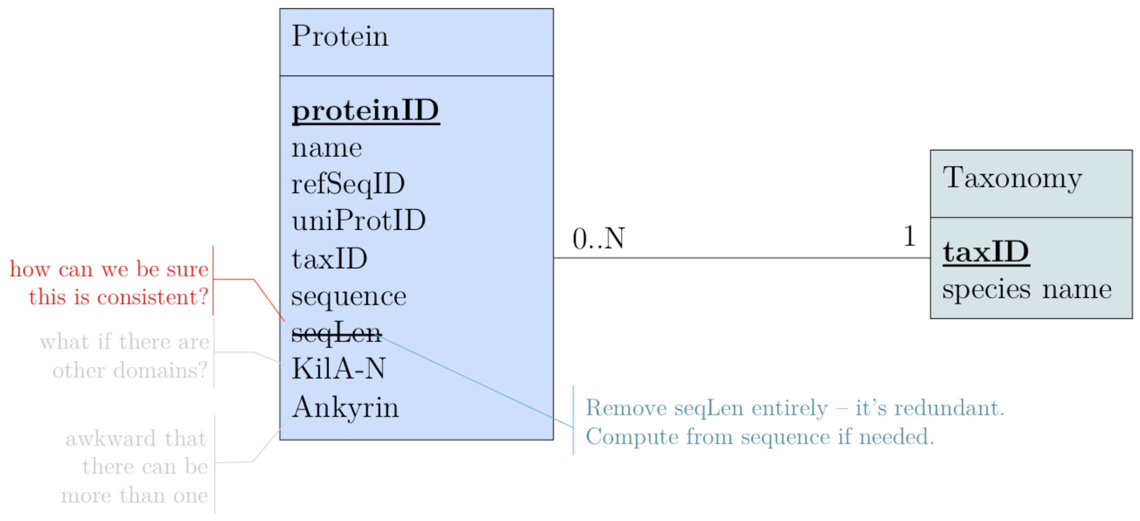

Protein
proteinID
name
refSeqID
uniProtID
~~species~~
taxID
sequence
seqLen
KilA-N
Ankyrin
doesn't depend on the protein
how can we be sure this is consistent?
what if there are other domains?
awkward that there can be more than one
Remove to separate table
0..N
1
Taxonomy
taxID
species name
Cardinality
Relationship
Foreign key (primary key in a different table)

The species name does not depend on the protein, but on the taxonomy ID (or vice versa). This means it should not be in the protein table in the first place. In fact, if you have species name / taxID combinations in two different protein records, you are storing the same name/ID relationship in two different places – and they could become inconsistent. So let's put the species name elsewhere.

We move the species name out of the **Protein** table and define a new table: **Taxonomy**. 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) – we couldn't even store species information previously if there was no protein obsreved i a species yet – but a protein must have exactly one species associated with it. "*Any number*" (0..N), "*exactly one*" (1) and "*at least one*" (1..N) are the most common 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.

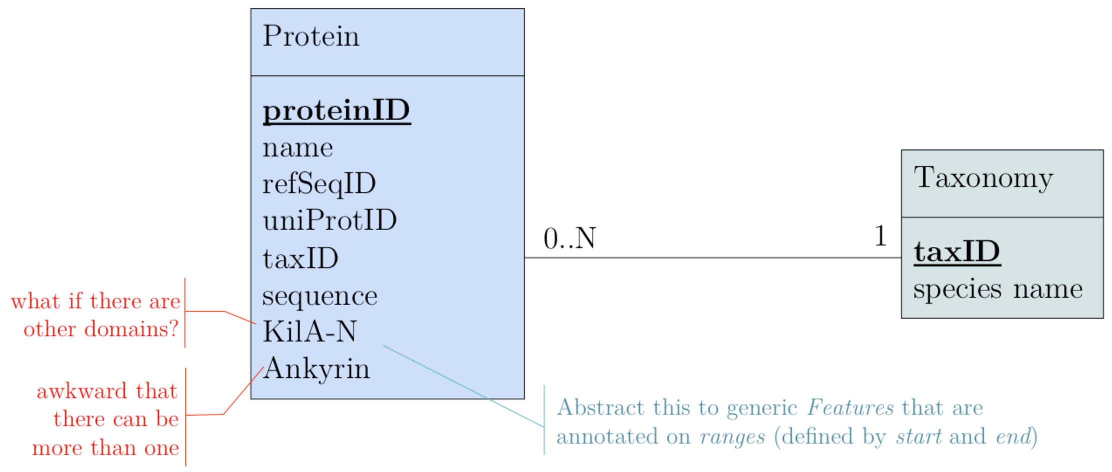
Relational Data Model


Protein
proteinID
name
refSeqID
uniProtID
taxID
sequence
~~seqLen~~
KilA-N
Ankyrin
0..N
1
Taxonomy
taxID
species name
how can we be sure this is consistent?
what if there are other domains?
awkward that there can be more than one
Remove seqLen entirely – it's redundant. Compute from sequence if needed.

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.

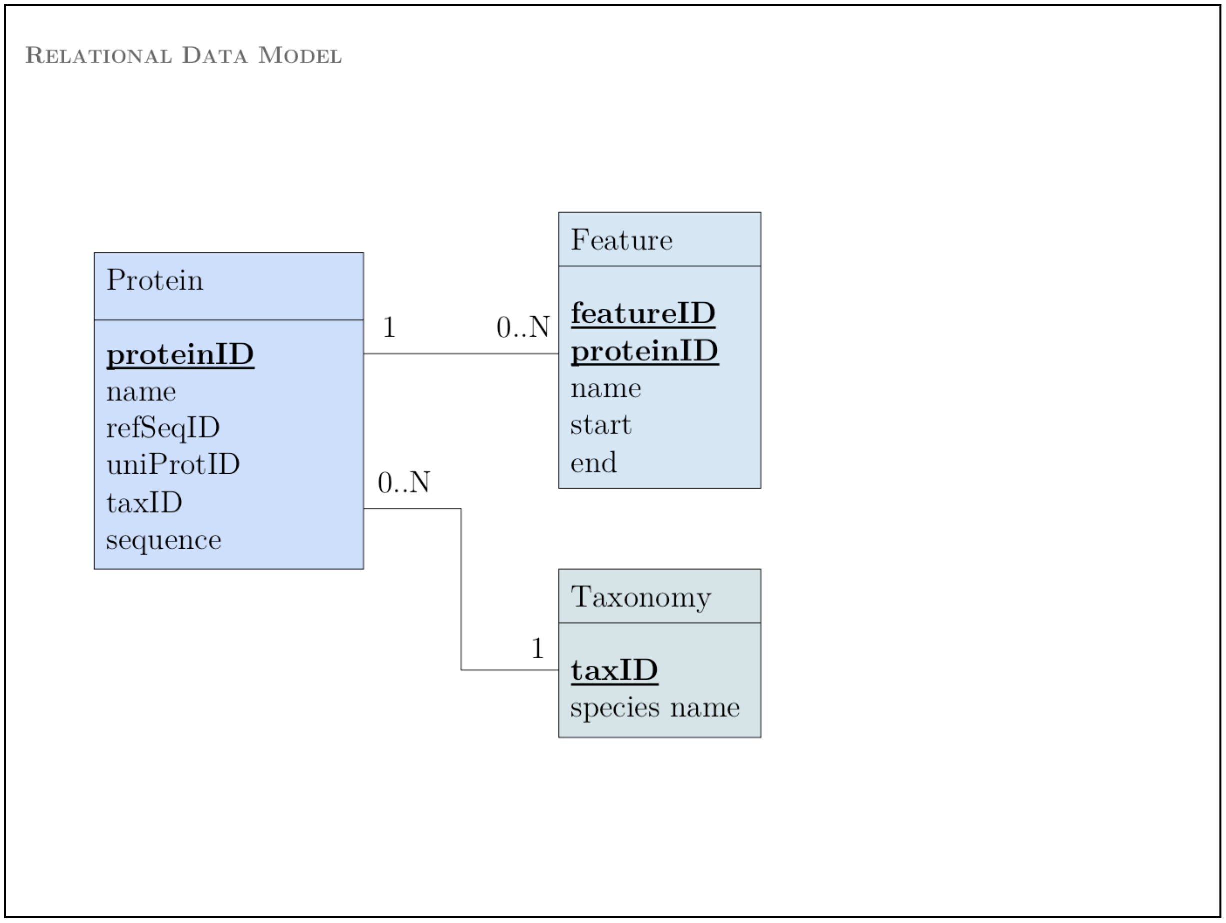
(We might violate this rule in case the computation is expensive, i.e. not perform the computation every time we need it but store its result, but in that case we need to ensure that fields are automatically updated if information in one of them changes.)

Relational Data Model

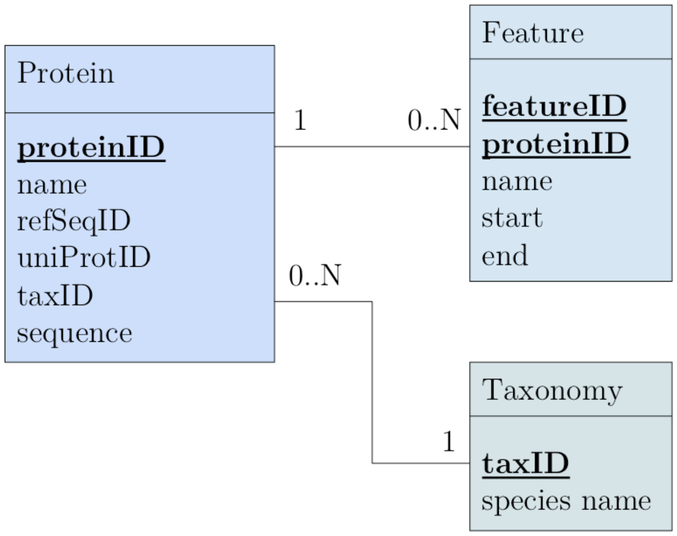

Protein
proteinID
name
refSeqID
uniProtID
taxID
sequence
KilA-N
Ankyrin
0..N
1
Taxonomy
taxID
species name
what if there are other domains?
awkward that there can be more than one
Abstract this to generic *Features* that are annotated on *ranges* (defined by *start* and *end*)

The domains we list here 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 domains 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, 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.

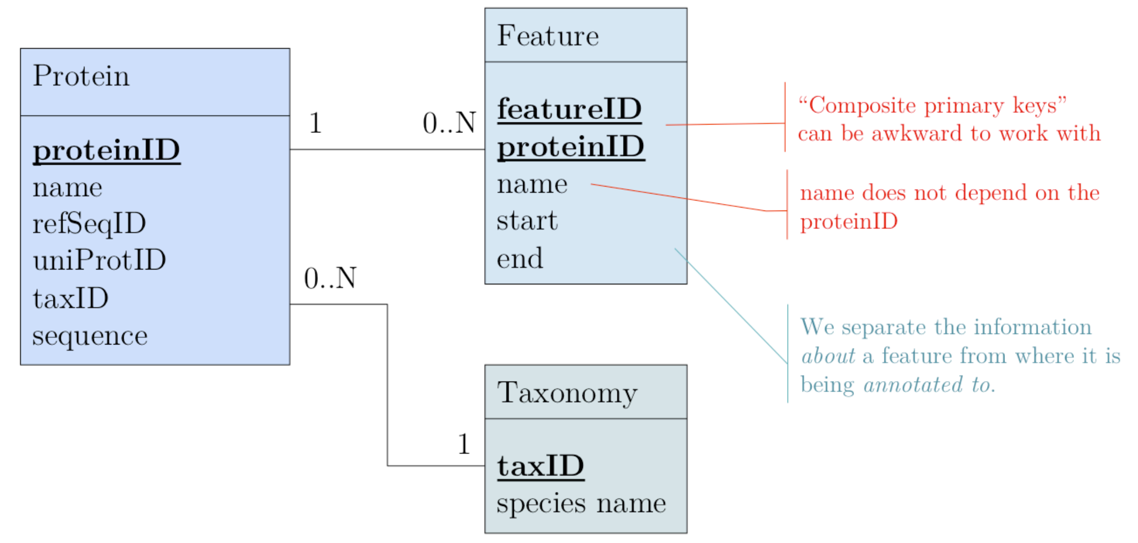


Relational Data Model



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?

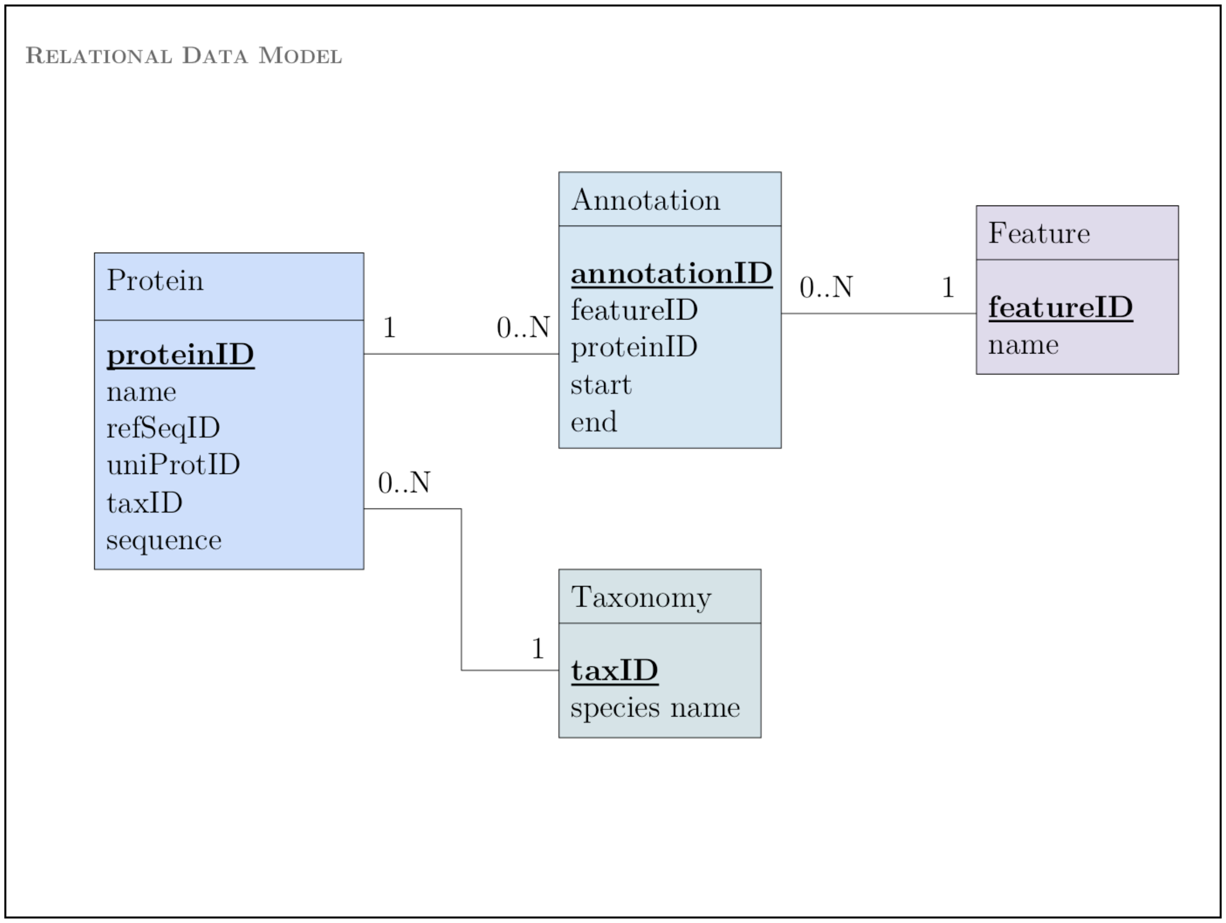
Relational Data Model


Protein
proteinID
name
refSeqID
uniProtID
taxID
sequence
1
0..N
Feature
featureID
proteinID
name
start
end
0..N
1
Taxonomy
taxID
species name
"Composite primary keys" can be awkward to work with
name does not depend on the proteinID
We separate the information *about* a feature from where it is being *annotated to.*

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 write 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. E.g we store "KilA-N" in the record for the Mbp1 protein, and "KilA-N", again in the record for the Swi4 protein etc. 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. All need to be duplicated for every single protein. And, what is even worse: the model does not **prevent** us from calling this particular feature an APSES domain in the PhD1 protein – and the database is no longer consistent at this point.

In general, the value of an attribute must depend on the entry's 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.

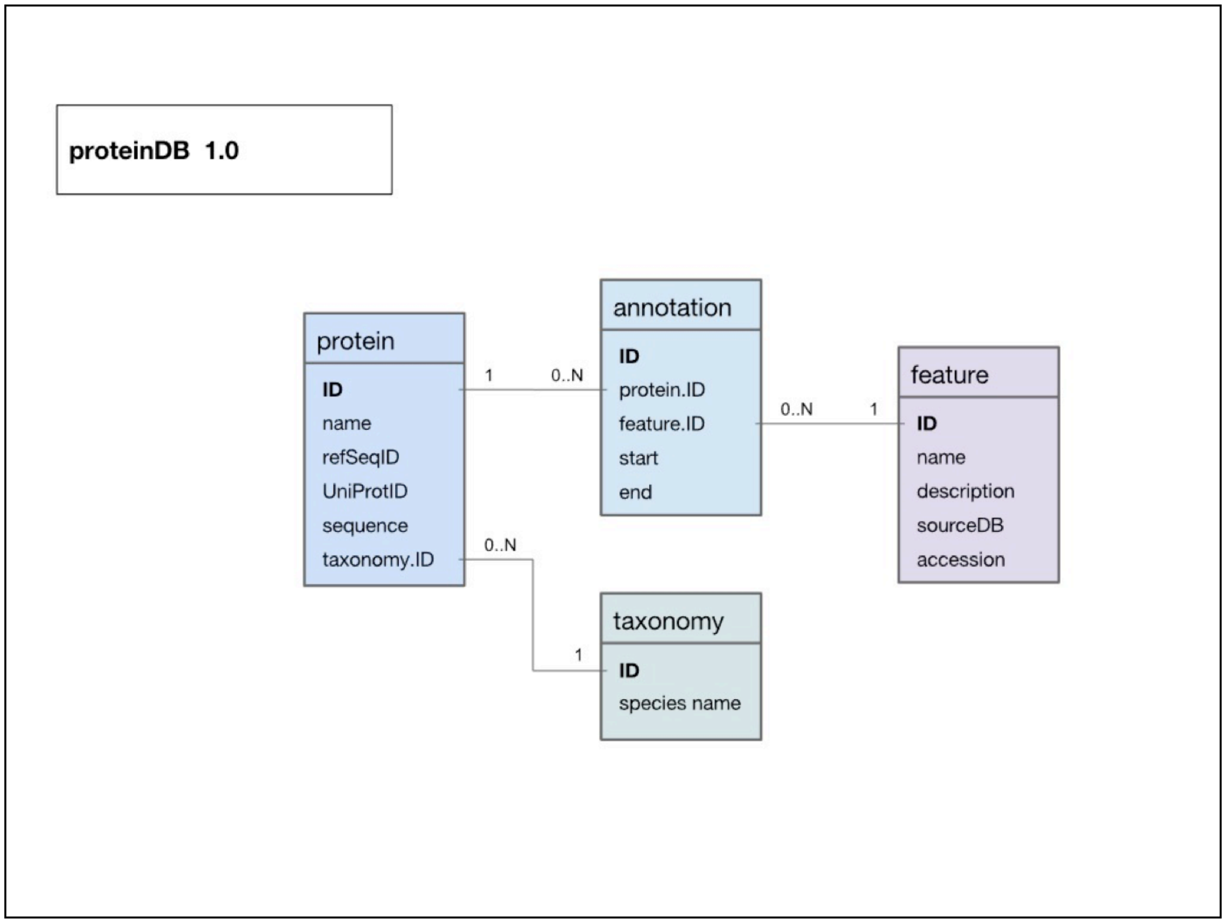
What we do instead is to employ a pattern that you will encounter very, very frequently in data models. All of the **information** about a particular entity (such as protein, feature ...) is kept in its own table. The actual **annotation** is stored in a separate table (called a "join table", a "junction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

A good way to recognize when a join table is needed is to think about this as a matrix: a list of proteins is a 1-Dimensional table. A list of domains is a 1-Dimensional table. But to record which domain is in what protein, we need a 2-Dimensional matrix. The join table simply represents every cell in that matrix that has a value – it flattens the 2-D matrix into a 1-D list, that becomes a table in our relational datamodel.

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.



This is a version of the same datamodel drawn with the Google Drawing service on the Web. The drawing lives here: https://docs.google.com/drawings/d/1uupNvz18_FYFwyyVPebTM0CUxcJCPDQuxuIJGpjWQWg and you can access it and make a copy for yourself that you can edit and extend for your own models.

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