‘Introduction to Database Concepts’, ‘Data Normalization Principles’ and ‘Relationships’, excerpts from Harvey Quaman ‘Digital Humanities Databases: Tutorials, References and Exercises’, Digital Humanities Summer Institute 4–8 June 2012, Victoria University, Victoria BC, 2012, pp. 2-3, 4-6, and 7-15

‘Introduction to Database Concepts’

Definitions & Concepts:

A database is a collection of data that is organized and stored according to some purpose.

The database is organized into tables (which look like HTML tables or spreadsheets). Each table stores data about some real-world ***entity***.

Each table is organized into rows and columns.

Each row in the table is a record, which usually corresponds to a one real-world object (like an animal or a car or an archival document).

A record can contain several pieces of information. A column (sometimes called an ***attribute*** or a ***field***) corresponds to one of those pieces of information. They are often adjectives.

A **relational database** (as opposed to a “flat-file database”) is very good at relating (that is, matching up) information stored in one table to information stored in another table by looking for elements common to each of them.

A **primary key** is a special column in a table whose values (perhaps numbers, perhaps names) are unique within that column. A primary key has three qualities: 1) it is unique across all records in the table; 2) it has a non-NULL value for each record in the table for the entire lifetime of the record; 3) its value never changes during the lifetime of the record.

A **foreign key** is a value in a table that is a primary key from a different table. Foreign keys need not be unique. Foreign keys make relational databases possible.

General Design Principles:

1. **Store only one piece of information in each field**. What counts as “one piece of information” will depend on the purpose of your database. Generally speaking, though, no groups, lists, or comma-separated values!

2. **Store each piece of data in only one place**. If you’re finding the same name or value typed out again and again in different places, then you need to create a new table, store that data there, assign it a primary key, and use foreign keys elsewhere in the database.

3. **Do not store anything that you can calculate**. Do not store dynamically changing data, such as the number of days until you graduate or someone’s age; store a birthday instead.

4. **Design tables so that adding new information creates new rows (records), not new columns.**

5. **Design tables to contain as few NULL values as possible**. If you notice lots of NULL values (or even lots of repeated values) your table probably needs further normalization.

Data Modeling Strategies:

1. **Real-world entities**, which may be range from the concrete (baseball cards, pets, contracts) to the abstract (friends, genres, rankings), become tables in the database. Entities are often nouns.

2. An entity's **attributes** (or qualities or properties or identifiers) become columns in its table. Attributes are often adjectives. Choose an appropriate data type for each column.

3. **Specific instances** of an entity become rows or records in the table. Whereas **person** is an entity and therefore signifies a table, **Bob** and **Sue** and **Jim** are specific instances and so are represented by individual records in the table.

4. Unique attributes (identifiers) become **primary keys**. Interestingly, most entities do not naturally have unique identifiers (there are a lot of Jim Smiths in the world, for example), so most of the time we just assign arbitrary sequences of numbers as primary keys. Sometimes users are fully aware of their primary keys (student numbers, social insurance or social security numbers, etc.), but many times users never know their primary keys. Primary keys are not always public information.

5. When stored in another table, a primary key is known as a **foreign key**. Relationships in the database are modeled using foreign keys.

Data Normalization Principles

The classic definition of a database is that it's a collection of data that is organized and stored according to some purpose. That sounds pretty vague and you might think that a definition like that opens the door for a chaotic data-driven free-for-all. Not so, in practice.

There are indeed some best practices when it comes to organizing data in a database. And we have Raymond F. Boyce and Edgar F. Codd to thank for that. The system they developed is sometimes called Boyce-Codd Normal Form or BCNF for short. (Knowing that is a great icebreaker at parties. Try it. You'll see.)

Those principles are actually defined in mathematical terms, though. Knowing that fact might come in handy if you're ever captured by space alien geniuses and your freedom hinges on your ability to provide a demonstrable proof that SQL (Structured Query Language) works and that any well-structured query is indeed guaranteed to give you the right answer. If you're just learning databases, though, the technical definitions of data normalization forms don't help at all.

So here's the shortcut. Quamen's over-simplified and metaphoric approach to data normalization:

Step 1. **Nouns**
Step 2. **Relationships**
Step 3. **Adjectives**

Step 1. **Create a separate table for each entity in your data**.

Entities are usually real-world objects like birds, employees, cars, concerts, contracts, or events. Any important noun in a prose description of your dataset is probably an entity and should therefore get its own table.

Achieving so-called *first normal form* (1NF) really involves striving for these three goals:

1. There are no duplicated rows in the table. Each concrete instance of your entity occupies one and only one row. Each row should have some kind of unique identifier, which we call a ***primary key***. Sometimes the dataset provides us with a good primary key but, if not, we can always invent one (usually an arbitrary but linear sequence of numbers).

2. Each cell (or field) contains only one value. There should no groups of data or comma-separated lists of data in any given cell.

3. Any given column contains the same kind of data. For example, avoid illicit mixing in one column of phone numbers and email addresses. Those should be two separate columns.

Step 2. **Move to new tables any repetitive information in your existing entity tables**.

After you've created an entity table, take it out for a test drive by filling it with some sample data. Look for information that gets repeated from row to row—data bits like addresses, course titles, names of bosses or instructors, dinner courses, authors of books, museum or archive names, event venues, movie genres, university names, information categories, building names, etc. These repeated bits of information should now be moved into their own tables. Often, they represent "sub-entities" that we simply didn't recognize as being important in Step 1. That's OK. That's why we have a Step 2.

More examples: Shakespeare's plays are often classified as comedies, histories or tragedies. Many different movies were all directed by Martin Scorsese. An interesting group of countries are all located in Europe. Animals are classified into reptiles, amphibians, birds, mammals. Your personal library probably has many books by the same publisher.

Moving repetitive information into new tables should suggest to you that those new tables need to go through the data normalization process as well. Repeat the process on those tables: ensure row and column consistency, guarantee row uniqueness with a primary key, etc.

And in order to maintain the relationship between the old, original entity and this new one that you've just created, you should store the new table's primary key back in the original table as a ***foreign key***. And, for me, that's the gist of *second normal form* (2NF): the creation of relationships between entities. Databases model relationships with a system of primary keys and foreign keys.

Data relationships are classically divided into three varieties: *one-to-one*, *one-to-many*, and *many-to-many*. The more you know about modeling data relationships, the easier 2NF

becomes. There are enough subtleties about relationships, though, that they warrant their own discussion. Check out the section on "Relationships" for more details.

Step 3. **Make sure all table columns are fully dependent upon the table's primary key**.

This rule is really about the consistency and integrity of data. The relationship between any given column and its table should be analogous to the relationship between an adjective and its noun. Any columns in your tables that don't contribute like adjectives probably need to be moved elsewhere. Any columns whose values could "go stale" over time (that is to say, that simply become incorrect over time) should be reconceptualized.

Here are a few common litmus tests:

1. If we delete a row, will we lose any data that other records might need?

If (or when) we delete Paris Hilton's autobiography from the library database, will we also lose the fact that autobiographies are housed in Johnson Library? If we delete dinner course #2 from the meal database, do we also lose track of which one is the salad fork? When Babe Ruth got traded to the Yankees, did we also accidentally lose the fact that the Red Sox play in Boston?

2. If we add a row, could we accidentally make any data entry mistakes that would compromise our database's integrity?

When we inevitably add Paris Hilton's autobiography back into the library database, could we accidentally record the mistake that autobiographies are housed in Campbell Library instead of Johnson Library? When we add a new course to the ambassador's fancy dinner party, do we need to know which one is the salad fork? When we logged the Babe Ruth trade, did we mistakenly assign him to Chicago because misspelling the word *socks* is weird enough that surely only one team would do it?

3. Have we stored any values that we should calculate instead?

Don't store someone's age; store a birthdate. We can always calculate a person's age on-the-fly when we need to know it. Don't store the average price of bananas; store all the prices of bananas and then calculate today's average price. Don't store the highest and lowest test scores; calculate them. Don't store the number of baseball trades so far this year; count them.

If the answer to any of those litmus test questions is "yes," then you should either move the problematic information to a new table or else (in the case that your data can accidentally go stale) try to decide how to store it in such a way that it doesn't implicitly have a "best before" date stamped on its forehead.

Reading the technical details about data normalization can turn you into a shark—you feel that you have to keep moving or else your eyes will roll back in your head and you'll die from lack of oxygen. That's because data normalization is, in its most fundamental form, about mathematical proof. And most of us don't care about the mathematics or the implicit guarantees about truthiness that math can provide.

Rather, we're interested in the heuristics—rules of thumb—and the "best practices" about how to create database tables effectively. The guidelines above simply translate mathematical concepts like *functional* and *transitive* dependencies into more metaphoric terms like *relationships* and *adjectives*.

Data normalization—especially the 2NF step—gets easier when you know more about how databases handle relationships between tables. Read on, MacDuff.

Relationships

> ***synonyms:*** *connected, associated, linked, coupled, allied,affiliated, corresponding, kindred, parallel, be relevant to, pertain to, concerned with, having a bearing on, appertain to, involved with, pertinent to, have a rapport with, identify with . . .*

Database people talk about relationships between entities as if they're obvious and self-evident. And maybe they are. But when I'm working on a new database design, I inevitably run through a list of synonyms for the term "relationship" as I try to wrangle my data into patterns.

The heart of a database relationship is that two different real-world entities are somehow linked or affiliated with one another. British Columbia is located in Canada, even though the province and the country are different entities and either can be discussed without reference to the other. John Lennon still has a connection to the Beatles even though neither entity, unfortunately, has survived. No matter how you say it, Uranus is still funny.

Despite the complexity of real-world relationships, though, databases model relationships as a connection between two tables using primary and foreign keys. That simplicity is perhaps why database people think relationships are self-evident. Nonetheless, database designers classify the relationship between entities into three types:

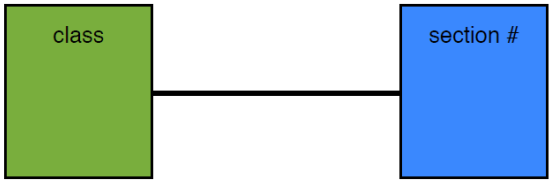
relationship	description	examples
one-to-one one	X has only one Y	• a person has one social insurance number • a building has one address
one-to-many	one X has many Y's	• a country has many cities • a student takes many classes
many-to-many	many X's have many Y's	• many different bands play at many different venues • many mythbusters bust many myths

One-to-One Relationships

A one-to-one relationship says that each entity of type X is connected to only one entity of type Y. For example, each person has one social insurance (or "social security") number.

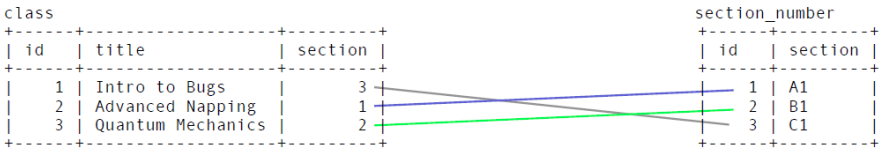
Each student at a university has one identification number. Each class at the university has only one section identifier. Each automobile has one license plate number.

If we were to model that in a database, we'd draw a picture like this:



Here, each rectangle represents a table and the horizontal line represents a relationship between them. It's a single line—it doesn't branch or divide—so we will interpret that type of line as a one relationship. Database people would say that its ***cardinality*** is **one**. Here, the line is **one** on both sides, and so this diagram suggests that the relationship between class and section number is **one-to-one**.

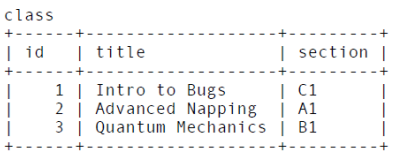
In the database, we'd model that relationship by taking the primary key from one of the entities (say**, section_number**) and putting it into the class table as a foreign key. And anytime we want to discover the section number of any given class, we'll retrieve the foreign key, look across to the **section_number** table, and learn the section number.



Here's the dilemma. There's no reason why we couldn't simply store the section number directly in the class table. There's no need here for a relationship at all. The introduction of foreign keys unnecessarily complicates an otherwise simple and straightforward table design. So the best practice in this case is to remove the **section_number** table entirely and move its data into the class table.

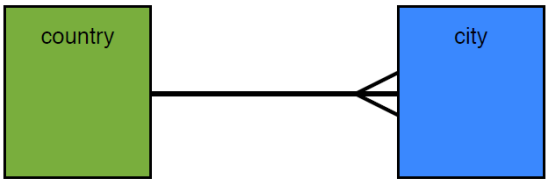
The table below is not only a better solution to this particular problem, but it's the best way to handle almost all one-to-one relationships. I'm not prepared to say that distributing a one-to-one relationship across multiple tables is *always* wrong, but it's uncommon enough that if you're tempted to do it, you should sit down over a nice beverage and contemplate the maneuver.

Our revised table looks like this:



One-to-Many Relationships

A one-to-many relationship says that each entity of type X has many entities of type Y. A country has many cities. A store sells many products. A person has many ancestors. When diagramming a one-to-many relationship, we use a line that branches on the end, sometimes called a "crow's foot." The crow's foot signifies the **many** side of a relationship:



To read one of these diagrams, we read across from the table name to the shape of the line on the opposite end. It's not obvious, but we ignore shape of the line at its beginning (here, where it connects to the country table). Going left to right, then, we read, "a country has many cities":

In the other direction, going from right to left, again we ignore the shape of the line at its beginning. We read the name of the table and read leftwards to the shape of the line at its finish: "a city has one country":

Using the diagram, we can now build tables to hold that data. One rule of thumb when building tables from these kinds of diagrams is that ***the foreign key gets stored on the crow's foot side of the relationship.*** Here, that means that we'll take the primary key from the country table and stash it into the city table as a foreign key:

```
country
+------+---------------+
| code | country       |
+------+---------------+
| CA   | Canada        |
| US   | United States |
| FR   | France        |
+------+---------------+
```

```
city
+------+------------+---------+
| id   | city       | country |
+------+------------+---------+
|    1 | Avignon    | FR      |
|    2 | Paris      | FR      |
|    3 | Marseilles | FR      |
|    4 | Victoria   | CA      |
+------+------------+---------+
```

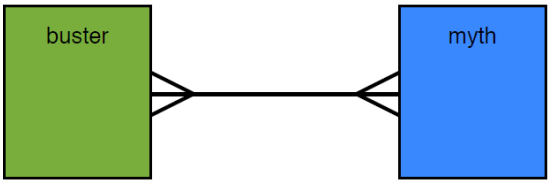
Although it's common to use a sequence of integers as primary keys, in this case I've used the two-letter country codes from the International Organization for Standardization (http://www.iso.org/iso/country_codes/iso_3166_code_lists.htm). These two-letter codes are guaranteed to be unique for all countries in the world, so they work perfectly well as primary keys.

Many-to-Many Relationships

In order to elaborate all the baroque subtleties of the many-to-many relationship, I'd like to work through three illuminating examples: 1) Many mythbusters bust many myths; 2) A pint of beer is a many-to-many relationship; and 3) a thesaurus.

Example 1: Many mythbusters bust many myths.

The popular Discovery Channel show, *Mythbusters*—hosted by Jamie Hyneman, Adam Savage, Tory Belleci, Kari Byron and Grant Imahara—puts urban myths to the test on a weekly basis. The hosts generally split into two teams to do their busting and so no one mythbuster ever works solo. Sometimes one episode busts several myths and so we have here the classic structure of a many-to-many relationship: many mythbusters bust many myths and many myths are busted by many mythbusters.



Let's build some tables and follow our one-to-many strategy of stashing foreign keys on the many side. Since we have here two tables with many relationships, we'll just pick one at random:

buster

id	buster	myths_busted
1	Jamie Hyneman	1, 2, 3, ...
2	Adam Savage	1, 2, 3, ...
3	Tory Belleci	14, 15, 16, ...
4	Kari Byron	14, 15, 16, ...
5	Grant Imahara	14, 15, 16, ...

myth

id	myth
14	Soda Cup Killer
15	Toilet Bomb
16	Superhero

Hmmmm. Not good. Since every mythbuster has worked on so many myths, we end up with a column that's just a comma-separated list of myths busted. We have the same problem even if we flip the responsibility of holding the foreign keys to the other table:

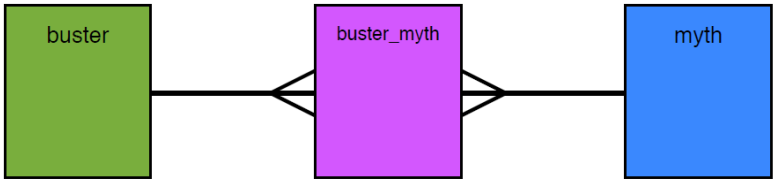
buster

id	buster
1	Jamie Hyneman
2	Adam Savage
3	Tory Belleci
4	Kari Byron
5	Grant Imahara

myth

id	myth	busted_by
14	Soda Cup Killer	3, 4, 5
15	Toilet Bomb	3, 4, 5
16	Superhero	3, 4, 5

Clearly, the rule of thumb that got us through the one-to-many examples is failing. Or is it? What if we could put another table in the middle? Wouldn't we break the many-to-many down into two one-to-many relationships? Yes, we would:



That middle table is called a **junction table** and its role is to help manage many-to-many relationships. In its simplest form, it's merely a 2-column stash of foreign keys (remember the rule: put the foreign key where the crow's foot is) but that's all we need in order to create two one-to-many relationships. In the process, it allows us to convert our comma-separated lists into individual table rows, which is one of the goals of data normalization.

buster

id	buster
1	Jamie Hyneman
2	Adam Savage
3	Tory Belleci
4	Kari Byron
5	Grant Imahara

buster_myth

buster	myth
3	14
4	14
5	14
3	15
4	15

myth

id	myth
14	Soda Cup Killer
15	Toilet Bomb
16	Superhero

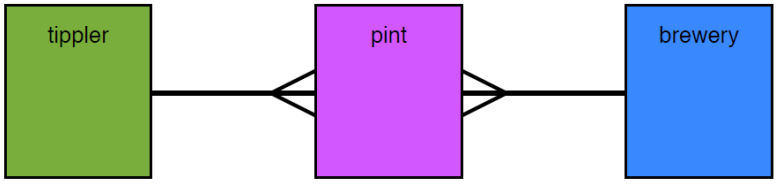
In this case, our junction table does not represent any recognizable real-world entity, but it still needs a name. One common strategy is to combine the names of the two tables it's joining and so I've called this junction table **buster_myth**.

You can call it anything you want, but just calling it **junction_table** is less descriptive than you'd want, especially if your database ends up with a bunch of these. Trying to decipher the difference between **junction_table_1**, **junction_table_2** and **junction_table_3** isn't something you want to do late at night.

Example 2: A pint of beer is a many-to-many relationship.

The junction table in our last example did not correspond to a real-world entity, but that's not always the case. In fact, that's not even the case most of the time. We see entities all around us every day that function as real-world junction tables. A pint of beer is just such a thing.

Let's imagine that Arthur, John and Georg hit the pub. Arthur like stouts, John prefers British bitters and Georg opts for German wheat beers. As the evening progresses, the gentlemen's minds expand, and they begin trying each other's brews. The result is a classic many-to-many relationship: many tipplers try many beers, and many beers were tried by many tipplers.



But because a pint is a real-world entity, we can store more information in our table than just foreign keys. Of course, the pint table still functions as a junction table, but its real-world existence means that we will probably think of it more as an entity than as a junction table.

Here's what our tables might look like:

tippler

id	tippler
1	Arthur
2	John
3	Georg

pint

tippler	brewery	cost
1	3	7.99
2	1	5.49
3	1	5.49
1	2	6.99
3	2	6.99

brewery

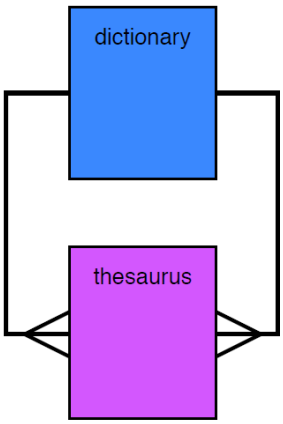
id	brewery
1	Guinness
2	Fuller's
3	Schneiderweis

We could store much more information on our **pint** table: the pub, the date and time, a beer review. The pint table transcends its lowly role as a junction table. It might even become the centrepiece of the whole database, containing more data and more columns than all the other tables combined.

Real-world entities act as junction tables all the time. A restaurant review manages a many-to-many relationship between restaurants and critics. A bird sighting table identifies a many-to-many relationship between people and birds. A movie enacts a many-to-many relationship between actors and directors.

Example 3: A thesaurus articulates a many-to-many relationship between words and other words.

The classic many-to-many relationship elaborates a relationship between three tables in the database. But in certain circumstances, a junction table can manage a many-to-many relationship with only one other table. A thesaurus is a good example. The thesaurus plucks a word from the dictionary (*relationship*, for instance) and points to other words elsewhere in the dictionary that mean roughly the same thing (*connected*, *associated*, *linked*, *coupled*, *allied*, *affiliated*, etc.)



dictionary

id	word
1	affiliated
2	allied
3	associated
4	connected
5	coupled
6	linked
7	relationship

thesaurus

word	synonym
7	1
7	2
7	3
18	9
23	87
37	4
72	94

One of the interesting decisions to make here is whether the relationship between words and their synonyms are one-way ("asymmetrical") or two-way ("symmetrical"). It makes sense that if *affiliated* is a synonym for *relationship*, then *relationship* must automatically be a synonym for *affiliated* too. That's a symmetrical relationship. In other words, the database designer needs to determine whether the one row (7, 1) is sufficient to cover both cases or whether the reverse, (1, 7), needs to be entered as well.

The best decision is probably to let the relationship work both ways. That makes the table smaller – half the size it would be otherwise, obviously. But that decision introduces new problems as well: any attempt to insert the pair (1, 7) into thesaurus should fail because (7, 1) is already there. And there's not a wholly elegant way to solve that problem.

To begin, though, you could add a multi-column unique index to the thesaurus table, but that prevents only new entries that are in the same order as a row already existing in the table. Unfortunately, you can still enter those values if you reverse the order. You could solve this dilemma by always sorting the two foreign keys so that the smallest number goes into the lefthand column and the larger number goes into the righthand column. (We can discount cases where the two numbers are the same because we probably don't want to log the fact that a word is its own synonym.) Sorting the numbers is a bit of extra work, but in tandem with the unique index, sorting the values before inserting would successfully prevent multiple entries in the table.

Relationships Redux

The topic of determining relationships between database tables is, I think, under-represented in the database literature and in database tutorials. I suspect that's because relationships seem to be so intimately connected with the data itself that writers and teachers assume that nothing useful can be said. But I don't think that's true. Relationships occur in fairly regular, repeating patterns and – as we've seen here – most of them resolve into one-to-many relationships anyway.

Even those elusive many-to-many relationships are built up from one-to-many relationships. Identifying them hinges on being able to see patterns between three different tables, some of which may or may not actually correspond to real-world entities.

Regardless, *relationships* are the heart and soul of *relational* databases and learning to see relational patterns amid the chaos of data is one of the most important skills you can develop as a database designer.