# Data Normalization, Denormalization, and the Forces of Darkness

a white paper

by Melissa Hollingsworth
melissa@fastanimals.com
http://www.fastanimals.com/melissa/

v. 0.4

This paper is in beta.  Feedback is welcome.

## Introduction

According to one SQL blogger, “normalization” and “denormalization” are the most common search terms bringing people to his site. Typing “normali” into Google brings up suggestions like “normalization” and “normalizing data” high on the list. If you actually search for “normalization,” your top search results include Wikipedia, overviews, tutorials, and basics. *All* the bases are covered!

Many of these overviews just skim the surface, never explaining why anyone would bother doing this. Some use examples which illustrate one principle while violating others, leading to confusion. Many use precisely the same examples for the higher forms, reworded slightly from each other, which might lead a more cynical person to wonder how well the writers grasp what they describe. If we can’t create our own examples, do we really understand the principle?

I do see a need to distribute basic information more widely. If it were common knowledge, we wouldn’t see so many people searching for it. We also wouldn’t see so many people complaining that it’s hard to understand.

This paper covers the basics in a manner which I hope is both accurate and easy to understand, and also slithers into intermediate territory. We need at least an intermediate level of knowledge before we can decide how far to normalize, the possible efficiency tradeoffs, if denormalization is needed, and (if it is) what denormalization methods to use.

## What Is Normalization?

“Normalization” just means making something more normal, which usually means bringing it closer to conformity with a given standard.

Database normalization doesn’t mean that you have have weird data, although you might. It’s the name for an approach for reducing *redundant* data in databases. If the same data is stored in more than one place, keeping it synchronized is a pain. Furthermore, if you can’t trust your synchronization process absolutely, you can’t trust the data you retrieve.

You may have heard that normalization is the enemy of search efficiency. This is kind of truish, but not really true. At the lower levels, imposing a bit of order on your chaotic heap of data will probably *improve* efficiency. At the higher levels you may see a performance slowdown, but it probably won’t be as bad as you fear and there are ways to address any problems you do find.

Let’s get to work on a data set.

I work for a company which sells widgets. We must track widget inventory, prices, order numbers, blah blah blah . . . . Lord, I’m falling asleep just writing about it.

You know what? You look trustworthy. I’m going to tell you the sinister truth. I’ve been retained by an organization called the Council of Light, which stands alone against the forces of evil and darkness which daily threaten to engulf our innocuous, daylit world.

The Council of Light wants me to put their data into a SQL database so that they can retrieve it more easily. They have lists of monsters, types of weapons, historical records of monster outbreaks, a registry of currently available monster fighters, and a great deal more. Unfortunately, it's all stored in a combination of dusty tomes and piles of index cards. I must design a database which models this information accurately and usefully.

Let's dive in.

A table of active monster fighters seems fairly obvious—full name, date of birth, formal training, date of joining the Council, fighting skills, and whatever else we come up with.

| LAST NAME | FIRST NAME | DATE OF BIRTH | TRAINING | DATE JOINED COUNCIL | FIGHTING SKILLS |
|---|---|---|---|---|---|
| Buttkicker | Brenda | 19850123 | Knights Templar Special Ops | 20050615 | blades, distance weapons, firearms, martial arts, munitions, stakes |
| Cutthroat | Cathy | 19760721 | Order of the Stake | 20020513 | blades, stakes |
| Killer | Karl | 19840812 | Demon Dojo | 20040914 | blades, distance weapons, martial arts |
| Martyr | Mel | 19790304 | Demon Dojo | 20000912 | blades, distance weapons, martial arts |
| Monsterbasher | Marvin | 19820118 | Order of the Stake | 20071120 | blades, stakes |
| Vampkiller | Valerie | 19810329 | School for Slayers | 20030502 | blades, firearms, stakes |
| And 50 others... | | | | | |

And I'd better have a table of those who provide support from the sidelines—students of the arcane, rogue fighters who sometimes partner with the Council, etc. Two tables so far.

| LAST NAME | FIRST NAME | DATE OF BIRTH | DATE JOINED COUNCIL | EXPERTISE |
|---|---|---|---|---|
| Clueful | Clarence | 19640723 | 19950312 | antique weapons, American folklore, American history, blades, martial arts |
| Elder | Ellen | 19440105 | 19680345 | world folklore, world history |
| Learned | Lucy | 19510821 | 19730103 | demons, fey, lycanthropes, undead |
| Rogue | Rodney | 19840323 | | firearms, munitions |
| Sage | Sally | 19710612 | 19980226 | lycanthropes, Romanian folklore, Romanian history |
| And 200 others... | | | | |

But wait. Many of the sideline personnel have at least some fighting ability. Clarence Clueful has retired from active fighting because middle age took away his edge, but he still knows how to use those weapons as well as discuss them. Rodney Rogue's index card is in the half-toppled pile of "other people" instead of "fighters" because he isn't a Council of Light member, but he does fight. Wouldn't it be simpler just to merge these, and have one table for all our human resources?

But then we have fields which are null for almost every entry, such as formal training, because only rows for full-fledged fighters need them. Those empty fields will lead to a performance penalty[i] which is unacceptable when designing for a secret society charged with keeping the world safe from the forces of darkness.

So are two tables the right choice? Isn't that a much worse performance hit, having to search at least two tables whenever we want to search all monster fighters?

Hmm. Maybe we should put this cowboy database design aside for the moment, to see if we can learn anything from the normalized approach.

## Normalization Forms

We usually speak of five normalization forms. This is a little misleading.[ii] The standard numbering is useful for discussion, though, since everyone knows what "Third Normal Form" or "Normal 3" means without needing an explanation. One might say the names are normalized.

## *First Normal Form*

The First Normal Form is really just common sense. It tells us two things: to eliminate duplicate columns, and that some column or combination of columns should be a unique identifier.

Eliminating columns which are flat-out duplicates is a no-brainer. Obviously you shouldn't create two different columns called "birthdate" and "date of birth," since a given individual is generally born only once.

Duplication may be a little less obvious than blatant copies, though. If my monster fighter table has a firstname field and a lastname field, it might be tempting to have a derived fullname field to speed up searching. Derived data is allowed at the Normal 1 level.

| LAST NAME | FIRST NAME | FULL NAME | DATE OF BIRTH | TRAINING | DATE JOINED COUNCIL | FIGHTING SKILLS |
|---|---|---|---|---|---|---|
| Buttkicker | Brenda | Buttkicker, Brenda | 19850123 | Knights Templar Special Ops | 20050615 | blades, distance weapons, firearms, martial arts, munitions, stakes |

But what happens if monster fighter Brenda Buttkicker gets married and takes her husband's name? Wouldn't you, if you were born with a name like "Buttkicker"? Now we need to update two fields.

We could rely on our application, and all other possible applications ever written by anyone else, to update both lastname and fullname. We could set up trigger functions to autochange fullname when lastname is changed, and hope nobody accesses the data through any other means. Or we could be normal and eliminate the duplication of the lastname and fullname columns by not having a fullname column, and combine firstname with lastname when we need a full name.

A "derivation" as simple as bunging two strings together is really just duplication, trying to be sneaky by wearing a Richard Nixon mask. Duplication, sneaky or otherwise, is disallowed by Normal 1.

The other requirement of the First Normal Form is that some column or combination of them be unique in the database. The birthdate field might seem like a good choice, since there are an awful lot of days in a span of several generations. But what if a pair of twins joins the Council? Also, as our team grows, chance makes it virtually certain that at least two were born on the same day. Consider the so-called Birthday Paradox—whenever 23 people are considered, the odds are better than even at that least two of them share a birthday, so when several hundred people of one generation are considered the odds of birthdate overlap are unacceptably high. firstname+lastname+birthdate might be a safe enough choice for uniqueness, at the Normal 1 level.

Why all this insistence on uniqueness at the most basic normalization level? Simple. We need a reliable way to retrieve any given single record.

## ***Second Normal Form***

Normal 2 includes all of Normal 1, and goes further. Normal 2 tells us that duplicate data should be removed and placed in a child table. For instance, I may make a table of known werewolf clans throughout history, starting with one in Rome.

| NAME | LOCATION | DATE DISCOVERED | DATE ELIMINATED |
|---|---|---|---|
| Dances with Death | Rome | 12940614 | 14020819 |

There's more than one Rome in the world, so we specify: Rome, Italy, Europe.

| NAME | LOCATION | DATE DISCOVERED | DATE ELIMINATED |
|---|---|---|---|
| Dances with Death | Rome, Italy, Europe | 12940614 | 14020819 |

Let's add more clans. Here's one in Athens, Greece, Europe. One in Cairo, Egypt, Africa. One in London, England, Europe. Hmm, we're racking up a lot of fields that say, "Europe." And here's one in Rome—heeeey, there was another one in Rome earlier in the list.

Locations can be duplicate data. There's no sense in having City, Country, and Continent columns when cities tend to stay put, but we still have to identify *which* Rome and *which* Cairo. We could cram all this into one column, with "Rome, Italy, Europe" as a string.

| NAME | LOCATION | DATE DISCOVERED | DATE ELIMINATED |
|---|---|---|---|
| Dances with Death | Rome, Italy, Europe | 12940614 | 14020819 |
| Family of Blood | Athens, Greece, Europe | 16930321 | 16980605 |
| Fury with Fur | Cairo, Egypt, Africa | 19040506 | 19750302 |
| White Fangs | London, United Kingdom, Europe | 17890817 | 18011225 |
| Zombies Are Lunch | Rome, Italy, Europe | 19540802 | 19680812 |

But sheesh, talk about your duplicate data. We're making it worse.

The smarter approach is to use a location identifier—perhaps modern country codes and the countries' postal codes,which are already designed to identify particular areas—and have a separate locations table which expands "20:11471" into the sixth district of Cairo.

| NAME | LOCATION | DATE DISCOVERED | DATE ELIMINATED |
|---|---|---|---|
| Dances with Death | 039:00123 | 12940614 | 14020819 |
| Family of Blood | 030:17675 | 16930321 | 16980605 |
| Fury with Fur | 020:11471 | 19040506 | 19750302 |
| White Fangs | 044:SW8 | 17890817 | 18011225 |
| Zombies Are Lunch | 039:00172 | 19540802 | 19680812 |

When we use a location ID instead of a human-readable string, we've also taken care of the problem of strings being entered inconsistently. "Philadelphia, United States of America, North America"? "Philadelphia, PA, U.S.A., North America"? Some other variation? Which is it? It's none of those; it's "001:19131" in a specific part of Philadelphia. By removing duplicate data into a child table, we've also taken care of the problem of sneaky duplicates which don't actually look alike. We also distinguish between different districts in large cities, and pre-solve the problem of what to enter when the Council finds a werewolf clan in London, Ontario, Canada.

Do you feel clever yet? Now we can retrieve data with confidence, knowing that searching one field for one string will return every werewolf clan which was centered in that area. If we want to search all of Egypt, we need only search for fields beginning with "020:". We might even want to go nuts and prepend a continent number, so that we can trivially search all of Africa.[iii]

Normal 2 also encourages us to make use of foreign keys. If I try to enter a new werewolf clan without a location ID, I should be at least warned and probably prevented. The same is true if I enter a location ID which isn't defined in the locations table. If I'm entering a new clan whose base of operations isn't yet known, I should use an appropriate reserved ID which is predefined in the locations table accordingly.

Removing duplicate data led us directly into a better system for categorizing that same data. Now we're really getting somewhere.

## Third Normal Form

Normal 3 includes all of Normal 2, and goes a big step further. It tells us that a given table should contain no information which is not directly related to the primary key.

In theory, that sounds like a no-brainer. Of course all the information in a row is related, or what would it be doing there? But think about the implications. One column is the primary key, and we should remove any columns which aren't dependent on the primary key. Every non-key field must provide a fact about the key.

Let's move on to a table of which weapons work well with which monsters. Our needs include chainsaws for zombies, bullets for werewolves, and stakes for vampires. Does composition matter? Absolutely, since vampires require wooden stakes and only a silver bullet will take out a werewolf. Does the manufacturer matter? Not really; any functional stake or chainsaw or bullet will do the job. "The stake is wooden" provides a fact we need to know about stakes when we're looking up weapons to use. "The stake was made by Oddball Weapons, LLC," applies only to some of our stakes, and isn't relevant to killing the nosferatu currently battering at our door.

| WEAPON | MATERIAL | MONSTER |
|---|---|---|
| axe | ANY | zombie |
| bullet | silver | werewolf |
| blade | ANY | zombie |
| blade | metal | vampire |
| blade | silver | werewolf |
| chainsaw | ANY | zombie |
| cross | ANY | vampire |
| garlic | ANY | vampire |
| stake | wood | vampire |
| sunlight | ANY | vampire |
| sunlight | ANY | werewolf |
| ANY | silver | werewolf |

We may have other weapons-related tables, such as a current inventory or a list of what vendors sell what. The manufacturer will be relevant to some of these, but it's not relevant to the table of which weapons to use on which monsters, and shouldn't be stuck in as an extra.

You can use one of your data fields as the primary key, if you know it's unique. But do you *know* it's unique? Social security numbers are supposed to be unique, but the U.S. government has issued duplicates by accident. We can still get away with combinations of fields at the Normal 3 level—although we're starting to see why we shouldn't—so perhaps weapon type combined with weapon composition would work.

## Fourth Normal Form

Normal Four includes Normal 3 and adds one more requirement: no multi-value dependencies. A record shouldn't contain two or more independent multi-valued facts about an entity.

Let's look at that table of fighters again.

| LAST NAME | FIRST NAME | DATE OF BIRTH | TRAINING | DATE JOINED COUNCIL | FIGHTING SKILLS |
|---|---|---|---|---|---|
| Buttkicker | Brenda | 19850123 | Knights Templar Special Ops | 20050615 | blades, distance weapons, firearms, martial arts, munitions, stakes |
| Cutthroat | Cathy | 19760721 | Order of the Stake | 20020513 | blades, stakes |
| Killer | Karl | 19840812 | Demon Dojo | 20040914 | blades, distance weapons, martial arts |
| Martyr | Mel | 19790304 | Demon Dojo | 20000912 | blades, distance weapons, martial arts |
| Monsterbasher | Marvin | 19820118 | Order of the Stake | 20071120 | blades, stakes |
| Vampkiller | Valerie | 19810329 | School for Slayers | 20030502 | blades, firearms, stakes |
| And 50 others... | | | | | |

Consider the Training and Fighting Skills columns. We as humans know that there's a pretty close relationship between these two things, but are they *guaranteed* related? If so, we don't need Fighting Skills at all because we can derive it from Training. If not, they're separate values (in the logical sense). The Council tells me that they're separate values, since the Order of the Stake sometimes drops the ball on training with blade weapons.

So. Let's pull them out.

First let's look at a one-to-many relationship. Each fighter attended one and only one training program, while each training program has many graduates.

| UID | LAST NAME | FIRST NAME | TRAINING |
|---|---|---|---|
| XXX | Buttkicker | Brenda | Knights Templar Special Ops |
| XXX | Cutthroat | Cathy | Order of the Stake |
| XXX | Killer | Karl | Demon Dojo |
| XXX | Martyr | Mel | Demon Dojo |
| XXX | Monsterbasher | Marvin | Order of the Stake |
| XXX | Vampkiller | Valerie | School for Slayers |
| XXX | And 50 others... | | |

I treat the fighter as the key, because the fighter is a "child" of the "parent" training program. If I treated the training program as the key, I'd either have multiple columns of fighter names for its graduates or I'd still have a separate entry for each fighter-program combination. Why borrow trouble?

Now we consider the set of fighting skills. This is a many-to-many relationship, so trouble is borrowing us. To keep the table short, we'll consider only Brenda and Mel.

| UID | LAST NAME | FIRST NAME | FIGHTING SKILLS |
|---|---|---|---|
| XXX | Buttkicker | Brenda | blades, distance weapons, firearms, martial arts, munitions, stakes |
| XXX | Martyr | Mel | blades, distance weapons, martial arts |

One fighter can have many skills. One skill can be had by many fighters. To avoid multi-value dependencies, we simply string them out down rows.

| UID | FIGHTER_ID | FIGHTING SKILL |
|---|---|---|
| AAA | YYY | blades |
| AAA | YYY | distance weapons |
| AAA | YYY | firearms |
| AAA | YYY | martial arts |
| AAA | YYY | munitions |
| AAA | YYY | stakes |
| AAA | ZZZ | blades |
| AAA | ZZZ | distance weapons |
| AAA | ZZZ | martial arts |

I can search to find out what skills a given fighter has, or what fighters have a given skill, with equal facility.

The names, which are now duplicate data, have been extracted into a child table. Each fighter has an assigned ID number. I'll also apply these FIGHTER_ID numbers in the table of training programs, table of biographical data, and whatever else we need. I can use the same ID number when looking up Cathy Cutthroat's fighting skills for mission assignment, her training program to find her fellow alums, or her birthday to plan her surprise party.

I've left the fighting skills as text for readability, but the duplication of data is pretty obvious. Fighting skills will probably also have ID numbers, unless we can guarantee that all "blades" are created equal and that there's nothing else we'll ever need to record about those.

Keen observers will note that we now require a guaranteed-unique ID in a single field. This isn't always mission-critical. In SQL, you can simulate it with views. Really, though, why *not* have a guaranteed-unique ID?

We now have a required primary key in a single field, with no more of this namby-pamby nonsense about some combination of fields being unique. A good practice is to have a column just to hold a unique numeric key. Another good practice is to let the database assign a number, via SQL's newid() function.

If you want IDs which are universally unique and not just unique to your table or database, another choice is to follow the UUID standard as defined in RFC 4122. Your programming language of choice probably already has a library/class/module/whatever to generate them (e.g., java.util.UUID).

Some databases, such as Postgres, support UUID as a native datatype. For others, just remember that it's 128 bits by definition and go from there. Some people simply store it as a CHAR(36), while others get fancier (e.g., stripping hyphens and performing hex conversions) to store in other formats.

A Universal UID isn't necessary to meet Normal 4, though. The UID need only be unique to the table, so if duplicates in other tables aren't a problem in your design, letting the database assign the ID is an admirably lazy technique.

Marvin+Monsterbasher+19820118 looks likely to stay unique, but I'm not convinced that stake+wood is good enough. I'm taking all of the Council's tables to Normal 4.

Let's look briefly at both approaches to a unique ID. If I choose the UUID approach, my primary keys will now all be something like

12345678-1234- 5678-1234-567812345678

and my other fields tell me some fact about that—starting with what the heck it is. As you can see, it's a bit much for a human to process visually or mentally. It need never be displayed in reports intended for human eyes, though.

If I let the database assign non-universal UIDs, the numbers will be smaller—probably small enough for humans to manage, so that people could memorize their own monster fighter ID numbers. If I want the UIDs human-usable and don't need them to be unique in the entire universe, that's probably the right approach. If any Council members complain about feeling like numbers instead of people, I'll just mumble something about the computer requiring it.

## *Fifth Normal Form*

Normal 5 doesn't come up often. At least in theory, any data set normed to Normal 4 usually conforms to Normal 5 without further work. There are those times, though, and the Council has presented me with some things I should consider.

There is no such thing as a wooden chainsaw, for obvious reasons. I can let my application carry the burden of preventing users from accidentally entering a wooden chainsaw as a new weapon type, or I can bite the bullet and tackle Normal 5. Also, in our monster fighters table, we have a near-opposite situation—any fighter who has finished one of the several anti-monster training programs is presumed competent in the particular fighting techniques taught by that program. I can let my application fill in duplicate fields of techniques, or I can normal up and conform to Normal 5.

A data set conforms with the Fifth Normal Form if and only if every join dependency in it is implied by the candidate keys.

Um, what?

When real-world constraints mean that some combination of allowed values is not a valid combination, that combination must be disallowed by the logical data structure itself in order to meet Normal 5. For instance, a real-world situation may be such that W=X means that Y can never equal Z, but the existing Normal 4 structure of the table doesn't disallow it because each field entry is valid. We could apply a constraint, but that's cheating. We're talking about the logical data design itself.

On the other side of the coin, a real-world constraint may mean that value A with value B always implies value C. The Normal 4 structure would duplicate value C for each such entry, even though it can be derived from the existence of values A and B. This isn't derived data in the usual sense—it's not A times B, for example. It's inferable because something in the real world tells us that it must always be that way.

To meet Normal 5, we must set up the tables such that Y=Z can never occur in the logical data design, and we must remove C on the grounds that it can be inferred from A and B.

Um. *What?*

It's hard to understand this abstractly, so let's talk about the Council of Light's weapon data. This is going to get detailed, but we don't have to consider every possible combination. That's the Council's job. Our job is to create a data model which accurately represents the data they'll put in.

Let's start with the wooden chainsaw problem. A monster is vulnerable to certain weapon-material combinations. We may choose to model this data by saying that it is vulnerable to some weapons and some materials. We derive its weapon-material vulnerabilities by checking its weapon vulnerabilities and its material vulnerabilities. Let's look at our weapons table again:

| UID | WEAPON | MATERIAL | MONSTER |
|---|---|---|---|
| XXX | axe | ANY | zombie |
| XXX | bullet | silver | werewolf |
| XXX | blade | ANY | zombie |
| XXX | blade | metal | vampire |
| XXX | blade | silver | werewolf |
| XXX | chainsaw | ANY | zombie |
| XXX | cross | ANY | vampire |
| XXX | garlic | ANY | vampire |
| XXX | stake | wood | vampire |
| XXX | sunlight | ANY | vampire |
| XXX | sunlight | ANY | werewolf |
| XXX | ANY | silver | werewolf |

We're seeing a fair bit of duplication. Consider the werewolf entries—all but one are for silver things. We have an ANY silver entry, but bullets are there separately because they're the weapon of choice, and we also have "blade" for consistency when searching for monsters who can be hurt by blades. This isn't needful duplication. Given that monster is werewolf and material is silver, we can derive the fact that the weapon will do damage in its usual manner.

Normal 5 tells us that we should remove any data which can be derived from other data—such as "silver [anything] hurts werewolves."

Is this universal? Is it something that will always be true? If so, this means that I split my weapons table into three tables. One is for weapon type and weapon composition, one is for weapon type and monster vulnerability, and one is for weapon composition and monster vulnerability.

| UID | WEAPON | MONSTER |
|---|---|---|
| XXX | axe | zombie |
| XXX | stake | vampire |

| UID | MATERIAL | MONSTER |
|---|---|---|
| XXX | silver | werewolf |
| XXX | wood | vampire |

| UID | WEAPON | MATERIAL |
|---|---|---|
| XXX | axe | metal |
| XXX | axe | silver |
| XXX | stake | metal |
| XXX | stake | wood |

One tells me that vampires are vulnerable to stakes. Another tells me that vampires are vulnerable to wood. The third tells me that stakes of wood are available. From the first two, we derive the fact that wooden stakes are handy anti-vampire weapons. Meanwhile, "wood+chainsaw" is coming up blank from the "weapon type and weapon composition" table.

What if someone creates a chainsaw with wood-tipped chain links specifically for anti-vampire use? That's pretty cool, and we have the case pre-covered. If we create an entry for a chainsaw whose relevant material is wood, it'll automatically come up as a possible anti-vampire weapon because of the wood vulnerability.

This leads to some problems, though. Vampires are vulnerable to the material ANY—but only if it's in the shape of a cross. What does that mean for our material-monster table? There aren't enough real-life material-monster constraints to make this a good data model, and so our previous table doesn't need to be split further to meet Normal 5—at least, not in that particular way. Still, humor me while I carry this a bit further.

Our hypothetical weapon-material table also can't represent reality as well as we might wish. Wooden sunlight, anyone? Are there any possible composition values for sunlight other than, well, "sunlight"? What's the weapon and what's the material?

But sunlight, although deadly to vampires, isn't really a weapon as such. You can't carry it around and deploy it. Perhaps it shouldn't be in our weapons table. We can create a separate table for environments, including those which are so hostile to the monster that they actively harm it.

*Nota bene: Applying Normal 5 breaks data up into smaller and smaller tables. Splitting up our data into such small and specialized tables forces us to think about edge and corner cases.*

Hmm. That's good to know. Our first attempt at further breakdown didn't work out so well, but let's follow this thought process further. What won't fit? *Why* won't it fit?

A moment ago we were thinking about crosses and vampires. From the most elaborate silver crucifix to a couple of sticks properly tied, any cross can hold a vampire at bay. Composition "ANY" is needed to represent that.

With werewolves, on the other hand, only the silver crucifix would be useful and then only if we can bash the monster on the noggin with it. In the werewolf case, it's the silver composition that matters and we'll need the semi-generic weapon "blunt instrument." The data model's failure taught us something about what we'll need for a successful one.

We could also think about axes. Only cold iron and cold steel work against evil fey, but that's a metal vulnerability. Anything sharp enough to cut flesh can dismember a zombie. A silver axe would be too soft to cut well but would still hurt a werewolf. Vampires can be cut by anything sharp but must be beheaded to kill them that way. A given axe could fall into any or all or none of the appropriate categories.

| UID | WEAPON | MATERIAL | MONSTER |
|---|---|---|---|
| XXX | axe | ANY | zombie |
| XXX | axe | gold | |
| XXX | axe | iron | fey, zombie |
| XXX | axe | metal | zombie |
| XXX | axe | silver | werewolf, zombie |
| XXX | axe | steel | fey, zombie |
| XXX | axe | wood | |

Hmm. Metal and iron aren't at the same level; most of those things *are* metals. And we aren't distinguishing between any old iron and the cold-worked iron needed to hurt the fey creatures. Perhaps what we really need is a hierarchy:

Everything -> inorganic -> metal -> iron -> cold

| UUID | KINGDOM | CONTAINS |
|---|---|---|
| XXXX | ANY | |
| XXXX | animal | YYYY, YYYY, YYYY |
| XXXX | energy | YYYY, YYYY |
| XXXX | **inorganic** | YYYY, YYYY, **YYYY**, YYYY, YYYY, YYYY |
| XXXX | vegetable | YYYY, YYYY, YYYY |

| UUID | INORGANIC | CONTAINS |
|---|---|---|
| YYYY | ANY | |
| YYYY | crystal | ZZZZ, ZZZZ, ZZZZ, ZZZZ, ZZZZ |
| YYYY | **metal** | ZZZZ, ZZZZ, ZZZZ, **ZZZZ**, ZZZZ |
| YYYY | other | ZZZZ, ZZZZ, ZZZZ, ZZZZ, ZZZZ |

| UUID | METAL | TYPES |
|---|---|---|
| ZZZZ | ANY | |
| ZZZZ | copper | |
| ZZZZ | gold | |
| ZZZZ | **iron** | AAAA, **BBBB** |
| ZZZZ | silver | CCCC, DDDD |
| ZZZZ | steel | AAAA, BBBB |

Now we're really getting somewhere. Mineral refers us to metal, which refers us to silver, which can refer us to blessed or unblessed if necessary. Note that I've traded in my table-unique IDs for universally unique IDs. The big win is that "energy, sunlight" and "mineral, metal, silver" can each be listed as a monster vulnerability, even though they're on different hierarchical levels.

For we designing the database, the task isn't to think up every possible combination. That's the Council's job. Our job is to think of what *kinds* of things we need to record, and examples which push the boundaries of standard rules help us do that.

You've probably noticed that we have a one-to-many or many-to-many relationship a single row, listing multiple children in a single field. I left it that way for readability, but a truly normalized table would consist of nothing but IDs until the leaf level(s). A slice might look like this:

| UUID | PARENT_ID | CHILD_ID |
|---|---|---|
| XXXX | AAAA | QQQQ |

| | | |
|---|---|---|
| XXXX | AAAA | RRRR |
| XXXX | BBBB | QQQQ |

Woo. We're rapidly moving further away from the human-readable. We may not choose to do it precisely this way.

Whether or not we choose to implement Normal 5, thinking it through is leading us to some good realizations. A team of vampire fighters, trapped in a cellar at midnight, would not be amused to learn that sunlight is a useful option. A team of werewolf hunters, on the other hand, may be pleasantly surprised to realize that the silver part of their vampire-hunting kit can be repurposed.

I don't recommend Normal 5 for most real-world situations even when it applies smoothly, since it doesn't allow for exceptions. Those real-world constraints on which we relied may turn out not to constrain universally and forever. What if an item is made of silver but is so small and light that it can't conceivably hurt a werewolf?

"Anything silver hurts werewolves" and "all crosses hurt vampires" may be true in theory, but the real world seldom matches theoretical constraints. A thrown dime isn't a useful anti-werewolf missile. A cross made by crossing your forefingers will only slow a vampire down if he's laughing too hard to attack you. A database for real use must let us store data for real cases, including exceptions to general rules.

Still, even without implementing it, trying to apply the Normal 5 process has shown us that improbable combinations should issue warnings, that some things we think of as weapons should be treated differently from others, that materials of composition are a hierarchy rather than a simple list, and that weaponry will require a great deal more attention than we realized back in Normal 4.

By using Normal 5 as a thought exercise, we get many of its planning benefits without its drawbacks of numerous microtables and performance slowdowns.

### Sixth Normal Form

Hey, didn't she say "five" earlier? Yeah, but I'm going to mention this for completeness.

You may have heard of a Sixth Normal Form. This term is used in two different ways, and both of them are for academic use. It's intended to break down data relations until they've been reduced to irreducible components. If you're designing real databases for real use, ignore it. Both of it. Please.

## Denormalization

Denormalization doesn't mean not normalizing. It's a step in the process. First we normalize, then we realize that we now have hundreds or thousands of small tables and that the performance cost of all those joins will be prohibitive, and then we carefully apply some denormalization techniques so that our application will return results before the werewolves have overrun the city.

## Deciding How Far to Normalize

I believe that running all the way up to Normal 5 can be a useful thought exercise for any database which is being designed instead of just tossed together. But when it's time to do the grunt work of mapping all the actual tables and columns, we first decide how far we need to go.

Even a spreadsheet should comply with Normal 1. There's really no excuse for duplicate columns and non-unique entries. If your hardware can't handle that and your data is understandable without it, do you need a real database at all? Searching a directory of text files might serve you better.

Normal 2 is a good idea for all but the smallest and most casual of databases. If you coordinate a group of half a dozen fighters who only battle zombies and operate entirely in Boise, then duplicate data may not be a problem and you may be able to ignore Normal 2. Everyone else should conform to it.

Normal 3 is necessary, at least in the long term, for any business operation or any other operation where the data is critical. It may seem a bit hardcore, but it's vital for any organization which has more data than it could reasonably process by hand. That's the crossover point at which we must treat the database as a vital part of the organization's mission.

Normal 4 isn't always necessary, but if you've done Normal 3, why *not* have a guaranteed-unique ID for each record? Unlike some of the other steps, this one is actually a performance win. It's faster to look up one field than to check several to see if the combination's unique, and you still have the option of searching by other means. Stringing data down rows instead of across columns may lead to performance slowdown, but there are ways to address this, and we'll look at some of them. Normal 4 is an excellent idea for any organization with large amounts of data—again, because keeping the data straight is vital to the organization's mission.

Normal 5 is rarely necessary. It reaches into rarified heights where the purity of the data model is more important than the real-world usability. Except in rare cases where permanent and inflexible real-world constraints match the logical data design beautifully, you'll probably end up needing to denormalize anyway. Think it through, because you'll realize some good stuff, but don't worry about meeting it to the letter. If you did Normal 4 correctly, you probably already met it.

To summarize— at least Normal 2 for tiny organizations, and at least Normal 4 for any operation where the data's critical.

I'll be conforming to Normal 4 for the Council of Light's final database design, although thinking about Normal 5 did lead me to some realizations I'll incorporate into my design.

## Should I Denormalize?

Since normalization is about reducing redundancy, denormalizing is about deliberately *adding* redundancy to improve performance. Before beginning, consider whether or not it's necessary.

- Is the system's performance unacceptable with fully normalized data? Mock up a client and do some testing.
- If the performance is unacceptable, will denormalizing make it acceptable? Figure out where your bottlenecks are.
- If you denormalize to clear those bottlenecks, will the system and its data still be reliable?

Unless the answers to all three are "yes," denormalization should be avoided.

Unfortunately for me, the Council of Light has some pretty old hardware and can't or won't upgrade to newer and more efficient SQL software. I'd better get to work.

## *Denormalization Strategies*

### Materialized Views

If we're lucky, we won't need to denormalize our logical data design at all. We can let the database management system store additional information on disk, and it's the responsibility of the DBMS software to keep the redundant information consistent.

Oracle does this with materialized views, which are pretty much what they sound like—SQL views, but made material on the disk. Like regular views, they change when the underlying data does. Microsoft SQL has something called indexed views, which I gather are similar although I've never used them. Other SQL databases have standard hacks to simulate materialized views; a little Googling will tell you whether or not yours does.

### Database Constraints

The more common approach is to denormalize some carefully chosen parts of the database design, adding redundancies to speed performance. Danger lies ahead. It is now the database designer's responsibility to create database constraints specifying how redundant pieces of data will be kept consistent.

These constraints introduce a tradeoff. They do speed up reads, just as we wish, but they slow down writes. If the database's regular usage is write-heavy, such as the Council of Light's message forum, then this may actually be worse than the non-denormalized version.

### Double Your Database Fun

If enough storage space is available, we can keep one fully-normalized master database, and another denormalized database for querying. No app ever writes directly to the denormalized one; it's read-only from the app's point of view. Similarly, no app ever reads directly from the normalized master; it's write-only from the app's point of view.

This is a good, if expensive, plan. The fully normalized master database updates the querying database upon change, or the querying database repopulates itself on a schedule. We no longer have slower writes, and we still have faster reads. Our problem now becomes keeping the two database models perfectly synchronized. That itself takes time, of course, but your particular situation may be such that you can do this.

Most commonly, the denormalized version populates itself by querying the normalized master on a controlled schedule. It will thus be the only thing which ever reads from the normalized master.

The danger is that data returned from the denormalized query base may be inaccurate if the data has changed in the normalized master but the denormalized query database hasn't repopulated itself yet. If your data doesn't generally change in a heartbeat, this may be acceptable. If the future of all humankind hinges on the currency of your data, this approach may not be acceptable.

Another possibility is for the normalized master to change the denormalized query database whenever the normalized master is changed. This requires a complex set of triggers interfacing the normalized and denormalized data designs, and the trigger set must be updated when the denormalized design is altered. The need to handle triggers will slow down writes, so this isn't a good option for write-heavy applications, but the upside is that your two databases are much less likely to be out of synch.

Space requirements often prohibit this strategy entirely.

### Let the App Handle It

We add redundancies to our database, but let the application worry about synchronizing redundant fields. We avoid the performance hit on writes, but now we're relying on a different piece of software to do the right thing at all times.

If this is the choice, document it thoroughly and unavoidably. Make a stern warning and a list of the redundancies the first thing any future developer will see. It may also help if you include a threat about noncompliant apps contributing to a future in which the world is overrun by creatures of the night.

## *Denormalization Tactics*

All right, so those are the basic approaches. All of them have one thing in common, which is that they deliberately introduce redundancies. How do we decide what redundancies to introduce?

Simple. We find out exactly what is causing those performance problems we noted—you know, the ones which must be present in order for us to consider denormalization. Then we introduce redundancies which will make those particular causes go away.

The two most common bottlenecks are the cost of joining multiple tables and, on very active systems, the activity on a particular table being too high.

### Pre-Joined Tables

Is it the joins? Which ones?

Perhaps we find that our users do a lot of queries which require joining the vampire table and the zombie table. If so, we could combine them into a single pre-joined table of the undead.

If our users often search all available personnel, joining combatants and former combatants and noncombatants, then we may want to revert to our idea of keeping all Council members and other helpers in a single personnel table.

This table is essentially the materialization of a join, so we populate if from the original tables either on a schedule or via triggers when changing the originals.

## Mirror Tables

If a particular table is too active, with constant querying and updating leading to slowdowns and timeouts and deadlocks, consider a miniature and internal version of "Double your database fun." You can have a background and foreground copy of the same table. The background table takes only writes, the foreground table is for reading, and synchronizing the two is a low-priority process that happens when less is going on.

The obvious problem is that the data you're reading has no guaranteed currency. In fact, given that we're doing this because a table is particularly active, it's fairly likely *not* to be current. It's something to consider when the data in question doesn't require up-to-the-second currency, though.

## Report Tables

Each day, between sunset and midnight, the Council's system is pummeled with queries on how to take out particular monsters. There's a lot of joining of weaponry tables, *and* many users are making the same queries to the same tables.

I can address this by making redundant tables just for these extremely common reports. My table on how to take out a vampire will be modeled after the desired report rather than around the logical data design, and might look something like this:

| UID | WEAPON | HOW TO USE | EFFECT |
|---|---|---|---|
| XXX | cross | expose monster to it | monster retreats in pain |
| XXX | stake, wood | stab monster through the heart | kills monster |
| XXX | sunlight | expose monster to it | kills monster |
| XXX | water, fresh, running | put between self and monster | barrier to monster |

Except for the UID for internal use, this table contains exactly what the user wanted to know and is therefore extremely fast to query.

The report table doesn't take writes from apps. It updates itself from the logical data design tables, regularly during off-peak hours and during peak hours as system resources permit.

## Split Tables

We may have distinct groups of users who query tables in distinct ways. Remember our table of werewolf clans? Monster fighters and coordinators may be interested only in clans which haven't been exterminated, while historians studying the effectiveness of extermination methods may be interested only in those which have. I could split that single table in two. The original table could also be maintained as a master, in which case the split tables are special cases of mirror tables (above) which happen to contain a subset instead of a complete copy. If few people want to query the original table, I could maintain it as a view which joins the split tables and treat the split tables as masters.

| UID | NAME | LOCATION | DATE DISCOVERED | DATE ELIMINATED |
|---|---|---|---|---|
| XXX | Dances with Death | 039:00123 | 12940614 | 14020819 |
| XXX | Family of Blood | 030:17675 | 16930321 | 16980605 |

| UID | NAME | LOCATION | DATE DISCOVERED |
|---|---|---|---|
| XXX | Fangs and Fur | 01:45701 | 20091001 |
| XXX | Howl | 044:SW1 | 20100105 |

That's a horizontal split, pulling rows out into different tables. For other needs we might do a vertical split, keeping all the same rows/keys but with each table having separate sets of columns of information.

This is only worthwhile when there are distinct kinds of queries which are, in effect, already treating the table as more than one table.

## Combined Tables

Instead of splitting one table into several, we might combine several into one. If tables have a one-to-one relationship, it might speed performance to combine them into one table even if that isn't part of the logical database design. We might even combine tables with a one-to-many relationship, although that would significantly complicate our update process and would likely work out to be a net loss.

For instance, I might combine the table of monster fighters with the table of training programs on the grounds that there are duplicate columns (school names, skills taught/learned). That could be useful for high-level coordination, which might require tracking current munitions training programs by evaluating both munitions programs and the fighters rated as skilled in munitions. A join would serve, but we might go with a combined table if we find that we're getting many such queries.

Combined tables differ from joined tables (above) in that they already share some data in some relationship, while any two tables which are frequently joined are potential candidates for joined tables. They also differ in that a joined table usually populates itself from the co-existing normalized tables, while a combined table will usually be the master copy.

So what's the difference between splitting tables and never combining them in the first place? Or between combining tables and just not splitting them? Hey, I never said this was a science. It's more of an art. Or possibly a craft. Whatever it is, it's definitely not for sissies. The redundancies you choose to introduce must be based on the particular bottlenecks in your own specific data set.

## Index Tables

These are nice for searching hierarchies. Remember how Normal 5 taught us that weapon composition is actually a hierarchy of materials rather than a simple list? Well, now we're stuck with searching that hierarchy by searching all the leaves under particular parent nodes and combining with a bunch of union statements.

Or are we? We could make a metatable consisting of nothing but parent-child pairs and any data which would speed searching. Like maybe this:

| UUID | PARENT_ID | CHILD_ID | LEVEL | LEAF |
|---|---|---|---|---|
| XXXX | AAAA | BBBB | 1 | N |
| XXXX | AAAA | CCCC | 1 | N |
| XXXX | BBBB | QQQQ | 2 | Y |

This table gets large fast, but it's all numeric and Boolean, so searching is a breeze. It's simply an index for that tree of materials we made. Technically it's derivable data (below), but this is a special case which is almost always worthwhile when more than a minimal hierarchy must be searched.

It can repopulate itself on a schedule, or changes to the main databases can trigger changes in the index table.

## Repeating Groups

Let's say that my table of monster fighters' skill eventually shook out like this:

| UID | FIGHTER_ID | SKILL_ID | RATING |
|---|---|---|---|
| XXX | AAA | 10 | 5 |
| XXX | AAA | 15 | 3 |
| XXX | BBB | 12 | 4 |
| XXX | BBB | 23 | 3 |
| XXX | BBB | 17 | 4 |

This can hold an infinite number of skills. I could instead choose to compress rows, stringing information across columns instead of down rows:

| UID | FIGHTER_ID | SKILL_ID_1 | RATING_1 | SKILL_ID_2 | RATING_2 | SKILL_ID_3 | RATING_3 |
|---|---|---|---|---|---|---|---|
| XXX | AAA | 10 | 5 | 15 | 3 |  |  |
| XXX | BBB | 12 | 4 | 23 | 3 | 17 | 4 |

Is this a good idea?

Well, it reduced the number of rows to search. The one-row-per-fighter structure seems like it would be efficient, since I won't normally need to do operations within a single row. The data in the new row format is often accessed collectively. Those are all plusses.

On the minus side, I've artificially limited the number of skills a fighter can have. There's no stable and predictable number of fighting skills, so putting a limit on it is a big minus. Even if I make the number unreachably huge, now I've got a table with tons of nulls.

This could be a good idea for data where there's an actual hard limit on the number of fields, though. If I'm making a report table (above) for monster activity within the past 12 months, a limit of 12 on the months is quite reasonable.

## Derivable Data

In rare cases, it's the cost of doing a calculation which is prohibitive. These are usually cases in which the algorithm for calculation is enormously complex and/or is itself dependent on expensive queries. In such cases, it might be worth storing value C even though value A with value B always works out to value C.

Changes to A or to B can trigger a function to rederive C. If C needs to be retrieved often, this can be a lot cheaper than rederiving it every time it's needed.

Index tables (above) are derivable data, but they're different in function. Those are for internal use only, containing no data that anyone is likely to want to use directly. These are derivable data which might be presented to a user.

#### Redundant Data

Yes, plain old duplicated data. A violation of Normal 1.

If we get a large number of queries which take most of their data from one table but have to join with a single column from another table, it *may* be worth duplicating that one column in the first table. This is almost never a good idea, since it goes against the basic principal of normalization and brings back all the problems we were trying to avoid in the first place. It may be worth a shot if normal database access is downright unbearable and some other constraint makes a join table (above) unacceptable.

### *Denormalization in Hyperspace: Star Schemas, OLAP Cubes, and Other Weirdness*

These are elegant, but a great deal of design work because they require math that most of us never learned. Modeling data into cubes? It sure sounds interesting, but it also sounds suspiciously like one of those approaches where the approach turns into the real project.

Lacking a Ph.D. in mathematics, I'm as likely to make things worse as I am to make them better. These things are fun to read about, but not suitable for most real-world deployments. If you think you're up for the task, Wikipedia is either a great starting place or a much-needed reality check.

## *Conclusion*

How does this help me with my original problem? I started with one question: Should all Council members be in one table, or should fighters and noncombatants be treated separately?

Well, I'd just about concluded that I should have several separate tables for people in different roles and use a Postgres hack to simulate a materialized view for searching all personnel at once. That will avoid both the join performance hit and the many-nulls performance hit. After going through the normalization and denormalization process, I see that that's the best answer.

But then I realized that the Council of Light never actually agreed to pay me. Tightwads. If they can't cough up a few bucks to save the world more efficiently, they can keep poring over their dusty tomes forever, and I don't care if all Hell breaks loose.

Um. Perhaps I should rephrase that . . . .

## Endnotes

i SQL nulls are just plain unintuitive. SQL has an odd aggregation of behaviors with respect to its nulls. It's never safe to extrapolate from what you know about them to what they ought to do in other cases. Other white papers have been written entirely on SQL's nulls.

Here are a couple of nutshell reasons why using lots of null values is not a big performance win: A null takes up the entire width of the column, so that storing the string "NULL" would actually be faster in many cases. If you use an IS NOT NULL condition, your SQL performance will be normal, but if your search allows for null values—as in our search for all monster fighters, regardless of training or active duty status—then performance slows measurably.

At least one major SQL vendor recommends never using the standard-mandated null, and always fudging via some other reserved value.

ii There's a "three and a half," better called the Boyce-Cobb standard, which is a little more restrictive than the Third Normal Form. Anything which complies with the Fourth Normal Form usually conforms to the Fifth Normal Form automatically, without further design effort, leading some to wonder if they really deserve different whole integers. There are also two different Sixth Normal Forms. We'll ignore the weirdest stuff and stick to the standard five; you can always do your own further research if you're really interested.

iii I'm using this colon-delineated format because country codes and postal codes already exist and are standardized. I see no sense in reinventing the wheel. As long as I'm keeping them in one field, a non-numeric delineator works best. This system is guaranteed unique in theory, since each country has a separate country code and each country bears responsibility for keeping its internal postal codes unique. I'd have to reserve values for "location unknown" and for remote locations without postal codes.