1 Robust Database Design

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation." —Frederick P. Brooks, Jr.

This tutorial will describe concepts behind designing database schemas. It is most applicable to relational databases, but can just as easily be applied to object databases.

Over the course of this tutorial we will build a simple database that might be used in an online store—to track customer purchases.

2 Why is it so difficult?

Databases organize and store information. However, before they can do that, we, as in humans, need to provide them with the structure for that information. That is, we, as in humans, need to figure out the structure ourselves—and this is where we, as in humans, very often get stuck and/or make lots of wrong decisions.

You must have noticed my emphasis on humans in the last paragraphs. Most humans have a problem when it comes to organizing their own thoughts! Not to mention organizing their thoughts on how some abstract information should be organized. As you can imagine, the process is full of pitfalls (and usually lots of frustration).

In addition to the essential difficulty of using our brains, the database usually stores data in a rigid format, with rules, and structures, and whatnot. However, the information that's in the database comes from the real world, which might not abide by the same rules the database imposes on it.

It is hard to make a good conceptual design. It is even harder to make that design work with reality.

3 Where does it all start?

Database design (as well as any sort of design) starts with requirements. We need to know what we are building, what information we are dealing with, and how we are dealing with it.

This is not a document on requirements gathering (which is a surprisingly difficult activity in itself), so we'll just skip along and assume you know what you're building and what information you're dealing with (and what you need to do with that information). Anyway, the process of defining the database schema starts with objects.

3.1 Objects

Knowing the information your system will deal with, you can start on the process of identifying objects. These are similar, but not quite the same objects as in Object Oriented Programming. These objects are distinct entities that exist by themselves. Usually, if you

can touch it, it's an object. If you can describe something without referring to anything else, then that's usually an object too. Objects tend to exist for extended periods of time.

Make a list of any objects you can think of. Be careful not to include things that cannot exist by themselves.

So, starting with the design of our online store database, a "`CUSTOMER`" might be an object; a "`PRODUCT`" might also be an object. But "`PRICE`" is not an object, because it cannot exist by itself—it needs to be a "`PRICE`" of something (and that something will very likely be an object).

So a preliminary list of objects for an online store might be:

- `CUSTOMER` - The person who purchases something.
- `PRODUCT` - The product a customer might want to buy.

This may be all there is, but as we move along, we might want to add something later on. Design is a recurring process, and all too often we need to go back and reexamine what we have and what we don't have.

One thing to note at this point is that we are not dealing with SQL, or any sort of implementation details. Once we are happy with the design, implementing it is easy—so leave off writing SQL until later.

3.2 Events

Now that we have a list of objects we can start figuring out how they interact. What happens when a customer makes a purchase?

An event is an interaction between one or more objects at a particular time. Events cannot exist by themselves (otherwise they'd be objects). Events must record the time of their occurrence—to differentiate themselves from other similar events but at different times. Of course they can also record other information about the occurrence.

Extending our previous example, in an online store, a "`PURCHASE`" is an obvious event. There might be others, and if we think of something, we'll add it later.

If we are unsure whether something is an object or an event, you can see if it requires a time-stamp. If something occurs at some time, then it's very likely an event. This does not mean that you cannot have a timestamp on an object—objects might have a creation time.

Now that you have a list of all objects and events, let's move on to properties.

3.3 Properties

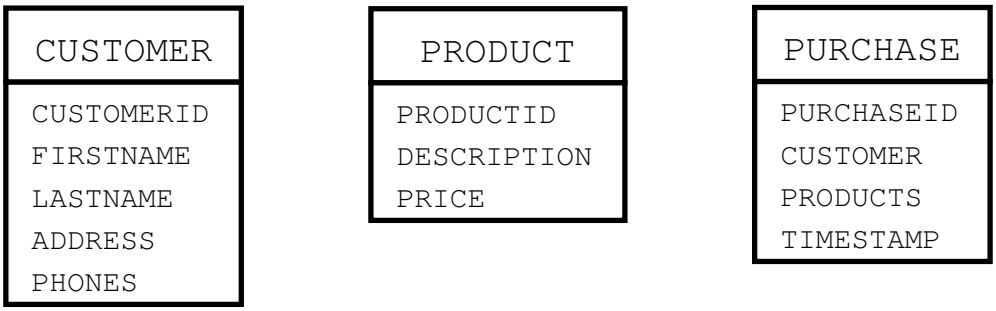
Properties are the details or characteristics of our objects and events. A customer objects might have a "`FISTNAME`" and "`LASTNAME`", just to name a few. Events also have properties: the time they occurred!

Let us go through our objects and events and add details to them:

Obviously we can add more details (like product "color", etc.), but let's leave it at that (no need to overcomplicate the example). Notice that `PURCHASE` has the time of purchase, *and* who made the purchase (the `CUSTOMER`) *and* what they bought (a bunch of `PRODUCTS`).

Ok, now that that's over, here are a few rules: Every object and event *must* have a unique ID. Whenever we say a `PURCHASE` event "has" a `CUSTOMER`, what we mean is that `PURCHASE` has a `CUSTOMERID` that uniquely identifies a particular customer.

More detailed view of our tables (yep, I said the T word):

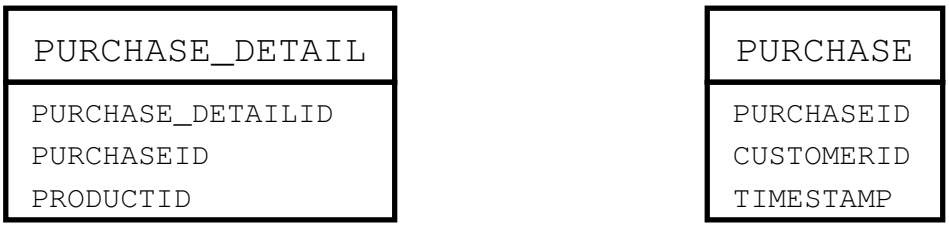


That's more like it. Now it's clearer that `PURCHASE` only has an ID of a `CUSTOMER` and not the actual `CUSTOMER`.

3.4 Repeating Properties

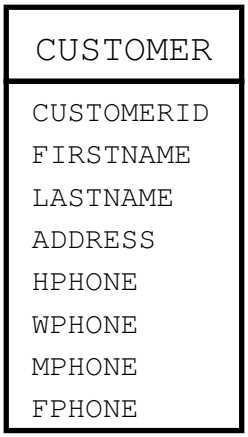
What probably bugged you (or should have bugged you) from the previous example is that we have this "`PRODUCTS`" property in `PURCHASE` table. No, we cannot store a number of products just like that. We need to break that up into separate tables.

The key to the concept is that we introduce a `PURCHASE_DETAIL` table, which will contain details about individual products. The table would look like this; along with the new `PURCHASE` table:



Anytime we need to know which products were purchased we just look them up in PURCHASE_DETAIL table using PURCHASEID of the PURCHASE. Neat, isn't it?

Now, not all repeating properties deserve such an aggressive treatment. For example, some can be fixed by simply listing several copies of a particular field. Notice that in our CUSTOMER table we have PHONES (plural). How can we fix that without introducing another table? I think you got the idea—we simply list different phones that a CUSTOMER may have:



We simply replace "PHONES" with HPHONE (for Home Phone), WPHONE (for Work Phone), MPHONE (for Mobile Phone), and FPHONE (for Fax). If a customer doesn't have one of those, we just set it to NULL. No big deal. A bit simpler to deal with than with a separate table—unfortunately we can't always do that as is illustrated in the PRODUCT_DETAIL example above.

4 Almost Done!

The above illustrates some basic concepts that you will more than likely encounter on *every* project you ever come across. The remainder of this tutorial will illustrate more contrived (and sometimes very useful) techniques to solving some less than common problems.

4.1 Property Value History

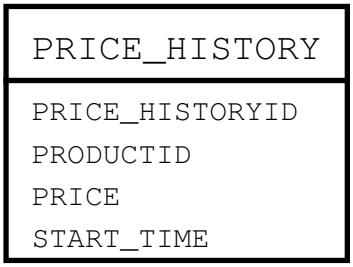
A somewhat rare, but very interesting requirement is the need to maintain the value history of a particular property. For example, our PRODUCT table has a PRICE. What if a product goes on *sale*? Purchases made before the sale must be charged the non-sale price, similarly, the sale price must be applied to purchases made during a sale.

There are several ways of handling this requirement. We can maintain a copy of the sale price along with the PURCHASE_DETAIL for that PRODUCT. Then you just change the PRODUCT.PRICE at will, and have CUSTOMERs purchase at the current prices. This design, while workable, is a bit too tedious when you need to analyze price histories. In order to observe price history you need to sift through PURCHASEs.

What if you had a sale for some item, and nobody bought it; how do you know the price was even lowered? (and by how much?).

When faced with such a situation, a good option is to maintain the price of an `PRODUCT` along with the `PRODUCT`. So we setup `PRODUCT` to look like this:

And move on to apply our Repeating Properties technique; which produces:



Notice that in addition to all the Repeating Properties things, we also added "`START_TIME`". This signifies the start of this price. So to find the current price, all we need to do is look for last `PRICE_HISTORY` item.

(Note that you can consider a price change as an event that might happen on the `PRODUCT`, and in the end, you'll end up with more or less the same table layout).

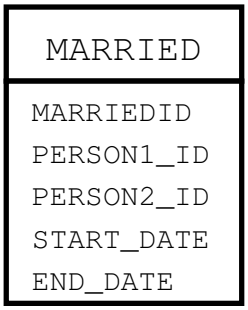
4.2 Object Relationships

Another relatively common situation is when objects have relationships to other objects. Consider a database with people objects—where `PERSON` objects may have family relations to other `PERSON` objects.

These types of relationships are very similar to events—except they are long term. Events happen at some instant in time, and go away. Object relationships are also formed at some time (so you might have a start date), and may be destroyed at some future time (so you might have an end date).

Think of a "`MARRIED`" or "`PARENTOF`" relationships, etc.

You also must be careful not to duplicate information. For example, if you define `MARRIED` as:



Then when you want to find if "John" is married to "Jane" do you have "John" being `PERSON1_ID`, or `PERSON2_ID`? Do you have a two way relationship—have two entries of `MARRIED`, in effect saying Jane is `MARRIED` to John, and John is `MARRIED` to Jane). Either way, you are making tradeoffs. If only one way relationship, then you need application logic to handle the reverse case.

Another alternative that avoids this issue is to implement these Repeating Properties as a separate table (as opposed to just listing two `PERSONID`s in the `MARRIED` table). Either way, you are making tradeoffs (in this case, speed of accessing an extra table).

Tip: A good approach is to go with the cleanest design that avoids data duplication. During design, don't worry about the number of tables you have.

Another important thing to realize is that Object-To-Object relationships cannot be required. An object can exist all by itself! If you find yourself needing required relationships, then you need to reconsider what you're treating as an object and what as properties of that object.

5 Optimization

DON'T START OPTIMIZING UNTIL YOU'RE HAPPY WITH THE DESIGN!

Relational databases are beautifully designed. Your design should work, and be just plain beautiful (at least to you). Most optimization tends to increase speed but at the expense of the beauty of design. If you prematurely start optimizing, you'll just end up with a poorly designed mess.

Some techniques: adding indexes in appropriate places, reducing number of tables, introducing redundancy (there are a few others, but mostly they all revolve around these three).

5.1 Adding Indexes

In improving performance, the thing to try first (before you go and break your design) is to add indexes. These can provide an enormous boost in performance.

(Indexes are special database files that contain column values in sorted order (B-Trees), allowing you to easily find a record in logarithmic time).

Just consider, how long would it take you to find a name in an unsorted phone book? Many times, databases are faced with just such a challenge. An index provides a sorted view of the data making data lookup a fairly trivial disk access (as opposed to looking through millions of records, the database may only look at a few to find a record).

If your database is sluggish, adding indexes on appropriate fields will more than likely improve performance.

Now, what are these appropriate fields (or columns)? If at any time you are doing a search on the value of some field, then you've identified a good candidate for an index.

Most primary and foreign keys should also be indexed (and most databases will actually do that by default). If you ever do searching by dates, then index those too.

Adding an index that you don't need will not decrease `SELECT` performance—the speed at which you retrieve records, but may severely impact `INSERT`, `UPDATE` or `DELETE` performance,

because indexes need to be inserted and deleted along with the record. Watch out for these tradeoffs.

Keeping that in mind, you should not go crazy and add indexes on everything without thoroughly understanding why you need it there. While indexes increase retrieval performance, they also waste space. Indexes can easily occupy 10% to 50% of the total space used by a database.

Which brings us to another issue: Add indexes on simple integer values (dates, etc.) first. Avoid indexing character strings—unless you really need them. And, when you do go and index them, make sure you don't index the entire string, but some small sized prefix of the string. Read the database documentation on how to setup such things.

5.2 Reducing Number of Tables

Another approach to improve performance (if indexes don't help) is reducing number of tables—or to put it another way: To redesign your database to use less tables (some may call this step: Refactoring).

This step is extremely dangerous and has very little chance of significantly improving performance (unless your initial design was horrible). Don't get it wrong, redesigning things is *very* beneficial (you leaned something in the first design iteration - you can improve things the second time around). However, redesigning with an explicit goal to use fewer tables is very dangerous, and has more chances of reducing design clarity (and ultimately performance) than increasing it.

Also remember that reduction at this stage may require additional logic in application code that uses the database (you're just shifting the performance burden away from the database to your clients).

That said (hopefully that was discouraging enough), here are some things to consider cutting:

Object-To-Object relations, as described above, are implemented via a separate table (often called a "link" table). If the relationship is many-to-many, then that's the way to do it. If the relationship is one-to-many, then you can eliminate the table, and simply have an ID field in one table.

Farther still, if the relationship is one-to-one, you might be able to absorb one object into another completely. This is usually the case with Generalization (or Specialization) relationships.

For example, a database that has `ANIMAL` and `MAMMAL` (which is a specialization of `ANIMAL`) might merge all `MAMMAL` fields into `ANIMAL` table.

Also, if two tables are similar enough, you can merge them into one table: table `DOG` and table `CAT` may be merged into a `PET` table, which will have all fields of `DOG` and `CAT` and a `TYPE` field, which will let the application know what we are dealing with.

The possibilities are endless—just remember not to go overboard, and still maintain a good design.

5.3 Data Redundancy

In good design, having redundancy is a big no-no. You should strive to eliminate every little byte that appears in two or more places. There is a good reason for it too: redundant data causes inconsistencies! (big problems) No matter how you prepare for it or try to avoid it, sooner or later, you will get caught.

That being said, sometimes bending the rules a bit provides a fairly big payoff. (it can also make your good & flexible "more tables than you think you need" design practical).

Redundancy is mostly used as a fast cache of data. Instead of re-computing some function or rerunning a query (sub-query) you simply grab an already made value. The next point is so important; it'll get it its own paragraph:

You must never update *the redundant data!*

Redundant data should be strictly read only. Updates should occur only on the source data. Editing source should update redundant data.

Each piece of redundant data *must* be documented. You must document what is the true and ultimate source of data—and what is just a copy of it. You *never* update the copy.

When the true value changes, you simply refresh the redundant fields with the new value (nice place for stored procedures). You can also do such a refresh via batch processing (at the end of the day or week). In fact, you should run such a batch process just for good measure—to ensure that your redundant data is *exactly* what your source data says it should be.

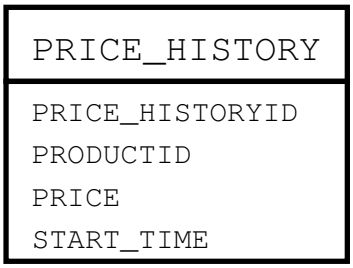
Let me repeat (and stress this point): *Redundant data must be read-only!* Redundant data *must be documented*! Procedure for update must be documented! Recovery procedures must also be documented!

Now, after following all the guidelines, you should still be prepared for inconsistency. This will happen no matter what (according to Murphy). Someone updates a value, but still sees the old value on the screen. Make sure that redundant data is not critical, and can handle being inconsistent for some periods of time.

Good candidates for redundancy are: totals, sums, averages, or any value the database (or your application) computes from the data already in the database.

As an example, let's consider our `PRODUCT` price dilemma. Most of the time we just want to see the price, but yet we also want to maintain the history of price changes. Previously we broke up the idea into two tables, saying that one of them will maintain the history of the `PRICE` field. That's all fine, except it takes an extra sub-query (and more complex application logic) to retrieve the current price.

We might need the price history once a week, but we will probably need the current price every few minutes. What do we do? Well... We define:



Where `PRODUCT.PRICE` is a redundant field of the latest price derived from finding the last entry in `PRICE_HISTORY` table for that `PRODUCTID`. Now, on an average use, we never have to even know `PRICE_HISTORY` table is even there. We deal with `PRODUCT`s and their `PRICE`s.

However, when we need to modify the `PRICE`, we need to know (and it *must be clearly documented*) that we need to add the price to the `PRICE_HISTORY` table. At that point, either a batch process, or a stored procedure (or just our application logic) will re-compute the price for the `PRODUCT` table.