

Database Design: The Quick and Easy Way (Well, Mostly)

Mac Newbold
mac@macnewbold.com
www.codegreene.com

UPHPU, November 17, 2006

Why Do We Care?

- The database is the core of a web application
 - Almost everything depends on it
- The way you design your database schema has a big impact on the rest of your project
 - It can make your life easy, or very, very hard
- Designing your database is one of the first steps you'll take in building a project
 - Then you're usually stuck with it for many years
- Bottom line: Do it right the first time!

Some Introductory Remarks

- We're going to be talking about database design principles more than specifics
 - The principles apply to all relational databases
 - If we get into the actual SQL, it will be MySQL
 - Cause that's what I know best
 - But I'll try not to get to that level much unless you want it
- If you've got questions, ask
- Be thinking as we go along about an example or problem you're having – a little later we'll go over some cases that you share with us

Key Considerations

- Database Integrity/Consistency (good)
 - Does the schema help us keep a consistent view of the data, or is it easy to get inconsistencies?
- Redundancy (bad)
 - Duplicating data is a bad thing in this case
 - Among other things, it hurts consistency
- Correctness (good)
 - Above all, the schema needs to correctly model the situation and relationships it is meant to represent

Relational Database Vocabulary 1

- Schema (a.k.a. Database)
 - The complete set of tables in a system
 - Most RDBMS engines (MySQL, etc.) can have more than one database that coexist separately
- Table, Column/Field, Row
 - Table Schema is the set of columns or fields
 - Made up of rows of data, one entry per column
 - Every column has a data type. Some systems allow a default value, and other special attributes.

Relational Database Vocabulary 2

- Primary Key
 - One or more fields that uniquely identify each row in a table. May be intrinsic or generated.
- Foreign Key
 - Field in a table that is used as a reference to a row in another table (i.e. what you do a join with)
- Key (a.k.a. Index)
 - May be “unique” or “multi-valued” usually
 - May enforce a constraint or just speed things up

Techniques for Database Design

- The first thing you need to do is understand the system the database needs to model
 - Then write it down somehow
- Entity-Relationship Diagrams (ERDs) are a common method to document the system
 - Entities have attributes, and participate in relationships
 - Entities usually map to tables, with attributes mapping to fields/columns, and relationships usually map to foreign keys

Relationships 1

- One-to-One
 - For each A, there is exactly one B
 - Each person has exactly one birthdate
 - Can often be represented as a field in a table
- Many-to-One / One-to-Many
 - Each A may have many B's, but each B is tied to exactly one A
 - A person can have more than one email address, but each address has only one owner
 - Usually represented by two tables and a foreign key

Relationships 2

- Many-to-Many
 - Each A may have several B's, and each B might belong to several A's
 - A person might have several houses, and a house might have several people
 - Typically represented by three tables, one for each entity and one to match them up

Relationships 3

- Person table has a PersonID and a birthdate field, among other things (1-to-1)
- Email table has an EmailAddress and a PersonID (foreign key) (1-to-many)
- House table has HouseID and other fields
- PeopleHouse table has PersonID and HouseID fields (many-to-many)

Relationships 4

- Entities usually have attributes
- Sometimes a relationship can have attributes that are separate from the attributes of the entities involved
 - Especially common with Many-to-Many
- Example: PeopleHouse may have fields for how long they've been there, what room they stay in, how well they like it, or how much they pay for it each month

Entity-Relationship Notation

- The notation varies, sometimes quite widely, but there are some commonalities usually
- Minimum requirements: a way to represent entities (rectangles?) and relationships (lines?) with their types (“crow's feet” are common)
 - Showing attributes is optional
- Example: 1-to-1 is a straight line, 1-to-Many is shown straight on the 1 side, but with a crow's foot on the Many side. Many-to-Many has a crow's foot on both sides.

How does an ERD help me?

- Easy way to write down and visualize the things you need to model in your system
- Easy to map to a relational database schema
- Gets you thinking about the requirements in your system, hopefully very thoroughly
 - The quality of the resulting database schema is directly proportional to accuracy and completeness of the model you construct
- Pretty easy for non-techies to understand

Let's try an ERD

- Let's have a simple example or two
- Something real is preferable, so you can answer questions about it
- What are our entities?
- How are they related?
- Which relationships are 1-to-1, 1-to-Many, and Many-to-Many?

From ERD to Schema

- First, all the entities become tables
 - They get columns for each attribute
 - Decide on the Primary Key (or add one)
 - Decide on any Unique Keys
 - Add column for a Foreign Key for each relationship where the entity is the 1 side
- Add tables for Many-to-Many relationships
 - Two foreign keys, and optionally some attributes
- Easy enough?

How do we know we did it right?

- Good question
- There isn't always one right answer
- But we know some answers are better than others, at least according to certain criteria
- Remember our criteria?
 - We want consistency, no redundancy, and above all, correctness

Normalization

- Anyone remember the definition of “normal” you used in geometry, trigonometry, calculus, or other math, or in a physics class?
 - Think “perpendicular”, see also “orthogonal”
- In other words, if it is “normalized”, the variables are orthogonal or independent of each other, rather than being interrelated
- It's like 2-d or 3-d graph paper – we can give simple coordinates that don't depend on each other, and allow us to reach any point

Database Normalization

- In the database world, they talk about “normal forms” and generally there are five of them
- Each normal form adds another rule about how the schema should be set up
- Sometimes it can be beneficial to break the rules, but there's usually a cost too
- Get a whole lot of experience in keeping all the rules before you try to break them
 - Over-optimizing is a common mistake

First Normal Form (1NF)

- Google “normal form” (no quotes) and you'll get a long list of places that will tell you all you ever want to know, and more, about normal forms
- First normal form requires data to be atomic
 - Columns can't be sets, lists, or aggregated data
 - Consider: One Name field vs separate First/Last name
- Every table must have a primary key
- No duplicate columns
 - Don't use 5 columns for a list of up to 5 items!

Second Normal Form (2NF)

- First, a definition: a “composite” key is made up of multiple columns (usually a primary key)
- 2NF requires that every field in a table be “functionally dependent” on every part of the composite key, or it should in a separate table
 - PartID (PK), StoreID (PK), PartName, StoreName
 - PartID (PK), StoreID (FK), PartName
- When you're not in 2NF, you'll have lots of redundant data in your tables (bad), which can be a source of inconsistency (bad)

Third Normal Form (3NF)

- Similar to 2NF but takes it further in subtle ways
- According to Bill Kent (courtesy of Wikipedia):
- “The relation is based on the key, the whole key, and nothing but the key, so help me Codd”
 - (Edgar F. Codd defined 1NF-3NF)
- Not only must the field depend on the key, but it can't depend on anything else, even transitively
- If you're not in 3NF, you have problems with redundancy, like not being in 2NF

Boyce-Codd Normal Form (BCNF)

- Some books/authors stick another form called BCNF between 3NF and 4NF
 - It's really just an expansion/clarification of 3NF in my opinion
- It says you can't have any non-trivial functional dependencies on anything but the key
- Try this: For each column, if you changed a value, are there other columns you'd have to update? Then you're probably breaking 3NF or BCNF and should see redundancies.

Fourth Normal Form (4NF)

- Has to do with “multi-valued attributes”
- A table should not have two or more independent multi-valued columns
- It is broken when you try and represent one or more 1-to-many or many-to-many relationships in a single table
- It is resolved by breaking the relationships into separate tables
- If you work from a good ERD, it isn't a problem

Fifth Normal Form (5NF)

- Eliminate cases where you can generate the same information with less redundancy
- Best explained with an example
 - Salesmen are agents for a company, and sell certain types of products, i.e. Ford and GM, and cars and trucks: Agent,Product,Company
 - Generally, Smith could sell Ford cars and GM trucks, but not Ford trucks or GM cars
 - Add a rule: If you sell cars, and sell for Ford, then you sell Ford Cars
 - Better layout: Agent,Product and Agent,Company

How and Why to Use Normal Forms

- The normal forms are rules for making your database be structured to eliminate redundancy and improve consistency
 - Note they don't help with correctness!
- Apply them step by step, solving the problems
 - With practice, you won't even think about making a table that violates them, you'll just do it right
- If you've answered the right questions in your ERD, many of these problems just can't crop up

Back to Our Examples

- Let's see if we can find any violations of the normal forms...
- What would have been some wrong ways to do these examples?
- How does the “right” answer change if we change the rules and requirements of the system we're modeling?

A Little More About Correctness

- Correctness is the hardest aspect
 - Defined as the accuracy with which your system models the real world you're trying to represent
 - It's very application specific
- The key is to understand all the rules up front about what is allowed and what isn't.
 - Whether a relationship is 1-to-1, 1-to-Many, or Many-to-Many is a key aspect
- “Correct” often changes as time goes by

Any questions so far?

- Before we move on in a different direction, are there any questions we can answer?
- Don't worry if you don't get it all right now. A lot of it only comes with practice as you internalize what the rules mean.
- Once you get that “sense” for what is redundant and what isn't, the rules are second nature
- A great way to practice is to have a friend review your plan for you

Topics by Request

- On the mailing list, several people expressed interest in certain topics related to database design:
- Performance and query optimization
- In Database vs. In Code
- OLAP/Data Warehousing, Star vs. Snowflake
 - Or, When Not to Normalize

Database Design and Performance

- The design/schema of your database has a lot to do with how fast it will perform for your needs
- How the data is broken up will determine what joins you'll need to do to perform certain queries
- If you normalize properly, it helps you pull exactly the data you need, without a lot of redundancy
 - Saves space, adds complexity (more tables/joins), and may save time
- Be careful about trading correctness for speed!

Keys/Indices 1

- Once you've got a nice normalized database schema, you've got a lot of primary keys
- You will probably also have foreign keys
- Defining a key/index on a field means your database will keep special track of those values, so it can find what you ask for without having to search through the whole list
- Slows insert/delete/update, speeds up select, and takes up more space

Keys/Indices 2

- Depending on your mix of reads vs writes, it can really provide a big benefit
- Prime candidates for keys are fields used as foreign keys in joins
- If you have a composite primary key, you may want to add an index on some of the fields that compose it - read your DB docs though
- Hint: Make sure foreign key fields have the exact same type/size in all tables!

SQL Explain

- If you're trying to optimize a query, “Explain” (a.k.a. “describe” sometimes) is your best friend
- “explain select ... from foo join bar...”
 - Shows you how many rows it has to touch in each table, what kind of join it can do (which indicates how hard/slow the join is) and any special things it had to do (like sorting, temporary tables, etc.)
- Rough estimate of how bad things are going is found by multiplying the number of rows examined in each table

Query Optimization

- MySQL has a chapter in the handbook on it
- Try using the slow query log maybe?
- Try explaining some of your queries
- Often the type of join or the keys being used will tell you where you might be able to add a key/index and really reap a benefit
- If you're having trouble, ask a friend for help
- Some queries won't ever be fast (!)

Performance Benchmarks/Tuning

- The most important benchmark is how well it performs in real life for the things you use it for
 - Everything else is an artificial test meant to mimic some reality that is probably different from yours
- The most practical way to approach it is the standard “If it ain't broke” method
- Don't worry about optimizing unless you need to
 - It can be a big time sink, and often unnecessary
- Be careful not to overoptimize

Performance Tuning 1

- The speed of your database depends on lots of things, and the weakest links in the chain (a.k.a. bottlenecks) will be the limiting factor
 - You can optimize non-bottlenecks all you want, but you will see very little improvement
- The database server has many components
 - Places to look: memory usage, hard disk speeds, processor speed, memory bandwidth, cache sizes, CPU load, and many more
 - Multi-threaded DBs or forking DBs might benefit from multi-processor or dual-core CPUs

Performance Tuning 2

- Comments about over-optimizing apply
- Easiest place to get performance gains is usually by optimizing your application (do fewer queries, be smarter) and by optimizing your queries, especially the slowest ones and the most common ones
- Depends a lot on your read/write mix, etc.
- Clustering/replication are a last resort!
 - If you're write-heavy, they may kill performance!

In Database vs In Code 1

- When you could make MySQL do something for you, or do it in PHP, which do you choose?
 - Some things are pretty obviously done better by MySQL, like joining tables, and doing most kinds of grouping and sorting of query results
 - Others could go either way: try it and see
- Example: auto-increment index vs PHP incrementing an index
 - Considerations: atomicity, race conditions, speed
 - Winner: auto-increment (on all 3 counts)

In Database vs In Code 2

- Often it is a lot easier to write/debug/maintain a snippet of PHP to do a particular operation than an SQL query for it
 - PHP is very general purpose compared to SQL
- Performance differences are usually lost in the noise, and insignificant in the big picture
- My opinion: if it is easy to do in SQL, do it there, otherwise, do it in the PHP
- Other examples or opinions?

In Database vs In Code 3

- A major consideration for where to do something depends on what else is using the same database
- If your web site is the only thing using that database, then it doesn't much matter
- If there are other “front ends” for your database, especially ones that modify it, push as much as you can into the database (don't reimplement)
 - Things like triggers, stored procedures, and views can really help a lot in this regard

In Database vs In Code 4

- One constant debate is regarding foreign keys
- Many databases will enforce foreign key constraints for you
- If you've got multiple front ends, use them
- If it's just one front end, it's up to you
 - If you don't use them, you've got to check for violations of the foreign keys, and tell the user
 - If you use them, you'll have to check for database errors when something tries to violate them

Data Warehousing 1

- One question had to do with really large scale stuff (think Amazon doing datamining in the database of purchases, page views, etc)
- It's a very different ball game than the stuff most of us will deal with
- I think it isn't worth spending much time on until you know that it is a problem
 - at least 99% of systems never get big enough to have these problems, chances are that you and I will never have to worry about it

Data Warehousing 2

- It does bring us to a good point though: When Not to Normalize
 - Or, The Right Ways to Break the Rules
- Normalization eliminates redundancy by adding complexity (more tables)
 - It usually saves space, and almost always improves consistency, and can go either way on time
- With really large scale problems, you might get better performance from something simpler
 - Note: “Might.” Don't assume. It might help or hurt.

Any More Questions?

- Is there anything else you want to talk about?
- Does anyone have another example or case study they'd like to talk about?

Thank You

- This concludes my prepared slides. Is there anything we missed that you'd like to discuss?
- Remember that UPHPU (via the mailing list or IRC) is a great resource for all of us for things like this – there are many experts lurking here and there who are happy to help you!
- If there are things we didn't cover today, we can easily start up a thread or two on the mailing list about the things that you want to know.