Relational Databases

This chapter introduces *relational databases* and the concepts they depend on. What is a database, a database server, a database client, a table, a row, a column? How do we map problems to these objects, and why? What is a join, a query, a transaction? If you know these fundamentals, or if you'll be using MySQL only as an non-relational document store, feel free to skim this chapter; otherwise take your time to make sure that you grasp this material before venturing further. We need the concepts and the terminology.

What is a database?

If you come to this chapter in a big hurry to start a project, then while reading, start at Step 1 of this ***how-to outline****.*

Databases are lists of lists. Even if you think you've never designed a database in your life, you already have hundreds of them: lists of people and their addresses and phone numbers, of favourite web addresses, of your insured items and insurance policies, of what to do tomorrow, of bills to pay, of books to read.

As any obsessive can tell you, the art of listmaking needs serious contemplation. Not just any fact makes it into your lists. What is worth writing down, and what is ephemeral? Will you record the current temperature and humidity in every diary entry, or instead focus on your problems, eliminating extraneous detail? Does it matter that a butterfly flapped her wings in Bangkok a week before your spouse decided to marry you, or leave you?

In designing databases, these are the questions you confront. What lists do you need? What levels of importance do they hold? How much do you need to know to survive, or how little can you live with?

Since we humans began writing on tablets, we have been storing information. You could say that the collection of printed books in the world is a database of sorts -- a collection of information, but a seriously stupid one, since you cannot ask a book to find all occurrences of a word like "love" in its own text. Even if the book has an index, how confident are we that it is complete?

The art of database design lies in elegant storage of information. To get maximum value from your lists of lists, you must be clever, and on occasion devious. For such talents, we turn to mathematicians.

A database is a set of sets

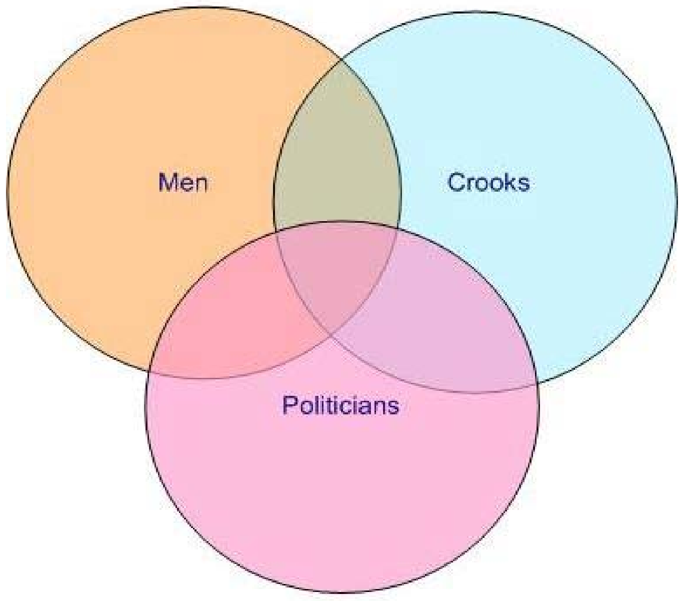

Fig. 1

A list represents a collection. Collections are sets. A database is a set of sets.

Consider the set of blue things–blue skies, blues music, blue jays, bachelor button flowers, blue jeans, blue movies.

Consider the set of vehicles–automobiles, trucks, forklifts, bicycles, golf carts.

Consider a more abstract set, the set of shapes–circles, triangles, rectangles, pentangles, ellipses, trapezoids. Each member of this set is precisely defined. A triangle consists of the straight lines connecting three points in space that are not in a straight line.

Now consider (Fig 1) three sets at once - men, crooks and politicians - and their areas of intersection.

Suppose you've just been hired as custodian of certain demographic data on the people of your town, and you have been told that the focus will be, for some reason, on which people are men, crooks and/or politicians. Your boss has made it clear that she expects you to record all instances of these three properties in all people for whom she will send data to you, and she further expects you to produce, at a moment's notice, a completely accurate list of people having any, some or none of the three attributes you will be tracking.

Well, some men are neither crooks nor politicians, some male crooks are not politicians, some politicians are neither crooks nor males, and so on for a total of seven subsets or lists corresponding to the seven different coloured areas in Fig 1. And, Fig 1 represents a set you may not have noticed yet: the white space around the coloured shapes. That white space represents the set of everything that is neither town man nor crook nor politician. That set is empty with respect to those three set attributes. A *null set*.

Yes, you could argue that the white space represents, not a null set, but actually a set of 7 billion people–all the folks in the world who are not men and not crooks and not politicians in your town. But you would be ignoring perspective and context. If every database model had to account for all possible members in the universe of each of its sets, we could never finish our first database job. Indeed we will expand some of the coloured sets and subsets, but having done so, we will still have that white space representing everything that belongs to none of our categories.

All in all, then, Fig. 1 specifies eight sets, for seven of which you might have to deliver membership lists. You have seven lists to maintain, somehow.

Are you going to keep seven different lists in seven different files, one labelled 'men', another labelled 'crooks', one called 'crooked men' and so on? Then every time your boss sends you data on someone new, or asks you to delete someone who's moved away, you will find and update the appropriate list or lists? Each deletion will require that you read up to eight lists, and so will each addition (to make sure that the person you are adding is not already filed or misfiled on a list).

You could do it that way. Would it be rude to some of our ancestors to call it a *neanderthal* database? Tomorrow, when your boss gives you three more attributes to track, say, 'owns property' and 'owns a firearm' and 'finished high school', you will have a lot more than twice as much work to do. With three attributes, a person can be on one of 2^3-1=7 lists. With six attributes, you have to maintain 2^6-1=63 lists. Doubling the number of attributes grew your workload ninefold, and it's only day two. You're in trouble.

Then you wake up: suppose you make just *one* list, and define each of these attributes—men, crooks, politicians, property owners, gun owners, high school graduates, whatever else is specified—as an item that can be filled in, yes-or-no, for each person?

That is the idea of a *record* or *row*. It contains exactly one *cell* for each attribute. A collection of such rows is a perfectly rectangular *table*. In such a table, we refer to each attribute as a *column*. A node where a row and column intersect is a *cell* containing a value for the corresponding row and column.

Rows and columns. In terms of sets, what have we done? We noticed that membership in any one of our sets is more or less logically independent of membership in any other set, and we devised a simple structure, a *table*, which maps set memberships to columns. Now we can record all possible combinations of membership in these sets for any number of persons, with a minimum of effort.

With this concept, you winnow your cumbersome 63 lists down to just one.

After you solve one small problem, and one larger problem.

The small problem is that, supposing we have a table, a collection of rows with data in them, we have not yet figured out a method for identifying specific rows, for distinguishing one row from another.

There seems to be a simple method at hand: add a column called `name`, so every row contains a person's name. The `name` cell in identifies each row, and the list of values in the column delivers a list of all recorded names.

Fine, but what if we have three John Smiths? Indeed, and again there is a simple answer. Add one more column, `person_id`, and assume, for now, a method of assigning, to each row, a unique number that is a `person_id` value. Each John Smith will then have his own `person_id`, so we can have as many John Smiths as we like without mixing up their data. So much for the smaller problem.

The larger problem is, how do we get information into this table, and retrieve information from it? In a real way, that is the subject of this book for the case where our array of rows and columns is a MySQL table. Suppose though, for now, a generalised mechanism that converts our table design to a data storage process, how do we instruct the mechanism to put rows of data into the table, and to retrieve rows, or summaries of rows, from it? How do you tell it to store

```
  ( 'Smith, John', male, crook, non-politician, owns property,
    owns gun, graduated high school )
```

in one row? And after you have stored many such rows, how do you ask it to list all the crooked male politicians with firearms?

In database lingo, such requests are called *queries*. We can call our system a database management system (DBMS) only if it supports these two functionalities:

1. creation and maintenance of *tables* as we have defined them, and
2. *queries* that update or retrieve information in the tables

Obviously if humans are going to use such a system, then table creation, table maintenance and queries need language interfaces. A language for table creation and maintenance is called a *Data Definition Language* or DDL, and a query language is called a *Data Manipulation Language*. For the moment, assume we have one of each.

With slight touchups, and ignoring for the moment the different kinds of data that will reside in each cell, our table, so far, has these columns:

```
person_id
name
gender
crook
politician
owns_property
has_firearm
hs_diploma
```

Call it the `persons` table. If the 63 separate lists constituted a neanderthal database, what is our table? We could give it the name *JPLDIS*, or *dBASE II*, or *Paradox*, or many another 1970s-1980s system that implemented simple tables with a simple DDL and DML. Clearly it fits our definition of a database as a set of sets.

But a *real* database is a set of sets *of sets*

Next day your boss greets you with more unwelcome news: you will need to store people's addresses, and a bunch of sub-attributes for politicians. Initial requests are:

- holds an electoral office of some sort, or
- running for electoral office, and
- belongs to a recognised political party,
- date of entry into politics
- amount of political pork available to the politician for redistribution

and there will be more to come. In the course of this conversation it may dawn on you that you will soon be getting similar requests about other columns–property ownership, or some other as-yet-undreamt attribute. Your success is about to burden you with an unmanageable explosion of more lists.

What to do? Are we to add columns for `elected, running, party, entry date` and `pork dollars` to the persons table? If so, what would we put in those cells for persons who are not politicians? Then ask yourself what the `persons` table will look like when you have done this over and over again for more `persons` attributes.

These questions are about what is called, in database theory, *normalisation*, which is, roughly, exporting sets of columns to other tables in order to minimise data redundancy. You are going to have to make a new `persons` table for personal info, and a `politicians` table that lists the political attributes your boss wishes to track. All references to politicians in the `persons` table will disappear. Eh? Here is how. One row in the new `politicians` table is named, say, `pol_person_id`. A person who is a politician gets a row in the `politicians` table, and the `pol_person_id` cell in that row contains that person's `person_id` from the `persons` table.

And what can we do with these relational databases, these *sets of sets of sets*?

Some basic facts about sets

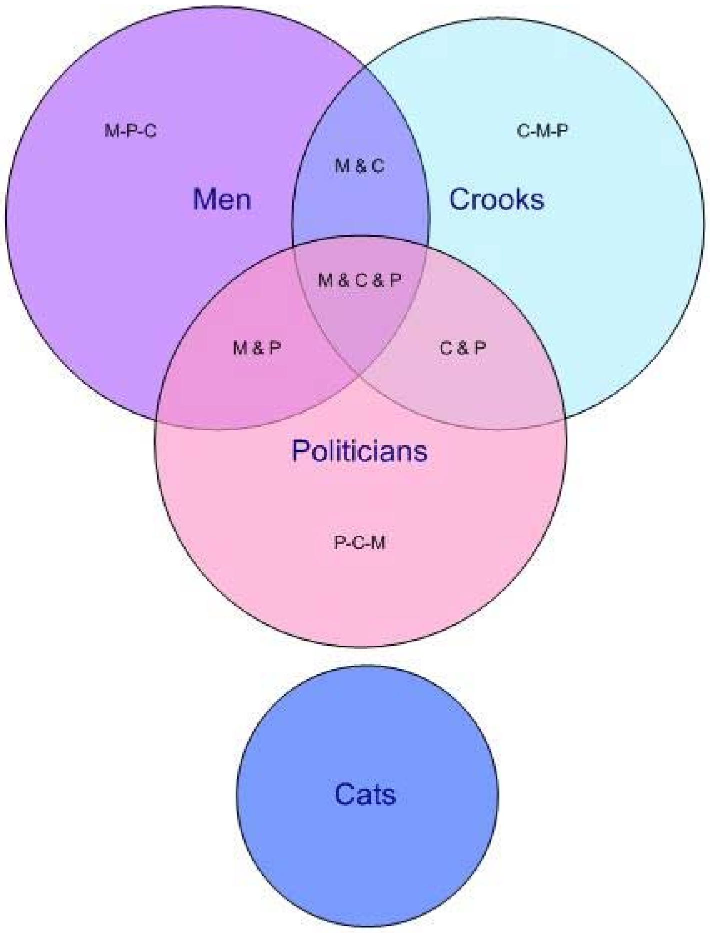

Fig. 2

A set is a collection of similar or related "things". The basics of the modern notion of a set come from a paper by Bernard Bolzano in 1847, who defined a set as *an embodiment of the idea or concept which we conceive when we regard the arrangement of its parts as a matter of indifference.*

As we shall see, that contains an idea that is crucial to the database engineer's idea of a database—the idea that *retrieval of information from any table must be completely independent of the physical order of rows and columns in the table.*

The mathematics of sets began in 1874 with Georg Cantor, who was trying to solve some mathematicial problems concerning infinity. We will not dive into Cantor's theory. We just want to indicate where relational database theory came from, and to list a few terms that will come up.

A set has zero or more *members*. If it has zero members, it is a *null set,* otherwise we can form a set of fewer than all members of the set, which is a *subset*.

Given two sets A and B, their *intersection* is defined by the members that belong to both. In Fig 2, M&P, C&P, C&M and C&M&P are intersection sets.

A set that has fewer than an infinite number of members is a *finite set*. Obviously, all databases are instances of finite sets.

If the set of members common to A and B is the null set, A and B are *disjoint* sets. For example, in Fig 2, no politician is a cat, and no cat is a politician. `Politicians` and `Cats` are disjoint.

Given sets A and B, members that belong to A and/or B constitute the *union* of A and B: in Fig 2, men who aren't crooks, plus crooks who aren't men, plus those who are both.

The intersection of A and B is always a subset of the union of A and B.

A relation R is *commutative* if a R b = b R a. Intersection and union are commutative.

Given two sets A and B, the *difference set* A-B is the set of elements of A that do not belong to B.

If B is a subset of A, A-B is called the *complement* of B in A.

A set has a *relation R* if, for each pair of elements (x,y) in A, x*R*y is true.

A relation *R* can have these properties:

- *reflexiveness*: for any element x of A, x*R*x
- *symmetry*: for any x and y in A, if x*R*y then y*R*x.
- *transitiveness*: for any x, y and z in A, if x*R*y and y*R*z then x*R*z.
- *equivalence*: a relation has equivalence if it is reflexive, symmetric and transitive.

If the idea of equivalence makes your eyes glaze over, don't worry. It will come clear when you need it.

Operations on sets

The most basic set operation is *membership*: (member) x *is in* (set) A.

Obviously, to speak of adding sets can be misleading if the sets have members in common, and if you expect set addition to work like numeric addition. So instead, we speak of *union*. In Fig 2, the union of men and crooks comprises the men who are not crooks, the crooks who are not men, and those who are both.

For similar reasons it is chancy to try to map subtraction to sets. Instead, we speak of the *difference set* A-B, the set of As that are not Bs.

To speak of multiplying sets can also be misleading, so instead we speak of *intersection*.

And with sets, division is a matter of *partitioning* into *subsets*.

Sets, tables, relations and databases

As we saw, a *table* is just an arrangement of data into rows (also called tuples, or records) and columns (also called attributes). It is a *set* of rows with data partitioned into columns. Relational database programmers, designers and managers everywhere refer to these structures as tables, but the usage is inexact—RDBMSs actually have *relations*, not tables.

What is the difference? Crucial question. A relation is a table that meets these three criteria:

1. Each row is unique.
2. Each row has exactly the same number of cells (the table is not ragged).
3. The data in each cell matches the data type (*domain*) defined for its column in the relation scheme of the database.

The first two are elementary. The third is a guarantee that there is a mapping from every cell in every table in the database to a data type, i.e. *a domain*, that has been defined generally or specifically for the database. Obviously, then, RDBMS tables are relations in definite ways that spreadsheet tables are not.

Having said that, we are going to follow the very common usage of referring to relations as *tables*. Be warned, though, that if you ever have the opportunity to discuss RDBMS issues with Chris Date, you'll be well advised to switch to correct usage.

We have just dipped our toes, ever so gingerly, into *relational algebra*, which mathematically defines database relations and the eight operations that can be executed on them—*partition, restriction, projection, product, union, intersection, difference*, and *division*. Each relational operation produces a new relation (table). In Chapters *6* and *9*, we will see how to perform these operations using the MySQL variant of the SQL language. Before doing that, we need to set down exactly what relational databases are, how they work, and how the MySQL RDBMS works. That is the task of the rest of this chapter, and of chapters 3, 4 and 6 through 9. We pick up the thread again, now, with the computer scientist who discovered—or if you prefer, invented—the relational database.

Introducing Dr. E.F. Codd

Dr. Edgar Frank Codd invented the relational database while working for IBM in 1970 [1]. Four years later, IBM began implementing some of Codd's ideas in a project called System R and a language they called SEQUEL ("Structured English Query Language"). In the next few years they rewrote the system and language for multiple tables and users, and for legal reasons abbreviated the language's name to SQL. Testing at customer sites began in 1978. IBM's first SQL products appeared in the 1980s, and relational database products from other vendors soon followed. In the mid-1980s Dr Codd published a two-part article [2] listing three axioms and twelve rules for determining whether a DBMS is relational. At the time, no commercial database system satisfied all 12 requirements. Now we are in a new century and a new millennium. Still there is not a single relational database management system that fully supports all 12 of Codd’s rules.

Codd’s theory of databases is a bit like the Law of Ideal Gases. No gas in the universe obeys the law perfectly, yet every gas is subject to it. No RDBMS in the world perfectly obeys Codd, yet not one of them can wriggle free from the impact of Codd’s laws.

Relational theory says how to organise information whilst hiding where it is and how to retrieve it. RDBMSs specify the wheres and hows. Basically, the bits of Codd that you need to know in order to design good databases are the three axioms and twelve rules, and how to normalise a database.

Codd's three axioms and 12 rules

Axiom 1: All stored values shall be atomic

To understand atomic values, consider their opposites. Suppose we have a column called `children` in an `employees` table. If we store all names of an employee's children in this single column—`'Rachel,Daniel,Jimi,Dylan,Jacqueline,Ornette'`—the column is not atomic. Relational databases exist to store data such that simple query expressions can retrieve it. If one table cell can contain any number of names, you have to write complex code to retrieve one of them. In this example, the correct approach is to create a `children` table, and there add a row for each child of each employee.

Axiom 2: All elements in a relation are unique

A relation differs from a table *in three ways*, one of which is that each row must be unique. Some attribute must distinguish one from another, otherwise the new row cannot be added to the set. In the terminology of relational databases, the attribute that uniquely distinguishes one row from *any* other row is called the *Primary Key*.(PK).

This rule imposes a little labour upon the application developer in the short term, but it saves much more development and maintainance labour, not to mention user labour, in the medium and long term.

Consider a table of Countries. What sense would it make to have two table entries named "Croatia"? Descending one geographic level, no two states or provinces of a single country may have the same name either. Descending one more level, no two cities in a single state of a single country may share a name.

In a classic instance of failed design logic, the developers of one application discovered that their database contained 19 different spellings of the city called Boston! In the real world, if it can go wrong, it will.

Axiom 3: No extra constructs

For any system that is claimed to be a relational database management system, that system must be able to manage data entirely through its relational capabilities.

No extra constructs are required. No black magic is permitted. Everything must be transparent.

Codd proved that a system following these three axioms has to obey the following twelve rules.

Rule 1: The information rule

All information in a relational database is represented explicitly at the logical level in exactly one way: by atomic values in tables.

The rule sharply distinguishes the logical level from the actual implementation method. The user of a database need be no more concerned with how the RDBMS is implemented than the driver of a car need be concerned with how the car engine is made, so the user of an RDBMS can treat the database's sets as Bolzano stipulated. You concern yourself solely with the database logic. *Rule 8* takes this concept a little further.

Rule 2: Guaranteed access

Each and every datum, or atomic value, in a relational database is guaranteed to be logically accessible by resorting to a table name, a primary key value and a column name.

Since every row in a table is distinct, the system must guarantee that every row may be accessed by means of its unique identifier (primary key). For example, given a table called `customers` with a unique identifier `CustomerID`, you must be able to view all the data concerning any given customer solely by referencing its `CustomerID`.

Rule 3: Systematic treatment of null values

Missing values are represented as `NULL` values, which are distinct from empty character strings or a string of blank characters, also distinct from zero or any other number. `NULL`s are required for concictent representation of missing data.

Suppose you have a customer whose surname is `Smith`, whose given name is `Beverly`, and whose gender is unknown. You cannot infer this customer's gender based on this information alone. `Beverly` is both a male and a female given name. In the absence of other data, you must leave the `Gender` column blank or `NULL`.

Take another case, a table called `employees`, with a column called `departmentID`. A new hire may not yet have been assigned to a `department`, in which case you cannot fill in this column.

Rule 3 says that no matter what kind of data is stored in a given column, in the absence of valid data the system should store the value `NULL`. It doesn't matter whether the column houses text, numeric or Boolean data - the absence of data is represented by the value `NULL`.

This turns out to be quite useful. The value Null indicates only that the value of that cell is unknown.

Rule 4: The database catalog must be relational too

The database description is represented at the logical level in the same way as ordinary data, so authorized users can apply the same relational language to its interrogation as they apply to regular data.

That is, you must be able to inquire about a database's meta-data using the same language that you use to inquire about the actual data. What tables are available? What columns do they contain?

This rule is a corollary of Axiom 2: no black boxes are permitted. The method of interrogating the structure of the data must be identical to the method of interrogating the data itself.

Rule 5: The system must implement a comprehensive data sublanguage

A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible as character strings using some well-defined syntax, supporting all of the following:

- data definition;
- view definition;
- data manipulation (interactive and by program);
- integrity constraints; and
- transaction boundaries (begin, commit, and rollback).

The ubiquitous language for such expressions is *Structured Query Language* or SQL. It is not the only possible language, and in fact SQL has several well-known problems which we explore later.

Rule 6: Data views must be updatable

A *view* is a specification of rows and columns of data from one or more tables. All views that are theoretically updatable, like this one,

```
DELETE
FROM invoices, customers
WHERE invoices.CustomerID = <some value>;
```

should be updatable by the system.

Rule 7: High-level Insert, Update, and Delete

The system must support not only set-at-a-time SELECT, but set-at-a-time INSERT, UPDATE, and DELETE on both base and derived tables.

A *base relation* is simply a table which physically exists in the database. A *derived relation* may be a view, or a table with one or more calculated columns. In either case, a fully relational system must be able to insert, update and delete base and derived tables with the same commands.

Rule 8: Physical data independence

Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representation or access methods.

That is, specific facts about the actual layout of data must be irrelevant from the viewpoint of the query language. Changing the column order or more radically, the way a relation is stored, should not adversely affect the application or any existing query.

In a world dominated by relational databases, this rule may seem so obvious that you might wonder how it could be any other way. But there were databases long before Dr. Codd issued his rules, and these older database models required explicit and intimate knowledge of how data was stored. You could not change anything about the storage method without seriously compromising or breaking any existing applications. This explains why it has taken so long, traditionally, to perform even the simplest modifications to legacy mainframe systems.

Stating this rule a little more concretely, imagine a `customers` table whose first three columns are `customerID`, `companyname` and `streetaddress`. This query would retrieve all three columns:

```
SELECT customerID, companyname, streetaddress
FROM customers;
```

Rule 8 says that you must be able to restructure the table, inserting columns between these three, or rearranging their order, without causing the `SELECT` to fail.

Rule 9: Logical data independence

Application programs and terminal activities (i.e. interactive queries) remain logically unimpaired when information-preserving changes of any kind are made to the base tables.

In other words, if I am in the act of inserting a new row to a table, you must be able to execute any valid query at the same time, without having to wait until my changes are complete.

Rule 10: Integrity independence

Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog, rather than in application programs.

The following two integrity constraints must be supported:

- *Entity integrity:* No component of a primary key is allowed to have a null value.
- *Referential integrity:* For each distinct non-null foreign key value in a relational database, there must exist a matching primary key value from the same domain.

There has been much contention and misunderstanding about the meaning of Rule 10. Some systems allow the first integrity constraint to be broken. Other systems interpret the second constraint to mean that no foreign keys may have a null value. In other cases, some application programmers violate the second constraint using deception: they add a *zeroth row* to the foreign table, whose purpose is to indicate missing data. *We think this approach is very wrong* because it distorts the meaning of data.

Rule 11: Distribution independence

A relational DBMS has distribution independence. Distribution independence implies that users should not have to be aware of whether a database is distributed.

To test this, begin with an undistributed database, then create some queries. Distribute the database (i.e., place one table on server A and a second table on server B), then re-issue the queries. They should continue to operate as expected.

Rule 12: Nonsubversion

If a relational system has a low-level (single-record-at-a-time) language (i.e. ADO, DAO, Java, C), that low-level language cannot be used to subvert or bypass the integrity rules or constraints expressed in the higher-level (multiple-records-at-a-time) relational language.

To put this rule simply: back doors are forbidden. If a constraint has any power at all, it must apply entirely, no exceptions.

MySQL and Codd's rules

How well do MySQL 5&Up implement Codd's axioms and rules?

Atomicity (Axiom 1): MySQL implements a column type, SET, to store multiple values in single data cells.

Row uniqueness (Axiom 2): Like most RDBMSs, MySQL provides proper primary keys, yet permits tables that don’t have them.

No back doors (Axiom 3, Rules 4 and 12): Since version 5.0.3, MySQL implements much of the current SQL INFORMATION SCHEMA standard for database metadata. There remain the previous backdoors for this functionality, in the form of custom commands (like SHOW) and utilities, but they are now necessary only for backward compatibility.

Information rule (Rules 1 and 2): All MySQL database data is available at the logical level as values in tables, and it is always possible, though as noted above not necessary, to enforce row uniqueness on any table.

NULLs (Rule 3): NULL in MySQL means data is missing, and unlike Microsoft SQL and Oracle, MySQL provides no backdoor escape from this treatment of NULLs.

Data language (Rule 5): The MySQL variant of SQL is comprehensive though it does not completely implement the current SQL specification.

Updateable views (Rules 6, 7): These are available since version 5.0.1.

Physical data independence (Rule 8): The current MySQL version provides some data storage engines which meet this requirement, and some which do not.

Logical data independence (Rule 9): In MySQL, different storage engines accept different commands. It may be impossible for *any* system to strictly obey Rule 9.

Integrity independence (Rule 10): MySQL's primary and foreign keys meet Codd's criteria, but it remains possible to create tables which bypass both requirements, and the MySQL 5 implementation of Triggers remains incomplete.

Distribution independence (Rule 11): Version 5.0.3 introduced storage engines which violate this requirement.

Would such a report card for other major RDBMSs be better or worse than the above? On the whole we think not. Like other major systems, MySQL supports most of Codd's rules.

Codd's 333 Rules

Over two decades, Dr. Codd refined and embellished the 12 rules into an imposing collection of 333 rules. Even a brief discussion of all of them would fill a book by itself. For those interested, we suggest that you go to the source[3], a gold mine of provocative thought.

Practical database design rules

Allowing for small departures from Codd conformance, your MySQL database will work better if you follow Codd's rules, and if you honour four practical rules of thumb that derive from Codd's rules: don't let the outside world set your primary keys, resist NULLs, keep storage of derived values to a minimum, and don't use zero-value keys.

1. Primary Keys should be meaningless outside the database

Parking management clerks at a major US movie studio liked to track regular parkers by Social Security Number (SSN). The manager wanted us to use the SSN as a primary key. But some employees provided incorrect SSNs, or none at all. "I know", said the manager, "when that happens, we just make up a new one, so you have to let us enter letters as well as digits in the SSN." To use SSN as a PK, they had to invent SSNs! So much for primary keys representing the real world.

That's the core of the argument for *surrogate keys*. The simplest and safest way to protect PKs from real world errors is to generate unique integer key values with MySQL's AUTO_INCREMENT feature (Chapters *4*, *6*). Each new table row gets a new unique value automatically, with no intervention by you. Here, lazy is smart.

Yes sometimes surrogate keys aren't needed, for example in a calendar table with exactly one row for every date in the year, an AUTO_INCREMENT key would be redundant. Aside from such cases, *resist the temptation to embed meaning in primary keys.*

In a t-shirt company, the first letter of a t-shirt code denotes its quality, the second its color, the third its size. All users know these codes by rote. Why not use these codes as the Primary Key for the t-shirt table? Let's count the reasons:

1. A real-world coding, as opposed to a purely logical one, will inevitably require adjustment. Keys derived from the coding will then have to be modified. Primary keys should never have to be edited.

2. Coding multiple attributes into one column violates Axiom 1 (atomicity).
3. An integer of a given length permits more unique values than a character string occupying the same number of bytes.
4. Searches on subsets are difficult: how will you list the white t-shirts, or all the colors and qualities available in extra-large?
5. At almost every turn, you encounter new problems, all created by the attempt to impose meaning on primary keys.

Data modellers object that an externally meaningless PK models nothing in the real world outside the database. Exactly! A PK models the uniqueness of a row, and nothing more. To the extent that you permit the external world to directly determine PK value, you permit external world errors to compromise your database.

2. Doubt NULLS

Seriously question any column that permits nulls. If you can tolerate the absence of data, why bother storing its presence? If its absence is permitted, it is clearly unessential.

This is not to say that columns that allow nulls are unacceptable, but rather to point out their expense. For any table that presents a mix of not-null and null columns, consider splitting it into two tables, not-nulls in the critical table, nulls in the associated table.

3. Do not store values that can be derived

To put it another way, wherever there is duplication, there is the opportunity for skew. Run report A and it states Reality R. Run report B and it states Reality S. If inventory reports 71 units, but stock purchases minus sales equals 67, which result is to be believed? *Wherever feasible, calculate and derive such quantities on the fly*. In cases where users cannot or will not tolerate waiting for derivation to complete, for example in databases optimised for reports (OLAP) rather than updates (OLTP), we suggest that you provide a datetime tag for display alongside the stored result.

4. Never succumb to the zeroth row

The Zeroth Row is a method of dealing with unknown parent column values. The argument goes like this: suppose you hire a new `employee`, but it is unclear which `department` she will work in. We want the best of both worlds:

- The columns `employees.departmentID` and `employees.title` are NOT NULL.
- We don't currently know values for this employee.

The solution known as the Zeroth Row is to add a row to the appropriate foreign key tables whose textual value is "Undefined" or something similar. The reason it is called the zeroth row is that typically it is implemented in a tricky way. This row is assigned the PK zero, but the PK column is AUTO_INCREMENT. Typically, the front end to such a system presents the zeroth row as the default value for the relevant picklists.

This violation is not occasional. It is often the considered response of experienced people.

Those who take this approach typically argue that it simplifies their queries. It does no such thing. It complicates almost every operation that you can perform upon a table, because 99% of the time your application must pretend this row doesn't exist.

Almost every SELECT statement will require modification. Suddenly you need special logic to handle these particular "special" non-existent rows. You can never permit the deletion of any row whose PK is zero. And finally, this solution is based on a misunderstanding of basic relational principles.

Every instant convenience has an unexpected cost. Never add meaningless rows to foreign tables. Nulls are nulls for a reason.

Other dimensions

Dr. Codd gave us a revolution but over the years it has become apparent that certain problems are not easy to model in relational databases. Two examples are geography and time. They introduce third or fourth dimensions to Relational Flatland. *Tables have rows and columns, but have neither depth nor time.*

MySQL 5&Up implement a subset of the *OpenGIS* specification; apart from covering that, we leave geographic data modelling to specialists. But change tracking is a universal, non-trivial problem. It has motivated development of post-relational and object-relational databases. Unless you instruct an RDBMS to the contrary, *it implements amnesia*: when updating, it destroys previous data. That's fine if you just want to know how many frammelgrommets are on the shelf right now, but what if you want to know how many lay there yesterday, or last month, or last year same day?

Accountants know the problem of change tracking as *the problem of the audit trail*. You can implement audit trails in relational databases, but it is by no means easy to do it efficiently with your own code. Some database vendors provide automated audit trails–all you have to do is turn the feature on, at the desired level of granularity. Like many RDBMSs, however, MySQL does not offer this feature. The MySQL implementation of stored procedures and views permits you to port change tracking functionality from client applications into your database. If just a few tables in your application require detailed change tracking and are not too large, you may be able to get by with having your update procedures copy timestamped row contents into audit tables before changes are written. If the inefficiences of this brute force approach are unacceptable, you will need to make change tracking a central feature of your database design (*Chapter 21*).

Normalisation and the Normal Forms

A *database server* serves data to clients as efficiently as possible from one or more computers. Normalisation is the process of progressively removing data redundancies from the tables of a database such that this efficiency is maximal.

The smaller the table, the faster the search. Store only information that you actually need. In some applications, your client's weight may matter. In most it won't.

In addition, you want to eliminate duplicated information. In an ideal design, each unique piece of information is stored precisely *once*. In theory, that will provide the most efficient and fastest possible performance. Call this the "Never type anything twice" rule.

A table which models a set is normalised if it records only facts which are attributes of the set. A set of tables is normalised if each fact they record is in one place at one time.

In a table called `customers`, you might have several columns for the customer address: `StreetAddress, CityID, StateID` and `CountryID`. But what if the customer has several addresses—one for billing, another for shipping, and so on? We have two choices. The wrong choice (as we saw in our first approach to a persons table *above*) is to add columns to the `customers` table for each kind of address. The right choice is to admit that an address is a different sort of object than a customer. Therefore we export address columns from `customers` to a new `addresses` table, and perhaps an associated table called `addresstypes`.

In formal database language, there are five main normal forms (and *many others*, as you might expect). Many database designers stop when they have achieved Third Normal Form (3NF), although this is often insufficient.

EF Codd and his colleague CJ Date vigorously disagree, but the normal forms are guidelines, not rules the breaking of which must cause calamity. Occasionally, it becomes necessary to disregard the rules to meet practical organisational requirements. Sometimes it's a question of supported syntax, sometimes of pure, raw performance.

Our rule of thumb is this: When you stray from the rules, you'd better have solid reasons and solid benchmarks.

That said, let's explore the main normal forms. Each successive form builds on the rules of the previous form. If a design satisfies normal form `i`, it satisfies normal form `i-1`.

Table 1-1: Normal Forms

Form	***Definition***
1NF	an unordered table of unordered atomic column values with no repeat rows
2NF	1NF + all non-key column values are uniquely determined by the primary key
3NF	2NF + all non-key column values are mutually independent
BCNF	Boyce-Codd: 3NF + every *determinant* is a candidate key
4NF	BCNF + at most one *multivalued dependency*
5NF	4NF + no *cyclic ("join") dependencies*

First Normal Form (1NF)

First normal form (1NF) requires that the table conform to Axioms *1* and *2*:

1. Every cell in every row contains atomic values.
2. No row is identical with any other row in the table.

Second Normal Form (2NF)

The formal definition of 2NF is 1NF *and* all non-key columns are fully *functionally dependent* on the primary key. This means: remove every non-key column which does not have exactly one value for a given value of the primary key.

Remove to where? To other tables. For any 2NF table, you must create separate 1NF tables for each group of related data, and to make those new 1NF tables 2NF, you have to do the same thing, and so on till you run out of attributes that do not depend on their primary keys.

Let's apply this to a common, non-trivial example. Consider an invoice, a document with

- an invoice number,
- the customer's name and address,
- the "ship to" address if different,
- a grid holding the invoice product and cost details (product numbers, descriptions, prices, discounts if any, extended amount)
- a subtotal of the extended amounts from the grid
- a list and sum of various taxes
- shipping charges if applicable
- the final total
- list of payments
- sublist of payment methods (credit card details, cheque number, amount paid)
- net amount due

Apart from normalisation, there is nothing to stop you from creating a single table containing all these columns. Virtually all commercial and free spreadsheet programs include several examples which do just that. However, such an approach presents huge drawbacks, as you would soon learn if you actually built and used such a "solution":

- You will have to enter the customer's name and address information over and over again, for each new sale. Each time, you risk spelling something incorrectly, transposing the street number, and so on.
- You have no guarantee that the invoice number will be unique. Even if you order preprinted invoice forms, you guarantee only that the invoices included in any particular package are unique.
- What if the customer wants to buy more than 10 products? Will you load two copies of the spreadsheet, number them both the same, and staple the printouts together?
- The price for any product could be wrong.
- Taxes may be applied incorrectly.
- These problems pale in comparison to the problem of searching your invoices. Each invoice is a separate file in your computer!
- Spreadsheets typically do not include the ability to search multiple files. There are tools available that can search multiple files, but even simple searches will take an inordinate amount of time.
- Suppose that you want to know which of your customers purchased a particular product. You will have to search all 10 rows of every invoice. Admittedly, computers can search very quickly, even if they have to search 10 columns per invoice. The problem is compounded by other potential errors. Suppose some of entries are misspelled. How will you find those? This problem grows increasingly difficult as you add items to each spreadsheet. Suppose you want the customers who have purchased both cucumber seeds and lawnmowers? Since either product

could be in any one of the ten detail rows, you have to perform ten searches for each product, per invoice!

- Modern spreadsheets have the ability to store multiple pages in a single file, which might at first seem a solution. You could just add a new page for every new invoice. As you would quickly discover, this approach consumes an astonishing amount of memory for basically no reason. Every time you want to add a new invoice, you have to load the entire group!

In short, spreadsheets and other un-normalised tabular arrangements are the wrong approach to solving this problem. What you need is a database in at least 2NF.

Starting with our list of invoice attributes, we take all the customer information out of the invoice and place it in a single row of a new table, called `customers`. We assign the particular customer a unique number, a *primary key*, and store only that bit of customer info in the invoice. Using that number we can retrieve any bit of information about the customer that we may need - name, address, phone number and so on.

The information about our products should also live in its own table. All we really need in the details table is the product number, since by using it we can look up any other bit of information we need, such as the description and price. We take all the product information out of the details grid and put it in a third table, called `products`, with its own primary key.

Similarly, we export the grid of invoice line items to a separate table. This has several advantages. Forget being limited to ten line items. We can add as many `invoice details` (or as few) as we wish. Searching becomes effortless; even compound searches (cucumber seeds and lawnmowers) are easy and quick.

Now we have *four* tables:

- `Customers` - customer number, name, address, phone number, credit card number and so on.
- `Invoices` - invoice number, date, customer number and so on.
- `Invoice Details` - detail number, invoice number, product number, quantity, price and extended amount (quantity times price).
- `Products` - product number, description, price, supplier, quantity available, etc.

Each row in each table is numbered uniquely. This number, its primary key, is the fastest possible way to find the row in the table. Even if there millions of rows, a database engine can find the one of interest in a fraction of second.

To normalise invoice data storage to 2NF, we had to take Humpty Dumpty apart. To produce an invoice we have to put him together again. We do this by providing relationship *links* amongst our tables:

- The `Customers` table has the PK `Customer_Id`.
- The `Products` table has the PK `Product_Id`.
- The `Invoices` table has a column `Inv_Customer_Id`, and every `Invoices` row has a `Inv_Customer_Id` value that points to a unique `Customers` row.
- The `Invoice Details` table has a column `InvDet_Invoicer_Id`, and every `Invoice Details` row has a `InvDet_Invoice_Id` value that points to a unique

`Invoices` row, and a `InvDet_Product_Id` value that points to a unique row in the `Products` table.

Third Normal Form (3NF)

Formally, 3NF is 2NF *and* all *non-key* columns are mutually independent.

What does this mean? Consider the `customers` table defined so far—`customer number, name, address, credit card number` and so on. Do you see any problems here?

Most people have more than one address: home address, mailing address, work address, summer cottage address, and so on. In the case of summer cottage addresses, there is the further qualification that it only applies for part of the year.

This suggests we create a new table `addresses`, with a primary key `AddressID` and a foreign key `CustomerID`. But we must be able to distinguish the kind of address each row represents. Thus we need another table, `addresstypes`, which will document each kind of address. This design would allow us to add as many addresses for each customer as we wish, identify each by type, and add new address types anytime we need them.

Now consider the phone number column. When was the last time you had only one phone number? Almost certainly, you have several—daytime, evening, cell, work, pager, fax, and whatever new technologies will emerge next week or month or year.

As with addresses, this suggests two more tables:

- `PhoneNumbers` - `PhoneNumberID, CustomerID, PhoneNo, PhoneTypeID.`
- `PhoneTypes` - `PhoneTypeID, Description`.

Our existing design glosses over the problem of suppliers, tucking it into the products table, where it will doubtless blow up in our faces. At the most basic level, we should export the supplier information into its own table, called `suppliers`.

Depending upon the business at hand, there may or may not emerge an additional difficulty. Suppose that you can buy product X from suppliers T, U and V. How will you model that fact?

The answer is, a table called `productsuppliers`, consisting of `ProductSupplierNumber`, `ProductNumber` and `SupplierNumber`.

Boyce-Codd Normal Form (BCNF)

BCNF is more demanding than 3NF, less demanding than 4NF.

The set of columns *b* is functionally dependent on the set of columns *a* if there is no more than one value of *b* for any *a*. We then say *a* determines *b*, or *a*→*b*. If the *city* is Toronto, the *province* is Ontario. A table is BCNF if for every dependency, *a*→*b* is trivial since *a* is in *b*, or *a* is unique—that is, *a* is a *superkey*. The *city*→*province* dependency meets the BCNF criterion if and only if every value of *city* is unique. If this uniqueness fails, then updates `WHERE city='Stratford'` will apply to cities of that name in Ontario, Quebec and Prince Edward Island—not likely the intention.

Fourth Normal Form

Fourth normal form (4NF) is 3NF, plus BCNF, plus one more step: the table may not contain more than one *multi-valued dependency* (column A depends on columns B and C, but B and C are independent).

What does this rule mean? Consider the table `addresses`, which we parcelled out from the table `customers`. If your database will have many addresses from multiple states, provinces or countries, then in the spirit of normalisation, you may have already realised the need for exporting lookup lists of cities, states and countries to their own tables. Depending on your problem domain, you might even consider exporting street names. An application confined to one suburb or one city might profit handsomely if its database exports street names to their own table. On the other hand, for a business with customers in many countries, this step might prove to be more a problem than a solution. Suppose you model addresses like this:

- `AddressID`
- `CustomerID`
- `AddressTypeID`
- `StreetNumber`
- `StreetID`
- `CityID`
- `StateID`
- `CountryID`

Clearly, `StreetID`, `CityID`, `StateID` and `CountryID` violate the 4NF rule. Multiple cities may have identical street names, multiple states can have identical city names, and so on. We have many possible foreign keys for `StreetID`, `CityID`, `StateID` and `CountryID`. Further, we can abstract the notion of address into a hierarchy. It becomes possible to enter nonsensical combinations like `New York City, New South Wales, England`.

The city New York resides in precisely one state, which in turn resides in precisely one country, the USA. On the other hand, a city named Springfield exists in 30 different states in the USA. Clearly, the name of a city, with or without an ID, is insufficient to identify the particular Springfield where Homer Simpson lives.

Fourth Normal Form requires that you export `stateID` to a `cities` table (`cityID`, `cityname`, `stateID`). `StateID` then points to exactly one row in the `states` table (`stateID`, `statename`, `countryID`), and `countryID` in turn points to exactly one row in a `countries` table (`countryID`, `countryname`).

With these 4NF refinements, `CityID` is enough to distinguish the 30 Springfields in the USA: each is situated in exactly one state, and each state is situated in exactly one country. In this model, given a `cityID`, we automatically know its state and its country. Further, in terms of data integrity, by limiting the user's choices to lookups in `cities` and `states` tables we make it impossible for a user to enter `Boston`, `Ontario` and `Hungary` for city, state and country. All we need is the `CityID` and we have everything else.

Fifth Normal Form (5NF)

Before we begin to explain fifth normal form, we must point out that you will rarely encounter it in the wild.

5NF is 4NF plus no *cyclic dependencies*. A cyclic dependency occurs in columns A, B, and C when the values of A, B and C are related in pairs. 5NF is also called *projection-join normal form* because the test for it is to project all dependencies to child tables then exactly reproduce the original table by rejoining those child tables. If the reproduction creates spurious rows, there is a cyclic dependency.

To illustrate 5NF, suppose we buy products to sell in our seed store, not directly from manufacturers, but indirectly through agents. We deal with many agents. Each agent sells many products. We might model this in a single table:

- `AgentID` - foreign key into the agents table
- `ProductID` - foreign key into the products table
- `SupplierID` - foreign key into the suppliers table
- `PrimaryKey` - a unique identifier for each row in this table

The problem with this model is that it attempts to contain two or more many-to-many relationships. Any given agent might represent *n* companies and *p* products. If she represents company `c`, this does not imply that she sells every product `c` manufactures. She may specialise in perennials, for example, leaving all the annuals to another agent. Or, she might represent companies `c`, `d` and `e`, offering for sale some part of their product lines (only their perennials, for example). In some cases, identical or equivalent products `p1`, `p2` and `p3` might be available from suppliers `q`, `r` and `s`. She might offer us cannabis seeds from a dozen suppliers, for example, each slightly different, all equally prohibited.

In the simple case, we should *create a table for each hidden relationship*:

- `AgentProducts` - the list of products sold by the agents. This table will consist of three columns minimally:
  - `AgentProductID` - the primary key
  - `AgentID` - foreign key pointing to the agents table.
  - `ProductID` - foreign key pointing to the products table.
- `ProductSuppliers` - the list of products made by the suppliers. Minimally:
  - `ProductSupplierID` - the primary key
  - `ProductID` - foreign key into the products table
  - `SupplierID` - foreign key into the suppliers table

Having refined the model to 5NF, we can now gracefully handle any combination of agents, products and suppliers.

3NF, BCNF, 4NF and 5NF are special cases of what is called **Domain/Key Normal Form**, which requires that only domain and key constraints limit column values

Not every database needs to satisfy 4NF or 5NF. That said, few database designs suffer from the extra effort required to satisfy these two rules.

Put the model to work

For a simple relational database design step-by-step, see *The basics of designing a database*. For a more advanced walkthrough see *Chapter 5*.

Limitations of the Relational Model

The CAP theorem

This theorem[8] says that of the three main requirements of a relational database …

1. Complete transactional consistency
2. Perfect availability
3. Perfect tolerance of partition failures

we can have just two at any one time. With most non-trivial database problems, some trade-offs are necessary. This topic is worth a book of its own. Indeed since MySQL first appeared, hundreds of non-relational database products have appeared, and some have become faddishly popular.

Is there a neat formula that decides which database model to apply to a problem? Unforunately not. We can only point to guidelines:

1. If the data consist largely of huge numbers of rather simple key-value pairs, or of time series, a key-value database like Apache Cassandra may be preferable to a relational database.

2. If the data consist largely of long chains of parent-child relationships, a graph database may be preferable.

3. If the data consist largely of documents, and especially if JSON is used, a document database may be preferable.

4. If the data consist of large amounts of raw data without many essential relationships, a "no-sql" database may be preferable.

Where is the "sweet spot" for relational databases? Roughly, data with necessary entity relationships that don't form huge time series or parent-child chains. Ultimately, you find out by applying a relational design algorithm[9] to the problem and appraising the result.

Dealing with Objects

Relational databases also fall short in the area of object storage. At the risk of cliché, objects are data with brains. An object's data may be mappable to tables, but its intelligence cannot be. Even object data tends to the complex. Typically it requires multiple rows in multiple tables.

To understand the complex issues involved here, consider a pulp and paper mill. A paper-making machine creates giant reels of paper (think of a roll of paper towels, 10 meters wide, 5 meters in diameter and unrolled, 100 kilometers long). These mammoth reels are then cut into numerous rolls (think of rolls of toilet tissue, one meter wide, one meter in diameter and 500 meters long). Some of these rolls are fed into sheeters, which cut them into large sheets that in turn are fed into machines that fold them and shape them and glue handles upon them until the final result is a bundle of shopping bags.

It is not impossible to model such complex relationships in a relational database, but it is difficult, and it is even more difficult to apply the practical rules of the pulp and paper business to this chain of products. You also face the non-trivial problem of mapping your class's attributes to a network of tables.

Once it looked as though object-relational databases would take hold. For many reasons that hasn't happened enough to significantly challenge the relational model's hold.

Dealing with Time

Even aside from time series, relational databases implement amnesia, leaving us two time problems to solve. One is the problem of *audit trails*. In some applications—especially accounting applications—the history of certain transactions or quantities is part of the current data. An interest-rate view, for example, must be able to show not only current rates but recent rates. A sales view must show not just the current view of any order, but its entire change history. The software requirement may include the capability to reconstruct any moment in the history of the database (point-in-time architecture, PITA). *Chapter 21* covers the basic methods. See “Audit trails” on our *MySQL Tips page* for example solutions.

The second time problem has to do with optimising for current changes or for history. Often a database designed for managing current information performs badly at historical reporting—and vice-versa. The bigger the organisation, the bigger the problem. It has given rise to an important distinction between two types of databases: those optimised for transactions (OLTP) and those optimised for analysis and reporting (OLAP).

Security, authentication

A relational database is a client-server application.Its databases must be secure, and writing passwords into client apps won’t cut it. Fundamentally, while client software may offer authentication user interfaces, the server needs to protect itself by maintaining sets of database privileges (*roles* in SQL) covering all actions on a database, define such roles in MySQL privilege tables, and define users in terms of usernames, login locations, pass-words and roles. Client authentication modules must work with this information. See *Chapter 3* and *19* for security basics; MySQL authentication architecture is detailed *here*.

Summary

Relational databases take their design principles from the branch of mathematics called set theory. They inherit the logical integrity of set theory and thus make it possible to store and retrieve the data that represents a given problem domain.

Normalisation is the process of designing an efficient and flexible database. Although often portrayed as rocket science and described in jargon whose first purpose is to support the continued employment of certain experts, normalisation is easy to understand and easy to implement.

At the least, your database should follow Codd's three axioms and twelve rules to the extent permitted by your RDBMS, and it should satisfy 3NF.

Not every database needs to satisfy 4NF and 5NF. Each of these refinements involves increased complexity as well as decreased performance. Weigh your choices carefully. Benchmark your results in each scenario, and factor in the likelihood that one day you may need to transform your database to 4NF or 5NF.

Successful databases change and evolve; failures whimper and die. Even if your design is flawless, the world outside your design will change. Your garden supplies company will come to sell wedding dresses, or (gasp) software. The more flexibility you build in, the less catastrophic such change will be.

References

1. Codd, EF. "A Relational Model of Data for Large Shared Data Banks." *CACM 13*, 6 (June 1970) pages 377-387.

2. Codd, EF. "Is Your DBMS Really Relational?", "Does Your DBMS Run By the Rules?" ComputerWorld, October 14/21, 1985.

3. Codd, EF. "The Relational Model for Database Management, Version 2." Addison-Wesley 1990.

4. Wikipedia, "The Relational Model", *http://en.wikipedia.org/wiki/Relational_model*

5. Database Tutorials on the Web, *http://www.artfulsoftware.com/dbresources.html*.

6. The Birth of SQL, *http://www.mcjones.org/System_R/SQL_Reunion_95/sqlr95-The.html*

7. Abbott, Edwin A. "Flatland". New York: Dover, 1990.

8. Wikipedia, “The CAP Theorem”, *http://en.wikipedia.org/wiki/CAP_theorem*

9. Brawley P, Fuller A, “Basics of designing a database”, *http://www.artfulsoftware.com/dbdesignbasics.html*

TOC ***Previous*** ***Next*** *Last updated 16 Jan 2021*