O'REILLY®

# What Is Database Design, Anyway?

C.J. Date

Strata+ Hadoop WORLD

Make Data Work

**strataconf.com**

Presented by O'Reilly and Cloudera, Strata + Hadoop World is where cutting-edge data science and new business fundamentals intersect—and merge.

- Learn business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Job # 15420

# What Is Database Design, Anyway?

*C.J. Date*

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

**What Is Database Design, Anyway?**
by C.J. Date

Copyright © 2016 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

**Editor:** Tim McGovern
**Production Editor:** Kristen Brown
**Interior Designer:** David Futato
**Cover Designer:** Karen Montgomery

December 2015: First Edition

**Revision History for the First Edition**
2015-12-04: First Release

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Cover photo by CEphoto Uwe Aranas / CC-BY-SA-3.0 Source: Wikimedia

978-1-491-94220-8

[LSI]

# What Is Database Design, Anyway?

> An earlier version of this essay appeared as a foreword to the book *Oracle SQL Developer Data Modeler for Database Design Mastery*, by Heli Helskyaho (Oracle Press, 2015). What follows is a revised and considerably expanded version of that foreword. My thanks to Heli and Oracle Press for allowing me to republish the essay here in its present form.

Databases lie at the heart of so much we do in the IT world that it's surely obvious that they need to be properly designed. Yet design theory—meaning database design theory specifically, of course—doesn't seem to be very well understood in the industry at large, and the same goes for design best practice also. You only have to look at the Wikipedia entry on database design to see the truth of these claims! In fact, before going any further, I'd like to quote a few sentences from that Wikipedia piece (with commentary by myself) as evidence in support of these claims:[1]

- **Database design** is the process of producing a detailed *data model* of a *database*. This *logical data model* contains all the needed logical and physical design choices and physical storage parameters needed to generate a design....

  *Comment:* So the "logical data model" contains "physical storage parameters"? Clearly, somebody is confused here, and I don't think it's me. Note too the circular nature of the foregoing "definition" (doing database design apparently consists of producing the things needed for doing database design). The fact that the

1 I've replaced the links in the original Wikipedia entry by italicized words and phrases, as in (e.g.) *logical data model*. Otherwise the extracts are quoted verbatim.

Wikipedia piece actually opens with the foregoing extract doesn't bode well for what's to come—but I suppose it might at least be argued that we've been given fair warning.

- The term database design can be used to describe many different parts of the design of an overall *database system*. Principally, and most correctly, it can be thought of as the logical design of the base data structures used to store the data. In the *relational model* these are the *tables* and *view* [*sic* singular "view"].

*Comment:* I'm going to argue later in this essay that database design isn't "principally and most correctly" about "the logical design of the base data structures" (at least, not exclusively), so I won't comment further on that particular issue now. I'm also going to say something later about the idea that "tables and views" are "used to store the data," so I won't comment on that issue now either. But I do want to say something about that phrase "tables and views." Sadly, that phrase appears all over the place in the database literature, including SQL documentation (even the SQL standard) in particular. But, clearly, anyone who talks this way is under the impression that tables and views are different things, and probably also that "tables" always means base tables specifically, and probably also that base tables are physically stored and views aren't (see my comments on the next quote below). But the whole point about a view is that it *is* a table—just as, in mathematics, the whole point about, say, the union of two sets is that it is a set. In mathematics we can perform the same kinds of operations on the union of two sets as we can on a regular set, because a union *is* a regular set. And in exactly the same kind of way, in the relational model we can perform the same kinds of operations on a view as we can on a regular table, because a view *is* a "regular table." So it's very important not to fall into the common trap of thinking that the term table always means a base table specifically. People who fall into that trap aren't thinking relationally, and they're likely to make mistakes as a consequence—mistakes in their database

designs, and mistakes in applications, and even, to some extent, mistakes in the design of the SQL language itself.[2]

- Once the relationships and dependencies amongst the various pieces of information have been determined, it is possible to arrange the data into a logical structure which can then be mapped into the storage objects supported by the *database management system*. In the case of *relational databases* the storage objects are *tables* which store data in rows and columns.

*Comment:* Tables in the relational model—even base tables—are most categorically *not* "storage objects"![3] The relational model deliberately has nothing to say regarding what's physically stored; in fact, it has nothing to say about physical storage matters at all. More specifically, it does *not* say that base tables are physically stored and views aren't. The only requirement is that there must be some mapping between whatever is physically stored and the base tables, so that those base tables can somehow be obtained when they're needed (conceptually, at any rate). If the base tables can be obtained from whatever's physically stored, then so can everything else. For example, we might physically store the join of the employees and departments base tables, instead of storing them separately; then those base tables could be obtained, conceptually, by taking projections of that join.

To repeat, the relational model has nothing to say about physical storage matters, and of course that omission was deliberate. The idea was to give implementers the freedom to implement the model in whatever way they chose—in particular, in whatever way seemed likely to yield good performance—without compromising on physical data independence. Unfortunately, most SQL product vendors seem not to have understood this point (or not to have risen to the challenge, at any rate); instead, they

2 Indeed, it could be argued that the very names of the operators CREATE TABLE and CREATE VIEW in SQL are and always were at least a psychological mistake, in that they tend to reinforce both (a) the idea that the term *table* means a base table specifically and (b) the idea that views and tables are different things.

3 I'm assuming here that (a) by "storage" the writer means physical storage specifically, and (b) by "tables" the writer means base tables specifically. Given the context, I believe these assumptions are both reasonable and noncontroversial.

do map base tables fairly directly to physical storage,[4] and their products thus provide far less physical data independence than relational systems are or should be capable of. But this state of affairs needs to be recognized for what it is—namely, a (major) defect in the products in question; it's not, and should not be taken to be, something that's intrinsic to the relational model as such.

- Each table may represent an implementation of either a logical object or a relationship joining one or more instances of one or more logical objects. Relationships between tables may then be stored as links connecting child tables with parents. Since complex logical relationships are themselves tables they will probably have links to more than one parent.

*Comment:* First, the writer is certainly playing pretty fast and loose with the language here. For example, an employee might perhaps be considered as a "logical object"; but then the employees table will "represent an implementation," not of that "logical object" as such, but rather of the set of all such "logical objects" currently existing in the business. (It would be better to use some other word than "joining" here too—perhaps "associating"?) Second, with respect to the phrase "logical object or a relationship": Well, it's one of the very great strengths of the relational model that it recognizes that what might be a "relationship" to one person, or one application, is a "logical object" to another (and vice versa). In other words, "relationships" *are* "logical objects" in the relational model, and they're represented in exactly the same way as all other "logical objects"—namely, by tables. Third, it follows that to talk of "relationships between tables" being "stored as links" is misleading in the extreme—in fact, totally wrongheaded. I mean, there's no such thing as a "link" in the relational model—there are only tables. Fourth, the (unexplained) terminology of "child and parent tables" is highly deprecated, for more reasons than I have space to go into here.

4 I say this knowing full well that today's SQL products do provide a variety of options for hashing, partitioning, indexing, clustering, and otherwise organizing the data as represented in physical storage. Despite this state of affairs, I still consider the mapping from base tables to physical storage in those products to be fairly direct. (For that very reason, in fact, elsewhere I've labeled those products "direct image systems." For further explanation see my book *Go Faster! The TransRelationalTM Approach to DBMS Implementation*, available as a free download from *http://bookboon.com*.)

Fifth, what's a "complex logical relationship"? More specifically, what would be an example of a relationship that's not "complex," or one that's not "logical"? As I've had occasion to write elsewhere, it's truly distressing in the relational context above all others—where precision of thought and articulation was always a key objective—to find such dreadfully sloppy phrasing. *Note:* The foregoing list of criticisms of this particular quote isn't meant to be complete. For example, what exactly does it mean to say (as the final sentence does) that relationships "are" tables? But I don't think any further deconstruction of the text is needed here. I think I've made my point.

- The physical design of the database specifies the physical configuration of the database on the storage media. This includes detailed specification of ... data types ... and other parameters....

  *Comment:* I'm sorry, but data types are most definitely a logical consideration, not a physical one! Unless—and this thought has only just crossed my mind, because it's almost beyond belief that someone could be so deeply muddled—by "data types" here the writer really means *representations?* (Well, I suppose I shouldn't be so surprised. In fact, I now recall that confusion over types vs. representations wasn't exactly unknown in certain earlier writings by certain other parties. But that was then and this is now, and I would have hoped that our understanding of such matters might have improved since then.)

Enough of Wikipedia; I think I've shown that I'm justified in complaining that database design theory and database design best practice seem not to be very well understood in the industry at large. In the rest of the present essay, therefore, what I'd like to do is try to inject some clarity into the debate; more specifically, I'd like to try to clarify exactly what database design really is, or ought to be. I'll start with some definitions.

- ***Database Design:*** Either logical database design or physical database design, as the context demands—though the unqualified term *database design*, or sometimes just *design*, is usually taken to mean logical database design specifically, unless the context demands otherwise.

- ***Logical Database Design (or just Logical Design):*** The process, or the result of the process, of deciding what tables some database should contain, what columns those tables should have, and what integrity constraints those tables and columns should be subject to. The goal of the logical design process is to produce a design that's independent of all considerations having to do with either physical implementation or specific applications (this latter objective being desirable for the very good reason that it's generally not the case that all uses to which the database will be put are known at design time). Overall, the logical design process can be summed up as one of (a) pinning down the table predicates and other business rules as carefully as possible, albeit necessarily somewhat informally, and then (b) mapping those informal predicates and rules to formally defined tables, columns, and integrity constraints—preferably in such a way as to ensure that the result of the process involves no uncontrolled redundancy. *Note:* I'll explain later what I mean by the terms *table predicate*, *business rule*, and *uncontrolled redundancy*.
- ***Physical Database Design (or just Physical Design):*** The process, or the result of the process, of deciding, given some logical design, how that design should map to whatever physical constructs the target DBMS happens to support. Observe, therefore, that the physical design should be derived from the logical design and not the other way around; ideally, in fact, it should be derived automatically, though I realize this might be a bit of a pipedream as far as most of today's commercial products are concerned.

For the remainder of this essay, I want to concentrate on logical design specifically. The first thing I want to say is that there does exist some science that can help with the logical design process; I refer, of course, to such matters as the principles of further normalization and the principle of orthogonal design. If you're a designer, therefore, you owe it to yourself—as well as to your clients, which is to say the people who are going to have to live with the databases you design—to be thoroughly familiar with those principles and to know how and when to apply them. (As an aside, I note that there's quite a bit more to the science than many people seem to realize. It's certainly not just a matter of making sure the tables are all in some

particular normal form. However, this isn't the place to go into details.[5])

The second thing I want to say is that although the science is important, there are, sadly, numerous aspects of design that the science doesn't address at all. And that's where practical experience comes in. If you do have a lot of personal experience in the design field, well, good for you—you'll have learned (possibly the hard way!) what works and what doesn't. But if you don't have much experience of your own to fall back on (and maybe even if you do), then you'll need sound advice you can follow, advice from someone who does have such experience. A good book on design, by a suitably qualified professional, can help meet that need. A word of caution, though: Books on database technology, as opposed to books on design specifically, might *not* be what you need here. Such books do often describe design concepts but fail to give much guidance on how to apply those concepts to the practical task of design. *Caveat lector*.

Let me now elaborate as I promised on those terms *table predicate*, *business rule*, and *uncontrolled redundancy*. First of all, the table predicate for a given table is simply a reasonably precise, but informal, statement in natural language of what the table in question means—in other words, it's a statement of how that table is supposed to be understood by users. For example, suppose we have a table called EMP ("employees"), with columns called ENO, ENAME, DNO, and SALARY. Then the predicate for that table EMP might look something like this:

> The person with employee number ENO is an employee of the company, is named ENAME, works in the department with department number DNO, and is paid salary SALARY.

ENO, ENAME, DNO, and SALARY are the *parameters* to this predicate, and of course they correspond to the columns of the table with those same names.

5 Those details can be found if you want them in my book *Database Design and Relational Theory: Normal Forms and All That Jazz* (O'Reilly, 2012). There's a video version too, also available from O'Reilly.

*Aside:* Perhaps I should take a moment to explain where this terminology of table predicates comes from. In logic, a predicate is basically just *a truth valued function*. Like all functions, it has a set of parameters; it returns a result when it's invoked; and (because it's truth valued) that result is either TRUE or FALSE. Here's a trivial example:

```
x > y
```

For this predicate, the parameters are *x* and *y*, and they stand for values of—let's agree for the sake of the example—type INTEGER. When we invoke this function, we substitute arguments (of the applicable types) for the parameters. Suppose we substitute the integers 8 and 5, respectively. We obtain the following statement:

```
8 > 5
```

This statement is in fact a *proposition*, which in logic is something that's unequivocally either true or false. (In the case at hand, of course, it's true; but if we substituted, say, 3 and 7 instead of 8 and 5 as the pertinent arguments, the resulting proposition would be false.)

Now let's get back to the predicate for table EMP. For that predicate the parameters are, as previously stated, ENO, ENAME, DNO, and SALARY, and they stand for values of (again let's agree for the sake of the example) types CHAR, CHAR, CHAR, and MONEY, respectively.[6] Now suppose we invoke this function—i.e., suppose we *instantiate this predicate*, as the logicians say—and substitute the arguments E4, Evans, D8, and 70K, respectively, for the parameters. We obtain the following proposition:

> The person with employee number E4 is an employee of the company, is named Evans, works in the department with department number D8, and is paid salary 70K.

And—here comes the point—the corresponding row (E4, Evans, D8, 70K) will appear in the EMP table if and only if this particular proposition is true. From a logical point of view, in fact, that's exactly what a "table" is—it's a set of rows, where the rows in question consist of all and only those rows whose column values correspond to true instantiations of some specified predicate; and that specified predicate is, precisely, the "table predicate" for the table in

6 See the subsequent discussion of business rules for more on the question of data types.

question. *Note:* Another way of saying the same thing is as follows: If row *r* appears in table *T*, then the proposition corresponding to *r* is true; conversely, if row *r* could appear in *T* but doesn't, then the proposition corresponding to *r* is false (where by "the proposition corresponding to *r*" I mean in both cases the instantiation of the table predicate for *T* that's obtained by substituting column values from *r* for the parameters of that predicate). This latter formulation constitutes what's usually known as *The Closed World Assumption. End of aside*.

Now I turn to the second of those terms I promised to explain, *business rule*. Like a table predicate, a business rule too is a reasonably precise but informal statement in natural language; however, it differs from a table predicate in its purpose, which is to capture some aspect of how the data in the database needs to be constrained:[7]

- To start with, there'll certainly be rules that specify what type of information is denoted by the parameters to those table predicates. In the case of employees, for example, there'll be a rule to the effect that the SALARY parameter ("salaries") denotes money values, expressed in, let's say, euros or U.S. dollars.
- Second, there'll be rules that constrain the values those parameters can take for a given employee considered in isolation. For example, there might a rule that says salaries mustn't be negative and must be less than some specified upper limit.
- Third, there'll be rules that constrain the set of employees taken as a whole, independent of other "entities" such as departments, that might be represented in the same database. For example, there might be a rule to the effect that employee numbers must be unique.
- Finally, there'll be rules that constrain employees considered in combination with other entities represented in the database. For example, there might be a rule to the effect that every employee must be assigned to some known department, or a rule to the effect that no employee can earn more than the manager of the department the employee in question is assigned to.

7 Actually, some writers regard table predicates as a special case of business rules. But there's much more to business rules in general than just the table predicates as such.

I'd like to say a bit more about this issue of business rules, because it's important—also because in practice it does tend to get somewhat overlooked. As the foregoing discussion should be sufficient to suggest, business rules can get quite complicated (as complicated as you like, in fact). As I've already said, however, they're necessarily somewhat informal. Their formal counterpart—i.e., the thing they map to in the logical design—is *integrity constraints* (constraints for short), which thus need to be stated in some formal language and enforced by the DBMS. In other words, I depart here from certain other writers in stating categorically that database design isn't just about choosing data structures—integrity constraints are crucial as well. (Of course, it's true that other writers usually do at least talk about key and foreign key constraints—sometimes cardinality constraints too—but these particular constraints are really nothing but important special cases of a much more general phenomenon.) In this connection, I'd like to draw your attention to the following remarks (somewhat paraphrased here) from *The Business Rule Book*, by Ron Ross (2nd edition, Business Rule Solutions Inc., 1997):

> Even though business rules (like the data itself) are "shared" and universal, traditionally they haven't been captured in database design. Instead, they've usually been stated vaguely (if at all) in largely uncoordinated analytical and design documents, and then buried deep in the logic of application programs. Since application programs are notoriously unreliable in the consistent and correct application of such rules, this has been the source of considerable frustration and error.

I couldn't agree more. Moreover, note the implicit but strong criticism of DBMS products that fail to provide adequate support for integrity constraints! (Interestingly, the support provided in this area by the SQL standard is actually not too bad. Unfortunately, however, SQL products have been rather slow, to say the least, in implementing this aspect of the standard.)

The third term I promised to explain is *uncontrolled redundancy*. Now, we often say, loosely, that the database displays redundancy if and only if "it says the same thing twice." We also often say, again loosely, that we don't want the database to display redundancy in this sense. However, it would be more accurate to say we don't want it to display any *uncontrolled* redundancy. Uncontrolled redundancy can be a problem, but controlled redundancy shouldn't be. Let me explain. First some more definitions:

- ***Controlled Redundancy:*** Redundancy in the database is controlled if the user is aware of it, but it's guaranteed never to lead to any inconsistencies.
- ***Uncontrolled Redundancy:*** Redundancy in the database is uncontrolled if it has the potential to lead to inconsistencies.
- ***Inconsistency:*** The database is inconsistent (at least from a formal point of view) if and only if there's some integrity constraint it's supposed to conform to but doesn't.

So if controlled redundancy means no inconsistencies, it must also mean no constraints are violated—at least, no constraints having to do with redundancy as such. Of course, not all constraints do have to do with redundancy as such; for example, a constraint to the effect that salaries mustn't be negative doesn't. Thus, if the database were to show some employee as having a negative salary it would certainly be inconsistent, but that particular inconsistency wouldn't be one that arises from redundancy. (It would, however, mean the database was incorrect, in the sense that it didn't faithfully reflect the state of affairs in the real world. Inconsistent implies incorrect, though the converse is false—the database can be incorrect without being inconsistent. For example, if it showed some employee as earning a salary different from that employee's true salary, it would be incorrect, but not inconsistent.)

To say it again, then, constraints don't always have to do with redundancy. But redundancy does always have to do with constraints. For example, suppose—very unrealistically!—that there's a constraint to the effect that all employees in the same department must earn the same salary. Suppose further that the database shows Heli and Chris as being in the same department. Then if it were also to show Heli and Chris as earning the same salary, it would be redundant; by contrast, if it were to show Heli and Chris as earning different salaries, it would be inconsistent (and incorrect).

So to say that the database involves some redundancy is to say some constraint is supposed to apply. The constraint in question, in the

case of the "same salary" example, might be formulated in SQL as follows:[8]

```
CREATE ASSERTION EX1 CHECK
   ( ( SELECT COUNT ( DISTINCT DNO ) FROM EMP ) =
     ( SELECT COUNT ( * ) FROM
            ( SELECT DISTINCT DNO , SALARY FROM EMP )
                                          AS POINTLESS ) ) ;
```

Stating this constraint explicitly serves to inform the user that the redundancy exists; enforcing it serves to ensure that it won't lead to any inconsistencies, thereby guaranteeing that the redundancy in question is controlled. Note, therefore, that we see once again, not incidentally, how important it is to be able to state integrity constraints formally and how important it is for the DBMS to be able to enforce them.

There's one more thing I want to say here. Some readers, I'm sure, will have found the foregoing remarks on consistency and redundancy a little puzzling, especially in view of the recent interest in what has come to be known as *eventual consistency* (in the context of those so called "NoSQL" systems in particular). So let me try to clarify those remarks:

- First of all, to repeat, to say that a database is consistent merely means, formally speaking, that the database conforms to all stated constraints. Now, it's crucially important that the database *always* be consistent in this formal sense; indeed, a database that's not consistent in this sense, at some particular time, is like a logical system that contains a contradiction. Well, actually, that's exactly what it is—a logical system with a contradiction. And in a logical system with a contradiction, you can prove *anything*; for example, you can prove that 1 = 0 (in fact, you can *prove* that you can prove that 1 = 0!). What this means in database terms is that if the database is ever inconsistent in the foregoing formal sense, you can never trust the answers you get to queries—they may be false, they may be true, and you have no way in general of knowing which they are. In other words, all bets are off. That's why consistency in the formal sense is so cru-

8 As you can see, the constraint in question is defined by means of a CREATE ASSERTION statement in SQL. For some reason, SQL sometimes (but not always!) calls constraints assertions. As for that AS *POINTLESS* specification, it's pointless, but it's required by the syntax rules of the SQL standard.

cial. It's also why, contrary to popular opinion, integrity checking must always be immediate—i.e., it must be done at the end of any update operation that has the potential to violate the integrity constraint in question. (In other words, so called "deferred checking" is a violation of the principles of the relational model; in fact, it's a logical error.)

- But consistency in the formal sense isn't necessarily the same thing as consistency as conventionally understood, meaning consistency as understood outside the world of databases in particular. Suppose there are two items *A* and *B* in the database that, in the real world, we believe should have the same value. They might, for example, both be the selling price for some commodity, stored twice because replication is being used to improve availability. If *A* and *B* in fact have different values at some given time, we might certainly say, informally, that there's an inconsistency in the data as stored at that time. But that "inconsistency" is an inconsistency as far as the system is concerned *only if the system has been told that A and B are supposed to be equal*—i.e., only if "$A = B$" has been declared as a formal constraint. If it hasn't, then (a) the fact that $A \neq B$ at some time doesn't in itself constitute a consistency violation as far as the system is concerned, and (b) importantly, the system will never rely on an assumption that *A* and *B* are equal.
- Thus, if all we want is for *A* and *B* to be equal "eventually"—i.e., if we're content for that requirement to be handled in the application layer—all we have to do as far as the database system is concerned is omit any declaration of "$A = B$" as a formal constraint. No problem, and in particular no violation of the relational model.

With that, I'll conclude these brief remarks on database design. I'd like to thank Heli for giving me the chance to air my opinions on this topic, and Hugh Darwen for taking the time to comment on what I had to say (his comments led to several improvements and clarifications in the text). And, of course, I'd like to wish Heli and her book every success in her own chosen field.

# About the Author

**C.J. Date** has a stature that is unique within the database industry. He is a prolific writer, and is well known for his best-selling textbook *An Introduction to Database Systems* (Addison Wesley). He is an exceptionally clear-thinking writer who can lay out principles and theory in a way easily understood by his audience.