Whitepaper

embarcadero

Normalization Heuristics

By Joe Celko

For Embarcadero Technologies

December, 2015

Table of Contents

The dictionary definition of a heuristic is
"Computers, Mathematics: pertaining to a trial-and-error method of problem solving used when an algorithmic approach is impractical." which means this is what you do when you do not know what to do next.

The term "normalization" comes from Dr. Codd, who borrowed it from the then-current political climate in which we were trying to normalize relations with the Soviet block. The goal is to remove redundancy from a database schema and to have the schema maintain some data integrity without procedural code.

Actually, these are the goals of all databases. When we had file systems, the same data was repeated in many different files and there was no reasonable way to bring the files together and create a consistent model of the data. Then we got the network model and the idea of a consistent, single picture of our data.

Dr. Codd's relational model in 1970 went one step further. The relational model could be formalized and we even got Armstrong's Axioms and math! The goal of normal forms is to avoid certain data anomalies that can occur in unnormalized tables. Data anomalies are easier to explain with examples. When Dr. Codd defined the relational model, he gave 0 to 12 rules for tables (Yes, there is a rule zero). Some of them are important for table design heuristics, so it is good to get feel for them.

The most important one for a working programmer is The Information Rule. This simply requires all information in the database to be represented in one and only one way, namely by scalar values in columns within rows of tables.

The problem is people do not know what "scalar" means, so they want to use structured data in the form of arrays and CSV lists. NO! They did not get a class on scales and measurements. It is how they handled data in an old FORTRAN or COBOL program, so why should it change now?

Dr. Codd's third rule is the "Systematic Treatment of NULL Values" in the data model. SQL has a NULL that is used for both missing information and inapplicable information, but you have to decide what each a NULL means as part of the data model. The heuristic is to declare every column as NOT NULL, then go back and deliberately decide to make it NULL-able.

Heuristic #0: A Schema Should Have Few, If Any, NULL-able Columns.

This is important because a key cannot have NULLs. It also tells us that we have not designed the data carefully. We ought to know as much as possible about the model before we started coding. I just had to do a zero in my list because Codd had one.

Rule 8. "Physical Data Independence" This is self-explanatory; users are never aware of the physical implementation and deal only with a logical model. Any real product is going to have some physical dependence, but SQL is better than most programming languages on this point.

In particular, auto-incrementing row identifiers based on physical insertions into a table like the IDENTITY table property in MS SQL Server are in total violation of this rule. So are GUIDs used inside the schema (they locate global resources, not local schema objects). So are byte and bit operators. Why would you think that the physical storage layout is part of a model? Is your hard ware High end? Low end? 16 bit? 32 bit? 64 bit ?

Rule 10."Integrity Independence" This means Declarative Referential Integrity (DRI) constraints must be specified separately from application programs and stored in the schema. This means keys, defaults, DRI actions, triggers, and check constraints are part of the design, you cannot skip them and expect to have a correct schema. I tell students that 85-95% of the real work in SQL is done in the DDL, not in the DML.

Codd also specified 9 structural features, 3 integrity features, and 18 manipulative features, all of which are required as well. He later extended the list from 12 rules to 333 in the second version of the relational model. This section is getting too long and you can look them up for yourself.

Normal forms are an attempt to make sure that you do not destroy true data or create false data in your database. One of the ways of avoiding errors is to represent a fact only once in the database, since if a fact appears more than

once, one of the instances of it is likely to be in error at some time – a man with two wrist watches can never be sure what time it is.

Functional and Multi-Valued Dependencies

A normal form is a way of classifying a table based on the functional dependencies (FDs for short) in it. A functional dependency means that if I know the value of one attribute, I can always determine the value of another. The notation used in relational theory is an arrow between the two attributes, for example A→ B, which can be read in English as "A determines B". If I know your employee number, I can determine your name; if I know a part number, I can determine the weight and color of the part; and so forth.

A multi-valued dependency (MVD) means that if I know the value of one attribute, I can always determine the values of a set of another attribute. The notation used in relational theory is a double-headed arrow between the two attributes, for instance A ↠ B , which can be read in English as "A determines many Bs". If I know a teacher's name, I can determine a list of her students; if I know a part number, I can determine the part numbers of its components; and so forth.

Okay, so much for abstractions. Let's clean up a file and turn it into an SQL database. The Normal forms are numbered and names, each one built on a simpler normal form. Just like Dr. Codd's rules, you do not have to know all of them by heart, but some of them are important.

First Normal Form (1NF)

Consider a requirement to maintain data about class schedules at a school. We are required to keep the course_name, class_section, dept_name, time, room_nbr, professor, student, student_major, and student_grade. Suppose that we initially set up a Pascal file with records that look like this:

```
Classes = RECORD
   course_name: ARRAY [1:7] OF CHAR;
  class_section: CHAR;
   time_period: INTEGER;
    room_nbr: INTEGER;
    room_size: INTEGER;
    professor: ARRAY [1:25] OF CHAR;
    dept_name: ARRAY [1:10] OF CHAR;
    students: ARRAY [1:class_size]
       OF RECORD
         student_name ARRAY [1:25] OF CHAR;
         student_major ARRAY [1:10] OF CHAR;
         student_grade CHAR;
         END;
   END;
```

If you do not read Pascal, it is easy. Records are read from files in left to right. There is no string data type; the language uses an array of characters. Integers explain themselves. Records can be structured inside each other as arrays; this is like the OCCURS clause in COBOL.

First Normal Form (1NF) means that the table has no repeating groups. That is, every column is a scalar value, not an array or a list or anything with its own structure. In SQL, it is impossible not to be in 1NF unless the vendor has added array or other extensions to the language. The Pascal record could be "flattened out" in SQL and the field names changed to data element names to look like this:

```
CREATE TABLE Classes
 (course_name CHAR(7) NOT NULL,
  class_section CHAR(1) NOT NULL,
  time_period INTEGER NOT NULL,
  room_nbr INTEGER NOT NULL,
  room_size INTEGER NOT NULL,
  professor_name CHAR(25) NOT NULL,
  dept_name CHAR(10) NOT NULL,
  student_name CHAR(25) NOT NULL,
  student_major CHAR(10) NOT NULL,
  student_grade CHAR(1) NOT NULL);
```

This table is acceptable to SQL. But it has no keys and repeats huge amounts of data. In fact, we can locate a row in the table with a combination of (course_name, class_section, student_name), so we have an undeclared key. But what we are doing is hiding the Students record array, which has not changed its nature by being flattened. And this thing has no keys, and no constraint declared. But at least there are no NULLs.

There are problems.

If Professor 'Jones' of the math department dies, we delete all his rows from the Classes table.

```
DELETE FROM Classes WHERE professor_name = 'Jones';
```

This also deletes the information that all his students were taking a math class and maybe not all of them wanted to drop out of school just yet. I am deleting more than one fact from the database. This is called a deletion anomaly.

If student 'Wilson' decides to change one of his math classes, formerly taught by Professor 'Jones', to English, we will show Professor 'Jones' as an instructor in both the math and the English departments.

```
UPDATE Classes
  SET course_name = 'English'
WHERE student_name = 'Wilson';
```

I could not change a simple fact by itself. This creates false information, and is called an update anomaly.

If the school decides to start a new department, which has no students yet, we cannot put in the data about the professor we just hired until we have classroom and student data to fill out a row. I cannot insert a simple fact by itself. This is called an insertion anomaly.

There are more problems in this table, but you see the point. Yes, there are some ways to get around these problems without changing the tables. We could permit NULLs in the table. We could write triggers to check the table for false

data. These are tricks that will only get worse as the data and the relationships become more complex. The solution is to break the table up into other tables, each of which represents one relationship or simple fact.

At this point I can do some functional and MVD dependencies, apply some axioms and split this monster table into more tables with math. But the heuristic is that you can now look at each column and decide if it is an attribute or a key. You need to find the entities and relationships mashed together in this table.

Heuristic #1 Look for things you can touch; they will be entitles.

In this example, a student is a thing that can be touched. We want nouns that are the classic "person, place or thing" we were taught in grade school.

Heuristic #2 Look for keys.

How do you identify that entity? Remember a key has to be subset of attributes; not a GUID, not an IDENTITY, not some other physical locator. This is the a direct application

of the Law of Identity from formal logic. "To be is to be something in particular; to be nothing in particular or everything in general is to be nothing at all" This was first used in Plato's dialogue "Theaetetus" and it is wrongly attributed to Aristotle.

Heuristic #3 Look for non key attributes.

This is actually trickier. Think about an author; to be an author, her has to have written book, right? Well, yes. But is the book an attribute? Does it grow out of his chest? We have "authorship" as a relationship between a book and an author.

These basic relationships can be one to one (1:1), one to many (1:m) or many to many (n:m). They can be enforces with constraints, but do not worry about this for now.

We now get our first table:

```
CREATE TABLE Students
(student_name CHAR(25) NOT NULL PRIMARY KEY,
 student_major CHAR(10) NOT NULL);
```

Not much to this table, is there? As a generalization, files will have lots of fields in a few records while an SQL schema will have a lot of small tables whose columns are interrelated and constrained. For now, just ignore using a student's name instead of an identifier number of some kind for the key.

Let's put the students into their classes and set up a roster with the enrollments. The enrollment has to have a student, a class to attend and a grade for that work.

Heuristic #4 Look for multi-columns keys.

In this example, we identify a class by the course name and a section number for the courses that overflowed. This is a common pattern, where we have a strong attribute (the course) and a weak attribute (the section within the course). Weak attributes and exist only with a strong attribute. The most common example in business is a document header, say an invoice, and its weak details, say invoice line items. In theory there is no limit to how far down the nesting can go.

But there is another kind of multi-column key, where the columns are on an equal level. Ever use (longitude, latitude) pairs? Let's set up the grade book for the class.

```
CREATE TABLE Enrollment
(student_name CHAR(25) NOT NULL,
 course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 student_grade CHAR(1) NOT NULL,
PRIMARY KEY (student_name, course_name, class_section));
```

Now that we have a grade book, we need a place and person to teach a class.

```
CREATE TABLE Classes
(course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 time_period INTEGER NOT NULL,
 room_nbr INTEGER NOT NULL,
 room_size INTEGER NOT NULL,
 professor_name CHAR(25) NOT NULL,
PRIMARY KEY (course_name, class_section));
```

At this point, we are in Second Normal Form (2NF). That means every attribute depends on the entire key in its table. Now if a student changes majors, it can be done in one place. Furthermore, a student cannot sign up for different sections of the same class, because we have changed the key of Enrollment. Unfortunately, we still have problems.

Notice that while room_size depends on the entire key of Classes, it also depends on room_nbr. If the room_nbr is changed for a course_name and class_section, we may also have to change the room_size, and if the room_nbr is modified (we knock down a wall), we may have to change room_size in several rows in Classes for that room_nbr.

This is bad. When we change one fact, we want to do it in place, one time, one way. And this leads to the next heuristic.

This query uses a characteristic function while my original version compares a count of Personnel under each manager to a count of Personnel under each project_id. The use of "GROUP BY M1.mgr_name, P1.dept_id_name, P2.project_id" with the "SELECT DISTINCT M1.mgr_name, P1.dept_id_name" is really the tricky part in this new query. What we have is a three-dimensional space with the (x, y, z) axis representing

Heuristic #5 a normalized schema has one fact, in one place, one way, one time

If I have to change other things along with the target data element, then I need to do some more work. In this example, the room number will determine the size

of the room. And this leads use to the most common normalization we want to have in a schema!

Third Normal Form (3NF)

A table is in Third Normal Form (3NF) if it is in 2NF and for all X→ Y, (the arrow reads "determines") where X and Y are columns of a table, X is a key or Y is part of a candidate key. (A candidate key is a unique set of columns that identify each row in a table; you cannot remove a column from the candidate key without destroying its uniqueness.) This implies that the table is in 2NF, since a partial key dependency is a type of transitive dependency.

Informally, all the non-key columns are determined by "the key, the whole key, and nothing but the key, so help you Codd!"; this phrase is attributed to Chris Date

The usual way that 3NF is explained is that there are no transitive dependencies, but this is not quite right. A transitive dependency is a situation where we have a table with columns (A, B, C) and (A→ B) and (B→ C), so we know that (A→ C). In our case, the situation is that (course_name, class_section)→ room_nbr and room_nbr→ room_size. This is not a simple transitive dependency, since only part of a key is involved, but the principle still holds. To get our example into 3NF and fix the problem with the room_size column, we make the following decomposition:

```
CREATE TABLE Rooms --- another thing I can touch!
(room_nbr INTEGER NOT NULL PRIMARY KEY,
 room_size INTEGER NOT NULL);
```

```
CREATE TABLE Students
(student_name CHAR(25) NOT NULL PRIMARY KEY,
 student_major CHAR(10) NOT NULL);
```

Let's start adding DRI actions to the schema.

```
CREATE TABLE Classes
(course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 PRIMARY KEY (course_name, class_section),
 time_period INTEGER NOT NULL,
 room_nbr INTEGER NOT NULL  -- can find the size in Rooms
 REFERENCES Rooms(room_nbr));
```

```
CREATE TABLE Enrollment
(student_name CHAR(25) NOT NULL
 REFERENCES Students(student_name),
 course_name CHAR(7) NOT NULL,
 class_section CHAR(1) NOT NULL,
 PRIMARY KEY (student_name, course_name, class_section),
 student_grade CHAR(1) NOT NULL);
```

A common misunderstanding about relational theory is that 3NF tables have no transitive dependencies. As indicated above, if X → Y, X does not have to be a key if Y is part of a candidate key. We still have a transitive dependency in the example -- (room_nbr, time_period) → (course_name, class_section) -- but since the right side of the dependency is a key, it is technically in 3NF. The unreasonable behavior that this table structure still has is that several course_names can be assigned to the same room_nbr at the same time.

Another form of transitive dependency is a computed column. For example:

```
CREATE TABLE Boxes
(width INTEGER NOT NULL,
 length INTEGER NOT NULL,
 height INTEGER NOT NULL,
 volume INTEGER NOT NULL
     CHECK (width * length * height = volume),
 PRIMARY KEY (width, length, height));
```

The volume column is determined by the other three columns, so any change to one of the three columns will require a change to the volume column. You can use

a computed column in this example which would look like:

```
(volume INTEGER COMPUTED AS (width * length * height) PERSISTENT)
```

Heuristics #6: Try not to store computed data; store what you need for the computations.

There are several other normal forms. Occasionally you will need to worry about Boyce-Codd Normal Form (BCNF) and Fifth Normal Form (5NF). Be aware they exists and you can handle them. Without the math, just know what they smell like. They deal with trying to do loss-less decomposition of the tables. In English, that means you can put the new tables back together with joins.

For example, we might give a fountain pen to a beginning salesman with a base pay rate between $15,000.00 and $20,000.00 and 100 gift_points, but give a car to a master salesman, whose salary is between $30,000.00 and $60,000.00 and who has 200 gift_points. The functional dependencies are, therefore,

(pay_step, gift_points)→ gift_name
gift_name → gift_points

Gifts

salary_amt	gift_points	gift_name
15,000.00	100	'Pencil'
17,000.00	100	'Pen'
30,000.00	200	'Car'
31,000.00	200	'Car'
32,000.00	200	'Car'

Let's start with a table that has all the data in it and normalize it.

```
CREATE TABLE Gifts
(salary_amt DECIMAL(8,2) NOT NULL
 gift_points INTEGER NOT NULL,
 PRIMARY KEY (salary_amt, gift_points),
 gift_name VARCHAR(10) NOT NULL);
```

This schema is in 3NF, but it has problems. You cannot insert a new gift into our offerings and points unless we have a salary to go with it. If you remove any sales points, you lose information about the gifts and salaries (e.g., only people in the $30,000.00 to $32,000.00 range can win a car). And, finally, a change in the gifts for a particular point score would have to affect all the rows within the same pay step. This table needs to be broken apart into two tables:

Pay_Gifts

Salary_amt	Gift_name
15,000.00	'Pencil'
17,000.00	'Pen'
30,000.00	'Car'
31,000.00	'Car'
32,000.00	'Car'

```
CREATE TABLE Gifts
(salary_amt DECIMAL(8,2) NOT NULL,
 gift_points INTEGER NOT NULL,
 PRIMARY KEY(salary_amt, gift_points),
 gift_name VARCHAR(10) NOT NULL);
```

Gift_points

gift_name	gift_points
'Pencil'	'100'
'Pen'	'100'
'Car'	'200'

(salary_amt, gift_points) → gift

gift → gift_points

```
CREATE TABLE GiftsPoints
(gift_name VARCHAR(10) NOT NULL PRIMARY KEY,
 gift_points INTEGER NOT NULL));
```

Fifth Normal Form (5NF)

Fifth Normal Form (5NF), also called the Join-Projection Normal Form or the Projection-Join Normal Form, is based on the idea of a lossless JOIN or the lack of a join-projection anomaly. This problem occurs when you have an n-way relationship, where (n > 2). A quick check for 5NF is to see if the table is in 3NF and all the candidate keys are single columns. This is not the only configuration, but it is the most common.

As an example of the problems solved by 5NF, consider a table of house notes that records the buyer, the seller, and the lender:

HouseNotes

buyer	seller	lender
'Smith'	'Jones'	'NationalBank'
'Smith'	'Wilson'	'HomeBank'
'Nelson'	'Jones'	'Homebank'

This table is a three-way relationship, but because older diagramming tools allow only binary relationships it might have to be expressed in an E-R diagram as three binary relationships, which would generate CREATE TABLE statements leading to these tables:

BuyerLender

buyer	lender
'Smith'	'NationalBank'
'Smith'	'HomeBank'
'Nelson'	'Homebank'

SellerLender

seller	lender
'Jones'	'NationalBank'
'Wilson'	'HomeBank'
'Jones'	'Homebank'

BuyerrLender

buyer	seller
'Smith'	'Jones'
'Smith'	'Wilson'
'Nelson'	'Jones'

The trouble is that when you try to assemble the original information by joining pairs of these three tables together, thus:

```
SELECT BS.buyer, SL.seller, BL.lender
  FROM BuyerLender AS BL,
    SellerLender AS SL,
    BuyerSeller AS BS
  WHERE BL.buyer = BS.buyer
  AND BL.lender = SL.lender
  AND SL.seller = BS.seller;
```

you will recreate all the valid rows in the original table, such as ('Smith', 'Jones', 'National Bank'), but there will also be false rows, such as ('Smith', 'Jones', 'Home Bank'), which were not part of the original table. This is called a join-projection anomaly.

Heuristics #7: The most important leg on three-legged stool is the leg that is missing

When you are trying to sell your house, you need a lender and a buyer to make a sale. You are the third leg. Likewise, banks are looking for buyers and sellers, while buyers want to find their bank and their dream house.

Using NULLs for the "missing legs" is not usually a good idea. If we had made all three columns NULL-able, we could never have had a key at all. Want to try to write

a constraint that keeps two of the three columns non-NULL? Now try to write a simple query on that

table. The DDL and DML become more and more complex until it is impossible to optimize the SQL or even maintain it. As God says to the Angles in THE GREEN PASTURES (ISBN: 0299079244, page 84), "Dat's always de trouble wid miracles. When you pass one you always gotta r'ar back an' pass another."

Heuristics #8: if it is not an entity, is it a relationship? Does the relationship have its own attributes?

The simple home loan example we just saw is a pure relationship among three entities. But in the real world, there would be a loan number, purchase price, signature dates, and lot of other stuff. The loan number will be a key, but not the only key, as we just saw.

Non-Normal Form Redundancy

Normalization prevents some redundancy in a table. But not all redundancy is based on Normal Forms. We saw how a computed column could be used to replace a base column when the base column is a redundant computation. The computation is done at processor speeds (nanoseconds today) while reading it off of a moving disk is done at mechanical speeds (microseconds).

Now, move up a level in the schema.

Aggregation Level Redundancy

A common example is the “Invoices” and “Invoice_Details” idiom which puts detail summary data in the order header. This is usually a column for “invoice_total” which has to be re-computed when an order item changes. What has happened is a confusion in levels of aggregation.

```
CREATE TABLE Invoices
(invoice_nbr CHAR(15) NOT NULL PRIMARY KEY,
 customer_name VARCHAR(35) NOT NULL,
 invoice_terms CHAR(7) NOT NULL
  CHECK (invoice_terms IN ('cash', 'credit', 'coupon')),
 invoice_amt_tot DECIMAL(12,2) NOT NULL);
```

```
CREATE TABLE Invoice_Details
(invoice_nbr CHAR(15) NOT NULL
  REFERENCES Invoices (invoice_nbr)
  ON DELETE CASCADE,
  line_nbr INTEGER NOT NULL
  CHECK (line_nbr > 0),
  item_gtin CHAR(15) NOT NULL,
 -- PRIMARY KEY (invoice_nbr, line_nbr),
-- PRIMARY KEY (invoice_nbr, item_gtin),
  invoice_qty INTEGER NOT NULL
 CHECK (invoice_qty > 0),
 unit_price DECIMAL(12,2) NOT NULL)
```

There is redundancy in line_ nbr and item_gtin as components in a key. (GTIN = Global Trade Item Number (GTIN) can be used by a company to uniquely identify all of its trade items. It is the UPC barcodes on steroids) The invoice line numbers are physical locations on paper forms or a screen. A line number lets you place one product (item_gtin) in several places on the order form. Line numbers are not part of a logical data model. Wrong!

But did you notice that Invoices.invoice_amt_tot = SUM (Invoice_Details.invoice_qty * Invoice_Details.unit_price)?

Entire Table Redundancy

Entire tables can be redundant. This often happens when there are two different ways to identify the same entity.

```
CREATE TABLE Map
(location_id CHAR(15) NOT NULL PRIMARY KEY,
 location_name VARCHAR(35) NOT NULL,
 location_longitude DECIMAL(9,5) NOT NULL),
 location_latitude DECIMAL(9,5) NOT NULL);
 CHECK (invoice_qty > 0),
 unit_price DECIMAL(12,2) NOT NULL)
```

location_id is the key, This might be a HTM (Hierarchical Triangular Mesh) number or a SAN (Standard Address Number, used in the Book Industry and other places). I can use a formula to compute the distance between two locations with this table. But I can also build a table of straight line distances directly:

```
CREATE TABLE Paths
(origin_location_id CHAR(15) NOT NULL,
 dest_location_id CHAR(15) NOT NULL,
 straight_line_dist DECIMAL(10,3) NOT NULL,
 PRIMARY KEY (origin_location_id, dest_location_id));
```

This is an actual case from Tom Johnston. The (longitude, latitude) coordinate pairs would get out of alignment with the distance computations because they were maintained by two different people. The solution was VIEW to construct Paths when needed.

Access Path Redundancy

A more subtle redundancy is in the roles an entity plays in a data model. Try this example: A sales team is responsible for every customer that a member of that team (a salesperson) is assigned to and not responsible for any other customer.

Draw this and you will see a cycle among

Now look at the redundant relationship. We have options.

1. Eliminate the redundancy: remove the is-responsible-for relationship from Sales-Team to Customer. The model is just as expressive as it was before

the redundancy was eliminated.

2. Control the redundancy: add DRI actions. Because the two foreign keys that must be kept synchronized are in the same row, only one update is required.

Here is the DDL for the possible solutions:

```
CREATE TABLE Sales_Teams
(sales_team_id INTEGER NOT NULL PRIMARY KEY,
 sales_team_name CHAR(10) NOT NULL);
```

```
    CREATE TABLE Salespersons
    (sales_person_id INTEGER NOT NULL PRIMARY KEY,
     sales_person_name CHAR(15) NOT NULL,
     sales_team_id INTEGER NOT NULL
          REFERENCES Sales_Teams(sales_team_id)
          ON UPDATE CASCADE);
```

```
   CREATE TABLE Customers
    (customer_id INTEGER NOT NULL PRIMARY KEY,
     sales_team_id INTEGER NOT NULL
           REFERENCES SalesPerson(sales_team_id)
           ON UPDATE CASCADE,
     sales_person_id INTEGER
        REFERENCES Salespersons(sales_person_id)
        ON UPDATE CASCADE
        ON DELETE SET NULL);
```

Another possible schema:

```
    CREATE TABLE Sales_Teams
     (sales_team_id INTEGER NOT NULL PRIMARY KEY,
      sales_team_name CHAR(10) NOT NULL);
```

```
CREATE TABLE Salespersons
 (sales_person_id INTEGER NOT NULL PRIMARY KEY,
  sales_person_name CHAR(15) NOT NULL,
  sales_team_id INTEGER NOT NULL
      REFERENCES Sales_Teams(sales_team_id)
      ON UPDATE CASCADE,
  UNIQUE (sales_person_id, sales_team_id));
```

```
CREATE TABLE Customers
(customer_id INTEGER NOT NULL PRIMARY KEY,
 sales_team_id INTEGER NOT NULL ,
 sales_person_id INTEGER
    REFERENCES Salespersons(sales_person_id)
    ON UPDATE CASCADE ON DELETE SET NULL,
  FOREIGN KEY (sales_person_id, sales_team_id)
 REFERENCES Salespersons (sales_person_id, sales_team_id)
       ON UPDATE CASCADE);
```

Attribute Splitting

Would you have a "Female_Personnel" and a "Male_Personnel" table in a schema? No, of course not! We need a "Personnel" table, not two tables constructed by using the values in a "sex_code" column as the splitter for table names. Making a table per calendar year (or month) is very common because it looks like how we did magnetic tapes. Another common split is a physical data source, such as each store in an enterprise.

Chris Date calls it "Orthogonal design" and I call it "Attribute Splitting", but I use this for more general errors than just tables.

This is not disk partitioning! That is a physical vendor feature for accessing the data, but the table is still one logical unit in the schema.

Splitting can be at the table level, too. In the school example, you can make a design decision to keep course name and class section as separate columns or concatenate them into one column. However, you will see clear splits in the case of a unit of measure in one column and the measurement in a secodn column.

The Summary

Broadly speaking, tables represent either entities, relationships or they are auxiliary tables. This is why E-R diagrams work so well as a design tool. The auxiliary tables do not show up on the diagrams, since they are functions, translations, and look-ups that support a declarative computational model.

The tables that represent entities should have a simple, immediate name suggested by their contents -- a table named Students has student data in it, not student data and their bowling scores. It is also a good idea to use plural or collective nouns as the names of such tables to remind you that a table is a set of entities; the rows are the single instances of them.

Tables which represent one or many to one or many relationships should be named by their contents and should be as minimal as possible. For example, Students are related to Courses by a third (relationship) table for their attendance. These tables might represent a pure relationship or they might contain attributes that exist within the relationship, such as a student_grade for the class attended. Since the only way to get a student_grade is to attend the class, the relationship is going to have a compound key made up of references to the entity keys. We will probably name it "ReportCards", "Grades" or something similar. Avoid naming entities based on m:m relationships by combining the two table names. for example, “Students_Courses” is an easy but really bad name for the “Enrollment” entity.

Avoid NULLs whenever possible. If a table has too many NULL-able columns, it is probably not normalized properly. Try to use a NULL only for a value which is missing now, but which will be resolved later. Even better, put missing values into the encoding schemes for that column. I have a whole book on this topic, SQL PROGRAMMING STYLE (ISBN 978-0120887972) and mention in other books.

As a gross generalization, normalized databases will tend to have a lot of tables with a small number of columns per table. Do not panic when you see that happen. People who first worked with file systems (particularly on computers that used magnetic tape) tend to design one monster file for an application and do all the work against its records. This made sense in the old days, since there was no

reasonable way to JOIN a number of small files together without having the computer operator mount and dismount lots of different magnetic tapes. The habit of designing this way carried over to disk systems, since the procedural programming languages were still the same for the databases as they had been for the sequential file systems.

The same non-key attribute in more than one table is probably a normalization problem. This is not a certainty, just a guideline. The key that determines that attribute should be in only one table, and therefore its attributes should be with it. The key attributes will be referenced and not repeated by related tables.

As a practical matter, you are apt to see the same attribute under different names and need to make the names uniform in the entire database. The columns "date_of_birth", "birthdate", "birthday", and "dob" are very likely the same attribute of an employee. You now have the ISO-11179 for naming guidelines, as discussed in SQL PROGRAMMING STYLE (Morgan Kaufmann; 2005 May 01; ISBN: 978-0120887972).