

# Design Your Own Database Concept to Implementation

## or How to Design a Database Without Touching a Computer

The following is an aggregation of several online resources with a bit of personal insight and experience thrown in for good measure. -m

### ***Keys to Successful Database Design***

Planning, Planning, and Planning. Oh did I mention Planning? Seriously, planning is the largest most significant aspect of database design and is often sold short for the *raison de jour*. This results in databases which do not meet requirements, do not meet expectations, or are otherwise unwieldy. Before switching on your computer and getting geeky using your tools of choice you should sit down and, with pencil and paper in hand, verify your requirements, diagram your data, and plan your database. Then proceed quickly with more paper and pointy pencils to refine a design. Then and only then should you turn on your computer and start coding the database. But I am getting ahead of myself.

**Note:** Regrettably, discussions on database design tend to suffer from a special, rather non-intuitive terminology. I will try to offer explanations of expressions that you are likely to encounter in the text as they come up.

### ***Pitfalls We Wish to Avoid***

#### *Redundancy Versus Loss of Data*

When designing our schema, we want to do so in such a way that we minimize redundancy of data without losing any data. By redundancy, I mean data that is repeated in different rows of a table or in different tables in the database.

Imagine that rather than having an employee table and a department table, we have a single table called employeeDepartment. We can accomplish this by adding a single departmentName column to the employee table so that the schema looks like this:

```
Table: employeeDepartment
employeeID  name  job      departmentID  departmentName
```

For each employee who works in the Department with the number 128, Research and Development, we will repeat the data "128, Research and Development." This will be the same for each department in the company. This schema design leads to redundantly storing the department name over and over.

We can change this design as shown here:

```
Table: employee
employeeID  name  job      departmentID
```

```
Table: department
departmentID  name
```

In this case, each department name is stored in the database only once, rather than many times, minimizing storage space and avoiding some problems.

Note that we must leave the departmentID in the employee table; otherwise, we lose information from

the schema, and in this case, we would lose the link between an employee and the department the employee works for. In improving the schema, we must always bear these twin goals in mind—that is, reducing repetition of data without losing any information.

#### *Anomalies*

Anomalies present a slightly more complex concept. Anomalies are problems that arise in the data due to a flaw in the database design. There are three types of anomalies that may arise, and we will consider how they occur with the flawed schema described above.

##### *Insertion Anomalies*

Insertion anomalies occur when we try to insert data into a flawed table. Imagine that we have a new employee starting at the company. When we insert the employee's details into the employeeDepartment table, we must insert both his department id and his department name. What happens if we insert data that does not match what is already in the table, for example, by entering an employee as working for Department 42, Development? It will not be obvious which of the rows in the database is correct. This is an insertion anomaly.

##### *Deletion Anomalies*

Deletion anomalies occur when we delete data from a flawed schema. Imagine that all the employees of Department 128 leave on the same day (walking out in disgust, perhaps). When we delete these employee records, we no longer have any record that Department 128 exists or what its name is. This is a deletion anomaly.

##### *Update Anomalies*

Update anomalies occur when we change data in a flawed schema. Imagine that a Department decides to change its name. We must change this data for every employee who works for this department. We might easily miss one. If we do miss one (or more), this is an update anomaly.

#### *Null Values*

A final rule for good database design is that we should avoid schema designs that have large numbers of empty attributes. For example, if we want to note that one in every hundred or so of our employees has some special qualification, we would not add a column to the employee table to store this information because for 99 employees, this would be NULL. We would instead add a new table storing only employeeIDs and qualifications for those employees who have those qualifications.

By the end of this document we will understand how to avoid these pitfalls.

In the following text we will cover:

- Database concepts and terminology
- Database design principles
- Normalization and the normal forms
- Database design exercises

## ***Database Concepts and Terminology***

To understand these principles we will look at in this text, we need to establish some basic concepts and terminology.

### ***First Things First: What is a database?***

Simply put a *database* is a structured body of related information. The software used to manage and manipulate that structured information is called a *DBMS* (Database Management System). A database is one component of a DBMS. You can think of a database simply as a list of information.

A fine example is the white pages of the phone book. The each listing in the whitepages contains several items of information – name, address and phone number – about each phone subscriber in a particular region (information). All subscriber information shares the same form (structure).

In database terms, the white pages comprise a *table* in which each subscriber is represented by a *record*. Each subscriber record contains three *fields*: name, address, and phone number. The records are sorted alphabetically by the name field, which is called the *key field*.

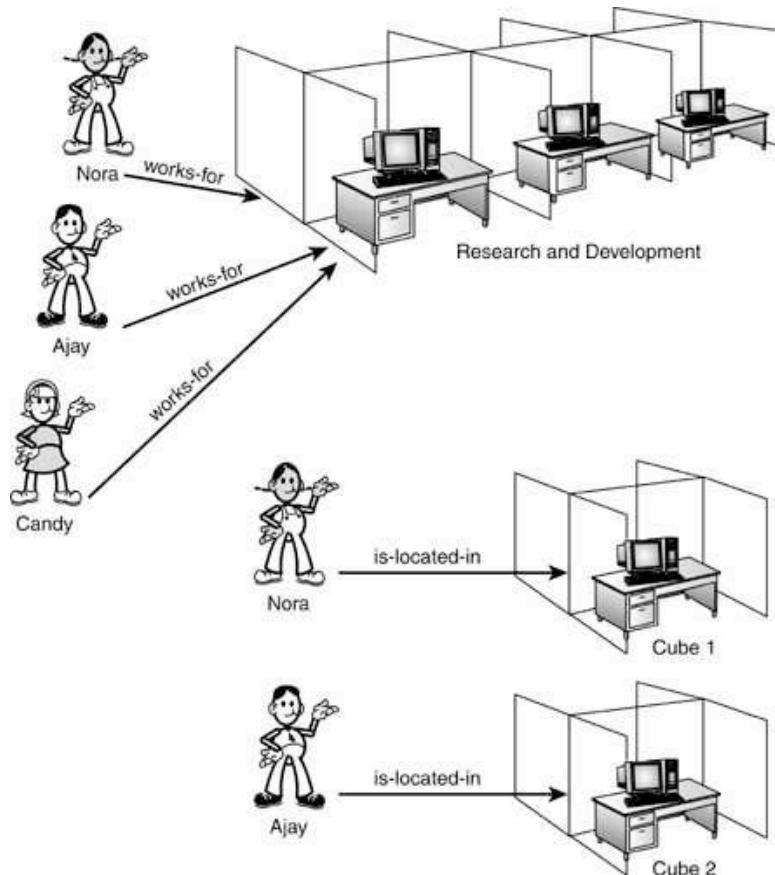
Other examples of databases are membership/customer lists, library catalogues, and web page content. The list is, in fact, infinite. You can model and design a database to store anything which can be represented as structured information.

### **Entities and Relationships**

The very basics of what we are trying to model are entities and relationships. *Entities* are the things in the real world that we will store information about in the database. For example, we might choose to store information about employees and the departments they work for. In this case, an employee would be one entity and a department would be another. *Relationships* are the links between these entities. For example, an employee works for a department. Works-for is the relationship between the employee and department entities.

Relationships come in different degrees. They can be one-to-one, one-to-many (or many-to-one depending on the direction you are looking at it from), or many-to-many. A one-to-one relationship connects exactly two entities. If employees in this organization had a cubicle each, this would be a one-to-one relationship. The works-for relationship is usually a many-to-one relationship in this example. That is, many employees work for a single department, but each employee works for only one department. These two relationships are shown in Figure 1.

Figure 1. The is-located-in relationship is one-to-one. The works-for relationship is many-to-one.



Note that the entities, the relationships, and the degree of the relationships depend on your environment and the business rules you are trying to model. For example, in some companies, employees may work for more than one department. In that case, the works-for relationship would be many-to-many. If anybody shares a cubicle or anybody has an office instead, the is-located-in relationship is not one-to-one. Note that we can't represent a many-to-many association directly in a relation scheme, because two tables can't be children of each other—there's no place to put the foreign keys. Instead, we put the foreign keys in a scheme that represents the association.

When you are coming up with a database design, you must take these rules into account for the system you are modeling. No two systems will be exactly the same.

## **Relations or Tables**

MySQL is a relational database management system (RDBMS)—that is, it supports databases that consist of a set of relations. A relation in this sense is not your auntie, but a table of data. Note that the terms table and relation mean the same thing. In this book, we will use the more common term table. If you have ever used a spreadsheet, each sheet is typically a table of data. A sample table is shown in Figure 2.

Figure 2. The employee table stores employee IDs, names, jobs, and the department each employee works for.

<b>employee</b>			
<b>employeeID</b>	<b>name</b>	<b>job</b>	<b>departmentID</b>
7513	Nora Edwards	Programmer	128
9842	Ben Smith	DBA	42
6651	Ajay Patel	Programmer	128
9006	Candy Burnett	Systems Administrator	128

As you can see, this particular table holds data about four employees at a particular company.

## **Columns or Attributes**

In database tables, each column or attribute describes some piece of data that each record in the table has. The terms column and attribute are used fairly interchangeably, but a column is really part of a table, whereas an attribute relates to the real-world entity that the table is modeling. In Figure 2 you can see that each employee has an employeeID, a name, a job, and a departmentID. These are the columns of the employee table, sometimes also called the attributes of the employee table.

## **Rows, Records, Tuples**

Look again at the employee table. Each row in the table represents a single employee record. You may hear these called rows, records, or tuples. Each row in the table consists of a value for each column in the table.

## **Keys**

Keys are a very important concept in a successful database design. Keys come in five basic flavors: Super Keys, Candidate Keys, Primary Keys, Foreign Keys, and Keys. Note: This is not a full explanation of keys and how to use them, there is a section on keys later in this document.

A Super Key is a column (or set of columns) that can be used to identify a row in a table. A Key is a minimal Super Key. For example, look at the employee table. We could use the employeeID and the name together to identify any row in the table. We could also use the set of all the columns (employeeID, name, job, departmentID). These are both Super Keys.

However, we don't need all those columns to identify a row. We need only (for example) the

employeeID. This is a minimal Super Key—that is, a minimized set of columns that can be used to identify a single row. So, employeeID is a key.

Look at the employee table again. We could identify an employee by name or by employeeID. These are both keys. We call these candidate keys because they are candidates from which we will choose the primary key. The primary key is the column or set of columns that we will use to identify a single row from within a table. In this case we will make employeeID the primary key. This will make a better key than name because it is common to have two people with the same name.

Foreign keys represent the links between tables. For example, if you look back at Figure 2, you can see that the departmentID column holds a department number. This is a foreign key: The full set of information about each department will be held in a separate table, with the departmentID as the primary key in that table.

### **Functional Dependencies**

The term *functional dependency* comes up less often than the ones previously mentioned, but we will need to understand it to understand the normalization process that we will discuss later.

If there is a functional dependency between column A and column B in a given table, which may be written  $A \rightarrow B$ , then the value of column A determines the value of column B. For example, in the employee table, the employeeID functionally determines the name (and all the other attributes in this particular example).

### **Schemas**

The term schema or database schema simply means the structure or design of the database—that is, the form of the database without any data in it. If you like, the schema is a blueprint for the data in the database.

We can describe the schema for a single table in the following way:

Table: employee  
employeeID    name    job    departmentID

## **Database Design Principles**

### **Planning**

#### **Think Ahead!!**

Perhaps the single most important thing to do when you start designing a database is to think ahead. Before you even switch on the computer, think about the type of information you have to work with and the types of questions you'll want your database to answer. To rephrase: What information needs to be stored or what things or entities do we need to store information about? And what questions will we need to ask of the database?

When thinking about these questions, we must bear in mind the rules of what we are trying to model — what the things are that we need to store data about and what specifically are the links between them.

A typical design cycle is identify data->set data types->normalize tables/assign keys->rinse/repeat

Let's revisit the white pages example presented earlier. It should be easy: all you need is the name, the address and the phone number of each person. Your initial table design thus consists of three fields: name, address and phone number. Right? That gives us the following structure:

Name  
Address  
Phone Number

That seems clear enough. But we need the names sorted alphabetically by last name. There's no easy way to do it, so it makes sense to break the Name into its component parts: First Name, Middle Initial, Last Name.

The same goes for the address. It, too, can be usefully broken down into Street Address, City, State and Postal Code.

The best way to see if the database design works is to test it with some sample data, so feed the following records into your hypothetical table:

NAME	ADDRESS	PHONE NUMBER
Jay T. Apples	100 Megalong Dr Etna	4992122
Beth York	2/53 Alice Lebanon	5050011
Mike. R. Sullivan	9 Jay Lebanon	4893892
Barry J. Anderson	71 Wally Rd Hanover	2298310

Now tell the database program to sort the information:

NAME	ADDRESS	PHONE NUMBER
Barry J. Anderson	71 Wally Rd Hanover	2298310
Beth York	2/53 Alice Lebanon	5050011
Jay T. Apples	100 Megalong Dr Etna	4992122
Mike. R. Sullivan	9 Jay Lebanon	4893892

Immediately, you can see this is not what you want. You want the table sorted alphabetically by last name, not first name.

How can you fix this? Well, you could do some complex manipulation using a database feature called 'string functions' – if your program supports such a feature. Or, you could come up with a better table design in the first place: last name, middle initial(s), First name, address and phone number. Feed your test data into this revised structure then tell the database program to sort your table using the last name followed by the initials.

Now our table structure is:

LastName  
FirstName  
Mid  
Address  
Phone

This time, you'll get the correct alphabetical listing:

LastName	FirstName	Mid	Address	Phone
Anderson	Barry	J.	71 Wally Rd Hanover	2298310
Apples	Jay	T.	100 Megalong Dr Etna	4992122
Sullivan	Mike.	R.	9 Jay Lebanon	4893892
York	Beth		2/53 Alice Lebanon	5050011

Don't stop there. The table can be even more effective if you break the structure down further. For instance, if you'd like to have an easy way to list only those people who live in Leichhardt, this design won't help you. But with a little more work, you can break your database structure down further into

first name, last name, middle initials, street address, city, and phone number.

Now our table structure is:

FirstName  
Mid  
LastName  
Street  
City  
Phone

LastName	FirstName	Mid	Street	City	Phone
Anderson	Barry	J.	71 Wally Rd	Hanover	2298310
Apples	Jay	T.	100 Megalong Dr	Etna	4992122
Sullivan	Mike	R.	9 Jay	Lebanon	4893892
York	Beth		2/53 Alice	Lebanon	5050011

With this structure, you'll be able to sort your database alphabetically by last name or by city, and you'll be able to pluck out all those people who live in a particular city.

Hopefully it is becoming obvious that whenever you create a database it is crucial to plan ahead. You need to look at the information you want to store and the ways you want to retrieve that information before you start working on the computer. The way you structure your data will affect every interaction with the database. It will determine how easy it is to enter information into the database; how well the database will trap inconsistencies and exclude duplicate records; and how flexibly you will be able to get information out of the database. A poorly structured database will hamstring you further down the track when you try to get your information back out in a usable form.

The above data could be stored in a single table, but there are benefits to further analyzing the data and our requirements and considering if our requirements would be better served using a Multi-table or Relational database. A relational database as we'll see soon enough, can provide exceptional power and flexibility in storing and retrieving information.

Let us have a look at a CD Database and how it is best served as a relational database.

### ***Woe are the Inadequacies of the Single-table Database! New Improved Relational Databases to the Rescue!***

When it comes to data entry, single-file databases in many cases present an easy to understand data structure. In the long run single-file databases often create more work for you. That is not to say that single table databases will not work for many use cases, just that for most of those cases a relational table eases many woes. For instance, if you create a flat-file database to catalogue your CDs, you have to put all the details, including the artist information, into one table. Say you want to include information such as the artist/band's recording label, band members, a discography and artist notes. How's that going to work? Your table structure might look something like this:

cd\_name  
cd\_date  
genre  
tracknumber  
trackname  
artist\_or\_band\_name  
band\_members  
recording\_label  
discography  
notes

For each Beatle's CD you own, you'll have to type in all those details! That means you'll have to type the names of all of the Beatle's releases repeatedly. Ugh.

### ***Multi-table flexibility***

On the other hand, if you use a multi-table relational database, you can store the CD details (name, date, tracks and so on) in a CD table and store the artist details once in an Artist table. Your CD table will look like this:

```
cd_name
cd_date
genre
tracks
artist_or_band_name
```

Your Artist table will look like this:

```
artist_or_band_name
band_members
recording_label
discography
notes
```

You then link the two tables using the artist/band name field (that's why it's called a relational database – you define relationships between the tables) as the Foreign Key and enter the artist information once only. Each time you add a subsequent Beatles CD to your collection, you type The Beatles in the artist field and the database looks up the other details for you. It not only minimizes effort on your part, it also ensures consistency of information and minimizes the chance of introducing errors into the data.

To further flexibility you could continue and create a songs table:

```
cd_name
song_title
duration
track_number
writer
vocals
```

and link it to the CD by using CD Name as the Foreign Key. Better yet would be to use a CD\_ID as the primary key of the CD table and use CD ID as the Foreign Key in the songs table. This is because you know, as an avid collector of covers, that CDs may share same name.

You can now get a list of all your CDs that contain a given song - great for us collectors of covers!

### ***What is it We Want to Store or "It's all in the Type, Data."***

The SQL standard defines a number of standard datatypes and most DB vendors support some additional ones which are specific to their own product. In the absence of truly compelling reasons to the contrary, avoid such extensions for the sake of portability.

### ***Strings and Numbers***

In general, numerical types pose few problems -- just select one that is large enough to support the necessary range of values.



The attempt to find the optimal width of a string column is usually not worth the effort. You can avoid a lot of confusion later on by making all text messages of type varchar(n) and limiting yourself to a few standard string lengths and introducing aliases for them, such as: 32 bytes ("Label"), 256 bytes ("Note"), and 4k ("Text").

Even if other business requirements restrict the maximum length of certain fields to specific values, the DB schema is arguably not the best place to enforce these rules. By the time the data reaches the DB, it is too late to do anything about it (except reject it). Individual restrictions, stemming from business rules and requirements, should be enforced by the business logic layer, which handles user interaction and input validation. On the other hand, maintenance of the DB schema is considerably simplified if it is restricted to a handful of different string attributes.

Limit the use of fixed-width strings to codes of all sorts (as opposed to variable-length strings for real text). Keep in mind however, that many seemingly fixed-length codes do actually become wider over time. The prudent DB designer tries to avoid anything similar to the Y2K problem for new development work.

### ***Time and Money***

A type to record timestamps (date/time combinations) is always necessary and is, fortunately, covered by the SQL standard. No fully satisfying way exists to record a monetary value, however.

Saving monetary values and treating them in program code as a floating-point values always leads to round-off errors. Recording the value as an exact integer of the smallest currency subdivision (such as "cent" for US dollars, as well as for Euros and other appropriate currencies) may not be sufficient either. Many values carry more digits behind the decimal point than the two for which actual coins exist (just visit your local gas station). A choice of decimal with 5 to 9 digits should work, though.

It goes without saying that no monetary value should ever be recorded without also recording the currency -- even if you think your application will never handle anything but US dollars. Consider setting up a currency table and relating it to the monetary values using foreign keys rather than embedding currency information directly. This helps with internationalization (different currency names and symbols), as well as with formatting issues.

### ***Complex Datatypes***

Finally, there are some common but complex datatypes -- such as phone numbers, postal addresses, contact information, and credit cards -- that occur in almost every database schema. Typically, such records need to be accessed from many tables in the database. In a typical eCommerce system, for instance, it might be necessary to store contact information for users, suppliers, warehouses, and admins.

Rather than including those attributes in the respective user, supplier, or other record. (and thereby repeating those columns throughout the database), it makes sense to set up a single table for the contact information that is referenced through foreign keys by all other tables. This has two immediate benefits:

- It is easier to later change the cardinality of the relationships.
- It localizes any future changes to the complex datatype.

Anticipating the attributes that will likely be required for each of those complex types is something of an art. My suggestion is to try to strive for completeness from the outset rather than being forced to change the schema each time an additional field becomes necessary.

A sampling of possible attributes for postal addresses includes:

Department  
Company  
MailStop  
AddressLine1  
AddressLine2  
AddressLine3  
City  
State  
PostalCode  
Country

Full contact information might include the following attributes:

Title  
FirstName  
MiddleName  
LastName  
Suffix  
HomeAddress  
WorkAddress  
HomePhone  
WorkPhone  
CellPhone  
Fax  
Pager  
Email

Finally, phone numbers should never be considered flat numbers. In fact, they break down into the following fields:

CountryCode  
AreaCode  
ExchangeCode  
LineNumber  
Extension

In a phone number such as 987-1234, the prefix is the 987 and the suffix is the 1234. The extension is the only part of the phone number that is optional. It is probably sufficient to use char(4) for all columns, but one might consider char(6) to be on the safe side. Note that area codes in the US are limited to three digits, but this is not true for other countries.

### ***A Note on Sensitive Data***

Any sensitive data should be kept in encrypted form. Even if the database system itself is compromised, the data is still protected from misuse. The most famous example of this kind of data management is the Unix password system which only stores hashes of the users' passwords rather than the passwords themselves. Some data, such as credit card numbers, needs to be encrypted in a recoverable fashion; however, a one-way encryption (as for the Unix password file) will not do. This leads to the problem of encryption key management -- clearly, it should not be stored in the DB, along with the secrets, but supplied at boot-time, for instance.

Each field we've included has its own data type. The data type defines the type of information which may be stored in a field. The majority of our fields are text data type. Text fields can hold alphanumeric information, including letters, numbers, spaces and punctuation marks.

Other common data types include numeric (also known as number), currency (a specialised form of numeric field), date/time, Boolean (also called Yes/No), hyperlink, memo (for storing large quantities of text) and picture/object. Not all database programs support all these data types and our simple data structure uses only four types: text, numeric, Boolean and date.

Boolean fields are logical fields which may contain either a 0 or 1, or another binary pair such as True/False or Yes/No. They're useful when you want Yes/No answers to questions. We've used them in our database in the ListHomePhone, ListWorkPhone, active and FeesPaid fields to answer the questions "Should I list the member's home/work number when printing reports?", "Is this an active member?" and "Are the member's fees up to date?"

Notice how we've used the text data type for both the phone numbers and postal codes. Why not use the numeric data type?

With phone numbers, the answer's obvious: These numbers frequently contain non-numeric characters, such as parentheses and hyphens: (02) 4782-0000 for example. By using text data type we allow for such characters, as well as allowing for additional details such as ext 34 (although you could, if you wish, create an additional field called WorkExtension to handle extension numbers).

As for the postcode, although this field will contain only numbers, we don't treat postcodes as numbers, that is, use them in numerical calculations. Because of this, and because of the way database sort and format numbers differently from text, always store this type of information in a text field.

### ***Field Sizes***

The most important thing about the size of your fields is that you make them big enough to accommodate the largest possible piece of information they will need to store.

With names and addresses, be generous. You may not be able to imagine a family name longer than 15 characters, but Ms Clarissa Worthington-Wettersley is going to be really annoyed when all her club correspondence is addressed to Ms Worthington-Wet.

As for fields where you're not quite sure how much info you need to store, such as the Skills field we've included, one approach is to allow the maximum permissible size for a text field, which is usually around 254 or 255 characters.

### ***Allowing for International Variations***

If your database may be used in more than one locale or for storing more than one language, keep in mind regional differences of data representation (phone numbers, currency, etc.), and make sure to use a DBMS capable of handling multi-byte characters.

### ***Field Names***

You'll notice that many field names are compound words, such as FirstName and MembershipType. Why is that so? Why not make them more readable by including spaces?

Well, although all three database programs allow you to create field names which contain spaces, if you end up using your database with scripting tools such as JavaScript or with advanced data access technologies such as ADO (ActiveX Data Objects), you'll find spaces in fieldnames are unacceptable. So, even if you don't think you'll end up using such esoterica, it pays to allow for the possibility by eliminating spaces.

When it comes to creating data entry forms, you can always change the field name which is displayed above a data entry box by adjusting its caption, something we'll look at later.

## **Formatting and Validation**

Some formatting and validation takes place as the result of the datatype you have selected. In the best of all circumstances the data that is passed to your database will be clean and present no problems. In the case of data collected from webforms you absolutely should pre-validate all data collected before presenting it to the database. This may be done using javascript, or the serverside scripting language that your site is built on.

## **Normalization or Less is More**

One of the most important factors in database design is definition. If your tables are not set up properly, it can cause you a lot of headaches down the road when you have to perform miraculous SQL calls in your code in order to extract the data you want. By understanding data relationships and the normalization of data, you will be better prepared to begin developing your application. A well-designed database minimizes redundancy without losing any data. That is, we aim to use the least amount of storage space for our database while still maintaining all links between data. Additionally, a normalized DB schema avoids certain anomalies when inserting, updating, or deleting data and, therefore, helps to keep consistent data in the database.

However, the absence of anomalies is only the tangible result of a deeper benefit of normalization -- namely the correct identification and modeling of entities. The insert, update, and delete anomalies I've just referred to are the consequences of the redundancy introduced by improper or inadequate separation between distinct entities. The normalization procedure is, therefore, not just a technical chore to be done out of principle, but it can actively help to improve the understanding of the business domain.

Whether you work with MySQL or Oracle, you should know the methods of normalizing the table schema in your relational database system. They can help make your code which accesses the database easier to understand, easier to expand upon, and in some cases, actually speed up your application. Interestingly after all the rigor the outcome of a normalization procedure often evokes the reaction that it all is nothing more than common sense.

Basically, the Rules of Normalization are enforced by eliminating redundancy and inconsistent dependency in your table designs. In the following example we will look at what that means by examining the five progressive steps to normalization you should be aware of in order to create a functional and efficient database. I'll detail the types of relationships your data structure can utilize.

### **Functional Dependence**

Before we jump into the Normalization Process, I should take a step back and clear a few things up. First, Normalization is not specific to any one type of database. These are rules that should be followed when using any database system, whether it is Oracle, MySQL, PostgreSQL, SQL Server, etc.

Let us first discuss Functional Dependence, which is crucial in understanding the Normalization Process. This is merely a big term for a relatively simple idea. To illustrate it, let's take a look at a small sample table.

Name	Pay_Class	Rate
Ward	1	.05
Maxim	1	.05
Cane	1	.05
Beechum	2	.07
Collins	1	.05
Cannery	3	.09

This relatively simple table is a good example of functional dependence, it can also be used to illustrate a point.

Definition: A column is functionally dependent on another column if a value 'A' determines a single value for 'B' at any one time.

Sound confusing? Let me explain. The field 'Rate' is functionally dependent on the field 'Pay Class'. In other words, Pay Class determines Rate.

To determine functional dependency, you can think of it like this: Given a value for Field A, can you determine the single value for B? If B relies on A, then A is said to functionally determine B.

Taking the same table as above, lets add to it.

Name	Sales_Rep_Number	Pay_Class	Rate
Ward	001	1	.05
Maxim	002	1	.05
Cane	003	1	.05
Beechum	004	2	.07
Collins	005	1	.05
Cannery	006	3	.09

Now, lets look at this table and find some more Functional Dependencies. We already know that Pay Class determines Rate. We can also say that Sales Rep Number determines Last Name. Only one Sales Rep Number for each Last Name. This fits the definition of a Functional Dependency.

But does Last Name functionally determine anything? At first glance, some people might say yes, however, this is not true. Currently, you can say that Ward will only give you one Sales Rep Number, however, what if we hired another person with the name Ward? Then you would have two values for your Sales Rep Number, and then Last Name would no longer functionally determine anything.

#### On Keys and Functional Dependencies

Now that we know what functional dependence is, we can clarify keys. Now, if you are working in databases, you probably already know what Primary Keys are. But, can you define them?

*Definition: Column A is the primary key for table T if:*

*Property 1. All columns in T are functionally dependent on A*

*Property 2. No sub collections of the columns in table T also have Property 1.*

This makes perfect sense. If all your fields in a database are dependent on one and only one field, then that field is the key. Now, occasionally Property 2 is broken, and two fields are candidates for the Primary Key. These keys are then called candidate keys. From these candidate keys, one key is chosen and the others are called alternate keys.

For example, in the same table as before:

Name	Sales_Rep_Number	Pay_Class	Rate
Ward	001	1	.05
Maxim	002	1	.05
Cane	003	1	.05
Beechum	004	2	.07
Collins	005	1	.05
Cannery	006	3	.09

Our primary key is the Sales Rep Number, as it fits the definition of a Primary Key. Everything in the table is dependent on the Sales Rep Number, and nothing else can claim the same thing. Now, let us take this one step further, and assume that we also have the Social Security number of the employee in the table as well.

Name	Sales_Rep_Number	Pay_Class	Rate	Soc.Sec. no.
Ward	001	1	.05	133-45-6789
Maxim	002	1	.05	122-46-6889
Cane	003	1	.05	123-45-6999
Beechum	004	2	.07	113-75-6889
Collins	005	1	.05	121-44-6789
Cannery	006	3	.09	111-45-9339

Now, we have two Candidate Keys, Sales Rep Number and Social Security Number. So, what we have to decide is which field to use, as both will be unique. In the end, it would be best to have the Sales Rep Number as the Primary Key for various reasons that I hope will become obvious with the following example.

Let's say we want to create a table of user information, and we want to store each users' Name, Company, Company Address, and some personal bookmarks, or urls. You might start by defining a table structure like this:

### Zero Form

Table: users

name	company	company_address	url1	url2
Joe	ABC	1 Work Lane	abc.com	xyz.com
Jill	XYZ	1 Job Street	abc.com	xyz.com

We would say this table is in Zero Form because none of our rules of normalization have been applied yet. Notice the url1 and url2 fields -- what do we do when our application needs to ask for a third url? Do you want to keep adding columns to your table and hard-coding that form input field into your code? Obviously not, you would want to create a functional system that could grow with new development requirements. Let's look at the rules for the First Normal Form, and then apply them to this table.

### First Normal Form

The first normal form, sometimes called 1NF, states that each attribute or column value must be atomic. That is, each attribute must contain a single value, not a set of values or another database row. First Normal Form Rules are:

1. Eliminate repeating groups in individual tables.
2. Create a separate table for each set of related data.
3. Identify each set of related data with a primary key.

Notice how we're breaking that first rule by repeating the url1 and url2 fields? And what about Rule Three, primary keys? Rule Three basically means we want to put some form of unique, auto-incrementing integer value into every one of our records. Otherwise, what would happen if we had two users named Joe and we wanted to tell them apart? When we apply the rules of the First Normal Form we come up with the following table:

### First Normal Form

Table: users

userId	name	company	company_address	url
1	Joe	ABC	1 Work Lane	abc.com
1	Joe	ABC	1 Work Lane	xyz.com
2	Jill	XYZ	1 Job Street	abc.com
2	Jill	XYZ	1 Job Street	xyz.com

Now our table is said to be in the First Normal Form. We've solved the problem of url field limitation, but look at the headache we've now caused ourselves. Every time we input a new record into the users table, we've got to duplicate all that company and user name data. Not only will our database grow much larger than we'd ever want it to, but we could easily begin corrupting our data by

misspelling some of that redundant information. Let's apply the rules of Second Normal Form.

### **Second Normal Form**

A schema is said to be in second normal form (also called 2NF) if all attributes that are not part of the primary key are fully functionally dependent on the primary key, and the schema is already in first normal form. What does this mean? It means that each non-key attribute must be functionally dependent on all parts of the key. That is, if the primary key is made up of multiple columns, every other attribute in the table must be dependent on the combination of these columns.

1. Create separate tables for sets of values that apply to multiple records.
2. Relate these tables with a foreign key.

We break the url values into a separate table so we can add more in the future without having to duplicate data. We'll also want to use our primary key value to relate these fields:

#### **Second Normal Form**

Table: users

userId	name	company	company_address
1	Joe	ABC	1 Work Lane
2	Jill	XYZ	1 Job Street

Table: urls

urlId	relUserId	url
1	1	abc.com
2	1	xyz.com
3	2	abc.com
4	2	xyz.com

Ok, we've created separate tables and the primary key in the users table, `userId`, is now related to the foreign key in the urls table, `relUserId`. We're in much better shape. But what happens when we want to add another employee of company ABC? Or 200 employees? Now we've got company names and addresses duplicating themselves all over the place, a situation just rife for introducing errors into our data. So we'll want to look at applying the Third

### **Third Normal Form**

Eliminate fields that do not depend on the key.

You may sometimes hear the saying "Normalization is about the key, the whole key, and nothing but the key." Second normal form tells us that attributes must depend on the whole key. Third normal form tells us that attributes must depend on nothing but the key.

Formally, for a schema to be in third normal form (3NF), we must remove all transitive dependencies, and the schema must already be in second normal form. Okay, so what's a transitive dependency?

Look at the following schema:

```
users
userId name  company    company_address
```

This schema contains the following functional dependencies:

```
userId -> name, company, company_address
```

```
company -> company_address
```

The primary key is `userId`, and all the attributes are fully functionally dependent on it—this is easy to see because there is only one attribute in the primary key!

However, we can see that we have

userID → name

userID → company

and

company → company\_address

Note also that the attribute company is not a key.

This relationship means that the functional dependency userID → company is a transitive dependency. Effectively, it has a middle step (the company → company\_address dependency).

To get to third normal form, we need to remove this transitive dependency.

Our Company Name and Address have nothing to do with the User Id, so they should have their own Company Id:

### Third Normal Form

Table: users

userID	name	companyID
1	Joe	1
2	Jill	2

Table: companies

companyID	company	company_address
1	ABC	1 Work Lane
2	XYZ	1 Job Street

Table: urls

urlId	relUserId	url
1	1	abc.com
2	1	xyz.com
3	2	abc.com
4	2	xyz.com

Now we've got the primary key comp'd in the companies table related to the foreign key in the users table called relComp'd, and we can add 200 users while still only inserting the name "ABC" once. Our users and urls tables can grow as large as they want without unnecessary duplication or corruption of data. Most developers will say the Third Normal Form is far enough, and our data schema could easily handle the load of an entire enterprise, and in most cases they would be correct.

But look at our url fields - do you notice the duplication of data? This is perfectly acceptable if we are not pre-defining these fields. If the HTML input page which our users are filling out to input this data allows a free-form text input there's nothing we can do about this, and it's just a coincidence that Joe and Jill both input the same bookmarks. But what if it's a drop-down menu which we know only allows those two urls, or maybe 20 or even more. We can take our database schema to the next level, the Fourth Form, one which many developers overlook because it depends on a very specific type of relationship, the many-to-many relationship, which we have not yet encountered in our application.

### Data Relationships

Before we define the Fourth Normal Form, let's look at the three basic data relationships: one-to-one, one-to-many, and many-to-many. Look at the users table in the First Normal Form example above.



For a moment let's imagine we put the url fields in a separate table, and every time we input one record into the users table we would input one row into the urls table. We would then have a one-to-one relationship: each row in the users table would have exactly one corresponding row in the urls table. For the purposes of our application this would neither be useful nor normalized.

Now look at the tables in the Second Normal Form example. Our tables allow one user to have many urls associated with his user record. This is a one-to-many relationship, the most common type, and until we reached the dilemma presented in the Third Normal Form, the only kind we needed.

The many-to-many relationship, however, is slightly more complex. Notice in our Third Normal Form example we have one user related to many urls. As mentioned, we want to change that structure to allow many users to be related to many urls, and thus we want a many-to-many relationship. Let's take a look at what that would do to our table structure before we discuss it:

#### Fourth Normal Form

Table: users

userId	name	companyID
1	Joe	1
2	Jill	2

Table: companies

companyID	company	company_address
1	ABC	1 Work Lane
2	XYZ	1 Job Street

Table: urls

urlId	url
1	abc.com
2	xyz.com
3	abc.com
4	xyz.com

Table: urls\_relations

urlId	relUrID	relUserId
1	1	1
2	1	2
3	2	1
4	2	2

In order to decrease the duplication of data (and in the process bring ourselves to the Fourth Form of Normalization), we've created a table full of nothing but primary and foreign keys in url\_relations. We've been able to remove the duplicate entries in the urls table by creating the url\_relations table. We can now accurately express the relationship that both Joe and Jill are related to each one of , and both of, the urls. So let's see exactly what the Fourth Form Of Normalization entails.

#### Fourth Normal Form

In a many-to-many relationship, independent entities can not be stored in the same table.

Since it only applies to the many-to-many relationship, most developers can rightfully ignore this rule. But it does come in handy in certain situations, such as this one. We've successfully streamlined our urls table to remove duplicate entries and moved the relationships into their own table.

Just to give you a practical example, now we can select all of Joe's urls by performing the following SQL call:

```
SELECT name, url
```

```
FROM users, urls, url_relations
WHERE url_relations.relatedUserId = 1
AND users.userId = 1
AND urls.urlId = url_relations.relatedUrlId
```

And if we wanted to loop through everybody's User and Url information, we'd do something like this:

```
SELECT name, url
FROM users, urls, url_relations
WHERE users.userId = url_relations.relatedUserId
AND urls.urlId = url_relations.relatedUrlId
```

#### *Fifth Normal Form*

There is one more form of normalization which is sometimes applied, but it is indeed very esoteric and is in most cases probably not required to get the most functionality out of your data structure or application. It's tenet suggests:

*The original table must be reconstructed from the tables into which it has been broken down. The benefit of applying this rule ensures you have not created any extraneous columns in your tables, and that all of the table structures you have created are only as large as they need to be. It's good practice to apply this rule, but unless you're dealing with a very large data schema you probably won't need it.*

#### **Normalization summary**

Notice how creating an efficient table structure consists of breaking down your fields into simpler and simpler components? You end up with a table with many more fields than you might originally have thought necessary, but each of those fields houses much more basic information.

In some ways, creating a database that's effective and simple to use is almost an anti-intuitive process. What you need to remember is that while the structure might look more complex, the contents of each field have been reduced to the simplest useful components.

## **Keys**

After you've identified all the subjects that the database will track and defined the table structures that will represent those subjects and you've put the structures through a screening process to control their makeup and quality. In this stage of the database-design process, you'll begin the task of assigning keys to each table. You'll soon learn that there are different types of keys, and each plays a particular role within the database structure. All but one key is assigned during this stage; you'll assign the remaining key later (in Chapter 10) as you establish relationships between tables.

### **Why Keys Are Important**

Keys are crucial to a table structure for the following reasons:

- They ensure that each record in a table is precisely identified. As you already know, a table represents a singular collection of similar objects or events. (For example, a CLASSES table represents a collection of classes, not just a single class.) The complete set of records within the table constitutes the collection, and each record represents a unique instance of the table's subject within that collection. You must have some means of accurately identifying each instance, and a key is the device that allows you to do so.
- They help establish and enforce various types of integrity. Keys are a major component of table-level integrity and relationship-level integrity. For instance, they enable you to ensure that a table has unique records and that the fields you use to establish a relationship between a pair of tables always contain matching values.
- They serve to establish table relationships. As you'll learn in Chapter 10, you'll use keys to establish a relationship between a pair of tables.

Always make certain that you define the appropriate keys for each table. Doing so will help you guarantee that the table structures are sound, that redundant data within each table is minimal, and that the relationships between tables are solid.

### Establishing Keys for Each Table

There are four main types of keys: candidate, primary, foreign, and non-keys. A key's type determines its function within the table.

### Candidate Keys

The first type of key you establish for a table is the candidate key, which is a field or set of fields that uniquely identifies a single instance of the table's subject. Each table must have at least one candidate key. You'll eventually examine the table's pool of available candidate keys and designate one of them as the official primary key for the table.

Before you can designate a field as a candidate key, you must make certain it complies with all of the Elements of a Candidate Key. These elements constitute a set of guidelines you can use to determine whether the field is fit to serve as a candidate key. You cannot designate a field as a candidate key if it fails to conform to any of these elements.

### Elements of a Candidate Key

- It cannot be a multipart field. You've seen the problems with multipart fields, so you know that using one as an identifier is a bad idea. However: It may comprise of a minimum number of fields necessary to define uniqueness. You can use a combination of fields (treated as a single unit) to serve as a candidate key, so long as each field contributes to defining a unique value. Try to use as few fields as possible, however, because overly complex candidate keys can ultimately prove to be difficult to work with and difficult to understand.
- It must contain unique values. This element helps you guard against duplicating a given record within the table and ensures that you can accurately reference any of the table's records from other tables in the database. Duplicate records are just as bad as duplicate fields, and you must avoid them at all costs.
- It cannot contain null values. As you already know, a null value represents the absence of a value. There's absolutely no way a candidate key field can identify a given record if its value is null. Its value is not optional in whole or in part. You can infer, then, that an optional value automatically violates the previous element and is, therefore, unacceptable. (This caveat is especially applicable when you want to use two or more fields as a candidate key.)
- Its value cannot cause a breach of the organization's security or privacy rules. Values such as passwords and Social Security Numbers are not suitable for use as a candidate key.
- Its value must exclusively identify the value of each field within a given record. This element ensures that the table's candidate keys provide the only means of identifying each field value within the record. (You'll learn more about this particular element in the section on primary keys.)
- Its value can be modified only in rare or extreme cases. You should never change the value of a candidate key unless you have an absolute and compelling reason to do so. A field is likely to have difficulty conforming to the previous elements if you can change its value arbitrarily.

Establishing a candidate key for a table is quite simple: Look for a field or set of fields that conforms to all of the Elements of a Candidate Key. You'll probably be able to define more than one candidate key for a given table. Loading a table with sample data will give you the means to identify potential candidate keys accurately. (You used this same technique in the previous chapter.)

See if you can identify any candidate keys for the table in Figure 3.

FIGURE 3 Are there any candidate keys in this table?

## Employees

Employee ID	Social Security Number	EmpFirst Name	EmpLast Name	EmpStreet Address	EmpCity	EmpState	EmpZipcode	EmpHome Phone
1000	856-91-9938	Kendra	Bonnicksen	1204 Bryant Road	Seattle	WA	98157	363-9948
1001	896-11-2231	Katherine	Erllich	101 C Street, Apt. 32	Bellevue	WA	98046	322-6992
1002	901-48-0039	Timothy	Ennis	7402 Kingman Drive	Redmond	WA	98115	527-4992
1003	816-96-1299	Shannon	McLain	4141 Lake City Way	Seattle	WA	98136	336-5992
1004	978-02-1129	Susan	McLain	2100 Mineola Avenue	Seattle	WA	98115	572-9948
1005	955-92-5583	Estela	Pundt	101 C Street, Apt. 32	Bellevue	WA	98046	322-6992
1006	801-22-1734	Timothy	Sherman	66 NE 120th	Bothell	WA	98216	522-3232

You probably identified EMPLOYEE ID, SOCIAL SECURITY NUMBER, EMPLAST NAME, EMPFIRST NAME and EMPLAST NAME, EMPZIPCODE, and EMPHOME PHONE as potential candidate keys. But you'll need to examine these fields more closely to determine which ones are truly eligible to become candidate keys. Remember that you must automatically disregard any field(s) failing to conform to even one of the Elements of a Candidate Key.

Upon close examination, you can draw the following conclusions:

- EMPLOYEE ID is eligible. This field conforms to every element of a candidate key.
- SOCIAL SECURITY NUMBER is ineligible because it could contain null values and will most likely compromise the organization's privacy rules. Contrary to what the sample data shows, this field could contain a null value. For example, there are many people working in the United States who do not have Social Security numbers because they are citizens of other countries.

NOTE: Despite its widespread use in many types of databases, I would strongly recommend that you refrain from using SOCIAL SECURITY NUMBER as a candidate key (or a primary key, for that matter) in any of your database structures. In many instances, it doesn't conform to the Elements of a Candidate Key. You can learn some very interesting facts about Social Security numbers (which will shed some light on why they make poor candidate/primary keys) by visiting the Social Security Administration's Web site at <http://www.ssa.gov>.

- EMPLAST NAME is ineligible because it can contain duplicate values. As you've learned, the values of a candidate key must be unique. In this case there can be more than one occurrence of a particular last name.
- EMPFIRST NAME and EMPLAST NAME are eligible. The combined values of both fields will supply a unique identifier for a given record. Although multiple occurrences of a particular first name or last name will occur, the combination of a given first name and last name will always be unique. (Some of you are probably saying, "This is not necessarily always true." You're absolutely right. Don't worry; we'll address this issue shortly.)
- EMPZIPCODE is ineligible because it can contain duplicate values. Many people live in the same zip code area, so the values in EMPZIPCODE cannot possibly be unique.
- EMPHOME PHONE is ineligible because it can contain duplicate values and is subject to

change. This field will contain duplicate values for either of these reasons:

1. One or more family members work for the organization.
2. One or more people share a residence that contains a single phone line.

You can confidently state that the EMPLOYEES table has two candidate keys: EMPLOYEE ID and the combination of EMPFIRST NAME and EMPLAST NAME.

Mark candidate keys in your table structures by writing the letters "CK" next to the name of each field you designate as a candidate key. A candidate key composed of two or more fields is known as a composite candidate key, and you'll write "CCK" next to the names of the fields that make up the key. When you have two or more composite candidate keys, use a number within the mark to distinguish one from another. If you had two composite candidate keys, for example, you would mark one as "CCK1" and the other as "CCK2."

Apply this technique to the candidate keys for the EMPLOYEES table in Figure 3. Figure 4 shows how your structure should look when you've completed

FIGURE 4 Marking candidate keys in the EMPLOYEES table structure.

<b>Table Structures</b>	
<b>Employees</b>	
Employee ID	<i>CK</i>
Social Security Number	
EmpFirst Name	<i>CCK</i>
EmpLast Name	<i>CCK</i>
EmpStreet Address	
EmpCity	
EmpState	
EmpZipcode	
EmpHome Phone	

Now, try to identify as many candidate keys as you can for the PARTS table in Figure 5.

FIGURE 5 Can you identify any candidate keys in the PARTS table?

### **Parts**

<b>Part Name</b>	<b>Model Number</b>	<b>Manufacturer Name</b>	<b>Retail Price</b>
Shimka XT Cranks	XT-113	Shimka Incorporated	199.95
Faust Brake Levers	BL / 45	Faust USA	53.79
MiniMite Pump		MiniMite	35.00
Hobo Fanny Pack		Hobo Bike Company	59.00
Diablo Bike Pedals	Mtn-A26	Diablo Sports	129.50
Shimka Truing Stand	SP-100		37.95
Faust Brake Levers	BL / 60	Faust USA	79.95

At first glance, you may believe that PART NAME, MODEL NUMBER, the combination of PART NAME and MODEL NUMBER, and the combination of MANUFACTURER and PART NAME are potential candidate keys. After investigating this theory, however, you come up with the following results:

- PART NAME is ineligible because it can contain duplicate values. A given part name will be duplicated when the part is manufactured in several models. For example, this is the case with Faust Brake Levers.
- MODEL NUMBER is ineligible because it can contain null values. A candidate key value must exist for each record in the table. As you can see, some parts do not have a model number.
- PART NAME and MODEL NUMBER are ineligible because either field can contain null values. The simple fact that MODEL NUMBER can contain null values instantly disqualifies this combination of fields.
- MANUFACTURER and PART NAME are ineligible because the values for these fields seem to be optional. Recall that a candidate key value cannot be optional in whole or in part. In this instance, you can infer that entering the manufacturer name is optional when it appears as a component of the part name; therefore, you cannot designate this combination of fields as a candidate key.

It's evident that you don't have a single field or set of fields that qualifies as a candidate key for the PARTS table. This is a problem because each table must have at least one candidate key.

Fortunately, there is a solution.

#### Artificial Candidate Keys

When you determine that a table does not contain a candidate key, you can create and use an artificial (or surrogate) candidate key. (It's artificial in the sense that it didn't occur "naturally" in the table; you have to manufacture it.) You establish an artificial candidate key by creating a new field that conforms to all of the Elements of a Candidate Key and then adding it to the table; this field becomes the official candidate key.

You can now solve the problem in the PARTS table. Create an artificial candidate key called PART NUMBER and assign it to the table. (The new field will automatically conform to the Elements of a Candidate Key because you're creating it from scratch.) Figure 6 shows the revised structure of the PARTS table.

FIGURE 6 The PARTS table with the artificial candidate key PART NUMBER.

#### Parts

Part Number	Part Name	Model Number	Manufacturer Name	Retail Price
41000	Shimka XT Cranks	XT-113	Shimka Incorporated	199.95
41001	Faust Brake Levers	BL / 45	Faust USA	53.79
41002	MiniMite Pump		MiniMite	35.00
41003	Hobo Fanny Pack		Hobo Bike Company	59.00
41004	Diablo Bike Pedals	Mtn-A26	Diablo Sports	129.50
41005	Shimka Truing Stand	SP-100		37.95
41006	Faust Brake Levers	BL / 60	Faust USA	79.95

When you've established an artificial candidate key for a table, mark the field name with a "CK" in the table structure, just as you did for the EMPLOYEES table in the previous example.

You may also choose to create an artificial candidate key when it would be a stronger (and thus, more appropriate) candidate key than any of the existing candidate keys. Assume you're working on an EMPLOYEES table and you determine that the only available candidate key is the combination of the EMPFIRST NAME and EMPLAST NAME fields. Although this may be a valid candidate key, using a single-field candidate key might prove more efficient and may identify the subject of the table more easily. Let's say that everyone in the organization is accustomed to using a unique identification number rather than a name as a means of identifying an employee. In this instance, you can choose to create a new field named EMPLOYEE ID and use it as an artificial candidate key. This is an absolutely acceptable practice—do this without hesitation or reservation if you believe it's appropriate.

#### NOTE

I commonly create an ID field (such as EMPLOYEE ID, VENDOR ID, DEPARTMENT ID, CATEGORY ID, and so on) and use it as an artificial candidate key. It always conforms to the Elements of a Candidate Key, makes a great primary key (eventually), and, as was shown in the section on Normalization (rUrID is an example), makes the process of establishing table relationships much easier.

Review the candidate keys you've selected and make absolutely certain that they thoroughly comply with the Elements of a Candidate Key. Don't be surprised if you discover that one of them is not a candidate key after all—incorrectly identifying a field as a candidate key happens occasionally. When this does occur, just remove the "CK" designator from the field name in the table structure. Deleting a candidate key won't pose a problem as long as the table has more than one candidate key. If you discover, however, that the only candidate key you identified for the table is not a candidate key, you must establish an artificial candidate key for the table. After you've defined the new candidate key, remember to mark its name with a "CK" in the table structure.

#### Primary Keys

By now, you've established all the candidate keys that seem appropriate for every table. Your next task is to establish a primary key for each table, which is the most important key of all.

- A primary key field exclusively identifies the table throughout the database structure and helps establish relationships with other tables. (You'll learn more about this in Chapter 10.)
- A primary key value uniquely identifies a given record within a table and exclusively represents that record throughout the entire database. It also helps to guard against duplicate records.

A primary key must conform to the exact same elements as a candidate key. This requirement is easy to fulfill because you select a primary key from a table's pool of available candidate keys. The process of selecting a primary key is somewhat similar to that of a presidential election. Every four years, several people run for the office of president of the United States. These individuals are known as "candidates" and they have all of the qualifications required to become president. A national election is held, and a single individual from the pool of available presidential candidates is elected to serve as the country's official president. Similarly, you identify each qualified candidate key in the table, run your own election, and select one of them to become the official primary key of the table. You've already identified the candidates, so now it's election time!

Assuming that there is no other marginal preference, here are a couple of guidelines you can use to select an appropriate primary key:

1. If you have a simple (single-field) candidate key and a composite candidate key, choose the simple candidate key. It's always best to use a candidate key that contains the least

number of fields.

2. Choose a candidate key that incorporates part of the table name within its own name. For example, a candidate key with a name such as SALES INVOICE NUMBER is a good choice for the SALES INVOICES table.

Examine the candidate keys and choose one to serve as the primary key for the table. The choice is largely arbitrary—you can choose the one that you believe most accurately identifies the table's subject or the one that is the most meaningful to everyone in the organization. For example, consider the EMPLOYEES table again in Figure 7.

FIGURE 7 Which candidate key should become the primary key of the EMPLOYEES table?

Table Structures	
<b>Employees</b>	
Employee ID	CK
Social Security Number	
EmpFirst Name	CCK
EmpLast Name	CCK
EmpStreet Address	
EmpCity	
EmpState	
EmpZipcode	
EmpHome Phone	

Either of the candidate keys you identified within the table could serve as the primary key. You might decide to choose EMPLOYEE ID if everyone in the organization is accustomed to using this number as a means of identifying employees in items such as tax forms and employee benefits programs. The candidate key you ultimately choose becomes the primary key of the table and is governed by the Elements of a Primary Key. These elements are exactly the same as those for the candidate key, and you should enforce them to the letter. For the sake of clarity, here are the Elements of a Primary Key:

#### **Elements of a Primary Key**

It cannot be a multipart field.

It must contain unique values.

It cannot contain null values.

Its value cannot cause a breach of the organization's security or privacy rules.

Its value is not optional in whole or in part.

It comprises a minimum number of fields necessary to define uniqueness.

Its values must uniquely and exclusively identify each record in the table.

Its value must exclusively identify the value of each field within a given record.

Its value can be modified only in rare or extreme cases.

Before you finalize your selection of a primary key, it is imperative that you make absolutely certain that the primary key fully complies with this particular element:

Its value must exclusively identify the value of each field within a given record.

Each field value in a given record should be unique throughout the entire database (unless it is participating in establishing a relationship between a pair of tables) and should have only one



exclusive means of identification—the specific primary key value for that record.

You can determine whether a primary key fully complies with this element by following these steps:

1. Load the table with sample data.
2. Select a record for test purposes and note the current primary key value.
3. Examine the value of the first field (the one immediately after the primary key) and ask yourself this question:

Does this primary key value exclusively identify the current value of <fieldname>?

1. If the answer is yes, move to the next field and repeat the question.
2. If the answer is no, remove the field from the table, move to the next field and repeat the question.
4. Continue this procedure until you've examined every field value in the record.

A field value that the primary key does not exclusively identify indicates that the field itself is unnecessary to the table's structure; therefore, you should remove the field and reconfirm that the table complies with the Elements of the Ideal Table. You can then add the field you just removed to another table structure, if appropriate, or you can discard it completely because it is truly unnecessary.

Here's an example of how you might apply this technique to the partial table structure in Figure 8. (Note that INVOICE NUMBER is the primary key of the table.)

FIGURE 8 Does the primary key exclusively identify the value of each field in this table?

### Sales Invoices

Invoice Number	Invoice Date	CustFirst Name	CustLast Name	EmpFirst Name	EmpLast Name	EmpHome Phone
13000	06/15/02	Frank	DeSoto	Estela	Pundt	363-9948
13001	06/15/02	Gregory	Mattson	Katherine	Erllich	322-6992
13002	06/15/02	Caroline	Coie	Kendra	Bonnicksen	527-4992
13003	06/16/02	David	Cunningham	Kendra	Bonnicksen	336-5992
13004	06/16/02	Caroline	Coie	Shannon	McLain	572-9948
13005	06/17/02	Frank	DeSoto	Estela	Pundt	322-6992

First, you load the table with sample data. You then select a record for test purposes—we'll use the third record for this example—and note the value of the primary key (13002). Now, pose the question above for each field value in the record.

Does this primary key value exclusively identify the current value of . . .

INVOICE DATE?

Yes, it does. This invoice number will always identify the specific date that the invoice was created.

CUSTFIRST NAME?

Yes, it does. This invoice number will always identify the specific first name of the particular customer who made this purchase.

CUSTLAST NAME?

Yes, it does. This invoice number will always identify the specific last name of the particular customer who made this purchase.

#### EMPFIRST NAME?

Yes, it does. This invoice number will always identify the specific first name of the particular employee who served the customer for this sale.

#### EMPLAST NAME?

Yes, it does. This invoice number will always identify the specific last name of the particular employee who served the customer for this sale.

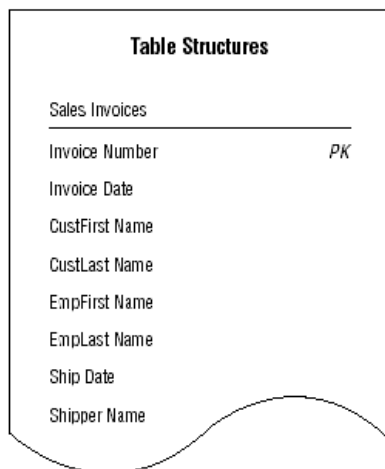
#### EMPHOME PHONE?

No, it doesn't! The invoice number indirectly identifies the employee's home phone number via the employee's name. In fact, it is the current value of both EMPFIRST NAME and EMLAST NAME that exclusively identifies the value of EMPHOME PHONE—change the employee's name and you must change the phone number as well. You should now remove EMPHOME PHONE from the table for two reasons: The primary key does not exclusively identify its current value and (as you've probably already ascertained) it is an unnecessary field. As it turns out, you can discard this field completely because it is already part of the EMPLOYEES table structure.

After you've removed the unnecessary fields you identified during this test, examine the revised table structure and make sure it complies with the Elements of the Ideal Table.

The primary key should now exclusively identify the values of the remaining fields in the table. This means that the primary key is truly sound and you can designate it as the official primary key for the table. Remove the "CK" next to the field name in the table structure and replace it with a "PK." (A primary key composed of two or more fields is known as a composite primary key, and you mark it with the letters "CPK.") Figure 9 shows the revised structure of the SALES INVOICE table with INVOICE NUMBER as its primary key.

FIGURE 9 The revised SALES INVOICES table with its new primary key.



The diagram shows a table structure for 'Sales Invoices'. The fields listed are Invoice Number, Invoice Date, CustFirst Name, CustLast Name, EmpFirst Name, EmpLast Name, Ship Date, and Shipper Name. The 'Invoice Number' field is marked with 'PK' as its primary key.

Table Structures	
Sales Invoices	
Invoice Number	PK
Invoice Date	
CustFirst Name	
CustLast Name	
EmpFirst Name	
EmpLast Name	
Ship Date	
Shipper Name	

As you create a primary key for each table in the database, keep these two rules in mind:  
Rules for Establishing a Primary Key

1. Each table must have one—and only one—primary key. Because the primary key must conform to each of the elements that govern it, only one primary key is necessary for a particular table.
2. Each primary key within the database must be unique—no two tables should have the same primary key unless one of them is a subset table. You learned at the beginning of this section that the primary key exclusively identifies a table throughout the database structure; therefore, each table must have its own unique primary key in order to avoid any possible confusion or ambiguity concerning the table's identity. A subset table is excluded from this

rule because it represents a more specific version of a particular data table's subject—both tables must share the same primary key.

### Alternate Keys

Now that you've selected a candidate key to serve as the primary key for a particular table, you'll designate the remaining candidate keys as alternate keys. These keys can be useful to you in an RDBMS program because they provide an alternative means of uniquely identifying a particular record within the table. If you choose to use an alternate key in this manner, mark its name with "AK" or "CAK" (composite alternate key) in the table structure; otherwise, remove its designation as an alternate key and simply return it to the status of a normal field. You won't be concerned with alternate keys for the remainder of the database- design process, but you will work with them once again as you implement the database in an RDBMS program. (Implementing and using alternate keys in RDBMS programs is beyond the scope of this work—our only objective here is to designate them as appropriate. This is in line with the focus of the book, which is the logical design of a database.)

Figure 10 shows the final structure for the EMPLOYEES table with the proper designation for both the primary key and the alternate keys.

FIGURE 10 The EMPLOYEES table with designated primary and alternate keys.

Table Structures	
<b>Employees</b>	
Employee ID	PK
Social Security Number	
EmpFirst Name	CAK
EmpLast Name	CAK
EmpStreet Address	
EmpCity	
EmpState	
EmpZipcode	
EmpHome Phone	

### Non-keys

A non-key is a field that does not serve as a candidate, primary, alternate, or foreign key. Its sole purpose is to represent a characteristic of the table's subject, and its value is determined by the primary key. There is no particular designation for a non-key, so you don't need to mark it in the table structure.

### Table-Level Integrity

This type of integrity is a major component of overall data integrity, and it ensures the following:

- There are no duplicate records in a table.
- The primary key exclusively identifies each record in a table.
- Every primary key value is unique.
- Primary key values are not null.

You began establishing table-level integrity when you defined a primary key for each table and ensured its enforcement by making absolutely certain that each primary key fully complied with the Elements of a Primary Key. In the next chapter, you'll enhance the table's integrity further as you establish field specifications for each field within the table.

### Reviewing the Initial Table Structures

Now that the fundamental table definitions are complete, you need to conduct interviews with users and management to review the work you've done so far. This set of interviews is fairly straightforward and should be relatively easy to conduct. During these interviews, you will accomplish these tasks:

Ensure that the appropriate subjects are represented in the database. Although it's highly unlikely that an important subject is missing at this stage of the database-design process, it can happen. When it does happen, identify the subject, use the proper techniques to transform it into a table, and develop it to the same degree as the other tables in the database.

Make certain that the table names and table descriptions are suitable and meaningful to everyone. When a name or description appears to be confusing or ambiguous to several people in the organization, work with them to clarify the item as much as possible. It's common for some table names and descriptions to improve during the interview process.

Make certain that the field names are suitable and meaningful to everyone. Selecting field names typically generates a great deal of discussion, especially when there is an existing database in place. You'll commonly find people who customarily refer to a particular field by a certain name because "that's what it's called on my screen." When you change a field name—you have good reasons for doing so—you must diplomatically explain to these folks that you renamed the field so that it conforms to the standards imposed by the new database. You can also tell them that the field can appear with the more familiar name once the database is implemented in an RDBMS program. What you've said is true; many RDBMSs allow you to use one name for the field's physical definition and another name for display purposes.

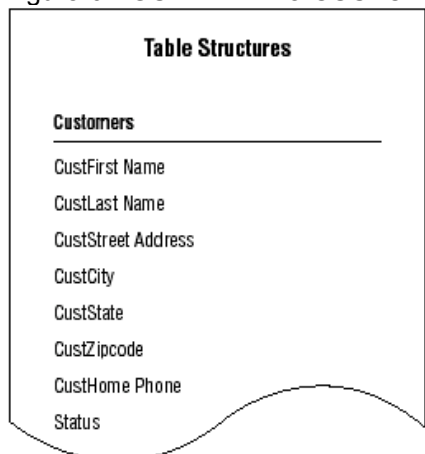
Verify that all the appropriate fields are assigned to each table. This is your best opportunity to make certain that all of the necessary characteristics pertaining to the subject of the table are in place. You'll commonly discover that you accidentally overlooked one or two characteristics earlier in the design process. When this happens, identify the characteristics, use the appropriate techniques to transform them into fields, and follow all the necessary steps to add them to the table.

When you've completed the interviews, you'll move to the next phase of the database-design process and establish field specifications for every field in the database.

### Case Study

It's now time to establish keys for each table in the Mike's Bikes database. As you know, your first order of business is to establish candidate keys for each table. Let's say you decide to start with the CUSTOMERS table in Figure 11.

Figure 9FIGURE 11 The CUSTOMERS table structure in the Mike's Bikes database.



As you review each field, you try to determine whether it conforms to the Elements of a Candidate Key. You determine that STATUS, CUSTHOME PHONE, and the combination of CUSTFIRST NAME and CUSTLAST NAME are potential candidate keys, but you're not quite certain whether any of them will completely conform to all of the elements. So you decide to test the keys by loading the table with sample data as shown in Figure 12.

Figure 10FIGURE 12 Testing candidate keys in the CUSTOMERS table.

### Customers

CustFirst Name	CustLast Name	CustStreet Address	CustCity	CustState	CustZipcode	CustHome Phone	Status
Bridget	Berlin	2121 NE 35th	Bellevue	WA	98004	422-4982	Valued
Phillip	Bradley	101 9th Avenue	Kent	WA	98126	322-1178	
Kel	Brigan	7525 Taxco Lane	Redmond	WA	98225	363-9360	Valued
Barbara	Carmichael	7525 Taxco Lane	Redmond	WA	98225	363-9360	Preferred
Daniel	Chavez	750 Pike Street	Bothell	WA	98001	441-3987	Valued
Daniel	Chavez	301 N Main	Seattle	WA	98115	365-7199	
Sandi	Cooper	115 Pine Place	Seattle	WA	98026	332-0499	Preferred

Always remember that a field must comply with all of the Elements of a Candidate Key in order to qualify as a candidate key. You must immediately disqualify the field if it does not fulfill this requirement.

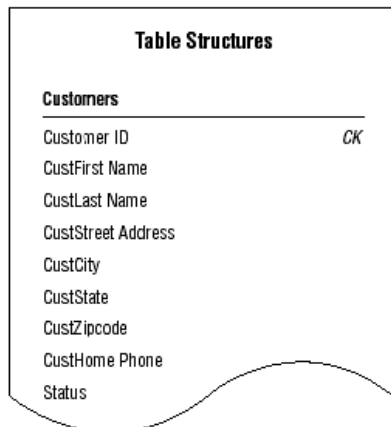
As you examine the table, you draw these conclusions:

- \* STATUS is ineligible because it will probably contain duplicate values. As business grows, Mike is going to have many "Valued" customers.
- \* CUSTHOME PHONE is ineligible because it will probably contain duplicate values. The sample data reveals that two customers can live in the same residence and have the same phone number.
- \* CUSTFIRST NAME and CUSTLAST NAME are ineligible because they will probably contain duplicate values. The sample data reveals that the combination of first name and last name can represent more than one distinct customer.

These findings convince you to establish an artificial candidate key for this table. You then create a field called CUSTOMER ID, confirm that it complies with the requirements for a candidate key, and add the new field to the table structure with the appropriate designation.

Figure 13 shows the revised structure of the CUSTOMERS table.

FIGURE 13 The CUSTOMERS table with the new artificial candidate key, CUSTOMER ID.



Now you'll repeat this procedure for each table in the database. Remember to make certain that every table has at least one candidate key.

The next order of business is to establish a primary key for each table. As you know, you select the primary key for a particular table from the table's pool of available candidate keys. Here are a few points to keep in mind when you're choosing a primary key for a table with more than one candidate key:

Choose a simple (single-field) candidate key over a composite candidate key.  
If possible, pick a candidate key that has the table name incorporated into its own name.  
Select the candidate key that best identifies the subject of the table or is most meaningful to everyone in the organization.

You begin by working with the EMPLOYEES table in Figure 14. As you review the candidate keys, you decide that EMPLOYEE NUMBER is a much better choice for a primary key than the combination of EMPFIRST NAME and EMPLAST NAME because Mike's employees are already accustomed to identifying themselves by their assigned numbers. Using EMPLOYEE NUMBER makes perfect sense, so you select it as the primary key for the table.

FIGURE 14 The EMPLOYEES table structure in the Mike's Bikes database.

Table Structures	
<b>Employees</b>	
Employee Number	CK
Social Security Number	
EmpFirst Name	CCK
EmpLast Name	CCK
EmpStreet Address	
EmpCity	
EmpState	
EmpZipcode	
EmpHome Phone	

Now you perform one final task before you designate EMPLOYEE NUMBER as the official primary key of the table: You make absolutely certain that it exclusively identifies the value of each field within a given record. So, you test EMPLOYEE NUMBER by following these steps:

1. Load the EMPLOYEES table with sample data.
2. Select a record for test purposes and note the current value of EMPLOYEE NUMBER.
3. Examine the value of the first field (the one immediately after EMPLOYEE NUMBER) and ask yourself this question:

Does this primary key value exclusively identify the current value of <fieldname>?

1. If the answer is yes, move to the next field and repeat the question.
2. If the answer is no, remove the field from the table, move to the next field and repeat the question. (Be sure to determine whether you can add the field you just removed to another table structure, if appropriate, or discard it completely because it is truly unnecessary.)
4. Continue this procedure until you've examined every field value in the record.

You know that you'll have to remove any field containing a value that EMPLOYEE NUMBER does not exclusively identify. EMPLOYEE NUMBER does exclusively identify the value of each field in the test record, however, so you use it as the official primary key for the EMPLOYEES table and mark its

name with the letters "PK" in the table structure. You then repeat this process with the rest of the tables in Mike's new database until every table has a primary key.

Remember to keep these rules in mind as you establish primary keys for each table:

Each table must have one—and only one—primary key.

Each primary key within the database should be unique—no two tables should have the same primary key (unless one of them is a subset table).

As you work through the tables in Mike's database, you remember that the SERVICES table is a subset table. You created it during the previous stage of the design process (in Chapter 7), and it represents a more specific version of the subject represented by the PRODUCTS table. The PRODUCT NAME field is what currently relates the PRODUCTS table to the SERVICES subset table. You now know, however, that a subset table must have the same primary key as the table to which it is related, so you'll use PRODUCT NUMBER (the primary key of the PRODUCTS table) as the primary key of the SERVICES table. Figure 15 shows the PRODUCTS and SERVICES tables with their primary keys.

FIGURE 15 Establishing the primary key for the SERVICES subset table.

<b>Table Structures</b>			
<b>Products</b>		<b>Services</b>	
Product Number	<i>PK</i>	Product Number	<i>PK</i>
Product Name		Service Type	
Product Description		Materials Charge	
Category		Service Charge	
Wholesale Price			
Retail Price			
Quantity On Hand			

The last order of business is to conduct interviews with Mike and his staff and review all the work you've performed on the tables in the database. As you conduct these interviews, make certain you check the following:

- That the appropriate subjects are represented in the database
- That the table names and descriptions are suitable and meaningful to everyone
- That the field names are suitable and meaningful to everyone
- That all the appropriate fields are assigned to each table

By the end of the interview, everyone agrees that the tables are in good form and that all the subjects with which they are concerned are represented in the database. Only one minor point came up during the discussions: Mike wants to add a CALL PRIORITY field to the VENDORS table. There are instances in which more than one vendor supplies a particular product, and Mike wants to create a way to indicate which vendor he should call first if that product is unexpectedly out of stock. So, you add the new field to the VENDORS table and bring the interview to a close.

### Keys Summary

The chapter opened with a discussion of the importance of keys. You learned that there are different types of keys, and each type plays a different role within the database. Each key performs a

particular function, such as uniquely identifying records, establishing various types of integrity, and establishing relationships between tables. You now know that you can guarantee sound table structure by making certain that the appropriate keys are established for each table.

We then discussed the process of establishing keys for each table. We began by identifying the four main types of keys: candidate, primary, foreign, and non-keys. First, we looked at the process of establishing candidate keys for each table. You learned about the Elements of a Candidate Key and how to make certain that a field (or set of fields) complies with these elements. Then you learned that you can create and use an artificial candidate key when none of the fields in a table can serve as a candidate key or when a new field would make a stronger candidate key than any of the existing candidate key fields.

The chapter continued with a discussion of primary keys. You learned that you select a primary key from a table's pool of candidate keys and that the primary key is governed by a set of specific elements. We then covered a set of guidelines that help you determine which candidate key to use as a primary key. Next, you learned how to ensure that the chosen primary key exclusively identifies a given record and its set of field values. When the primary key does not exclusively identify a particular field value, you know that you must remove the field from the table in order to ensure the table's structural integrity. You also know that each table must have a single, unique primary key.

You then learned that you designate any remaining candidate keys as alternate keys. These keys will be most useful to you when you implement the database in an RDBMS program because they provide an alternate means of identifying a given record. We then discussed the non-key field, which is any field not designated as a candidate, primary, alternate, or foreign key. You now know that a non-key field represents a characteristic of the table's subject and that the primary key exclusively identifies its value.

Table-level integrity was the next subject of discussion, and you learned that it is established through the use of primary keys and enforced by the Elements of a Primary Key.

The chapter closed with some guidance on conducting further interviews with users and management. You now know that these interviews provide you with a means of reviewing the work you have performed on the tables and help you to verify and validate the current database structure.

### ***Permissible Design Infractions or Now That You Know the Rules...***

Hmmm don't know of any.

### ***Summary***

#### ***Computer-less Design***

One thing I hope you've noticed is that we've done all our design without the aid of a computer. This is as it should be: it lets you focus on the significance of the task without the distractions of trying to learn a database program at the same time.

You can design and test your database structure without going near a computer. The only thing you really need to know is the type of database program you'll use: if it's a flat-file database you'll be limited to single-table database design. If it's a relational program you can design single- or multi-table databases.

Also, by using Normalization, breaking down your fields into simpler components in a single table can make it easier to get useful information out of your database, eliminate redundant information, and exclude inconsistencies.

You also learned that your design should minimize redundancy without losing data, that Insertion, deletion, and update anomalies are problems that occur when trying to insert, delete, or update data in a table with a flawed structure. And that you should avoid designs that will lead to large quantities of null values.



## GLOSSARY

database	A collection of related information stored in a structured format. Database is often used interchangeably with the term table (Lotus Approach, for instance, uses the term database instead of table). Technically, they're different: a table is a single store of related information; a database can consist of one or more tables of information that are related in some way. For instance, you could track all the information about the students in a school in a students table. If you then created separate tables containing details about teachers, classes and classrooms, you could combine all four tables into a timetabling database. Such a multi-table database is called a relational database.
data entry	The process of getting information into a database, usually done by people typing it in by way of data-entry forms designed to simplify the process.
dbms	Database management system. A program which lets you manage information in databases. Lotus Approach, Microsoft Access and FileMaker Pro, for example, are all DBMSs, although the term is often shortened to 'database'. So, the same term is used to apply to the program you use to organise your data and the actual data structure you create with that program.
entities	Things in the real world that we will store information about in the database.
field	Fields describe a single aspect of each member of a table. A student record, for instance, might contain a last name field, a first name field, a date of birth field and so on. All records have exactly the same structure, so they contain the same fields. The values in each field vary from record to record, of course.
flat file	A database that consists of a single table. Lightweight database programs such as the database component in Microsoft Works are sometimes called 'flat-file managers' (or list managers) because they can only handle single-table databases. More powerful programs, such as Access, FileMaker Pro and Approach, can handle multi-table databases, and are called relational database managers, or RDBMSs.
index	A summary table which lets you quickly locate a particular record or group of records in a table. Think of how you use an index to a book: as a quick jumping off point to finding full information about a subject. A database index works in a similar way. You can create an index on any field in a table. Say, for example, you have a customer table which contains customer numbers, names, addresses and other details. You can make indexes based on any information, such as the customers' customer number, last name + first name (a composite index based on more than one field), or postal code. Then, when you're searching for a particular customer or group of customers, you can use the index to speed up the search.
key field	You can sort and quickly retrieve information from a database by choosing one or more fields to act as keys. For instance, in a students table you could use a combination of the last name and first name fields as a key field. The database program will create an index containing just the key field contents. Using the index, you can quickly find any record by typing in the student's name. The database will locate the correct entry in the index and then display the full record.
primary key	A field that uniquely identifies a record in a table. In a students table, a key built from last name + first name might not give you a unique identifier (two or more Jane Does in the school, for example). To uniquely identify each student, you might add a special

Student ID field to be used as the primary key.

record A record contains all the information about a single 'member' of a table. In our students table, each student's details (name, date of birth, contact details, and so on) will be contained in its own record.

relational database A database consisting of more than one table. In a multi-table database, you not only need to define the structure of each table, you also need to define the relationships between each table in order to link those tables correctly.

relationship The link between the entities in a database.

schema

table A single store of related information. A table consists of records, and each record is made up of a number of fields. You can think of the phone book as a table: It contains a record for each telephone subscriber, and each subscriber's details are contained in three fields – name, address and telephone.

## **REFERENCES**

SQL Tutorial:

[http://www.highcroft.com/highcroft/sql\\_intro.pdf](http://www.highcroft.com/highcroft/sql_intro.pdf)

<http://www.phpbuilder.com/columns/barry20000731.php3>

<http://www.dbpd.com/vault/9805extra.htm>

The Case for the Surrogate Key A simple key to the flexible enterprise database

[http://www.geekgirls.com/databases\\_from\\_scratch\\_1.htm](http://www.geekgirls.com/databases_from_scratch_1.htm)

<http://www.tomjewett.com/dbdesign/dbdesign.php?page=intro.html&imgsize=medium>

## Requirements Gathering and Design Steps

1. Define the Scope as the Area of Interest,(e.g. the HR Department in an organization).
2. Define the "Things of Interest",(e.g. Employees), in the Area of Interest.
3. Analyze the Things of Interest and identify the corresponding Tables.
4. Consider cases of 'Inheritance', where there are general Entities and Specific Entities. For example, a Customer is a General Entity, and Commercial Customer and Personal Customer would be Specific Entities. If you are just starting out, I suggest that you postpone this level of analysis.
5. At this point, you can produce a List of Things of Interest.
6. Establish the relationships between the Tables.  
For example, "A Customer can place many Orders", and "A Product can be purchased many times and appear in many Orders."
7. Determine the characteristics of each Table,(e.g. an Employee has a Date-of-Birth).
8. Identify the Static and Reference Data, such as Country Codes or Customer Types.
9. Obtain a small set of Sample Data, e.g. "John Doe is a Maintenance Engineer and was born on 1st. August, 1965 and lives at 22 Woodland Street, New Haven.  
"He is currently assigned to maintenance of the Air-Conditioning and becomes available in 4 weeks time"
10. Review Code or Type Data which is (more or less) constant, which can be classified as Reference Data.  
For example, Currency or Country Codes. Where possible, use standard values, such as ISO Codes.
11. Look for 'has a' relationships. These can become Foreign Keys, or 'Parent-Child' relationships.
12. You need to define a Primary Key for all Tables.  
For Reference Tables, use the'Code' as the Key, often with only one other field, which is the Description field.  
I recommend that names of Reference Data Tables all start with 'REF\_'.  
For all non-Reference Data Tables, I suggest that you simply assign an Auto-increment Integer for each Primary Key.  
This has some benefits, for example, it provides flexibility, and it's really the only choice for a Database supporting a Web Site.  
However, it complicates life for developers, which have to use the natural key to join on, as well as the 'surrogate' key.  
It also makes it possible to postpone a thorough analysis of what the actual Primary Key should be. Which means, of course, that it often never gets done.
13. Confirm the first draft of the Database design against the Sample Data.
14. Review the Business Rules with Users,(if you can find any Users).
15. Obtain from the Users some representative enquiries for the Database, e.g. "How many Maintenance Engineers do we have on staff coming available in the next 4 weeks ?"
16. Review the Results of Steps 1) to 9) with appropriate people, such as Users, Managers, Development staff, etc. and repeat until the final Database design is reached.
17. Define User Scenarios and step through them with some sample data to check that that Database supports the required functionality.

## Database Program Tools

A database program gives you the tools to:

- Design the structure of your database
- Create data entry forms so you can get information into the database
- Validate the data entered and check for inconsistencies
- Sort and manipulate the data in the database
- Query the database (that is, ask questions about the data)
- Produce flexible reports, both on screen and on paper, that make it easy to comprehend the information stored in the database.

Most of the more advanced database programs have built-in programming or macro languages, which let you automate many of their functions.

### Using a database

If the mention of programming languages makes you feel you're getting out of your depth, don't worry! Most of the database programs you're likely to encounter can be used at a variety of levels.

If you're a beginner, you'll find built-in templates, sample databases, 'wizards' and 'experts' that will do much of the hard work for you. If you find the built-in databases don't quite work for you, it's easy to modify an existing database so it fits your needs, and it's not at all difficult to learn to create your own simple database structure from scratch.

For more advanced users, the more powerful database programs enable you to create complete, custom-built, application-specific systems which can be used by others in your organisation or business.

## EXAMPLE CASES

### ***A sample design process***

Let's step through a sample database design process. We'll design a database to keep track of students' sports activities. We'll track each activity a student takes and the fee per semester to do that activity.

Step 1: Create an Activities table containing all the fields: student's name, activity and cost. Because some students take more than one activity, we'll make allowances for that and include a second activity and cost field. So our structure will be: Student, Activity 1, Cost 1, Activity 2, Cost 2

Step 2: Test the table with some sample data. When you create sample data, you should see what your table lets you get away with. For instance, nothing prevents us from entering the same name for different students, or different fees for the same activity, so do so. You should also imagine trying to ask questions about your data and getting answers back (essentially querying the data and producing reports). For example, how do I find all the students taking tennis?

Testing our first table design

### **Activities Table**

<b>Student</b>	<b>Activity1</b>	<b>Cost1</b>	<b>Activity2</b>	<b>Cost2</b>
John Smith	Tennis	\$36	Swimming	\$17
Jane Bloggs	Squash	\$40	Swimming	\$17
John Smith	Tennis	\$36		
Mark Antony	Swimming	\$15	Golf	\$47

Step 3: Analyse the data. In this case, we can see a glaring problem in the first field. We have two John Smiths, and there's no way to tell them apart. We need to find a way to identify each student uniquely.

### ***Uniquely identify records***

Let's fix the glaring problem first, then examine the new results.

Step 4: Modify the design. We can identify each student uniquely by giving each one a unique ID, a new field that we add, called ID. We scrap the Student field and substitute an ID field. Note the asterisk (\*) beside this field in the table below: it signals that the ID field is a key field, containing a unique value in each record. We can use that field to retrieve any specific record. When you create such a key field in a database program, the program will then prevent you from entering duplicate values in this field, safeguarding the uniqueness of each entry.

Our table structure is now:

ID      Activity 1      Cost 1      Activity 2      Cost 2

While it's easy for the computer to keep track of ID codes, it's not so useful for humans. So we're going to introduce a second table that lists each ID and the student it belongs to. Using a database program, we can create both table structures and then link them by the common field, ID. We've now turned our initial flat-file design into a relational database: a database containing multiple tables linked together by key fields. If you were using a database program that can't handle relational databases, you'd basically be stuck with our first design and all its

attendant problems. With a relational database program, you can create as many tables as your data structure requires.

The Students table would normally contain each student's first name, last name, address, age and other details, as well as the assigned ID. To keep things simple, we'll restrict it to name and ID, and focus on the Activities table structure.

Step 5: Test the table with sample data.

Testing our revised table structure

### Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

### Activities Table

ID*	Activity1	Cost1	Activity2	Cost2
084	Tennis	\$36	Swimming	\$17
100	Squash	\$40	Swimming	\$17
182	Tennis	\$36		
219	Swimming	\$15	Golf	\$47

Step 6: Analyse the data. There's still a lot wrong with the Activities table:

1. Wasted space. Some students don't take a second activity, and so we're wasting space when we store the data. It doesn't seem much of a bother in this sample, but what if we're dealing with thousands of records?
2. Addition anomalies. What if #219 (we can look him up and find it's Mark Antony) wants to do a third activity? School rules allow it, but there's no space in this structure for another activity. We can't add another record for Mark, as that would violate the unique key field ID, and it would also make it difficult to see all his information at once.
3. Redundant data entry. If the tennis fees go up to \$39, we have to go through every record containing tennis and modify the cost.
4. Querying difficulties. It's difficult to find all people doing swimming: we have to search through Activity 1 and Activity 2 to make sure we catch them all.
5. Redundant information. If 50 students take swimming, we have to type in both the activity and its cost each time.
6. Inconsistent data. Notice that there are conflicting prices for swimming? Should it be \$15 or \$17? This happens when one record is updated and another isn't.

### **Eliminate recurring fields**

The Students table is fine, so we'll keep it. But there's so much wrong with the Activities table let's try to fix it in stages.

Step 7: Modify the design. We can fix the first four problems by creating a separate record for each activity a student takes, instead of one record for all the activities a student takes.

First we eliminate the Activity 2 and Cost 2 fields. Then we need to adjust the table structure so we can enter multiple records for each student. To do that, we redefine the key so that it consists of two fields, ID and Activity. As each student can only take an activity once, this combination gives us a unique key for each record.

Our Activities table has now been simplified to: ID, Activity, Cost. Note how the new structure lets students take any number of activities – they're no longer limited to two.

Step 8: Test sample data.

Table design: version III

Students Table		Activities Table		
Student	ID*	ID*	Activity*	Cost
John Smith	084	084	Swimming	\$17
Jane Bloggs	100	084	Tennis	\$36
John Smith	182	100	Squash	\$40
Mark Antony	219	100	Swimming	\$17
		182	Tennis	\$36
		219	Golf	\$47
		219	Swimming	\$15
		219	Squash	\$40

Step 9: Analyse the data. We know we still have the problems with redundant data (activity fees repeated) and inconsistent data (what's the correct fee for swimming?). We need to fix these things, which are both problems with editing or modifying records.

### ***Eliminate data entry anomalies***

As well, we should check that other data entry processes, such as adding or deleting records, will function correctly too.

If you look closely, you'll find that there are potential problems when we add or delete records:

- Insertion anomalies. What if our school introduces a new activity, such as sailing, at \$50. Where can we store this information? With our current design we can't until a student signs up for the activity.
- Deletion anomalies. If John Smith (#182) transfers to another school, all the information about golf disappears from our system, as he was the only student taking this activity.

Step 10: Modify the design. The cause of all our remaining problems is that we have a non-key field (cost) which is dependent on only part of the key (activity). Check it out for yourself: The cost of each activity is not dependent on the student's ID, which is part of our composite key (ID + Activity). The cost of tennis, for example, is \$36 for each and every student who takes the sport – so the student's ID has no bearing on the value contained in this field. The cost of an activity is purely dependent on the activity itself. This is a design no-no, and it's causing us problems. By checking our table structures and ensuring that every non-key field is dependent on the whole key, we will eliminate the rest of our problems.

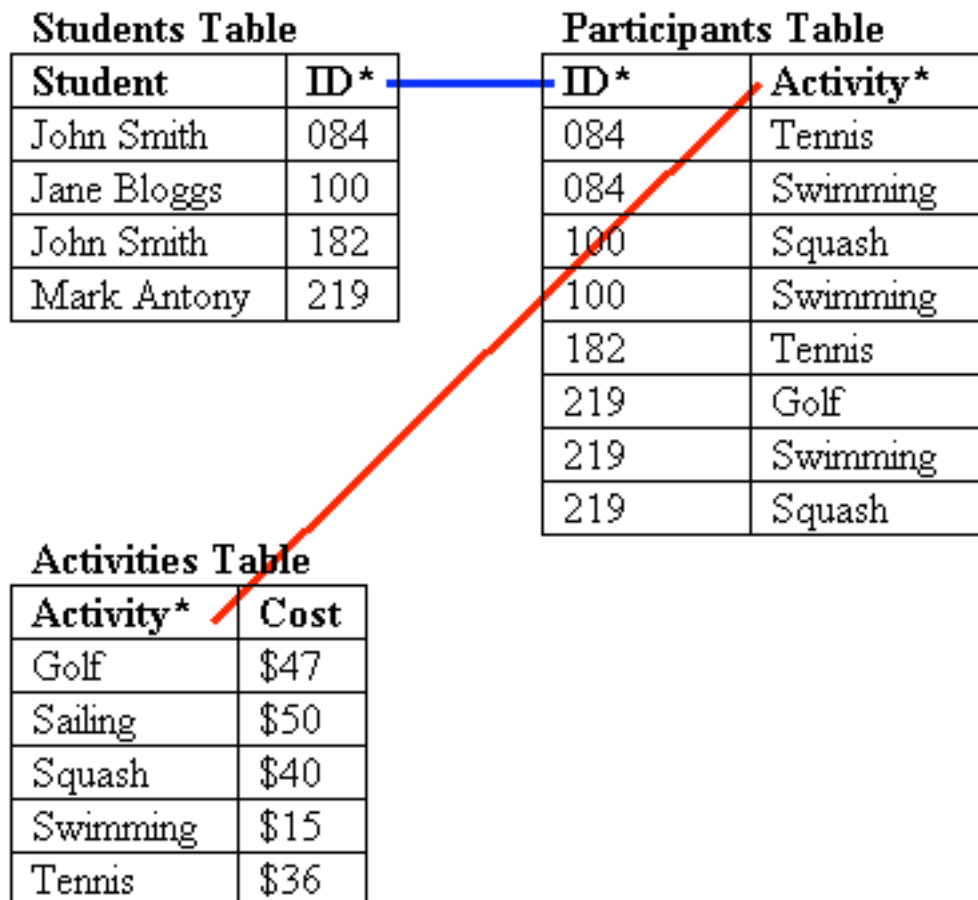


Our final design will thus contain three tables: the Students table (Student, ID), a Participants table (ID, Activity), and a modified Activities table (Activity, Cost).

If you check these tables, you'll see that each non-key value depends on the whole key: the student name is entirely dependent on the ID; the activity cost is entirely dependent on the activity. Our new Participants table essentially forms a union of information drawn from the other two tables, and each of its fields is part of the key. The tables are linked by key fields: the Students table:ID corresponds to the Participants table:ID; the Activities table:Activity corresponds to the Participants table:Activity.

Step 11: Test sample data.

Testing data in the final table design



Step 12: Analyse the results. This looks good:

- No redundant information. You need only list each activity fee once.
- No inconsistent data. There's only one place where you can enter the price of each activity, so there's no chance of creating inconsistent data. Also, if there's a fee rise, all you need to do is update the cost in one place.
- No insertion anomalies. You can add a new activity to the Activities table without a student signing up for it.
- No deletion anomalies. If John Smith (#219) leaves, you still retain the details about the golfing activity.

Keep in mind that to simplify the process and focus on the relational aspects of designing our database structure, we've placed the student's name in a single field. This is not what you'd normally do: you'd divide it into firstname, lastname (and initials) fields. Similarly, we've excluded other fields that you would normally store in a student table, such as date of birth, address, parents' names and so on.

### ***A summary of the design process***

Although your ultimate design will depend on the complexity of your data, each time you design a database, make sure you do the following:

- Break composite fields down into constituent parts. Example: Name becomes lastname and firstname.
- Create a key field which uniquely identifies each record. You may need to create an ID field (with a lookup table that shows you the values for each ID) or use a composite key.
- Eliminate repeating groups of fields. Example: If your table contains fields Location 1, Location 2, Location 3 containing similar data, it's a sure warning sign.
- Eliminate record modification problems (such as redundant or inconsistent data) and record deletion and addition problems by ensuring each non-key field depends on the entire key. To do this, create a separate table for any information that is used in multiple records, and then use a key to link these tables to one another.

### Another Normalization Example

To follow the normalization process, we take our database design through the different forms in order. Generally, each form subsumes the one below it. For example, for a database schema to be in second normal form, it must also be in first normal form. For a schema to be in third normal form, it must be in second normal form and so on. At each stage, we add more rules that the schema must satisfy.

#### First Normal Form

The first normal form, sometimes called 1NF, states that each attribute or column value must be atomic. That is, each attribute must contain a single value, not a set of values or another database row.

Consider the table shown in Figure 3.4.

Figure 3.4 This schema design is not in first normal form because it contains sets of values in the skill column.

#### employee

employeeID	name	job	departmentID	skills
7513	Nora Edwards	Programmer	128	C, Perl, Java
9842	Ben Smith	DBA	42	DB2
6651	Ajay Patel	Programmer	128	VB, Java
9006	Candy Burnett	Systems Administrator	128	NT, Linux

This is an unnormalized version of the employee table we looked at earlier. As you can see, it has one extra column, called skill, which lists the skills of each employee.

Each value in this column contains a set of values—that is, rather than containing an atomic value such as Java, it contains a list of values such as C, Perl, Java. This violates the rules of first normal form.

To put this schema in first normal form, we need to turn the values in the skill column into atomic values. There are a couple of ways we can do this. The first, and perhaps most obvious, way is shown in Figure 3.5.

Figure 3.5 All values are now atomic.

#### employee

employeeID	Name	job	departmentID	skill
7513	Nora Edwards	Programmer	128	C
7513	Nora Edwards	Programmer	128	Perl
7513	Nora Edwards	Programmer	128	Java
9842	Ben Smith	DBA	42	DB2
6651	Ajay Patel	Programmer	128	VB
6651	Ajay Patel	Programmer	128	Java
9006	Candy Burnett	Systems Administrator	128	NT
9006	Candy Burnett	Systems Administrator	128	Linux

Here we have made one row per skill. This schema is now in first normal form.

Obviously, this arrangement is far from ideal because we have a great deal of redundancy—for each skill-employee combination, we store all the employee details.

A better solution, and the right way to put this data into first normal form, is shown in Figure 3.6.

Figure 3.6 We solve the same problem the right way by creating a second table.

### employee

employeeID	name	job	departmentID
7513	Nora Edwards	Programmer	128
9842	Ben Smith	DBA	42
6651	Ajay Patel	Programmer	128
9006	Candy Burnett	Systems Administrator	128

### employeeSkills

employeeID	skill
7513	C
7513	Perl
7513	Java
9842	DB2
6651	VB
6651	Java
9006	NT
9006	Linux

In this example, we have split the skills off to form a separate table that only links employee ids and individual skills. This gets rid of the redundancy problem.

You might ask how we would know to arrive at the second solution. There are two answers. One is experience. The second is that if we take the schema in Figure 3.5 and continue with the normalization process, we will end up with the schema in Figure 3.6. The benefit of experience allows us to look ahead and just go straight to this design, but it is perfectly valid to continue with the process.

#### Second Normal Form

After we have a schema in first normal form, we can move to the higher forms, which are slightly harder to understand.

A schema is said to be in second normal form (also called 2NF) if all attributes that are not part of the primary key are fully functionally dependent on the primary key, and the schema is already in first normal form. What does this mean? It means that each non-key attribute must be functionally dependent on all parts of the key. That is, if the primary key is made up of multiple columns, every other attribute in the table must be dependent on the combination of these columns.

Let's look at an example to try to make things clearer.

Look at Figure 3.5. This is the schema that has one line in the employee table per skill. This table is in first normal form, but it is not in second normal form. Why not?

What is the primary key for this table? We know that the primary key must uniquely identify a single row in a table. In this case, the only way we can do this is by using the combination of the employeeID and the skill. With the skills set up in this way, the employeeID is not enough to uniquely identify a row—for example, the employeeID 7513 identifies three rows. However, the combination of

employeeID and skill will identify a single row, so we use these two together as our primary key. This gives us the following schema:

```
employee(employeeID, name, job, departmentID, skill)
```

We must next ask ourselves, "What are the functional dependencies here?" We have

```
employeeID, skill → name, job, departmentID
```

but we also have

```
employeeID → name, job, departmentID
```

In other words, we can determine the name, job, and departmentID from the employeeID alone. This means that these attributes are partially functionally dependent on the primary key, rather than fully functionally dependent on the primary key. That is, you can determine these attributes from a part of the primary key without needing the whole primary key. Hence, this schema is not in second normal form.

The next question is, "How can we put it into second normal form?"

We need to decompose the table into tables in which all the non-key attributes are fully functionally dependent on the key. It is fairly obvious that we can achieve this by breaking the table into two tables, to wit:

```
employee(employeeID, name, job, departmentID)
```

```
employeeSkills(employeeID, skill)
```

This is the schema that we had back in Figure 3.6.

As already discussed, this schema is in first normal form because the values are all atomic. It is also in second normal form because each non-key attribute is now functionally dependent on all parts of the keys.

### Third Normal Form

You may sometimes hear the saying "Normalization is about the key, the whole key, and nothing but the key." Second normal form tells us that attributes must depend on the whole key. Third normal form tells us that attributes must depend on nothing but the key.

Formally, for a schema to be in third normal form (3NF), we must remove all transitive dependencies, and the schema must already be in second normal form. Okay, so what's a transitive dependency?

Look back at Figure 3.3. This has the following schema:

```
employeeDepartment(employeeID, name, job, departmentID, departmentName)
```

This schema contains the following functional dependencies:

```
employeeID → name, job, departmentID, departmentName
```

```
departmentID → departmentName
```

The primary key is employeeID, and all the attributes are fully functionally dependent on it—this is easy to see because there is only one attribute in the primary key!

However, we can see that we have

employeeID  $\rightarrow$  departmentName

employeeID  $\rightarrow$  departmentID

and

departmentID  $\rightarrow$  departmentName

Note also that the attribute departmentID is not a key.

This relationship means that the functional dependency employeeID  $\rightarrow$  departmentName is a transitive dependency. Effectively, it has a middle step (the departmentID  $\rightarrow$  departmentName dependency).

To get to third normal form, we need to remove this transitive dependency.

As with the previous normal forms, to convert to third normal form we decompose this table into multiple tables. Again, in this case, it is pretty obvious what we should do. We convert the schema to two tables, employee and department, like this:

employee(employeeID, name, job, departmentID)

department(departmentID, departmentName)

This brings us back to the schema for employee that we had in Figure 3.2 to begin with. It is in third normal form.

Another way of describing third normal form is to say that formally, if a schema is in third normal form, then for every functional dependency in every table, either

The left side of the functional dependency is a superkey (that is, a key that is not necessarily minimal).

or

The right side of the functional dependency is part of any key of that table.

The second part doesn't come up terribly often! In most cases, all the functional dependencies will be covered by the first rule.

Boyce-Codd Normal Form

The final normal form we will consider—briefly—is Boyce-Codd normal form, sometimes called BCNF. This is a variation on third normal form. We looked at two rules previously. For a relation to be in BCNF, it must be in third normal form and come under the first of the two rules. That is, all the functional dependencies must have a superkey on the left side.

This is most frequently the case without our having to take any extra steps, as in this example. If we have a dependency that breaks this rule, we must again decompose as we did to get into 1NF, 2NF, and 3NF.

Higher Normal Forms

There are higher normal forms (fourth, fifth, and so on), but these are more useful for academic pursuits than practical database design. 3NF (or BCNF) is sufficient to avoid the data redundancy problems you will encounter.

**And Yet Another Normalization Sample**

## Normalization

### Overview

Normalization is a process in which an initial DB design is transformed, or decomposed, into a different, but equivalent, design. The resulting schema is equivalent to the original one in the sense that no information is lost when going from one to the other.

The normalization procedure consists of a sequence of projections -- that is, some attributes are extracted from one table to form a new one. In other words, tables are split up vertically. The decomposition is lossless, only if you can restore the original table by joining its projections.

Through such non-loss decompositions it is possible to transform an original schema into a resulting one that satisfies certain conditions, known as Normal Forms:

The First Normal Form (1NF) addresses the structure of an isolated table.

The Second (2NF), Third (3NF), and Boyce-Codd (BCNF) Normal Forms address one-to-one and one-to-many relationships.

The Fourth (4NF) and Fifth (5NF) Normal Forms deal with many-to-many relationships.

These Normal Forms form a hierarchy in such a way that a schema in a higher normal form automatically fulfills all the criteria for all of the lower Normal Forms.

The Fifth Normal Form is the ultimate normal form with respect to projections and joins -- it is guaranteed to be free of anomalies that can be eliminated by taking projections.

In the following discussion, any mention of keys refers to the conceptual keys formed from business data, not to any plainly technical surrogate keys which might have been defined.

### First Normal Form

A table is said to be in First Normal Form (1NF), if all entries in it are scalar-valued. Relational database tables are 1NF by construction since vector-valued entries are forbidden. Vector-valued data (that is, entries which have more than one value in each row) are referred to as repeating groups.

The following relation violates 1NF because the SupplierID forms a repeating group (here and in the following examples and text, primary key fields are in bold):

```
{ PartID, Supplier1ID, Supplier2ID, Supplier3ID }
```

Repeating groups indicate a one-to-many relationship -- in other words, a relationship which in relational databases is treated using foreign keys. Note that the problem of repeating groups cannot be solved by adding any number of fields to a record; even if the number of elements of the vector-valued data was fixed, finite, and predetermined, searching for a value in all these parallel fields is prohibitively cumbersome.

To achieve 1NF, eliminate repeating groups by creating separate tables for each set of related data.

To demonstrate the typical anomalies that occur in tables that are only 1NF, consider the following example:

```
{ CustomerID, OrderID, CustomerAddress, OrderDate }
```

Note the following problems:

Insert: It is not possible to add a record for a customer who has never placed an order.

Update: To change the address for a customer, this change has to be repeated for all of the

customer's existing orders.

Delete: Deleting the last order for a customer loses all information about the customer.

### Functional dependency

The Second and Third Normal Forms address dependencies among attributes, specifically between key and non-key fields.

By definition, a key uniquely determines a record: Knowing the key determines the values of all the other attributes in the table row, so that given a key, the values of all the other attributes in the row are fixed.

This kind of relationship can be formalized as follows. Let X and Y be attributes (or sets of attributes) of a given relationship. Then Y is functionally dependent on X if, whenever two records agree on their X-values, they must also agree on their Y-values. In this case, X is called the determinant and Y is called the dependent. Since for any X there must be a single Y, this relationship represents a single-valued functional dependency. If the set of attributes in the determinant is the smallest possible (in the sense that after dropping one or more of the attributes from X, the remaining set of attributes does no longer uniquely determine Y), then the dependency is called irreducible.

Note that functional dependency is a semantic relationship: It is the business logic of the problem domain, represented by the relation, which determines whether a certain X determines Y.

### Second Normal Form

A table is in Second Normal Form (2NF) if every non-key field is a fact about the entire key. In other words, a table is 2NF if it is 1NF and all non-key attributes are functionally dependent on the entire primary key (that is, the dependency is irreducible).

Clearly, 2NF is only relevant when the key is composite (that is, consisting of several fields). The following example describes a table which is not 2NF since the WarehouseAddress attribute depends only on WarehouseID but not on PartID:

```
{ PartID, WarehouseID, Quantity, WarehouseAddress }
```

To achieve 2NF, create separate tables for sets of values that apply to multiple records and relate these tables through foreign keys. The determinants of the initial table become the primary keys of the resulting tables.

### Third Normal Form

A relation is in Third Normal Form (3NF) if it is 2NF and none of its attributes is a fact about another non-key field. In other words, no non-key field functionally depends on any other non-key field. (Such indirect dependencies are known as transitive dependencies.)

The following example violates 3NF since the Location is functionally dependent on the DepartmentID:

```
{ EmployeeID, DepartmentID, Location }
```

To achieve 3NF, eliminate fields that do not depend on the key from the original table and add them to the table whose primary key is their determinant.

To summarize the normalization procedure up to and including Third Normal Form:

Every field in a record must depend on The Key (1NF), the Whole Key (2NF), and Nothing But The Key (3NF).

### Boyce-Codd Normal Form



Boyce-Codd Normal Form (BCNF) is an extension of 3NF in the case with two or more candidate keys which are composite and overlapping (that is, they have at least one field in common). If these conditions are not fulfilled, 3NF and BCNF are equivalent. A table is BCNF if, and only if its only determinants are candidate keys.

In the following table, both {SupplierID, PartID}, as well as {SupplierName, PartID}, are candidate keys. The table is not BCNF since it contains two determinants (SupplierID and SupplierName) which are not candidate keys. (SupplierID and SupplierName are determinants, since they determine each other.)

{ SupplierID, PartID, SupplierName, Quantity }

However, either of the following decompositions is BCNF:

{ SupplierID, SupplierName }  
{ SupplierID, PartID, Quantity }

or

{ SupplierName, SupplierID }  
{ SupplierName, PartID, Quantity }

To achieve BCNF, remove the determinants which are not candidate keys.

Many-to-many relationships and higher Normal Forms

Fourth and Fifth Normal Forms apply to situations involving many-to-many relationships. In relational databases, many-to-many relationships are expressed through cross-reference tables.

As an example, consider a case of class enrollment. Each student can be enrolled in one or more classes and each class can contain one or more students. Clearly, there is a many-to-many relationship between classes and students. This relationship can be represented by a Student/Class cross-reference table:

{ StudentID, ClassID }

The key for this table is the combination of StudentID and ClassID. To avoid violation of 2NF, all other information about each student and each class is stored in separate Student and Class tables, respectively.

Note that each StudentID determines not a unique ClassID, but a well-defined, finite set of values. This kind of behavior is referred to as multi-valued dependency of ClassID on StudentID.

Fourth Normal Form

Finally, we are up to the big one. 4NF is the father of the forms, as it is the be all and end all. This takes care of any problem that may occur. Lets start this time by defining a very important term, multivalued dependence (MD). MD is when field B is multidependent on A if each value of A is associated with a specific list of values for B, and this collection is independent of any values of C.

A table is in Fourth Normal Form (4NF) if it is 3NF and it does not represent two or more independent many-to-many relationships.

Consider an example with two many-to-many relationships, between students and classes and between classes and teachers. Also, a many-to-many relationship between students and teachers is

implied. However, the business rules do not constrain this relationship in any way -- the combination of StudentID and TeacherID does not contain any additional information beyond the information implied by the student/class and class/teacher relationships. Consequentially, the student/class and class/teacher relationships are independent of each other -- these relationships have no additional constraints. The following table is, then, in violation of 4NF:

```
{ StudentID, ClassID, TeacherID }
```

As an example of the anomalies that can occur, realize that it is not possible to add a new class taught by some teacher without adding at least one student who is enrolled in this class.

To achieve 4NF, represent each independent many-to-many relationship through its own cross-reference table.

#### Fifth Normal Form

A table is in Fifth Normal Form (5NF) if it is 4NF and its information content cannot be reconstructed from several tables containing fewer attributes.

Consider again the student/class/teacher example, but now assume that there is an additional relationship between students and teachers. The previous example table is now 4NF, since all the relationships it describes are interrelated. However, it is not 5NF, since it can be reconstructed from three cross-reference tables, each representing one of the three many-to-many relationships:

```
{ StudentID, ClassID }  
{ ClassID, TeacherID }  
{ TeacherID, StudentID }
```

To achieve 5NF, isolate interrelated many-to-many relationships, introducing the required number of new tables to represent all business domain constraints.

#### Normalization in context

In practice, many databases are de-normalized to greater or lesser degree. The reason most often stated has to do with performance -- a de-normalized database may require fewer joins and can, therefore, be faster for retrievals.

While this reasoning may be true, the usual caveats against premature optimization apply here as well as everywhere else. First, you should determine sufficiently that a performance problem exists and that the proposed de-normalization improves it before introducing a conceptually suboptimal design.

Furthermore, a de-normalized schema can be harder to update. The additional integrity checks that are necessary in this case may offset the performance gains for queries obtained through denormalization.

Finally, it should be noted that dealing with many-to-many relationships raises some issues that cannot be fully resolved through normalization (Chris Date's article, "Normalization is no Panacea," in Resources covers this topic).

#### History tables and event logging

Besides holding the data that is necessary to support the primary business purpose of the system under construction, the DB is also a possible location to record information that is useful primarily for internal technical purposes, such as administration and maintenance of the system itself.

#### History tables

In a production system, you may desire to preserve the history of changes to the data in the live

database. This can be achieved through the use of history (or backup) tables, and the appropriate INSERT, DELETE, and UPDATE triggers.

Each table in the DB should have a history table, mirroring the entire history of the primary table. If entries in the primary table are to be updated, the old contents of the record are first copied to the history table before the update is made. In the same way, deleted records in the primary table are copied to the history table before being deleted from the primary one. The history tables always have the name of the corresponding primary one, but with `_Hist` appended.

Entries to the history table are always appended at the end. The history table, therefore, grows strictly monotonically in time. It will become necessary to periodically spool ancient records to tape for archiving. Such records may, as a result, not be immediately available for recall.

The attributes of the history table should agree exactly with the attributes of the primary table. In addition, the history table records the date and type of the change to the primary table. The type is one of the following: Create, Update, or Delete.

Changes to the structure of the primary table affect the history table. When an attribute is added to the primary table, it is added to the history table as well. When an attribute is deleted from the primary table, the corresponding attribute is not deleted from the history table. Instead, this field is left blank (NULL) in all future records. Consequentially, the history table not only grows in length over time, but also in width.

Note that the choice to use such a history mechanism affects neither the overall DB layout, nor applications that access only the primary tables. During development, you can probably dispense with recording changes in this way and leave the creation of the history tables and the necessary triggers until installation time.

#### Event logging for fun and profit

A database can be used as an event logger. The notion of event is broad, ranging from common debugging and system specific runtime information, to events which are specific to the business domain. Possible candidates for events to be logged to the database include:

- Transactions making changes to persistent data
- Transactions crossing component boundaries
- Errors and exceptions
- Dispatching of messages to the user
- Events involving financial transactions
- State changes to business entities

An EventLog table to log such information contains at least these fields to record:

- Timestamp
- EventType (a type code)
- Details (a descriptive string)

Optionally, it may identify an owner or originator of the event. The owner concept can either identify a logged-in user or admin, but it may as well describe a part or module of the system itself. In applications dealing with financial transactions, additional (optional) fields identifying the from- and to-accounts can be useful.

#### System config tables

Finally, it is possible to use the database as centralized storage for configurational data. Usually this information is kept distributed in miscellaneous plain-text files, such as start-up scripts or property files. The database can provide a single, managed storage facility for such information.

Besides start-up parameters, which are usually supplied to the system at boot-time, one may also

think of properties that are required at runtime, such as localized strings and messages.

Lastly, the database is a possible place to keep system documentation. This is most useful, of course, for information that is naturally in tabular form (rather than free text), such as lists of assigned port numbers or shared memory keys, for instance. But this approach is not limited to codes. A data dictionary, defining the permissible values for each field, is a necessity on any non-trivial project. This also can be made accessible to all developers and administrators by storing it in the database.

In any case, the data is stored in simple key/value pairs. Additional table attributes can contain comments or pointers (URLs) to relevant offline documentation.

## More on Keys and Datatypes

What are the best choices when designing the schema for a relational database? What is the rationale in deciding in favor of one and against some other alternative? Given the amount of vendor-specific recommendations, it is all too easy to overlook basic relational database fundamentals. This section presents simple and complex datatypes, and primary and foreign keys -- the plumbing that holds the entire database together.

### Primary keys and related matters

A relational database (DB) stores two kinds of information -- data and plumbing. Data comprises the customer names, inventory numbers, item descriptions, and so on, that the application uses. Plumbing refers to the primary and foreign keys that the DB needs to find database records and relate them to one another.

### Basic plumbing

For the purpose of data modeling, the plumbing should be largely transparent. In fact, purist DB lore makes no distinction between data and plumbing. However, you will see that it is more efficient for administration and maintenance, as well as in terms of runtime performance, to have some additional fields to serve as DB keys.

Every table must have a primary key: an attribute or combination of attributes that are guaranteed to be unique and not-null. It is generally helpful to introduce a surrogate key -- a table attribute which has no business meaning, but simply serves as unique identifier for each record in the table. This is the plumbing that I have been referring to.

The requirements for a primary key are very strict. It must:

- Exist
- Be unique
- Not change over time

Surrogate keys help to mitigate the fact that real business data never reliably fulfills these requirements. Not every person has a Social Security Number (think of those outside the U.S.), people change their names, and other important information.

Business data might also simply be bad -- glitches in the Social Security Administration's system may lead to different persons having the same Social Security Number. A surrogate key helps to isolate the system from such problems.

The second reason that surrogate keys are favorable has to do with efficiency and ease of maintenance, since you can choose the most efficient datatype for the surrogate key. Furthermore, the surrogate key typically is a single field (not a compound key), which simplifies the schema (particularly when the key is used in other tables as a foreign key).

Every table should have a dedicated column to serve as this table's primary key. This column may be called `id` or `pk` (or possibly `<table_name>_id` or `<table_name>_pk`). Most databases are tuned for queries on integers, so it makes sense to use this datatype as primary key. Many databases, including Postgres and Oracle, also provide a special serial or sequence integer type, which generates a sequence of unique integers. Declaring a column to be of this type guarantees that a unique key is generated for each inserted row.

Foreign keys are table attributes, the values of which are the primary keys of another table. It often makes sense to label foreign key columns explicitly, for instance, by adopting a naming convention such as `<other_table_name>_fk`. A referential integrity constraint (references) should be declared as part of the `CREATE` statement when creating the table.

It bears repeating that the surrogate keys discussed earlier are part of the plumbing only -- their existence does not obviate the modeling requirement to be able to form a primary key from the

business data alone. Such a business data candidate key is a subset of all attributes, the values of which are never null, and each combination of values is unique. As a check on correct data modeling, such a candidate key must exist and should be documented for every table.

Strictly speaking, you may not always find a candidate key among the business data. Imagine a table recording the first and last name for each user, but having no further attributes. Now assume that there are two different persons, both of whom have the first name "Joe" and last name "Blow." In such a case, there exists no combination of table attributes that can form a suitable candidate key.

The underlying problem here is whether you are talking about the uniqueness of datasets or about the uniqueness of the underlying entities -- users, in this example. It is generally more intuitive, in particular to developers used to object-oriented analysis, to model the uniqueness of the underlying entities. Surrogate keys as discussed earlier can help to achieve this.

#### Alternate keys and visible identifiers

As part of the plumbing, the surrogate key has no need to ever be visible outside the DB. In particular, it should never be revealed to the user. This allows the DB administrator to change the representation of the keys at will if necessary. If a business need arises for providing the user with a unique identifier to a particular dataset, this identifier should be considered real business data and kept separate from the plumbing. For instance, an additional column called VisibleAccountNumber or the like can be introduced. Of course, this attribute should be non-null and unique so that it forms an alternative candidate key (an alternate key). Having a separate column for visible identifiers also makes it possible to generate and format the values for this attribute in a user-friendly way so that it is easy to read over the phone to a customer support person, for instance.

A borderline case is when the identifier is not directly visible, but may still be accessible to the user. Examples include hidden fields in Web pages in which an identifier is shuttled to the client to be used as a parameter in the following request. Although there is no need for the user to handle the identifier, a malicious user may read and attempt to spoof it. Using the numerical values of a primary key directly, in principle, allows any attacker to walk the entire table!

Defences against this problem include either encrypting and decrypting the value of the primary key, or protecting the key by appending a Message Authentication Code (MAC). An alternative is to use a hard-to-spoof visible identifier attribute for the table, such as the hash of the record's primary key or creation timestamp. (Of course, the uniqueness of this attribute must be assured.)

Whether the key is visible to the application (as opposed to the end user) depends on the specifics of the project. Using a numeric type directly carries the key's database representation straight into the application code and should be avoided to prevent coupling. In small-scale developments, a String representation of the key's value may be acceptable (all datatypes that can be stored in a DB must be able to be serialized).

But a better solution is a simple wrapper object that adds very little complexity, but provides strong decoupling of the database keys' representation from their interfaces. A danger exists in making the wrapper object too smart. The intention with surrogate keys is to make them simple and efficient for the database to handle. Settings from a database value and possibly from a String, comparing with another key object, and possibly serializing are all the methods that are required. Smarts, such as the ability to verify the contents based on a checksum calculation, suggest that this object probably belongs to the business data domain (like the visible record identifiers, introduced earlier).

#### The problem of the Universally Unique Identifier

A final consideration concerns the possible need for a Universally Unique Identifier (UUID). The short answer is that relational databases do not require UUIDs at all. In fact, the entire UUID concept is somewhat unrelated to relational database management. Relational database keys -- the plumbing -- need only be unique per table, which can be achieved by using an auto-incrementing datatype such as the serial type mentioned earlier.

UUIDs can have some technical difficulties. To ensure uniqueness, all UUIDs must be generated by a centralized service -- which leads to scalability problems and can become a single point of failure. (The scalability issue can be mitigated by a stratified approach in which a central master is used to give out seeds to several slaves, which in turn generate the final identifiers in batches, and so on.) To represent a UUID in a database, use either a string attribute or a compound key comprising several integer columns. Both approaches are significantly slower than operations based on keys made up of long integers. Compound keys also increase the complexity of the DB schema when used as foreign keys.

In the end, whether or not a record in a database needs to have a truly globally unique ID is dictated by the business rules, not the database architecture. Some records may already contain some form of UUID (merchandise items, for instance, typically possess a Universal Product Code as barcode). Some other records, possibly corresponding to principal business entities, may otherwise already contain a unique identifier as part of their business data (such as the combination of timestamp and account name for a ledger entry). If this is not the case, a UUID can be generated and stored alongside the business data for those records that require it. In any case, UUIDs should be considered part of the business data -- not of the plumbing.

Even if (and this is a big if) the object-relational mapping approach chosen requires every business object to have a persistent, unique ID, there is no need to base the internal workings of the underlying relational database engine on this fact.

In summary, I argue to keep business data apart from the database's internal plumbing. Building a relational database around UUIDs breaks this principle by using attributes, which, in the end, are really part of the business data, as internal infrastructure. (For a totally different point of view on this issue and a careful discussion of the problems involved in generating UUIDs in a scalable fashion, see Scott Ambler's paper, "Mapping objects to relational databases," in Resources.)