Chapter 8

Relational Database Design

In general, the goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. This is accomplished by designing schemas that are in an appropriate normal form. To determine whether a relation schema is in one of the desirable normal forms, we need information about the real-world enterprise that we are modeling with the database. Some of this information exists in a well-designed E-R diagram, but additional information about the enterprise may be needed as well.

Design Alternative: Larger Schemas

Now, let us explore features of this relational database design as well as some alternatives. Suppose that instead of having the schemas instructor and department, we have the schema:

inst_dept(ID, name, salary, dept_name, building, budget)

This represents the result of a natural join on the relations corresponding to instructor and department. This seems like a good idea because some queries can be expressed using fewer joins, until we think carefully about the facts about the university that led to our E-R design. Let us consider the instance of the inst_dept relation shown in Figure 8.2. Notice that we have to repeat the department information ("building" and "budget") once for each instructor in the department. For example, the information about the Comp. Sci. department (Taylor, 100000) is included in the tuples of instructors Katz, Srinivasan, and Brandt. It is important that all these tuples agree as to the budget amount since otherwise our database would be inconsistent. In our original design using instructor and department, we stored the amount of each budget exactly once. This suggests that using inst_dept is a bad idea since it stores the budget amounts redundantly and runs the risk that some user might update the budget amount in one tuple but not all, and thus create inconsistency. Even if we decided to live with the redundancy problem, there is still another problem with the inst_dept schema. Suppose we are creating a new department in the university. In the alternative design above, we cannot represent directly the information concerning a department (dept_name, building, budget) unless that department has at least one instructor at the university. This is because tuples in the inst_dept table require values for ID, name, and salary. This means that we cannot record information about the newly created department until the first instructor is hired for the new department. In the old design, the schema department can handle this, but under the revised design, we would have to create

a tuple with a null value for building and budget. In some cases null values are troublesome, as we saw in our study of SQL. However, if we decide that this is not a problem to us in this case, then we can proceed to use the revised design.

Design Alternative: Smaller Schemas

Suppose again that, somehow, we had started out with the schema inst_dept. How would we recognize that it requires repetition of information and should be split into the two schemas instructor and department? By observing the contents of actual relations on schema inst_dept, we could note the repetition of information resulting from having to list the building and budget once for each instructor associated with a department. However, this is an unreliable process. A real-world database has a large number of schemas and an even larger number of attributes. The number of tuples can be in the millions or higher. Discovering repetition would be costly. There is an even more fundamental problem with this approach. It does not allow us to determine whether the lack of repetition is just a "lucky" special case or whether it is a manifestation of a general rule. In our example, how would we know that in our university organization, each department (identified by its department name) must reside in a single building and must have a single budget amount? Is the fact that the budget amount for the Comp. Sci. department appears three times with the same budget amount just a coincidence? We cannot answer these questions without going back to the enterprise itself and understanding its rules. In particular, we would need to discover that the university requires that every department (identified by its department name) must have only one building and one budget value. In the case of inst_dept, our process of creating an E-R design successfully avoided the creation of this schema. However, this fortuitous situation does not always occur. Therefore, we need to allow the database designer to specify rules such as "each specific value for dept_name corresponds to at most one budget" even in cases where dept_name is not the primary key for the schema in question. In other words, we need to write a rule that says "if there were a schema (dept_name, budget), then dept_name is able to serve as the primary key." This rule is specified as a functional dependency

$$\text{dept\_name} \rightarrow \text{budget}$$

Given such a rule, we now have sufficient information to recognize the problem of the inst_dept schema. Because dept_name cannot be the primary key for inst_dept (because a department may need several tuples in the relation on schema inst_dept), the amount of a budget may have to be repeated. Observations such as these and the rules (functional dependencies in particular) that result from them allow the database designer to recognize situations where a schema ought to be split, or decomposed, into two or more schemas. It is not hard to see that the right way to decompose inst_dept is into schemas instructor and department as in the original design. Finding the right decomposition is much harder for schemas with a large number of attributes and several functional dependencies. To deal with this, we shall rely on a formal methodology that we develop later in this chapter.

Not all decompositions of schemas are helpful. Consider an extreme case where all we had were schemas consisting of one attribute. No interesting relationships of any kind could be expressed. Now consider a less extreme case where we choose to decompose the employee schema (Section 7.8):

employee(ID, name, street, city, salary)

into the following two schemas:

employee1(ID, name)
employee2(name, street, city, salary)

The flaw in this decomposition arises from the possibility that the enterprise has two employees with the same name. This is not unlikely in practice, as many cultures have certain highly popular names. Of course each person would have a unique employee-id, which is why ID can serve as the primary key. As an example, let us assume two employees, both named Kim, work at the university and have the following tuples in the relation on schema employee in the original design:

(57766, Kim, Main, Perryridge, 75000)
(98776, Kim, North, Hampton, 67000)

Figure 8.3 shows these tuples, the resulting tuples using the schemas resulting from the decomposition, and the result if we attempted to regenerate the original tuples using a natural join. As we see in the figure, the two original tuples appear in the result along with two new tuples that incorrectly mix data values pertaining to the two employees named Kim. Although we have more tuples, we actually have less information in the following sense. We can indicate that a certain street, city, and salary pertain to someone named Kim, but we are unable to distinguish which of the Kims. Thus, our decomposition is unable to represent certain important facts about the university employees. Clearly, we would like to avoid such decompositions. We shall refer to such decompositions as being lossy decompositions, and, conversely, to those that are not as lossless decompositions.

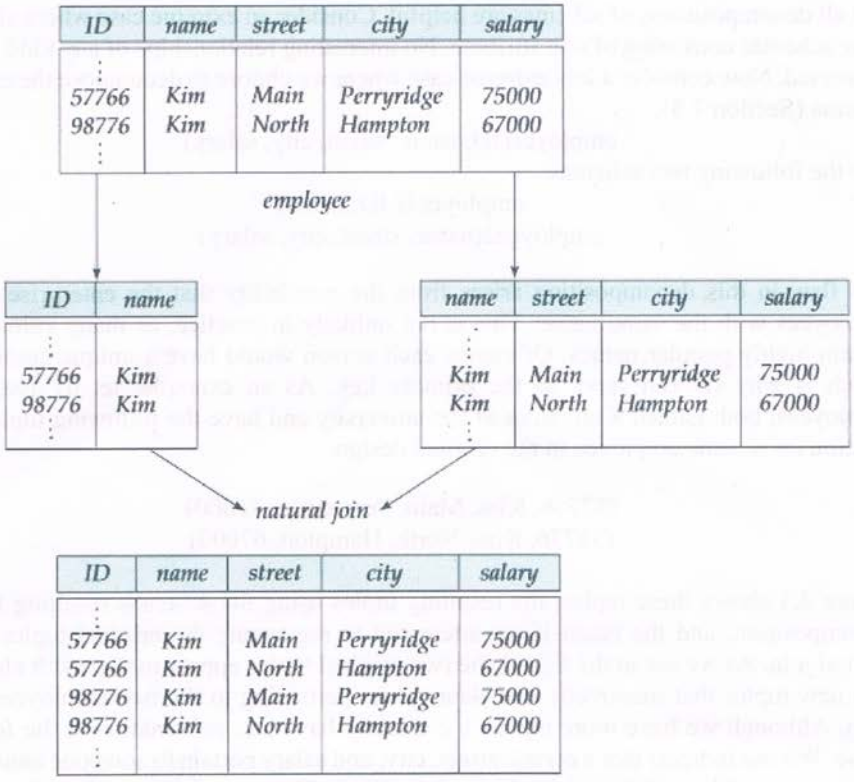

Figure 8.3 Loss of information via a bad decomposition.

Database Normalization

Database normalization is the process of efficiently organizing data in a database. There are two reasons of the normalization process:

- Eliminating redundant data, for example, storing the same data in more than one tables.
- Ensuring data dependencies make sense.

Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored. Normalization consists of a series of guidelines that help guide you in creating a good database structure.

Normalization guidelines are divided into normal forms; think of form as the format or the way a database structure is laid out. The aim of normal forms is to organize the database

structure so that it complies with the rules of first normal form, then second normal form, and finally third normal form.

It's your choice to take it further and go to fourth normal form, fifth normal form, and so on, but generally speaking, third normal form is enough.

- **First Normal Form (1NF)**
- **Second Normal Form (2NF)**
- **Third Normal Form (3NF)**

First Normal Form (1NF):

First normal form (1NF) sets the very basic rules for an organized database:

Define the data items required, because they become the columns in a table. Place related data items in a table.

Ensure that there are no repeating groups of data.

Ensure that there is a primary key.

First Rule of 1NF:
You must define the data items. This means looking at the data to be stored, organizing the data into columns, defining what type of data each column contains, and finally putting related columns into their own table.

For example, you put all the columns relating to locations of meetings in the Location table, those relating to members in the Member Details table, and so on.

Second Rule of 1NF:
The next step is ensuring that there are no repeating groups of data. Consider we have the following table:

```
CREATE TABLE CUSTOMERS(
ID  INT          NOT NULL,
NAME VARCHAR (20)    NOT NULL,
AGE  INT          NOT NULL,
ADDRESS  CHAR (25),
ORDERS   VARCHAR(155)
);
```

So if we populate this table for a single customer having multiple orders, then it would be something as follows:

ID	NAME	AGE	ADDRESS	ORDERS
100	Sachin36	Lower	West Side	Cannon XL-200
100	Sachin36	Lower	West Side	Battery XL-200
100	Sachin36	Lower	West Side	Tripod Large

But as per 1NF, we need to ensure that there are no repeating groups of data. So let us break above table into two parts and join them using a key as follows:

CUSTOMERS table:

```
CREATE TABLE CUSTOMERS(
ID  INT          NOT NULL,
NAME VARCHAR (20)    NOT NULL,
AGE  INT          NOT NULL,
ADDRESS  CHAR (25),
PRIMARY KEY (ID)
);
```

This table would have the following record:

ID	NAME	AGE	ADDRESS
100	Sachin36	Lower	West Side

ORDERS table:

```
CREATE TABLE ORDERS(
ID  INT          NOT NULL,
CUSTOMER_ID INT      NOT NULL,
ORDERS  VARCHAR(155),
PRIMARY KEY (ID)
);
```

This table would have the following records:

ID	CUSTOMER_ID	ORDERS
10	100	Cannon XL-200
11	100	Battery XL-200
12	100	Tripod Large

Third Rule of 1NF:

The final rule of the first normal form, create a primary key for each table which we have already created.

Second Normal Form (2NF):

Second normal form states that it should meet all the rules for 1NF and there must be no partial dependences of any of the columns on the primary key:

Consider a customer-order relation and you want to store customer ID, customer name, order ID and order detail, and date of purchase:

```
CREATE TABLE CUSTOMERS(
CUST_ID   INT          NOT NULL,
CUST_NAME VARCHAR (20)     NOT NULL,
ORDER_ID  INT          NOT NULL,
ORDER_DETAIL VARCHAR (20)  NOT NULL,
SALE_DATE  DATETIME,
PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

This table is in first normal form, in that it obeys all the rules of first normal form. In this table, the primary key consists of CUST_ID and ORDER_ID. Combined, they are unique assuming same customer would hardly order same thing.

However, the table is not in second normal form because there are partial dependencies of primary keys and columns. CUST_NAME is dependent on CUST_ID, and there's no real link between a customer's name and what he purchased. Order detail and purchase date are also dependent on ORDER_ID, but they are not dependent on CUST_ID, because there's no link between a CUST_ID and an ORDER_DETAIL or their SALE_DATE.

To make this table comply with second normal form, you need to separate the columns into three tables.

First, create a table to store the customer details as follows:

```
CREATE TABLE CUSTOMERS(
CUST_ID   INT          NOT NULL,
CUST_NAME VARCHAR (20)     NOT NULL,
PRIMARY KEY (CUST_ID)
);
```

Next, create a table to store details of each order:

```
CREATE TABLE ORDERS(
ORDER_ID  INT          NOT NULL,
ORDER_DETAIL VARCHAR (20)  NOT NULL,
PRIMARY KEY (ORDER_ID)
);
```

Finally, create a third table storing just CUST_ID and ORDER_ID to keep track of all the orders for a customer:

```
CREATE TABLE CUSTMERORDERS(
CUST_ID   INT          NOT NULL,
ORDER_ID  INT          NOT NULL,
SALE_DATE  DATETIME,
PRIMARY KEY (CUST_ID, ORDER_ID)
);
```

Third Normal Form (3NF):

A table is in third normal form when the following conditions are met:

It is in second normal form.

All non-primary fields are dependent on the primary key.

The dependency of non-primary fields is between the data. For example, in the below table, street name, city, and state are unbreakably bound to the zip code.

```
CREATE TABLE CUSTOMERS(
CUST_ID     INT          NOT NULL,
CUST_NAME   VARCHAR (20)     NOT NULL,
DOB         DATE,
STREET      VARCHAR(200),
CITY        VARCHAR(100),
STATE       VARCHAR(100),
ZIP         VARCHAR(12),
EMAIL_ID    VARCHAR(256),
PRIMARY KEY (CUST_ID)
);
```

The dependency between zip code and address is called a transitive dependency. To comply with third normal form, all you need to do is move the Street, City, and State fields into their own table, which you can call the Zip Code table:

```
CREATE TABLE ADDRESS(
ZIP         VARCHAR(12),
STREET      VARCHAR(200),
CITY        VARCHAR(100),
STATE       VARCHAR(100),
PRIMARY KEY (ZIP)
);
```

Next, alter the CUSTOMERS table as follows:

```
CREATE TABLE CUSTOMERS(
CUST_ID      INT            NOT NULL,
CUST_NAME    VARCHAR (20)     NOT NULL,
DOB          DATE,
ZIP          VARCHAR(12),
EMAIL_ID     VARCHAR(256),
PRIMARY KEY (CUST_ID)
);
```

The advantages of removing transitive dependencies are mainly twofold. First, the amount of data duplication is reduced and therefore your database becomes smaller.

The second advantage is data integrity. When duplicated data changes, there's a big risk of updating only some of the data, especially if it's spread out in a number of different places in the database. For example, if address and zip code data were stored in three or four different tables, then any changes in zip codes would need to ripple out to every record in those three or four tables.