1 Multiple Tables and the Relational Model

While the table is the basic unit in the relational database, a database typically contains a collection of tables. Up to this point in the chapter the focus has been on understanding the table. In this section, we broaden our view to examine information kept in multiple tables and how the relationships between these tables is modeled. To make this notion concrete, consider a simple example of a bank database based on an example found in Rolland [**?**]. This database contains four tables: a customer table, an account table, a branch table, and the registration table which links the customers to their accounts (see Figure 1).

The bank has two branches, and the branch table contains data specific to each branch, such as its name, location, and manager. Information on customers, i.e. name and address, is found in the customer table, and the account table contains account balances and the branch to which the account belongs. A customer may hold more than one account, and accounts may be jointly held by two or more customers. The registration table registers accounts with customers; it contains one tuple for each customer-account relation. Notice that customer #1 and customer #2 jointly hold account 201, and customer #2 holds an additional account, #202. Customer #3 holds 3 accounts, none of which are shared: #203 at the downtown branch of the bank and #301 and #302 at the suburban branch.

All of this data could have been included in one larger table (see Figure 2) rather than four separate tables. However Figure 2 contains a lot of redundancies: it has one tuple for each customer-account relation, and each tuple includes the address and manager of the branch to which the account belongs, as well as the customer's name and address. There may be times when all of this information is needed in this format, but typically space constraints and efficiency considerations make the multiple table database a better design choice.

Customers Table

CustNo	Name	Address
1	Smith, J	101 Elm
2	Smith, D	101 Elm
3	Brown, D	17 Spruce

Accounts Table

AcctNo	Balance	Branch
201	$12	Downtown
202	$1000	Downtown
203	$117	Downtown
301	$10	Suburb
302	$170	Suburb

Branches Table

Branch	Address	Manager
Downtown	101 Main St	Reed
Suburb	1800 Long Ave	Green

Registration Table

CID	AcctNo
1	201
2	201
2	202
3	203
3	301
3	302

Figure 1: The simple example of a bank database is inspired and adapted from Rolland. It contains four tables with information on customers, accounts, branches, and the customer-account relations.

CID	Name	Address	AcctNo	Balance	Branch	BAddr	Manager
1	Smith, J	101 Elm	201	$12	Downtown	101 Main St	Reed
2	Smith, D	101 Elm	201	$12	Downtown	101 Main St	Reed
2	Smith, D	101 Elm	202	$1000	Downtown	101 Main St	Reed
3	Brown, D	17 Spruce	203	$117	Downtown	101 Main St	Reed
3	Brown, D	17 Spruce	301	$10	Suburb	1800 Long Ave	Green
3	Brown, D	17 Spruce	302	$170	Suburb	1800 Long Ave	Green

Figure 2: All of the information in the four bank database table could be combined into one larger table with a lot of redundant information

The registration of accounts to customers is a very important aspect of this database design. Without it, the customers in the customer table could not be linked to the accounts in the account table. If we attempt to place this information in either the account or the customer table, then the redundancy will reappear, as more than one customer can share an account and a customer can hold more than one account.

Recall that a key to a table uniquely identifies the tuples in the table. The customer identification number is the key to the customer table, the account number is the key to the account table, and the customer-account relation has a composite key made up of both the account number and the customer number. These keys allow us to join the information in one table to that in another via the SELECT statement. We provide three examples.

Example For the first example, we find the total balance of all accounts held by a customer. To do this, we need to join the Account table, which contains balances, with the Registration table, which contains customer-account registrations. The following SELECT statement accomplishes this task. There are several things to notice about it. The two tables are listed in the FROM clause to denote that they are to be joined together. The WHERE clause specifies how these two tables are to be joined, namely matches are to be made on account number. The GROUP BY clause groups those accounts belonging to the same customer and the aggregate function

SUM reports the total balance of all accounts owned by the customer.

```
SELECT CID, SUM(Balance) AS Total
       FROM Registration, Accounts
       WHERE Accounts.AcctNo = Registration.AcctNo GROUP BY CID;
```

The results table will be as follows:

CID	Total
1	$12
2	$1012
3	$297

Since both the Registration and Accounts tables have an attribute called AcctNo, they need to be distinguished in the SELECT query. We do this by including the table name when we reference the attribute, e.g.

```
Accounts.AcctNo
```

refers to the AcctNo attribute in the Accounts table. Also note that the aggregate function is renamed as the attribute Total via the **AS** clause.

Example For the next example, the problem is to find the names and addresses of all customers with accounts in the downtown branch of the bank. To do this we need to select those accounts at the downtown branch, match them to their respective customers, and pick up the customer names and addresses. This information appears in three different tables, Accounts, Customers, and Registration, so we need to join these tables to subset and retrieve the data of interest. These three tables are listed in the FROM clause of the SELECT statement below. The WHERE clause joins customer tuples to account tuples according to the pairing of account number and customer number in the Registration table. It also limits the tuples to those accounts in the Downtown branch. The GROUP BY clause makes sure that a customer with more than one account in the branch of interest appears only once in the results table.

```
SELECT CustNo, Name, Address
    FROM Accounts A, Customers C, Registration R
    WHERE A.Branch = 'Downtown' AND A.AcctNo = R.AcctNo AND
     C.CustNo = R.CID GROUP BY CustNo;
```

A couple of comments on the syntax of this statement. Aliases for table names are provided in the FROM clause. The Registration table has been given the alias "R", Accounts has alias "A", and Customers can be referred to as "C". The alias gives us a shorthand name for a table. The A.Acctno refers to the AcctNo attribute in the A (Accounts) table and R.AcctNo refers to AcctNo in the Registration table. Since the customer number is labelled CID in the Registration table and CustNo in the Customers table, we do not need to include the table prefix in

```
R.CID = C.CustNo
```

. We do so for clarity. But we do not need this extra precaution for clarity sake when we list the attributes to be selected from the joined tables, **SELECT CustNo, Name, Address ...**

Example For the final example, consider the special case where a table is joined to itself in order to provide a list of customers sharing an account. That is, join the Registration table to itself, matching on account number and pulling out those tuples with the same account number but different customer numbers.

```
SELECT First.CustNo, Second.CustNo, First.AcctNo
   FROM Registration First, Registration Second
   WHERE First.AcctNo = Second.AcctNo
        AND First.CustNo < Second.CustNo;
```

Notice that the join does not join a tuple to itself because of the specification that the customer number in the First table must be less than the customer number in Second table.

The R language offers the **merge** function to merge two data frames by common columns or row names or do other versions of database join operations. However, database management systems are specially designed to handle these table operations, and if the data are in a database, for efficiency reasons, it usually makes sense to use the database facilities to subset, join, and group records in data tables.

1.1 Sub-queries

Intermediate tables can be created in a query by nesting one SELECT statement within another, which can useful for constructing complex searches and for optimizing a query.

Example Suppose we wish to find the name and address of those customers without accounts. We build the SELECT statement to accomplish this task by progressively nesting SELECTs. First, we produce a table of customer numbers in the Registration table,

```
SELECT CID FROM Registration;
```

Then we use this results table to find those customers in the Customers table that do not appear in this table,

```
SELECT * FROM Customer WHERE CustNo NOT IN
          (SELECT CID FROM Registration);
```

Notice that the SELECT statement used above to pull the disqualifying customer numbers is nested in the WHERE clause of the outer SELECT statement.

Subqueries can be further nested, as in the next example, where we re-visit an earlier example of joining multiple tables to produce a table of customers with accounts in the downtown branch. To start, first produce a table of account numbers for those accounts in the downtown branch:

```
SELECT AcctNo FROM Accounts WHERE Branch = 'Downtown';
```

With this list of accounts, we pull from the Registration table the customer numbers of the cusotmers who hold these accounts. The following nested SELECT statement does just that.

```
SELECT CID FROM Registration WHERE AcctNO IN
          (SELECT AcctNo FROM Accounts WHERE Branch = 'Downtown');
```

The final step requires acquisition of the names and addresses for these customers from the Customer table. A further nesting of SELECT statements accomplishes this goal.

```
SELECT CustNo, Name, Address
   FROM Customers C WHERE CustNo IN
      (SELECT CID FROM Registration WHERE AcctNO IN
        (SELECT AcctNo FROM Accounts WHERE Branch = 'Downtown'));
```

This query contains two nested SELECT statements which each create a temporary table. The decision as to whether to use these nested subqueries over the join of the three tables shown earlier depends on issues of efficiency and readability.

1.2 Virtual Tables and Temporary Tables

In addition to base tables in the database and the results table from a query to the database, we have views, virtual tables that can be used just as database tables. A view can be thought of as a named subquery expression that exists in the database for use where-ever one would use a database table. The view may be a projection or restriction of a single table, or the result of a more complex join of tables. Views can be used to remove attributes or tuples that a user is not allowed to see, or to provide a shorthand means to obtain a commonly used query. The CREATE VIEW statement defines a view via a select statement.

A similar type of table, is the temporary table. Temporary tables allow users to store intermediate results rather than having to submit the same query or subquery again and again. Unlike the viw, the temporary table is a real table in the database which is seen only by the user and which disappears at the end of the user's session. This is especially useful if the query is needed for many other queries and it is time consuming to complete it. The CREATE TEMPORARY TABLE command is a special case of the CREATE TABLE query discussed in Section **??**.