Solution Manual Database System Concepts 6th Edition

Introduction to Computer Science (Patuakhali Science and Technology University)

INSTRUCTOR'S MANUAL
TO ACCOMPANY

Database System Concepts

Sixth Edition

Abraham Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
Indian Institute of Technology, Bombay

Contents

Preface

This volume is an instructor's manual for the 6^{th} edition of *Database System Concepts* by Abraham Silberschatz, Henry F. Korth and S. Sudarshan. It contains answers to the exercises at the end of each chapter of the book. (Beginning with the 5^{th} edition, solutions for Practice Exercises have been made available on the Web; to avoid duplication, these are not included in the instructors manual.)

Before providing answers to the exercises for each chapter, we include a few remarks about the chapter. The nature of these remarks vary. They include explanations of the inclusion or omission of certain material, and remarks on how we teach the chapter in our own courses. The remarks also include suggestions on material to skip if time is at a premium, and tips on software and supplementary material that can be used for programming exercises.

The Web home page of the book, at http://www.db-book.com, contains a variety of useful information, including laboratory relation information such as sample data, lab exercises, and links to database software, online appendices describing the network data model, the hierarchical data model, and advanced relational database design, model course syllabi, and last but not least, up-to-date errata. We will periodically update the page with supplementary material that may be of use to teachers and students.

We would appreciate it if you would notify us of any errors or omissions in the book, as well as in the instructor's manual. Internet electronic mail should be addressed to db-book-authors@cs.yale.edu. Physical mail may be sent to Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT, 06520, USA.

Although we have tried to produce an instructor's manual which will aid all of the users of our book as much as possible, there can always be improvements. These could include improved answers, additional questions, sample test questions, programming projects, suggestions on alternative orders of presentation of the material, additional references, and so on. If you would like to suggest any such improvements to the book or the instructor's manual, we would be glad to hear from you. All contributions that we make use of will, of course, be properly credited to their contributor.

Several students at IIT Bombay contributed to the instructor manual for the 6^{th} edition, including Mahendra Chavan, Karthik Ramachandra, Bikmal Harikrishna, Ankush Jain, Manas Joglekar, Parakram Majumdar, Prashant Sachdeva, and Nisarg Shah. This manual is derived from the manuals for the earlier editions. John Corwin and Swathi Yadlapalli did the bulk of the work in preparing the instructors manual for the 5^{th} edition. The manual for the 4^{th} edition was prepared by Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia and Arvind Hulgeri. The manual for the 3^{th} edition was prepared by K. V. Raghavan with help from Prateek R. Kapadia. Sara Strandtman helped with the instructor manual for the 2^{nd} and 3^{rd} editions, while Greg Speegle and Dawn Bezviner helped us to prepare the instructor's manual for the 1^{th} edition.

A. S.
H. F. K.
S. S.

CHAPTER 1

Introduction

Chapter 1 provides a general overview of the nature and purpose of database systems. The most important concept in this chapter is that database systems allow data to be treated at a high level of abstraction. Thus, database systems differ significantly from the file systems and general purpose programming environments with which students are already familiar. Another important aspect of the chapter is to provide motivation for the use of database systems as opposed to application programs built on top of file systems. Thus, the chapter motivates what the student will be studying in the rest of the course.

The idea of abstraction in database systems deserves emphasis throughout, not just in discussion of Section 1.3. The overview of the structure of databases is, of necessity, rather brief, and is meant only to give the student a rough idea of some of the concepts. The student may not initially be able to fully appreciate the concepts described here, but should be able to do so by the end of the course.

The specifics of the E-R, relational, and object-oriented models are covered in later chapters. These models can be used in Chapter 1 to reinforce the concept of abstraction, with syntactic details deferred to later in the course.

If students have already had a course in operating systems, it is worthwhile to point out how the OS and DBMS are related. It is useful also to differentiate between concurrency as it is taught in operating systems courses (with an orientation towards files, processes, and physical resources) and database concurrency control (with an orientation towards granularity finer than the file level, recoverable transactions, and resources accessed associatively rather than physically). If students are familiar with a particular operating system, that OS's approach to concurrent file access may be used for illustration.

Exercises

1.7 List four applications you have used that most likely employed a database system to store persistent data.

Answer:

- Banking: For account information, transfer of funds, banking transactions.
- Universities: For student information, online assignment submissions, course registrations, and grades.
- Airlines: For reservation of tickets, and schedule information.
- Online news sites: For updating new, maintainence of archives.
- Online-trade: For product data, availability and pricing informations, order-tracking facilities, and generating recommendation lists.

1.8 List four significant differences between a file-processing system and a DBMS.

Answer: Some main differences between a database management system and a file-processing system are:

- Both systems contain a collection of data and a set of programs which access that data. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.
- A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, whereas data written by one program in a file-processing system may not be readable by another program.
- A database management system is designed to allow flexible access to data (i.e., queries), whereas a file-processing system is designed to allow pre-determined access to data (i.e., compiled programs).
- A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file.

1.9 Explain the concept of physical data independence, and its importance in database systems.

Answer: Physical data independence is the ability to modify the physical scheme without making it necessary to rewrite application programs. Such modifications include changing from unblocked to blocked record storage, or from sequential to random access files. Such a modification might be adding a field to a record; an application program's view hides this change from the program.

1.10 List five responsibilities of a database-management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.

Answer: A general purpose database-management system (DBMS) has five responsibilities:

a. interaction with the file manager.

b. integrity enforcement.

c. security enforcement.

d. backup and recovery.

e. concurrency control.

If these responsibilities were not met by a given DBMS (and the text points out that sometimes a responsibility is omitted by design, such as concurrency control on a single-user DBMS for a micro computer) the following problems can occur, respectively:

a. No DBMS can do without this, if there is no file manager interaction then nothing stored in the files can be retrieved.

b. Consistency constraints may not be satisfied, for example an instructor may belong to a non-existent department, two students may have the same ID, account balances could go below the minimum allowed, and so on.

c. Unauthorized users may access the database, or users authorized to access part of the database may be able to access parts of the database for which they lack authority. For example, a low-level user could get access to national defense secret codes, or employees could find out what their supervisors earn (which is presumably a secret).

d. Data could be lost permanently, rather than at least being available in a consistent state that existed prior to a failure.

e. Consistency constraints may be violated despite proper integrity enforcement in each transaction. For example, incorrect bank balances might be reflected due to simultaneous withdrawals and deposits on the same account, and so on.

1.11 List at least two reasons why database systems support data manipulation using a declarative query language such as SQL, instead of just providing a a library of C or C++ functions to carry out data manipulation.
Answer:

a. Declarative languages are easier for programmers to learn and use (and even more so for non-programmers).

b. The programmer does not have to worry about how to write queries to ensure that they will execute efficiently; the choice of an efficient execution technique is left to the database system. The declarative specification makes it easier for the database system to make a proper choice of execution technique.

1.12 Explain what problems are caused by the design of the table in Figure 1.4.
Answer:

- If a department has more than one instructor, the building name and budget get repeated multiple times. Updates to the building name and budget may get performed on some of the copies but not others, resulting in an inconsistent state where it is not clear what is the actual building name and budget of a department.
- A department needs to have at least one instructor in order for building and budget information to be included in the table. Nulls can be used when there is no instructor, but null values are rather difficult to handle.
- If all instructors in a department are deleted, the building and budget information are also lost. Ideally, we would like to have the department information in the database irrespective of whether the department has an associated instructor or not, without resorting to null values.

1.13 What are five main functions of a database administrator?
Answer:

- To backup data
- In some cases, to create the schema definition
- To define the storage structure and access methods
- To modify the schema and/or physical organization when necessary
- To grant authorization for data access
- To specify integrity constraints

1.14 Explain the difference between two-tier and three-tier architectures. Which is better suited for Web applications? Why?
Answer: In a two-tier application architecture, the application runs on the client machine, and directly communicates with the database system running on server. In contrast, in a three-tier architecture, application code running on the client's machine communicates with an application server at the server, and never directly communicates with the database. The three-tier archicture is better suited for Web applications.

1.15 Describe at least 3 tables that might be used to store information in a social-networking system such as Facebook.
Answer: Some possible tables are:

a. A *users* table containing users, with attributes such as account name, real name, age, gender, location, and other profile information.

b. A *content* table containing user provided content, such as text and images, associated with the user who uploaded the content.

c. A *friends* table recording for each user which other users are connected to that user. The kind of connection may also be recorded in this table.

d. A *permissions* table, recording which category of friends are allowed to view which content uploaded by a user. For example, a user may share some photos with family but not with all friends.

CHAPTER 2

Introduction to the Relational Model

This chapter presents the relational model and a brief introduction to the relational-algebra query language. The short introduction to relational algebra is sufficient for courses that focus on application development, without going into database internals. In particular, the chapters on SQL do not require any further knowledge of relational algebra. However, courses that cover internals, in particular query processing, require a more detailed coverage of relational algebra, which is provided in Chapter 6.

Exercises

2.9 Consider the bank database of Figure 2.15.

a. What are the appropriate primary keys?

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)

Figure 2.14 Relational database for Exercises 2.1, 2.7, and 2.12.

branch(*branch_name*, *branch_city*, *assets*)
customer (*customer_name*, *customer_street*, *customer_city*)
loan (*loan_number*, *branch_name*, *amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name*, *balance*)
depositor (*customer_name*, *account_number*)

Figure 2.15 Banking database for Exercises 2.8, 2.9, and 2.13.

b. Given your choice of primary keys, identify appropriate foreign keys.

Answer:

a. The primary keys of the various schema are underlined. Although in a real bank the customer name is unlikely to be a primary key, since two customers could have the same name, we use a simplified schema where we assume that names are unique. We allow customers to have more than one account, and more than one loan.

branch(*branch_name*, *branch_city*, *assets*)
customer (*customer_name*, *customer_street*, *customer _city*)
loan (*loan_number*, *branch_name*, *amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name*, *balance*)
depositor (*customer_name*, *account_number*)

b. The foreign keys are as follows
   i. For *loan*: *branch_name* referencing *branch*.
   ii. For *borrower*: Attribute *customer_name* referencing *customer* and *loan_number* referencing *loan*
   iii. For *account*: *branch_name* referencing *branch*.
   iv. For *depositor*: Attribute *customer_name* referencing *customer* and *account_number* referencing *account*

2.10 Consider the *advisor* relation shown in Figure 2.8, with *s_id* as the primary key of *advisor*. Suppose a student can have more than one advisor. Then, would *s_id* still be a primary key of the *advisor* relation? If not, what should the primary key of *advisor* be?
Answer: No, *s_id* would not be a primary key, since there may be two (or more) tuples for a single student, corresponding to two (or more) advisors. The primary key should then be *s_id, i_id*.

2.11 Describe the differences in meaning between the terms *relation* and *relation schema*.
Answer: A relation schema is a type definition, and a relation is an instance of that schema. For example, *student* (*ss#*, *name*) is a relation schema and

123-456-222	John
234-567-999	Mary

is a relation based on that schema.

2.12 Consider the relational database of Figure 2.14. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who work for "First Bank Corporation".

b. Find the names and cities of residence of all employees who work for "First Bank Corporation".

c. Find the names, street address, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

Answer:

a. $\Pi_{person\_name}(\sigma_{company\_name = \text{"First Bank Corporation"}}(works))$

b. $\Pi_{person\_name, city}(employee \bowtie (\sigma_{company\_name = \text{"First Bank Corporation"}}(works)))$

c. $\Pi_{person\_name, street, city}(\sigma_{(company\_name = \text{"First Bank Corporation"} \wedge salary > 10000)}(works \bowtie employee))$

2.13 Consider the bank database of Figure 2.15. Give an expression in the relational algebra for each of the following queries:

a. Find all loan numbers with a loan value greater than $10,000.

b. Find the names of all depositors who have an account with a value greater than $6,000.

c. Find the names of all depositors who have an account with a value greater than $6,000 at the "Uptown" branch.

Answer:

a. $\Pi_{loan\_number}(\sigma_{amount > 10000}(loan))$

b. $\Pi_{customer\_name}(\sigma_{balance > 6000}(depositor \bowtie account))$

c. $\Pi_{customer\_name}(\sigma_{balance > 6000 \wedge branch\_name = \text{"Uptown"}}(depositor \bowtie account))$

2.14 List two reasons why null values might be introduced into the database.

Answer: Nulls may be introduced into the database because the actual value is either unknown or does not exist. For example, an employee whose address has changed and whose new address is not yet known should be retained with a null address. If employee tuples have a composite attribute *dependents*, and a particular employee has no dependents, then that tuple's *dependents* attribute should be given a null value.

2.15 Discuss the relative merits of procedural and nonprocedural languages.

Answer: Nonprocedural languages greatly simplify the specification of queries (at least, the types of queries they are designed to handle). The free the user from having to worry about how the query is to be evaluated; not only does this reduce programming effort, but in fact in most situations the query optimizer can do a much better task of choosing the best way to evaluate a query than a programmer working by trial and error.

On the other hand, procedural languages are far more powerful in terms of what computations they can perform. Some tasks can either not be

done using nonprocedural languages, or are very hard to express using nonprocedural languages, or execute very inefficiently if specified in a nonprocedural manner.

CHAPTER 3

Introduction to SQL

Chapter 3 introduces the relational language SQL. Further details of the SQL language are provided in Chapters 4 and 5.

Although our discussion is based on SQL standards, no database system implements the standards exactly as specified, and there are a number of minor syntactic differences that need to be kept in mind. Although we point out some of these differences where required, the system manuals of the database system you use should be used as supplements.

Although it is possible to cover this chapter using only handwritten exercises, we strongly recommend providing access to an actual database system that supports SQL. A style of exercise we have used is to create a moderately large database and give students a list of queries in English to write and run using SQL. We publish the actual answers (that is the result relations they should get, not the SQL they must enter). By using a moderately large database, the probability that a "wrong" SQL query will just happen to return the "right" result relation can be made very small. This approach allows students to check their own answers for correctness immediately rather than wait for grading and thereby it speeds up the learning process. A few such example databases are available on the Web home page of this book, http://db-book.com.

Exercises that pertain to database design are best deferred until after Chapter 8.

Exercises

3.11 Write the following queries in SQL, using the university schema.

a. Find the names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

Answer:

a. SQL query:

```
select  name
from    student natural join takes natural join course
where   course.dept = 'Comp. Sci.'
```

b. SQL query:

```
select  id, name
from    student
except
select  id, name
from    student natural join takes
where   year < 2009
```

Since the **except** operator eliminates duplicates, there is no need to use a **select distinct** clause, although doing so would not affect correctness of the query.

c. SQL query:

```
select  dept, max(salary)
from    instructor
group by dept
```

d. SQL query:

```
select  min(maxsalary)
from    (select dept, max(salary) as maxsalary
         from instructor
         group by dept)
```

3.12 Write the following queries in SQL, using the university schema.

a. Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.

b. Create a section of this course in Autumn 2009, with *section_id* of 1.

c. Enroll every student in the Comp. Sci. department in the above section.

d. Delete enrollments in the above section where the student's name is Chavez.

e. Delete the course CS-001. What will happen if you run this delete statement without first deleting offerings (sections) of this course.

f. Delete all *takes* tuples corresponding to any section of any course with the word "database" as a part of the title; ignore case when matching the word with the title.

Answer:

a. SQL query:

```
insert into course
    values ('CS-001', 'Weekly Seminar', 'Comp. Sci.', 0)
```

b. SQL query:

```
insert into section
    values ('CS-001', 1, 'Autumn', 2009, null, null, null)
```

Note that the building, roomnumber and slot were not specified in the question, and we have set them to null. The same effect would be obtained if they were specified to default to null, and we simply omitted values for these attributes in the above insert statement. (Many database systems implicitly set the default value to null, even if not explicitly specified.)

c. SQL query:

```
insert into takes
    select id, 'CS-001', 1, 'Autumn', 2009, null
    from student
    where dept_name = 'Comp. Sci.'
```

d. SQL query:

```
delete from takes
where course_id= 'CS-001' and section_id = 1 and
      year = 2009 and semester = 'Autumn' and
      id in (select id
             from student
             where name = 'Chavez')
```

Note that if there is more than one student named Chavez, all such students would have their enrollments deleted. If we had used = instead of **in**, an error would have resulted if there were more than one student named Chavez.

e. SQL query:

```
delete from takes
where course_id = 'CS-001'

delete from section
where course_id = 'CS-001'

delete from course
where course_id = 'CS-001'
```

If we try to delete the course directly, there will be a foreign key violation because *section* has a foriegn key reference to *course*; similarly, we have to delete corresponding tuples from *takes* before deleting sections, since there is a foreign key reference from *takes* to *section*. As a result of the foreign key violation, the transaction that performs the delete would be rolled back.

f. SQL query:

```
delete from takes
where course_id in
      (select course_id
      from course
      where lower(title) like '%database%')
```

3.13 Write SQL DDL corresponding to the schema in Figure 3.18. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

Answer:

a. SQL query:

person (*driver_id*, *name*, *address*)
car (*license*, *model*, *year*)
accident (*report_number*, *date*, *location*)
owns (*driver_id*, *license*)
participated (*report_number*, *license*, *driver_id*, *damage_amount*)

Figure 3.18 Insurance database for Exercises 3.4 and 3.14.

```
create table person
        (driver_id varchar(50),
        name      varchar(50),
        address   varchar(50),
        primary key (driver_id))
```

b. SQL query:

```
create table car
        (license varchar(50),
        model    varchar(50),
        year     integer,
        primary key (license))
```

c. SQL query:

```
create table accident
        (report_number integer,
        date           date,
        location       varchar(50),
        primary key (report_number))
```

d. SQL query:

```
create table owns
        (driver_id varchar(50),
        license    varchar(50),
        primary key (driver_id,license)
        foriegn key (driver_id) references person
        foriegn key (license) references car)
```

e. SQL query:

```
create table participated
        (report_number integer,
        license         varchar(50),
        driver_id       varchar(50),
        damage_amountinteger,
        primary key (report_number,license)
        foriegn key (license) references car
        foriegn key (report_number) references accident))
```

3.14 Consider the insurance database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the number of accidents in which the cars belonging to "John Smith" were involved.

b. Update the damage amount for the car with license number "AABB2000" in the accident with report number "AR2197" to $3000.

Answer: Note: The *participated* relation relates drivers, cars, and accidents.

a. SQL query:

```
select    count (*)
from      accident
where     exists
          (select *
           from participated, owns, person
           where owns.driver_id = person.driver_id
                 and person.name = 'John Smith'
                 and owns.license = participated.license
                 and accident.report_number = participated.report_number)
```

The query can be written in other ways too; for example without a subquery, by using a join and selecting **count**(**distinct** *report_number*) to get a count of number of accidents involving the car.

b. SQL query:

```
update participated
set damage_amount = 3000
where report_number = "AR2197" and
      license = "AABB2000")
```

3.15 Consider the bank database of Figure 3.19, where the primary keys are underlined. Construct the following SQL queries for this relational database.

branch(*branch_name*, *branch_city*, *assets*)
customer (*customer_name*, *customer_street*, *customer_city*)
loan (*loan_number*, *branch_name*, *amount*)
borrower (*customer_name*, *loan_number*)
account (*account_number*, *branch_name*, *balance*)
depositor (*customer_name*, *account_number*)

Figure 3.19 Banking database for Exercises 3.8 and 3.15.

a. Find all customers who have an account at *all* the branches located in "Brooklyn".

b. Find out the total sum of all loan amounts in the bank.

c. Find the names of all branches that have assets greater than those of at least one branch located in "Brooklyn".

Answer:

a. SQL query:

```
with branchcount as
        (select count(*)
        branch
        where branch_city = 'Brooklyn')
select customer_name
from customer c
where branchcount =
        (select count(distinct branch_name)
        from (customer natural join depositor natural join account
                natural join branch) as d
        where d.customer_name = c.customer_name)
```

There are other ways of writing this query, for example by first finding customers who do not have an account at some branch in Brooklyn, and then removing these customers from the set of all customers by using an **except** clause.

b. SQL query:

```
select sum(amount)
from loan
```

c. SQL query:

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 3.20 Employee database for Exercises 3.9, 3.10, 3.16, 3.17, and 3.20.

```
select branch_name
from branch
where assets > some
      (select assets
      from branch
      where branch_city = 'Brooklyn')
```

The keyword **any** could be used in place of **some** above.

3.16 Consider the employee database of Figure 3.20, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names of all employees who work for First Bank Corporation.

b. Find all employees in the database who live in the same cities as the companies for which they work.

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

d. Find all employees who earn more than the average salary of all employees of their company.

e. Find the company that has the smallest payroll.

Answer:

a. Find the names of all employees who work for First Bank Corporation.

```
select employee_name
from works
where company_name = 'First Bank Corporation'
```

b. Find all employees in the database who live in the same cities as the companies for which they work.

```
select e.employee_name
from employee e, works w, company c
where e.employee_name = w.employee_name and e.city = c.city and
      w.company_name = c.company _name
```

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

```
select P.employee_name
from employee P, employee R, manages M
where P.employee_name = M.employee_name and
      M.manager_name = R.employee_name and
      P.street = R.street and P.city = R.city
```

d. Find all employees who earn more than the average salary of all employees of their company.

```
select employee_name
from works T
where salary > (select avg (salary)
                from works S
                where T.company_name = S.company_name)
```

The primary key constraint on *works* ensures that each person works for at most one company.

e. Find the company that has the smallest payroll.

```
select company_name
from works
group by company_name
having sum (salary) <= all (select sum (salary)
                            from works
                            group by company_name)
```

3.17 Consider the relational database of Figure 3.20. Give an expression in SQL for each of the following queries.

a. Give all employees of First Bank Corporation a 10 percent raise.

b. Give all managers of First Bank Corporation a 10 percent raise.

c. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

Answer:

a. Give all employees of First Bank Corporation a 10-percent raise. (the solution assumes that each person works for at most one company.)

```
update works
set salary = salary * 1.1
where company_name = 'First Bank Corporation'
```

b. Give all managers of First Bank Corporation a 10-percent raise.

```
update works
set salary = salary * 1.1
where employee_name in (select manager_name
                        from manages)
      and company_name = 'First Bank Corporation'
```

c. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

```
delete from works
where company_name = 'Small Bank Corporation'
```

3.18 List two reasons why null values might be introduced into the database.

Answer:

- "null" signifies an unknown value.
- "null" is also used when a value does not exist.

3.19 Show that, in SQL, <> **all** is identical to **not in**.

Answer: Let the set S denote the result of an SQL subquery. We compare (x <> **all** S) with (x **not in** S). If a particular value x_1 satisfies (x_1 <> **all** S) then for all elements y of S $x_1 \neq y$. Thus x_1 is not a member of S and must satisfy (x_1 **not in** S). Similarly, suppose there is a particular value x_2 which satisfies (x_2 **not in** S). It cannot be equal to any element w belonging to S, and hence (x_2 <> **all** S) will be satisfied. Therefore the two expressions are equivalent.

3.20 Give an SQL schema definition for the employee database of Figure 3.20. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

Answer:

```
create table    employee
(employee_name  varchar(20),
 street         char(30),
 city           varchar(20),
 primary key    (employee_name))

create table    works
(employee_name  person_names,
 company_name   varchar(20),
 salary         numeric(8, 2),
 primary key    (employee_name))

create table    company
(company_name   varchar(20),
 city           varchar(20),
 primary key    (company_name))
```

member(*memb_no*, *name*, *age*)
book(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

Figure 3.21 Library database for Exercise 3.21.

```
create table    manages
(employee_name  varchar(20),
 manager_name   varchar(20),
 primary key    (employee_name))
```

3.21 Consider the library database of Figure 3.21. Write the following queries in SQL.

a. Print the names of members who have borrowed any book published by "McGraw-Hill".

b. Print the names of members who have borrowed all books published by "McGraw-Hill".

c. For each publisher, print the names of members who have borrowed more than five books of that publisher.

d. Print the average number of books borrowed per member. Take into account that if an member does not borrow any books, then that member does not appear in the *borrowed* relation at all.

Answer:

a. Print the names of members who have borrowed any book published by McGraw-Hill.

```
select name
from member m, book b, borrowed l
where m.memb_no = l.memb_no
      and l.isbn = b.isbn and
          b.publisher = 'McGrawHill'
```

b. Print the names of members who have borrowed all books published by McGraw-Hill. (We assume that all books above refers to all books in the *book* relation.)

```
select distinct m.name
from member m
where not exists
    ((select isbn
    from book
    where publisher = 'McGrawHill')
    except
    (select isbn
    from borrowed l
    where l.memb_no = m.memb_no))
```

c. For each publisher, print the names of members who have borrowed more than five books of that publisher.

```
select publisher, name
from (select publisher, name, count (isbn)
    from member m, book b, borrowed l
    where m.memb_no = l.memb_no
    and l.isbn = b.isbn
    group by publisher, name) as
    membpub(publisher, name, count_books)
where count_books > 5
```

The above query could alternatively be written using the **having** clause.

d. Print the average number of books borrowed per member.

```
with memcount as
    (select count(*)
    from member)
select count(*)/memcount
from borrowed
```

Note that the above query ensures that members who have not borrowed any books are also counted. If we instead used **count(distinct** *memb_no*) from *borrowed*, we would not account for such members.

3.22 Rewrite the **where** clause

```
where unique (select title from course)
```

without using the **unique** construct.

Answer:

```
where(
    (select count(title)
    from course) =
    (select count (distinct title)
    from course))
```

3.23 Consider the query:

```
select course_id, semester, year, section_id, avg (credits_earned)
from takes natural join student
where year = 2009
group by course_id, semester, year, section_id
having count (ID) >= 2;
```

Explain why joining *section* as well in the **from** clause would not change the result.

Answer: The common attributes of *takes* and *section* form a foreign key of *takes*, referencing *section*. As a result, each *takes* tuple would match at most one one *section* tuple, and there would not be any extra tuples in any group. Further, these attributes cannot take on the null value, since they are part of the primary key of *takes*. Thus, joining *section* in the **from** clause would not cause any loss of tuples in any group. As a result, there would be no change in the result.

3.24 Consider the query:

```
with dept_total (dept_name, value) as
        (select dept_name, sum(salary)
        from instructor
        group by dept_name),
dept_total_avg(value) as
        (select avg(value)
        from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Rewrite this query without using the **with** construct.

Answer:

There are several ways to write this query. One way is to use subqueries in the where clause, with one of the subqueries having a second level subquery in the from clause as below.

```
select distinct dept_name d
from instructor i
where
      (select sum(salary)
      from instructor
      where department = d)
      >=
      (select avg(s)
      from
            (select sum(salary) as s
            from instructor
            group by department))
```

Note that the original query did not use the *department* relation, and any department with no instructors would not appear in the query result. If we had written the above query using *department* in the outer **from** clause, a department without any instructors could appear in the result if the condition were <= instead of >=, which would not be possible in the original query.

As an alternative, the two subqueries in the where clause could be moved into the from clause, and a join condition (using >=) added.

CHAPTER 4

Intermediate SQL

In this chapter, we continue our study of SQL. We consider more complex forms of SQL queries, view definition, transactions, integrity constraints, more details regarding SQL data definition, and authorization.

As in Chapter 3 exercises, students should be encouraged to execute their queries on the sample database provided on http://db-book.com, to check if they generate the expected answers. Students could even be encouraged to create sample databases that can expose errors in queries, for example where an inner join operation is used erroneously in place of an outerjoin operation.

Exercises

4.12 For the database of Figure 4.11, write a query to find those employees with no manager. Note that an employee may simply have no manager listed or may have a *null* manager. Write your query using an outer join and then write it again using no outer join at all.

Answer:

a.

select *employee_name*
from *employee* **natural left outer join** *manages*
where *manager_name* **is null**

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 4.11 Employee database for Figure 4.7 and 4.12.

b.

```
select employee_name
from employee e
where not exists
    (select employee_name
    from manages m
    where e.employee_name = m.employee_name and
            m.manager_name is not null)
```

4.13 Under what circumstances would the query

```
select *
from student natural full outer join takes
                    natural full outer join course
```

include tuples with null values for the *title* attribute?

Answer: We first rewrite the expression with parentheses to make clear the order of the left outer join operations (the SQL standard specifies that the join operations are left associative).

```
select *
from (student natural full outer join
    takes) natural full outer join course
```

Given the above query, there are 2 cases for which the *title* attribute is null

a. Since *course_id* is a foreign key in the *takes* table referencing the *course* table, the title attribute in any tuple obtained from the above query can be null if there is a course in *course* table that has a null title.

b. If a student has not taken any course, as it is a **natural full outer join**, such a student's entry would appear in the result with a **null** *title* entry.

4.14 Show how to define a view *tot_credits* (*year, num_credits*), giving the total number of credits taken by students in each year.

Answer:

```
create view tot_credits(year, tot_credits)
as
    (select year, sum(credits)
    from takes natural join course
    group by year)
```

Note that this solution assumes that there is no year where students didn't take any course, even though sections were offerred.

salaried_worker (*name*, *office*, *phone*, *salary*)
hourly_worker (*name*, *hourly_wage*)
address (*name*, *street*, *city*)

Figure 4.12 Employee database for Exercise 4.16.

4.15 Show how to express the **coalesce** operation from Exercise 4.10 using the **case** operation.

Answer:

select
  case *Result*
    when (A_1 is not null) **then** A_1
    when (A_2 in not null) **then** A_2
    .
    .
    .
    when (A_n is not nulll) **then** A_n
    else null
  end
from *A*

4.16 Referential-integrity constraints as defined in this chapter involve exactly two relations. Consider a database that includes the relations shown in Figure 4.12. Suppose that we wish to require that every name that appears in *address* appears in either *salaried_worker* or *hourly_worker*, but not necessarily in both.

a. Propose a syntax for expressing such constraints.

b. Discuss the actions that the system must take to enforce a constraint of this form.

Answer:

a. For simplicity, we present a variant of the SQL syntax. As part of the **create table** expression for *address* we include

foreign key (*name*) **references** *salaried_worker* **or** *hourly_worker*

b. To enforce this constraint, whenever a tuple is inserted into the *address* relation, a lookup on the *name* value must be made on the *salaried_worker* relation and (if that lookup failed) on the *hourly_worker* relation (or vice-versa).

4.17 Explain why, when a manager, say Satoshi, grants an authorization, the grant should be done by the manager role, rather than by the user Satoshi.

Answer: Consider the case where the authorization is provided by the user Satoshi and not the manager role. If we revoke the authorization from Satoshi, for example because Satoshi left the company, all authorizations that Satoshi had granted would also be revoked, even if the grant was to an employee whose job has not changed.
If the grant is done by the manager role, revoking authorizations from Satoshi will not result in such cascading revocation.
In terms of the authorization graph, we can treat Satoshi and the role manager as nodes. When the grant is from the manager role, revoking the manager role from Satoshi has no effect on the grants from the manager role.

4.18 Suppose user A, who has all authorizations on a relation r, grants select on relation r to **public** with grant option. Suppose user B then grants select on r to A. Does this cause a cycle in the authorization graph? Explain why.
Answer: Yes, it does cause a cycle in the authorization graph. The grant to public results in an edge from A to public. The grant to the **public** operator provides authorization to everyone, B is now authorized. For each privilege granted to *public*, an edge must therefore be placed between *public* and all users in the system. If this is not done, then the user will not have a path from the root (DBA). And given the with grant option, B can grant select on r to A resultin in an edge from B to A in the authorization graph. Thus, there is now a cycle from A to public, from public to B, and from B back to A.

4.19 Database systems that store each relation in a separate operating-system file may use the operating system's authorization scheme, instead of defining a special scheme themselves. Discuss an advantage and a disadvantage of such an approach.
Answer:

- Advantages:
  - Operations on the database are speeded up as the authorization procedure is caried out at the OS level.
  - No need to implement the security and authorization inside the DBMS. This may result in reduced cost.
  - Administrators need not learn new commands or use of a new UI. They can create and administer user accounts on the OS they may already be familiar with.
  - No worry of unauthorized database users having direct access to the OS files and thus bypassing the database security.
- Disadvantages:
  - Database users must correspond to operating system users.
  - Fine control on authorizations is limited by what the operating system provides and is dependent on it, For example, most operating systems

do not distinguish between insert, update and delete, they just have a coarse level privilege called "modify". Privileges such as "references" cannot be provided.

- Columnwise control is not possible. You cannot differentiate update/delete and insert authorizations.
- Cannot store more than one relation in a file.
- The with grant option is limited to what the OS provides (if any: most OS's don't provide such options) and cannot be controlled by the user or administrator.

CHAPTER 5

Advanced SQL

In this chapter we address the issue of how to access SQL from a general-purpose programming language, which is very important for building applications that use a database to store and retrieve data. We describe how procedural code can be executed within the database, either by extending the SQL language to support procedural actions, or by allowing functions defined in procedural languages to be executed within the database. We describe triggers, which can be used to specify actions that are to be carried out automatically on certain events such as insertion, deletion, or update of tuples in a specified relation. We discuss recursive queries and advanced aggregation features supported by SQL. Finally, we describe online analytic processing (OLAP) systems, which support interactive analysis of very large datasets.

Given the fact that the JDBC and ODBC protocols (and variants such as ADO.NET) are have become the primary means of accessing databases, we have significantly extended our coverage of these two protocols, including some examples. However, our coverage is only introductory, and omits many details that are useful in practise. Online tutorials/manuals or textbooks covering these protocols should be used as supplements, to help students make full use of the protocols.

Exercises

5.12 Consider the following relations for a company database:

- *emp* (*ename*, *dname*, *salary*)
- *mgr* (*ename*, *mname*)

and the Java code in Figure 5.26, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like "It finds the manager of the toy department," not a line-by-line description of what each Java statement does.)

```
import java.sql.*;
public class Mystery {
   public static void main(String[] args) {
        try {
            Connection con=null;
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con=DriverManager.getConnection(
                "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
            Statement s=con.createStatement();
            String q;
            String empName = "dog";
            boolean more;
            ResultSet result;
            do {
                q = "select mname from mgr where ename = '" + empName + "'";
                result = s.executeQuery(q);
                more = result.next();
                if (more) {
                    empName = result.getString("mname");
                    System.out.println (empName);
                }
            } while (more);
            s.close();
            con.close();
        } catch(Exception e){e.printStackTrace();} }}
```

Figure 5.26 Java code for Exercise 5.12.

Answer: It prints out the manager of "dog." that manager's manager, etc. until we reach a manager who has no manager (presumably, the CEO, who most certainly is a cat.) NOTE: if you try to run this, use your OWN Oracle ID and password, since Star, crafty cat that she is, changes her password.

5.13 Suppose you were asked to define a class MetaDisplay in Java, containing a method static void printTable(String r); the method takes a relation name *r* as input, executes the query "**select** * **from** *r*", and prints the result out in nice tabular format, with the attribute names displayed in the header of the table.

a. What do you need to know about relation *r* to be able to print the result in the specified tabular format.

b. What JDBC methods(s) can get you the required information?

c. Write the method printTable(String r) using the JDBC API.

Answer:

a. We need to know the number of attributes and names of attributes of *r* to decide the number and names of columns in the table.

b. We can use the JDBC methods getColumnCount() and getColumnName(int) to get the required information.

c. The method is shown below.

```
static void printTable(String r)
{
    try
    {
        Class.forName("oraclejdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@db.yale.edu:2000:univdb",user,passwd);
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.ExecuteQuery(r);
        ResultSetMetaData rsmd = rs.getMetaData();
        int count = rsmd.getColumnCount();
        System.out.println("<tr>");
        for(int i=1;i<=count;i++){
            System.out.println("<td>"+rsmd.getColumnName(i)+"</td>");
        }
        System.out.println("</tr>");
        while(rs.next(){
            System.out.println("<tr>");
            for(int i=1;i<=count;i++){
                System.out.println("<td>"+rs.getString(i)+"</td>");
            }
            System.out.println("</tr>");
        }
        stmt.close();
        conn.close();
    }
    catch(SQLException sqle)
    {
        System.out.println("SQLException : " + sqle);
    }
}
```

5.14 Repeat Exercise 5.13 using ODBC, defining void printTable(char *r) as a function instead of a method.

Answer:

a. Same as for JDBC.

b. The function SQLNumResultCols(hstmt, &numColumn) can be used to find the number of columns in a statement, while the function SQLColAttribute() can be used to find the name, type and other information about any column of a result set. set, and the names

c. The ODBC code is similar to the JDBC code, but significantly longer. ODBC code that carries out this task may be found online at the URL http://msdn.microsoft.com/en-us/library/ms713558.aspx (look at the bottom of the page).

5.15 Consider an employee database with two relations

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)

where the primary keys are underlined. Write a query to find companies whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

a. Using SQL functions as appropriate.

b. Without using SQL functions.

Answer:

a.

```
create function avg_salary(cname varchar(15))
    returns integer
    declare result integer;
        select avg(salary) into result
        from works
        where works.company_name = cname
    return result;
end
select company_name
from works
where avg_salary(company_name) > avg_salary("First Bank Corporation")
```

b.

```
select company_name
from works
group by company_name
having avg(salary) > (select avg(salary)
                      from works
                      where company_name="First Bank Corporation")
```

5.16 Rewrite the query in Section 5.2.1 that returns the name and budget of all departments with more than 12 instructors, using the **with** clause instead of using a function call.

Answer:

```
with instr_count (dept_name, number) as
        (select dept_name, count (ID)
                from instructor
                group by dept_name)
select dept_name, budget
        from department, instr_count
        where department.dept_name = instr_count.dept_name
        and number > 12
```

5.17 Compare the use of embedded SQL with the use in SQL of functions defined in a general-purpose programming language. Under what circumstances would you use each of these features?

Answer: SQL functions are primarily a mechanism for extending the power of SQL to handle attributes of complex data types (like images), or to perform complex and non-standard operations. Embedded SQL is useful when imperative actions like displaying results and interacting with the user are needed. These cannot be done conveniently in an SQL only environment. Embedded SQL can be used instead of SQL functions by retrieving data and then performing the function's operations on the SQL result. However a drawback is that a lot of query-evaluation functionality may end up getting repeated in the host language code.

5.18 Modify the recursive query in Figure 5.15 to define a relation

$$prereq\_depth(course\_id, prereq\_id, depth)$$

where the attribute *depth* indicates how many levels of intermediate prerequisites are there between the course and the prerequisite. Direct prerequisites have a depth of 0.

Answer:

```
with recursive prereq_depth(course_id, prereq_id, depth) as
        (select course_id, prereq_id, 0
                from prereq
        union
        select prereq.course_id, prereq_depth.prereq_id, (prereq_depth.depth + 1)
                from prereq, prereq_depth
                where prereq.prereq_id= prereq_depth.course_id)

select *
from prereq_depth
```

5.19 Consider the relational schema

part(*part_id*, *name*, *cost*)
subpart(*part_id*, *subpart_id*, *count*)

A tuple (p_1, p_2, 3) in the *subpart* relation denotes that the part with part-id p_2 is a direct subpart of the part with part-id p_1, and p_1 has 3 copies of p_2. Note that p_2 may itself have further subparts. Write a recursive SQL query that outputs the names of all subparts of the part with part-id "P-100".

Answer:

```
with recursive total_part(name) as
    (select part.name
        from subpart, part
        where subpart.part_id = "P-100" and
        subpart.part_id = part.part_id
    union
    select p2.name
        from subpart s, part p1, part p2
        where s.part_id = p1.part_id
        and p1.name total_part.name
        and s.subpart_id = p2.part_id)

    select *
        from total_part
```

5.20 Consider again the relational schema from Exercise 5.19. Write a JDBC function using non-recursive SQL to find the total cost of part "P-100", including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.

Answer: The SQL function 'total_cost' is called from within the JDBC code.

SQL function:

```
create function total_cost(id char(10))

returns table(number integer)

begin
  create temporary table result (name char(10), number integer);
  create temporary table newpart (name char(10), number integer);
  create temporary table temp (name char(10), number integer);
  create temporary table final_cost(number integer);

  insert into newpart
    select subpart_id, count
```

```
      from subpart
      where part_id = id
   repeat
      insert into result
      select name, number
      from newpart;

      insert into temp
      (select subpart.subpart_id, count
         from newpart, subpart
         where newpart.subpart_id = subpart.part_id;
      )
      except(
         select subpart_id, count
            from result;
         );

      delete from newpart;
      insert into newpart
         select *
         from temp;
      delete from temp;

   until not exists(select * from newpart)
   end repeat;

   with part_cost(number) as
      select (count*cost)
         from result, part
         where result.subpart_id = part.part_id);
   insert into final_cost
      select *
      from part_cost;
   return table final_cost;
end
```

JDBC function:

```
Connection conn = DriverManager.getConnection(
   ”jdbc:oracle:thin:@db.yale.edu:2000:bankdb”,
   userid,passwd);
Statement stmt = conn.createStatement();

ResultSet rset = stmt.executeQuery(
   "SELECT SUM(number) FROM TABLE(total_cost("P-100"))");

System.out.println(rset.getFloat(2));
```

5.21 Suppose there are two relations r and s, such that the foreign key B of r references the primary key A of s. Describe how the trigger mechanism can be used to implement the **on delete cascade** option, when a tuple is deleted from s.

Answer: We define triggers for each relation whose primary-key is referred to by the foreign-key of some other relation. The trigger would be activated whenever a tuple is deleted from the referred-to relation. The action performed by the trigger would be to visit all the referring relations, and delete all the tuples in them whose foreign-key attribute value is the same as the primary-key attribute value of the deleted tuple in the referred-to relation. These set of triggers will take care of the **on delete cascade** operation.

5.22 The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.

Answer: It is possible that a trigger body is written in such a way that a non-terminating recursion may result. An example of such a trigger is a *before insert* trigged on a relation that tries to insert another record into the same relation.

In general, it is extremely difficult to statically identify and prohibit such triggers from being created. Hence database systems, at runtime, put a limit on the depth of nested trigger calls.

5.23 Consider the relation, r, shown in Figure 5.27. Give the result of the following query:

select *building*, *room_number*, *time_slot_id*, **count**(*)
from r
group by rollup (*building*, *room_number*, *time_slot_id*)

Answer:

Garfield	359	P	1
Garfield	359	*null*	1
Garfield	*null*	*null*	1
Painter	705	N	1
Painter	705	*null*	1
Painter	*null*	*null*	1
Saucon	550	D	1
Saucon	550	*null*	1
Saucon	651	N	1
Saucon	651	*null*	1
Saucon	*null*	*null*	2

5.24 For each of the SQL aggregate functions **sum, count, min**, and **max**, show how to compute the aggregate value on a multiset $S_1 \cup S_2$, given the aggregate values on multisets S_1 and S_2.

On the basis of the above, give expressions to compute aggregate values with grouping on a subset S of the attributes of a relation $r(A, B, C, D, E)$, given aggregate values for grouping on attributes $T \supseteq S$, for the following aggregate functions:

a. **sum, count, min,** and **max**

b. **avg**

c. Standard deviation

Answer: Given aggregate values on multisets S_1 and S_2, we can calculate the corresponding aggregate values on multiset $S_1 \cup S_2$ as follows:

- $\textbf{sum}(S_1 \cup S_2) = \textbf{sum}(S_1) + \textbf{sum}(S_2)$
- $\textbf{count}(S_1 \cup S_2) = \textbf{count}(S_1) + \textbf{count}(S_2)$
- $\textbf{min}(S_1 \cup S_2) = \textbf{min}(\textbf{min}(S_1), \textbf{min}(S_2))$
- $\textbf{max}(S_1 \cup S_2) = \textbf{max}(\textbf{max}(S_1), \textbf{max}(S_2))$

Let the attribute set $T = (A, B, C, D)$ and the attribute set $S = (A, B)$. Let the aggregation on the attribute set T be stored in table *aggregation_on_t* with aggregation columns *sum_t*, *count_t*, *min_t*, and *max_t* storing **sum, count, min** and **max** resp.

a. The aggregations *sum_s*, *count_s*, *min_s*, and *max_s* on the attribute set S are computed by the query:

```
select A, B, sum(sum_t) as sum_s, sum(count_t) as count_s,
        min(min_t) as min_s, max(max_t) as max_s
from aggregation_on_t
groupby A, B
```

b. The aggregation *avg* on the attribute set S is computed by the query:

```
select A, B, sum(sum_t)/sum(count_t) as avg_s
from aggregation_on_t
groupby A, B
```

c. For calculating standard deviation we use an alternative formula:

$$stddev(S) = \frac{\sum_{s \in S} s^2}{|S|} - avg(S)^2$$

which we get by expanding the formula

$$stddev(S) = \frac{\sum_{s \in S} (s^2 - avg(S))^2}{|S|}$$

If S is partitioned into n sets $S_1, S_2, \ldots S_n$ then the following relation holds:

$$stddev(S) = \frac{\sum_{S_i} |S_i|(stddev(S_i)^2 + avg(S_i)^2)}{|S|} - avg(S)^2$$

Using this formula, the aggregation **stddev** is computed by the query:

```
select A, B,
        (sum(count_t * (stddev_t*stddev_t+ avg_t* avg_t))/sum(count_t)) -
            (sum(sum_t)/sum(count_t))
from aggregation_on_t
groupby A, B
```

5.25 In Section 5.5.1, we used the *student_grades* view of Exercise 4.5 to write a query to find the rank of each student based on grade-point average. Modify that query to show only the top 10 students (that is, those students whose rank is 1 through 10).

Answer:

```
with s_grades as
    select ID,rank() over (order by (GPA)desc) as s_rank
    from student_grades
select ID,s_rank
from s_grades
where s_rank <= 10
```

5.26 Give an example of a pair of groupings that cannot be expressed by using a single **group by** clause with **cube** and **rollup**.

Answer: Consider an example of hierarchies on dimensions from Figure 5.19. We can not express a query to seek aggregation on groups (*City*, *Hour of day*) and (*City*, *Date*) using a single **group by** clause with **cube** and **rollup**.

Any single **groupby** clause with **cube** and **rollup** that computes these two groups would also compute other groups also.

5.27 Given relation $s(a, b, c)$, show how to use the extended SQL features to generate a histogram of c versus a, dividing a into 20 equal-sized partitions (that is, where each partition contains 5 percent of the tuples in s, sorted by a).

Answer:

```
select tile20, sum(c)
from (select c, ntile(20) over (order by (a)) as tile20
        from r) as s
groupby tile20
```

5.28 Consider the bank database of Figure 5.25 and the *balance* attribute of the *account* relation. Write an SQL query to compute a histogram of *balance* values, dividing the range 0 to the maximum account balance present, into three equal ranges.

Answer:

```
(select 1, count(*)
 from account
 where 3* balance <= (select max(balance)
                      from account)
)
union
(select 2, count(*)
 from account
 where 3* balance > (select max(balance)
                     from account)
       and 1.5* balance <= (select max(balance)
                            from account)
)
union
(select 3, count(*)
 from account
 where 1.5* balance > (select max(balance)
                       from account)
)
```

CHAPTER 6

Formal Relational Query Languages

In this chapter we study three additional formal relational languages. Relational Algebra, tuple relational calculus and domain relational calculus.

Of these three formal languages, we suggest placing an emphasis on relational algebra, which is used extensively in the chapters on query processing and optimization, as well as in several other chapters. The relational calculi generally do not merit as much emphasis.

Our notation for the tuple relational calculus makes it easy to present the concept of a safe query. The concept of safety for the domain relational calculus, though identical to that for the tuple calculus, is much more cumbersome notationally and requires careful presentation. This consideration may suggest placing somewhat less emphasis on the domain calculus for classes not focusing on database theory.

Exercises

6.10 Write the following queries in relational algebra, using the university schema.

a. Find the names of all students who have taken at least one Comp. Sci. course.

b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

d. Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.

Answer:

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*person_name*, *manager_name*)

Figure 6.22 Relational database for Exercises 6.2, 6.8, 6.11, 6.13, and 6.15

a. $\Pi_{name}\ (student \bowtie takes \bowtie \Pi_{course\_id}(\sigma_{dept\_name\ =\ 'Comp.Sci.'}(course)))$
Note that if we join *student*, *takes*, and *course*, only students from the Comp. Sci. department would be present in the result; students from other departments would be eliminated even if they had taken a Comp. Sci. course since the attribute *dept_name* appears in both *student* and *course*.

b. $\Pi_{ID,name}\ (student) - \Pi_{ID,name}\ (\sigma_{year<2009}(student \bowtie takes)$ Note that Spring is the first semester of the year, so we do not need to perform a comparison on *semester*.

c. ${}_{dept\_name}\mathcal{G}_{\textbf{max}(salary)}(instructor)$

d. $\mathcal{G}_{\textbf{min}(maxsal)}({}_{dept\_name}\mathcal{G}_{\textbf{max}(salary)}\ \textbf{as}\ maxsal(instructor))$

6.11 Consider the relational database of Figure 6.22, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who work for "First Bank Corporation".

b. Find the names and cities of residence of all employees who work for "First Bank Corporation".

c. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

d. Find the names of all employees in this database who live in the same city as the company for which they work.

e. Assume the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

Answer:

a. $\Pi_{person\_name}\ (\sigma_{company\_name\ =\ \text{"First Bank Corporation"}}\ (works))$

b. $\Pi_{person\_name,\ city}\ (employee \bowtie$
$(\sigma_{company\_name\ =\ \text{"First Bank Corporation"}}\ (works)))$

c. $\Pi_{person\_name,\ street,\ city}$
$(\sigma_{(company\_name\ =\ \text{“First Bank Corporation”}\ \wedge\ salary\ >\ 10000)}$
$works \bowtie employee)$

d. $\Pi_{person\_name}\ (employee \bowtie works \bowtie company)$

e. Note: Small Bank Corporation will be included in each answer.
$\Pi_{company\_name}\ (company \div$
$(\Pi_{city}\ (\sigma_{company\_name\ =\ \text{“Small Bank Corporation”}}\ (company))))$

6.12 Using the university example, write relational-algebra queries to find the course sections taught by more than one instructor in the following ways:

a. Using an aggregate function.

b. Without using any aggregate functions.

Answer:

a. $\sigma_{instrcnt>1}\ (_{course\_id,section\_id,year,semester}\mathcal{G}_{\textbf{count}(*)\ \textbf{as}\ instrcnt}(teaches))$

b. $\Pi_{course\_id,section\_id,year,semester}(\sigma_{ID<>ID2}(takes \bowtie$
$\rho_{takes1(ID2,course\_id,section\_id,year,semester)}(takes)))$

6.13 Consider the relational database of Figure 6.22. Give a relational-algebra expression for each of the following queries:

a. Find the company with the most employees.

b. Find the company with the smallest payroll.

c. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

Answer:

a. $t_1 \leftarrow\ _{company\_name}\mathcal{G}_{\textbf{count-distinct}(person\_name)}(works)$
$t_2 \leftarrow\ \mathcal{G}_{\textbf{max}(num\_employees)}(\rho_{company\_strength(company\_name,num\_employees)}(t_1))$
$\Pi_{company\_name}(\rho_{t_3(company\_name,num\_employees)}(t_1) \bowtie \rho_{t_4(num\_employees)}(t_2))$

b. $t_1 \leftarrow\ _{company\_name}\mathcal{G}_{\textbf{sum}(salary)}(works)$
$t_2 \leftarrow\ \mathcal{G}_{\textbf{min}(payroll)}(\rho_{company\_payroll(company\_name,payroll)}(t_1))$
$\Pi_{company\_name}(\rho_{t_3(company\_name,payroll)}(t_1) \bowtie \rho_{t_4(payroll)}(t_2))$

c. $t_1 \leftarrow\ _{company\_name}\mathcal{G}_{\textbf{avg}(salary)}(works)$
$t_2 \leftarrow\ \sigma_{company\_name\ =\ \text{“First Bank Corporation”}}(t_1)$
$\Pi_{t_3.company\_name}((\rho_{t_3(company\_name,avg\_salary)}(t_1))$
$\bowtie_{t_3.avg\_salary\ >\ first\_bank.avg\_salary}\ (\rho_{first\_bank(company\_name,avg\_salary)}(t_2)))$

6.14 Consider the following relational schema for a library:

member(*memb_no*, *name*, *dob*)
books(*isbn*, *title*, *authors*, *publisher*)
borrowed(*memb_no*, *isbn*, *date*)

Write the following queries in relational algebra.

a. Find the names of members who have borrowed any book published by "McGraw-Hill".

b. Find the name of members who have borrowed all books published by "McGraw-Hill".

c. Find the name and membership number of members who have borrowed more than five different books published by "McGraw-Hill".

d. For each publisher, find the name and membership number of members who have borrowed more than five books of that publisher.

e. Find the average number of books borrowed per member. Take into account that if an member does not borrow any books, then that member does not appear in the *borrowed* relation at all.

Answer:

a. $t_1 \leftarrow \Pi_{isbn}(\sigma_{publisher="McGraw-Hill"}(books))$
$\Pi_{name}((member \bowtie borrowed) \bowtie t_1))$

b. $t_1 \leftarrow \Pi_{isbn}(\sigma_{publisher="McGraw-Hill"}(books))$
$\Pi_{name,isbn}(member \bowtie borrowed) \div t_1$

c. $t_1 \leftarrow member \bowtie borrowed \bowtie (\sigma_{publisher="McGraw-Hill"}(books))$
$\Pi_{name}\ (\sigma_{countisbn > 5}\ ((\ _{memb\_no}\ G_{\textbf{count-distinct}(isbn)\ \textbf{as}\ countisbn}(t_1))))$

d. $t_1 \leftarrow member \bowtie borrowed \bowtie books$
$\Pi_{publisher,name}\ (\sigma_{countisbn > 5}\ ((\ _{publisher,memb\_no}\ G_{\textbf{count-distinct}(isbn)\ \textbf{as}\ countisbn}(t_1)))$

6.15 Consider the employee database of Figure 6.22. Give expressions in tuple relational calculus and domain relational calculus for each of the following queries:

a. Find the names of all employees who work for "First Bank Corporation".

b. Find the names and cities of residence of all employees who work for "First Bank Corporation".

c. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

d. Find all employees who live in the same city as that in which the company for which they work is located.

e. Find all employees who live in the same city and on the same street as their managers.

f. Find all employees in the database who do not work for "First Bank Corporation".

g. Find all employees who earn more than every employee of "Small Bank Corporation".

h. Assume that the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

Answer:

a. Find the names of all employees who work for First Bank Corporation:

i. $\{t \mid \exists s \in works\ (t[person\_name] = s[person\_name] \wedge s[company\_name] = \text{"First Bank Corporation"})\}$

ii. $\{< p > \mid \exists c, s\ (< p, c, s > \in works \wedge c = \text{"First Bank Corporation"})\}$

b. Find the names and cities of residence of all employees who work for First Bank Corporation:

i. $\{t \mid \exists r \in employee\ \exists s \in works\ (t[person\_name] = r[person\_name] \wedge t[city] = r[city] \wedge r[person\_name] = s[person\_name] \wedge s[company\_name] = \text{'First Bank Corporation'})\}$

ii. $\{< p, c > \mid \exists co, sa, st\ (< p, co, sa > \in works \wedge < p, st, c > \in employee \wedge co = \text{"First Bank Corporation"})\}$

c. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum:

i. $\{t \mid t \in employee \wedge (\exists s \in works\ (s[person\_name] = t[person\_name] \wedge s[company\_name] = \text{"First Bank Corporation"} \wedge s[salary] > 10000))\}$

ii. $\{< p, s, c > \mid < p, s, c > \in employee \wedge \exists co, sa\ (< p, co, sa > \in works \wedge co = \text{'First Bank Corporation'} \wedge sa > 10000)\}$

d. Find the names of all employees in this database who live in the same city as the company for which they work:

i. $\{t \mid \exists e \in employee\ \exists w \in works\ \exists c \in company$
$(t[person\_name] = e[person\_name]$
$\wedge e[person\_name] = w[person\_name]$
$\wedge w[company\_name] = c[company\_name] \wedge e[city] = c[city])\}$

ii. $\{< p > \mid \exists st, c, co, sa\ (< p, st, c > \in employee$
$\wedge < p, co, sa > \in works \wedge < co, c > \in company)\}$

e. Find the names of all employees who live in the same city and on the same street as do their managers:

i. $\{t \mid \exists l \in employee\ \exists m \in manages\ \exists r \in employee$
$(l[person\_name] = m[person\_name] \wedge m[manager\_name] = r[person\_name]$
$\wedge l[street] = r[street] \wedge l[city] = r[city] \wedge t[person\_name] = l[person\_name])\}$

ii. $\{< t > \mid \exists s, c, m\ (< t, s, c > \in employee \wedge < t, m > \in manages \wedge < m, s, c > \in employee)\}$

f. Find the names of all employees in this database who do not work for First Bank Corporation:
If one allows people to appear in the database (e.g. in *employee*) but not appear in *works,* the problem is more complicated. We give solutions for this more realistic case later.

i. $\{t \mid \exists w \in works\ (w[company\_name] \neq$ "First Bank Corporation"
$\wedge t[person\_name] = w[person\_name])\}$

ii. $\{< p > \mid \exists c, s\ (< p, c, s > \in works \wedge c \neq$ "First Bank Corporation"$)\}$

If people may not work for any company:

i. $\{t \mid \exists e \in employee\ (t[person\_name] = e[person\_name] \wedge \neg \exists w \in works$
$(w[company\_name] =$ "First Bank Corporation"
$\wedge w[person\_name] = t[person\_name]))\}$

ii. $\{< p > \mid \exists s, c\ (< p, s, c > \in employee) \wedge \neg \exists x, y$
$(y =$ "First Bank Corporation"$\wedge < p, y, x > \in works)\}$

g. Find the names of all employees who earn more than every employee of Small Bank Corporation:

i. $\{t \mid \exists w \in works\ (t[person\_name] = w[person\_name] \wedge \forall s \in works$
$(s[company\_name] =$ "Small Bank Corporation" $\Rightarrow w[salary] > s[salary]))\}$

ii. $\{< p > \mid \exists\, c,\ s\ (< p, c, s > \in works \wedge \forall\, p_2,\ c_2,\ s_2$
$(< p_2, c_2, s_2 > \notin works \vee c_2 \neq$ "Small Bank Corporation" $\vee$
$s > s_2))\}$

h. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.
Note: Small Bank Corporation will be included in each answer.

i. $\{t \mid \forall s \in company\,(s[company\_name] =$ "Small Bank Corporation" $\Rightarrow$
$\exists\, r \in company\ (t[company\_name] = r[company\_name] \wedge$
$r[city] = s[city]))\}$

ii. $\{< co > \mid \forall\, co_2,\ ci_2\ (< co_2, ci_2 > \notin company$
$\vee co_2 \neq$ "Small Bank Corporation" $\vee\ < co, ci_2 > \in company)\}$

6.16 Let $R = (A,\ B)$ and $S = (A,\ C)$, and let $r(R)$ and $s(S)$ be relations. Write relational-algebra expressions equivalent to the following domain-relational-calculus expressions:

a. $\{< a > \mid \exists\, b\ (< a, b > \in r \wedge b = 17)\}$

b. $\{< a, b, c > \mid\ < a, b > \in r \wedge\ < a, c > \in s\}$

c. $\{< a > \mid \exists\, b\ (< a, b > \in r) \vee \forall\, c\ (\exists\, d\ (< d, c > \in s) \Rightarrow < a, c > \in s)\}$

d. $\{< a > \mid \exists\, c\ (< a, c > \in s \wedge \exists\, b_1, b_2\ (< a, b_1 > \in r \wedge\ < c, b_2 >$
$\in r \wedge b_1 > b_2))\}$

Answer:

a. $\Pi_A\,(\sigma_{B\,=\,17}\,(r))$

b. $r \bowtie s$

c. $\Pi_A(r) \cup (r \div \sigma_B(\Pi_C(s)))$

d. $\Pi_{r.A}\,((r \bowtie s) \bowtie_{c\,=\,r2.A \wedge r.B\,>\,r2.B} (\rho_{r_2}(r)))$
It is interesting to note that (d) is an abstraction of the notorious query "Find all employees who earn more than their manager." Let $R = (emp,\ sal)$, $S = (emp,\ mgr)$ to observe this.

6.17 Repeat Exercise 6.16, writing SQL queries instead of relational-algebra expressions.
Answer:

a. **select** a
from r
where $b = 17$

b. **select** a, b, c
from r, s
where $r.a = s.a$

c. (**select** a
from r)
union
(**select** a
from s)

d. **select** a
from r **as** $r1, r$ **as** $r2, s$
where $r1.a = s.a$ **and** $r2.a = s.c$ **and** $r1.b > r2.b$

6.18 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Using the special constant *null*, write tuple-relational-calculus expressions equivalent to each of the following:

a. $r \ltimes s$

b. $r \rtimes s$

c. $r \rtimes\!\ltimes s$

Answer:

a. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee$
$\exists s \in S(\neg \exists r \in R(r[A] = s[A]) \wedge t[A] = s[A] \wedge t[C] = s[C] \wedge t[B] = null)\}$

b. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee$
$\exists r \in R(\neg \exists s \in S(r[A] = s[A]) \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = null) \vee$
$\exists s \in S(\neg \exists r \in R(r[A] = s[A]) \wedge t[A] = s[A] \wedge t[C] = s[C] \wedge t[B] = null)\}$

c. $\{t \mid \exists r \in R\, \exists s \in S\, (r[A] = s[A] \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = s[C]) \vee$
$\exists r \in R(\neg \exists s \in S(r[A] = s[A]) \wedge t[A] = r[A] \wedge t[B] = r[B] \wedge t[C] = null)\}$

6.19 Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.
Answer: $\{< a > \mid < a > \in r \wedge \forall < b > \in R\, a >= b\}$

CHAPTER 7

Database Design and the E-R Model

This chapter introduces the entity-relationship model in detail. A significant change in the 6th edition is the change in the E-R notation used in the book. There are several alternative E-R notations used in the industry. Increasingly, however, the UML class diagram notation is used instead of the traditional E-R notation used in earlier editions of the book. Among the reasons is the wide availability of UML tools, as well as the conciseness of the UML notation compared to the notation with separate ovals to represent attributes. In keeping with this trend, we have changed our E-R notation to be more compatible with UML.

The chapter covers numerous features of the model, several of which can be omitted depending on the planned coverage of the course. Extended E-R features (Section 7.8) and all subsequent sections may be omitted in case of time constraints in the course, without compromising the students understanding of basic E-R modeling. However, we recommend covering specialization (Section 7.8.1) at least in some detail, since it is widely used in object-oriented modeling.

The E-R model itself and E-R diagrams are used often in the text. It is important that students become comfortable with them. The E-R model is an excellent context for the introduction of students to the complexity of database design. For a given enterprise there are often a wide variety of E-R designs. Although some choices are arbitrary, it is often the case that one design is inherently superior to another. Several of the exercises illustrate this point. The evaluation of the goodness of an E-R design requires an understanding of the enterprise being modeled and the applications to be run. It is often possible to lead students into a debate of the relative merits of competing designs and thus illustrate by example that understanding the application is often the hardest part of database design.

Among the points that are worth discussing when coming up with an E-R design are:

1. Naming of attributes: this is a key aspect of a good design. One approach to design ensures that no two attributes share a name by accident; thus, if ID appears as an attribute of person, it should not appear as an attribute of

another relation, unless it references the ID of person. When natural joins are used in queries, this approach avoids accidental equation of attributes to some extent, although not always; for example, students and instructors share attributes ID and name (presumably inherited from a generalization *person*), so a query that joins the student and instructor relations would equate the respective attribute names.

2. Primary keys: one approach to design creates identifier values for every entity, which are internal to the system and not normally made visible to end users. These internal values are often declared in SQL as **auto increment**, meaning that whenever a tuple is inserted to the relation, a unique value is given to the attribute automatically.
   In contrast, the alternative approach, which we have used in this book, avoids creating artificial internal identifiers, and instead uses externally visible attributes as primary key values wherever possible.
   As an example, in any university employees and students have externally visible identifiers. These could be used as the primary keys, or alternatively, the application can create identifiers that are not externally visible, and use them as the value for the primary key.
   As another example, the *section* table, which has the combination of (*course_id*, *section_id*, *semester*, *year*) as primary key, could instead have a section identifier that is unique across all sections as primary key, with the *course_id*, *section_id*, *semester*, *year* as non-primary key attributes. The difference would be that the relations that refer to *section*, namely *teaches* and *takes*, would have a single unique section id attribute as a foreign key referring to *section*, and would not need to store *course_id*, *section_id*, *semester*, and *year*.

Considerable emphasis is placed on the construction of tables from E-R diagrams. This serves to build intuition for the discussion of the relational model in the subsequent chapters. It also serves to ground abstract concepts of entities and relationships into the more concrete concepts of relations. Several other texts places this material along with the relational data model, rather than in the E-R model chapter. Our motivation for placing this material here is help students to appreciate how E-R data models get used in reality, while studying the E-R model rather than later on.

Exercises

7.14 Explain the distinctions among the terms primary key, candidate key, and superkey.

Answer: A *superkey* is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. A superkey may contain extraneous attributes. If K is a superkey, then so is any superset of K. A superkey for which no proper subset is also a superkey is called a *candidate key*. It is possible that several distinct sets of attributes could

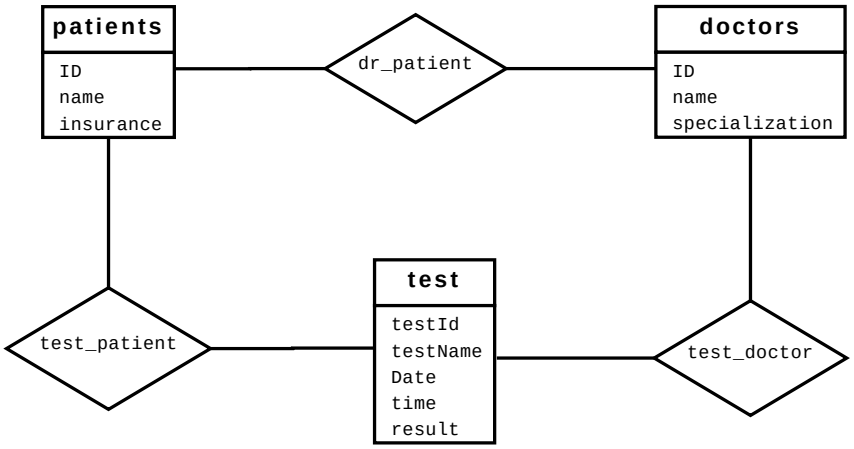


Figure 7.1 E-R diagram for a hospital.

serve as candidate keys. The *primary key* is one of the candidate keys that is chosen by the database designer as the principal means of identifying entities within an entity set.

7.15 Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

Answer:

An E-R diagram for the hospital is shown in Figure 7.1. Although the diagram meets the specifications of the question, a real-world hospital would have many more requirements, such as tracking patient admissions and visits, including which doctor sees a patient on each visit, recording results of tests in a more structured manner, and so on.

7.16 Construct appropriate relation schemas for each of the E-R diagrams in Practice Exercises 7.1 to 7.3.

Answer:

a. Car insurance tables:

customer (customer_id, name, address)
car (license, model)
owns(customer_id, license_no)
accident (report_id, date, place)
participated(license_no, report_id) policy(policy_id)
covers(policy_id, license_no)
premium_payment(policy_id, payment_no, due_date, amount, received_on)

Note that a more realistic database design would include details of who was driving the car when an accident happened, and the damage amount for each car that participated in the accident.

b. Student Exam tables:
   i. Ternary Relationship:

student (student_id, name, dept_name, tot_cred)
course(course_id, title, credits)
section(course_id, section_id, semester, year)
exam(exam_id, name, place, time)
exam_marks(student_id, course_id, section_id, semester, year, exam_id, marks)

   ii. Binary relationship:

student (ID, name, dept_name, tot_cred)
course(course_id, title, credits)
section(course_id, section_id, semester, year)
exam_marks(student_id, course_id, sec_id, semester, year, exam_id, marks)

c. Player Match tables:

match(match_id, date, stadium, opponent, own_score, opp_score)
player(player_id, name, age, season_score)
played(match_id,player_id, score)

7.17 Extend the E-R diagram of Practice Exercise 7.3 to track the same information for all teams in a league.

Answer: See Figure 7.2. Note that we assume a player can play in only one team; if a player may switch teams, we would have to track for each match which team the player was in, which we could do by turning the relationship *played* into a ternary relationship.

7.18 Explain the difference between a weak and a strong entity set.

Answer: A strong entity set has a primary key. All tuples in the set are distinguishable by that key. A weak entity set has no primary key unless attributes of the strong entity set on which it depends are included. Tuples in a weak entity set are partitioned according to their relationship with tuples in a strong entity set. Tuples within each partition are distinguishable by a discriminator, which is a set of attributes.

7.19 We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?

Answer: We have weak entities for several reasons:

- We want to avoid the data duplication and consequent possible inconsistencies caused by duplicating the key of the strong entity.
- Weak entities reflect the logical structure of an entity being dependent on another entity.
- Weak entities can be deleted automatically when their strong entity is deleted.

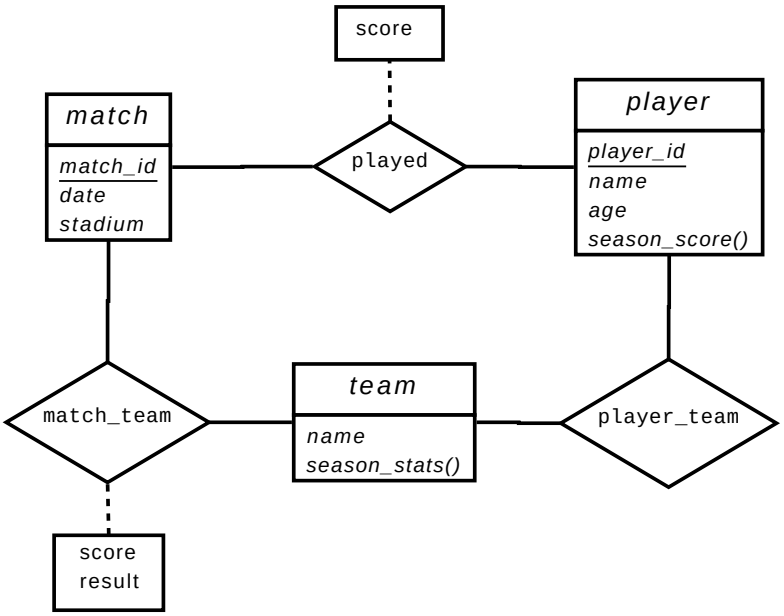


Figure 7.2 E-R diagram for all teams statistics.

- Weak entities can be stored physically with their strong entities.

7.20 Consider the E-R diagram in Figure 7.29, which models an online bookstore.

a. List the entity sets and their primary keys.

b. Suppose the bookstore adds Blu-ray discs and downloadable video to its collection. The same item may be present in one or both formats, with differing prices. Extend the E-R diagram to model this addition, ignoring the effect on shopping baskets.

c. Now extend the E-R diagram, using generalization, to model the case where a shopping basket may contain any combination of books, Blu-ray discs, or downloadable video.

Answer: Interpret the second part of the question as the bookstore adds videos, which may be in Blu-ray disk format or in downloadable format; the same video may be present in both formats.

a. Entity sets:

author(name, address, URL)
publisher(name, address, phone, URL)
book(ISBN, title, year, price)
customer(email, name, address, phone)
shopping_basket(basket_id)
warehouse(code, address, phone)

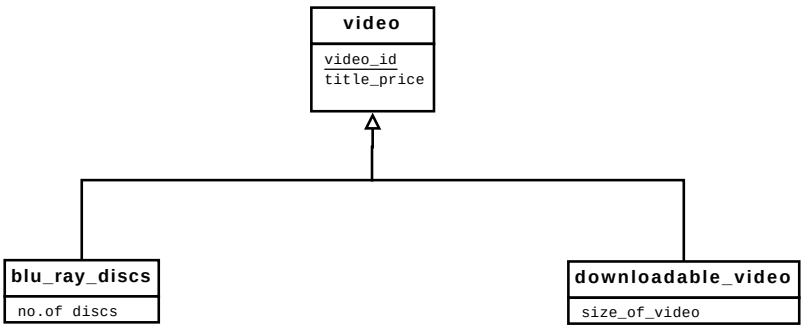


Figure 7.3 ER Diagram for Exercise 7.20 (b)

b. The ER-diagram portion related to videos is shown in Figure 7.3.

c. The E-R diagram shown in Figure 7.4 should be added to the E-R diagram of Figure 7.29. Entities that are shown already in Figure 7.29 are shown with only their names, omitting the attributes. The *contains* relationship in Figure 7.29 should be replaced by the version in Figure 7.4. All other parts of Figure 7.29 remain unchanged.

7.21 Design a database for an automobile company to provide to its dealers to assist them in maintaining customer records and dealer inventory and to assist sales staff in ordering cars.

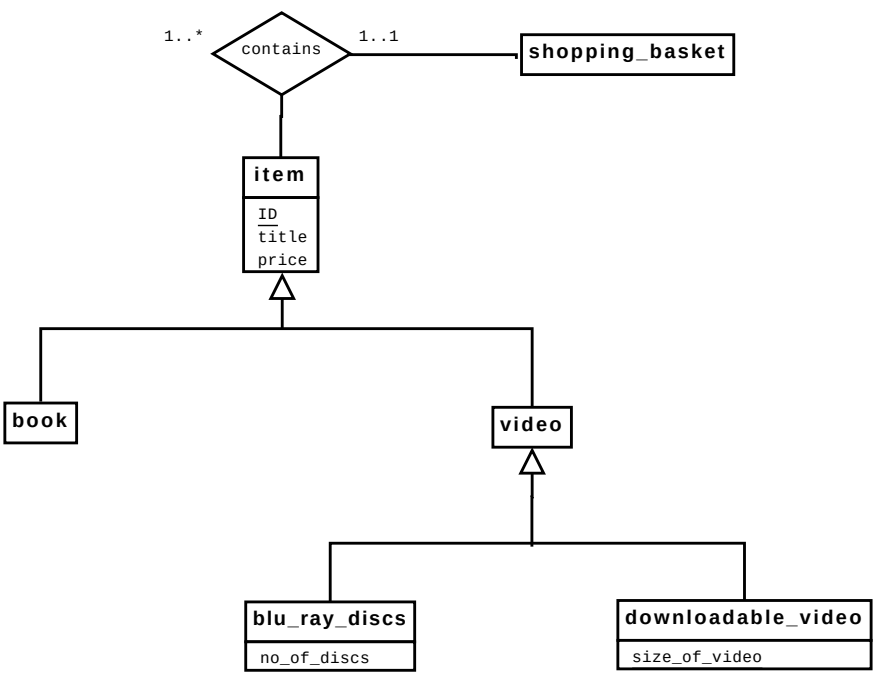


Figure 7.4 ER Diagram for Exercise 7.20 (c)

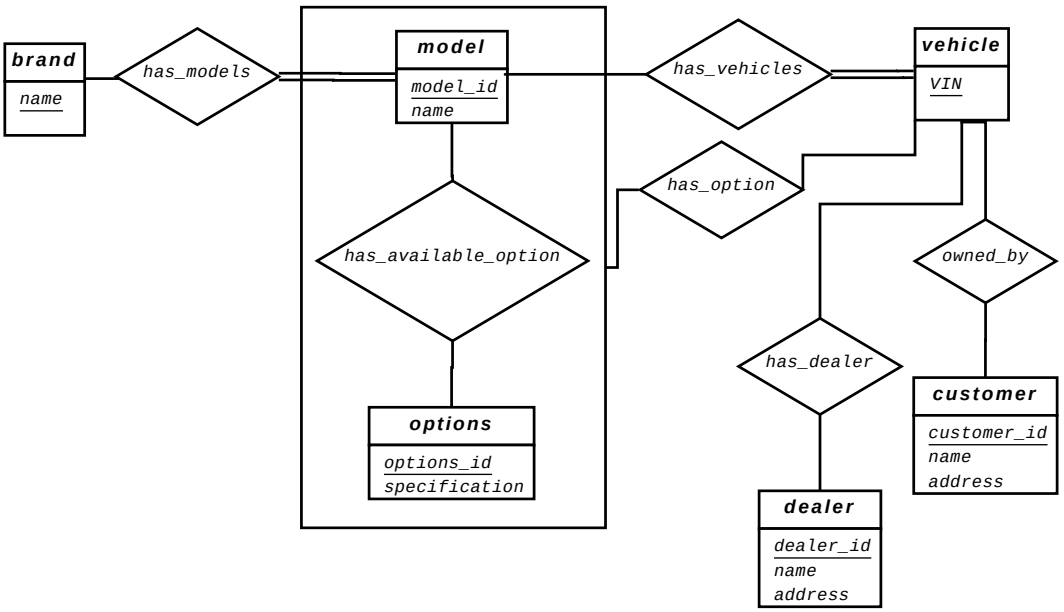


Figure 7.5 ER Diagram for Exercise 7.21

Each vehicle is identified by a vehicle identification number (VIN). Each individual vehicle is a particular model of a particular brand offered by the company (e.g., the XF is a model of the car brand Jaguar of Tata Motors). Each model can be offered with a variety of options, but an individual car may have only some (or none) of the available options. The database needs to store information about models, brands, and options, as well as information about individual dealers, customers, and cars.

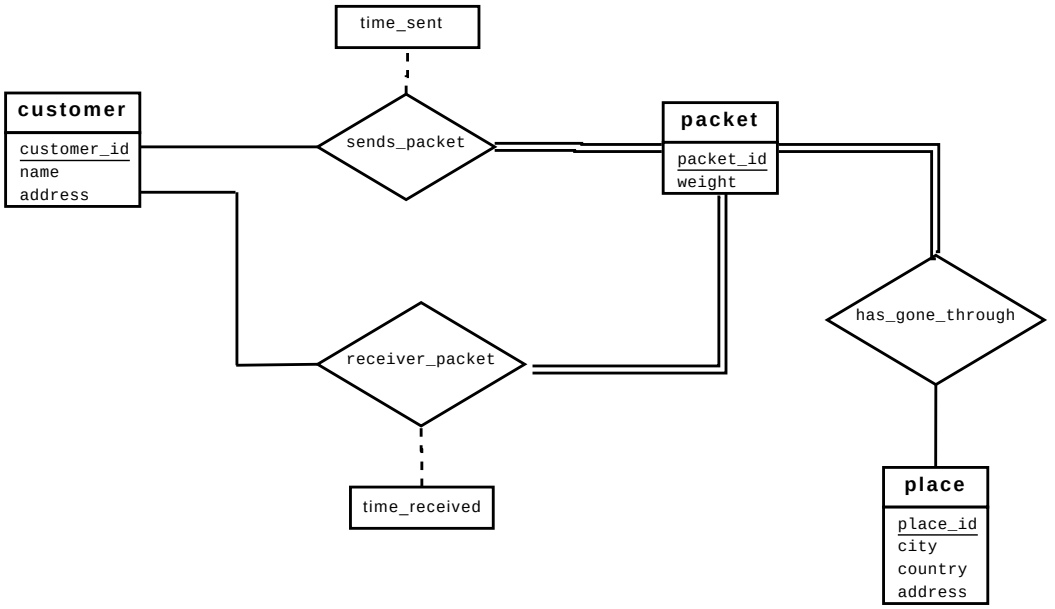
Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

Answer:

The E-R diagram is shown in Figure 7.5. Note that the *has_option* relationship links a vehicle to an aggregation on the relationship *has_available_option*, instead of directly to the entity set *options*, to ensure that a particular vehicle instance cannot get an option that does not correspond to its model. The alternative of directly linking to *options* is acceptable if ensuring the above integrity constraint is not critical.

The relational schema, along with primary-key and foreign-key constraints is shown below.

brand(*name*)
model(*model_id*,
    name)
vehicle(*VIN*)
option(*option_id*,
    specification)
customer(*customer_id*,
    name,
    address)
dealer(*dealer_id*,
    name,
    address)
has_models(*name*,
    model_id ,
    foreign key *name* **references** *brand* ,
    foreign key *model_id* **references** *model*
)
has_vehicles(*model_id*,
    VIN,
    foreign key *VIN* **references** *vehicle*,
    foreign key *model_id* **references** *model*
)
available_options(*model_id*,
    option_id,
    foreign key *option_id* **references** *option*,
    foreign key *model_id* **references** *model*
)
has_options(*VIN*,
    model_id,
    option_id,
    foreign key *VIN* **references** *vehicle*,
    foreign key (*model_id*, *option_id*) **references** *available_options*
)
has_dealer(*VIN*,
    dealer_id ,
    foreign key *dealer_id* **references** *dealer*,
    foreign key *VIN* **references** *vehicle*
)
owned_by(*VIN*,
    customer_id,
    foreign key *customer_id* **references** *customer*,
    foreign key *VIN* **references** *vehicle*
)



As an alternative:

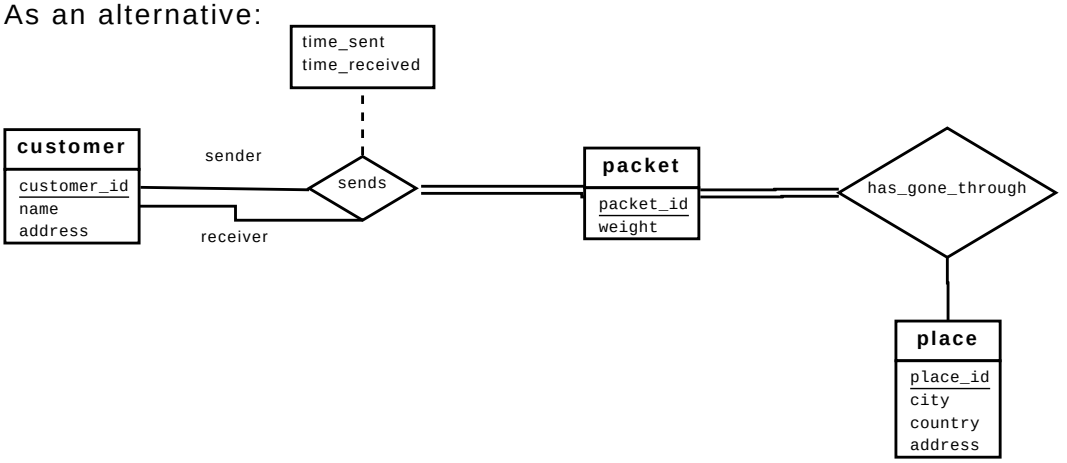


Figure 7.6 ER Diagram Alternatives for Exercise 7.22

7.22 Design a database for a world-wide package delivery company (e.g., DHL or FedEX). The database must be able to keep track of customers (who ship items) and customers (who receive items); some customers may do both. Each package must be identifiable and trackable, so the database must be able to store the location of the package and its history of locations. Locations include trucks, planes, airports, and warehouses.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

Answer:

Two alternative E-R diagrams are shown in Figure 7.6.
The relational schema, including primary-key and foreign-key constraints, corresponding to the second alternative is shown below.

*customer(*customer_id*,*
name,
address)
*packet(*packet_id*,*
weight)
*place(*place_id*,*
city,
country,
address)
*sends(*sender_id*,*
receiver_id,
packet_id,
time_received,
time_sent
foreign key *sender_id* **references** *customer,*
foreign key *receiver_id* **references** *customer,*
foreign key *packet_id* **references** *packet*
)
has_gone_through(
packet_id,
place_id
foreign key *packet_id* **references** *packet,*
foreign key *place_id* **references** *place*
)

7.23 Design a database for an airline. The database must keep track of customers and their reservations, flights and their status, seat assignments on individual flights, and the schedule and routing of future flights.

Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

Answer:

The E-R diagram is shown in Figure 7.7. We assume that the schedule of a flight is fixed across time, although we allow specification of on which days a flight is scheduled. For a particular instance of a flight however we record actual times of departure and arrival. In reality, schedules change with time, so the schedule and routing should be for a particular flight for specified dates, or for a specified range of dates; we ignore this complexity.

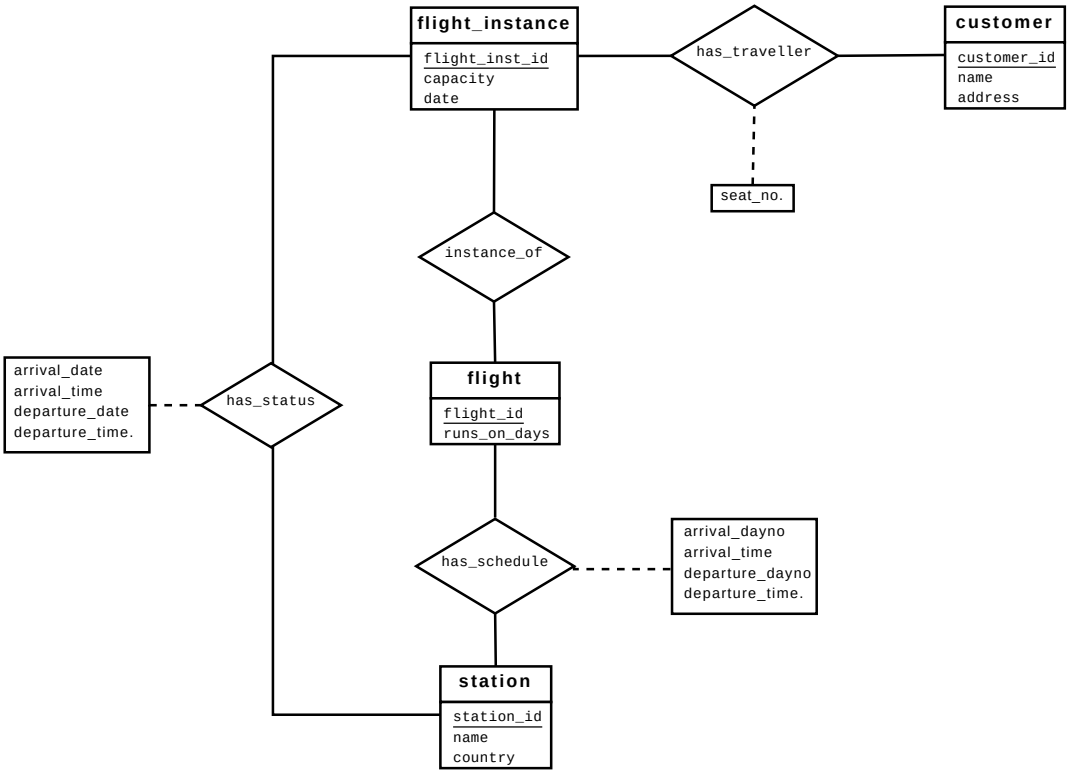


Figure 7.7 ER Diagram for Exercise 7.23

flight_instance(*flight_inst_id*, *capacity*, *date*)
customer(*customer_id*, *name*, *address*)
flight(*flight_id*, *runs_on_days*)
station(*station_id*, *name*, *country*)
has_traveller(
    flight_inst_id,
    customer_id,
    seat_number,
    foreign key *flight_inst_id* **references** *flight_instance*,
    foreign key *customer_id* **references** *customer*
)
instance_of(
    flight_inst_id,
    flight_id,
    foreign key *flight_inst_id* **references** *flight_instance*,
    foreign key *flight_id* **references** *flight*
)

has_schedule(
flight_id,
station_id,
order,
arrival_dayno,
arrival_time,
departure_dayno,
departure_time,
foreign key *flight_id* **references** *flight*,
foreign key *station_id* **references** *station*
)
has_status(
flight_inst_id,
station_id,
arrival_date,
arrival_time,
departure_date
departure_time
foreign key *flight_inst_id* **references** *flight_instance*,
foreign key *station_id* **references** *station*
)

7.24 In Section 7.7.3, we represented a ternary relationship (repeated in Figure 7.27a) using binary relationships, as shown in Figure 7.27b. Consider the alternative shown in Figure 7.27c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.

Answer: In the model of Figure 7.27b, there can be instances where E, A, B, C, R_A, R_B and R_C cannot correspond to any instance of A, B, C and R.

The model of Figure 7.27c will not be able to represent all ternary relationships. Consider the ABC relationship set below.

A	B	C
1	2	3
4	2	7
4	8	3

If ABC is broken into three relationships sets AB, BC and AC, the three will imply that the relation (4, 2, 3) is a part of ABC.

7.25 Consider the relation schemas shown in Section 7.6, which were generated from the E-R diagram in Figure 7.15. For each schema, specify what foreign-key constraints, if any, should be created.

Answer: The foreign-key constraints are as specified below.

teaches(
foreign key *ID* **references** *instructor,*
foreign key *(course_id, sec_id, semester, year)* **references** *sec_course*
)

takes(
foreign key *ID* **references** *student,*
foreign key *(course_id, sec_id, semester, year)* **references** *sec_course*
)

prereq(
foreign key *course_id* **references** *course,*
foreign key *prereq_id* **references** *course*
)

advisor(
foreign key *s_ID* **references** *student,*
foreign key *i_id* **references** *instructor*
)

sec_course(
foreign key *course_id* **references** *course,*
foreign key *(sec_id, semester, year)* **references** *section*
)

sec_time_slot(
foreign key *(course_id, sec_id, semester, year)* **references** *sec_course*
foreign key *time_slot_id* **references** *time_slot*
)

sec_class(
foreign key *(course_id, sec_id, semester, year)* **references** *sec_course*
foreign key *(building, room_number)* **references** *classroom*
)

inst_dept(
foreign key *ID* **references** *instructor*
foreign key *dept_name* **references** *department*
)

stud_dept(
foreign key *ID* **references** *student*
foreign key *dept_name* **references** *department*
)

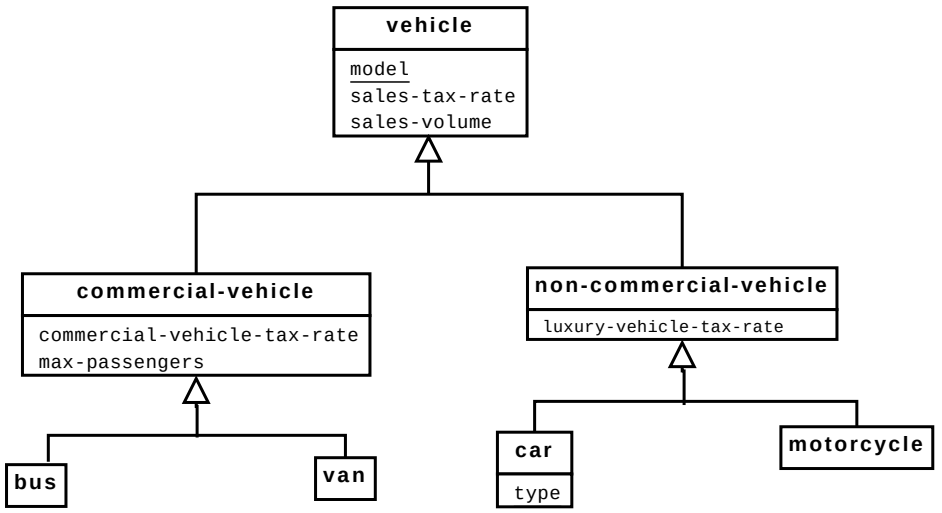


Figure 7.8 E-R diagram of motor-vehicle sales company.

course_dept(
foreign key *course_id* **references** *course*
foreign key *dept_name* **references** *department*
)

7.26 Design a generalization–specialization hierarchy for a motor vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.

Answer: Figure 7.8 gives one possible hierarchy; note that there could be many alternative solutions. The generalization–specialization hierarchy for the motor-vehicle company is given in the figure. *model*, *sales-tax-rate* and *sales-volume* are attributes necessary for all types of vehicles. Commercial vehicles attract commercial vehicle tax, and each kind of commercial vehicle has a passenger carrying capacity specified for it. Some kinds of non-commercial vehicles attract luxury vehicle tax. Cars alone can be of several types, such as sports-car, sedan, wagon etc., hence the attribute *type*.

7.27 Explain the distinction between condition-defined and user-defined constraints. Which of these constraints can the system check automatically? Explain your answer.

Answer: In a generalization–specialization hierarchy, it must be possible to decide which entities are members of which lower level entity sets. In a condition-defined design constraint, membership in the lower level entity-sets is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate.User-defined lower-level entity sets are not constrained by a membership condition; rather, entities are assigned to a given entity set by the database user.

Condition-defined constraints alone can be automatically handled by the system. Whenever any tuple is inserted into the database, its membership in the various lower level entity-sets can be automatically decided by evaluating the respective membership predicates. Similarly when a tuple is updated, its membership in the various entity sets can be re-evaluated automatically.

7.28 Explain the distinction between disjoint and overlapping constraints.

Answer: In a disjointness design constraint, an entity can belong to not more than one lower-level entity set. In overlapping generalizations, the same entity may belong to more than one lower-level entity sets. For example, in the employee-workteam example of the book, a manager may participate in more than one work-team.

7.29 Explain the distinction between total and partial constraints.

Answer: In a generalization–specialization hierarchy, a total constraint means that an entity belonging to the higher level entity set must belong to the lower level entity set. A partial constraint means that an entity belonging to the higher level entity set may or may not belong to the lower level entity set.

CHAPTER 8

Relational Database Design

This chapter presents the principles of relational database design. Undergraduates frequently find this chapter difficult. It is acceptable to cover only Sections 8.1 and 8.3 for classes that find the material particularly difficult. However, a careful study of data dependencies and normalization is a good way to introduce students to the formal aspects of relational database theory.

There are many ways of stating the definitions of the normal forms. We have chosen a style which we think is the easiest to present and which most clearly conveys the intuition of the normal forms.

Exercises

8.19 Give a lossless-join decomposition into BCNF of schema R of Exercise 8.1.

Answer: From Exercise 8.6, we know that $B \rightarrow D$ is nontrivial and the left hand side is not a superkey. By the algorithm of Figure 8.11 we derive the relations $\{(A, B, C, E), (B, D)\}$. This is in BCNF.

8.20 Give a lossless-join, dependency-preserving decomposition into 3NF of schema R of Practice Exercise 8.1.

Answer: First we note that the dependencies given in Practice Exercise 8.1 form a canonical cover. Generating the schema from the algorithm of Figure 8.12 we get

$$R' = \{(A, B, C), (C, D, E), (B, D), (E, A)\}.$$

Schema (A, B, C) contains a candidate key. Therefore R' is a third normal form dependency-preserving lossless-join decomposition.

Note that the original schema $R = (A, B, C, D, E)$ is already in 3NF. Thus, it was not necessary to apply the algorithm as we have done above. The single original schema is trivially a lossless join, dependency-preserving decomposition.

8.21 Normalize the following schema, with given constraints, to 4NF.

books(*accessionno*, *isbn*, *title*, *author*, *publisher*)
users(*userid*, *name*, *deptid*, *deptname*)
accessionno $\rightarrow$ *isbn*
isbn $\rightarrow$ *title*
isbn $\rightarrow$ *publisher*
isbn $\rightarrow\!\!\!\rightarrow$ *author*
userid $\rightarrow$ *name*
userid $\rightarrow$ *deptid*
deptid $\rightarrow$ *deptname*

Answer: In *books*, we see that

$$isbn \rightarrow\!\!\!\rightarrow title, publisher, author$$

and yet, *isbn* is not a super key. Thus, we break *books* into

books_accnno(*accessionno*, *isbn*)
books_details(*isbn*, *title*, *publisher*, *author*)

After this, we still have

$$isbn \rightarrow\!\!\!\rightarrow author$$

but neither is *isbn* a primary key of *book_details*, nor are the attributes of *book_details* equal to $\{isbn\} \cup \{author\}$. Therefore we decompose *book_details* again into

books_details1(*isbn*, *title*, *publisher*)
books_authors(*isbn*, *author*)

Similarly, in *users*,

$$deptid \rightarrow deptname$$

and yet, *deptid* is not a super key. Hence, we break *users* to

users(*userid*, *name*, *deptid*)
departments(*deptid*, *deptname*)

We verify that there are no further functional or multivalued dependencies that cause violation of 4NF, so the final set of relations are:

books_accnno(*accessionno*, *isbn*)
books_details1(*isbn*, *title*, *publisher*)
books_authors(*isbn*, *author*) *users*(*userid*, *name*, *deptid*)
departments(*deptid*, *deptname*)

8.22 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

Answer:

- Repetition of information is a condition in a relational database where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult, and can lead to inconsistent data if updates are done to one copy of the value, but not to another.
- Inability to represent information is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values otherwise a tuple without the unrelated information cannot be inserted into the relation.
- Inability to represent information can also occur because of loss of information which results from the decomposition of one relation into two relations, which cannot be combined to recreate the original relation. Such a lossy decomposition may happen implicitly, even without explicitly carrying out decomposition, if the initial relational schema itself corresponds to the decomposition.

8.23 Why are certain functional dependencies called *trivial* functional dependencies?

Answer: Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations.

8.24 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

Answer: The definition of functional dependency is: $\alpha \rightarrow \beta$ holds on R if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Reflexivity rule: if α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.
Assume $\exists\, t_1$ and t_2 such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	since $\beta \subseteq \alpha$
$\alpha \rightarrow \beta$	definition of FD

Augmentation rule: if $\alpha \rightarrow \beta$, and γ is a set of attributes, then $\gamma\,\alpha \rightarrow \gamma\,\beta$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\gamma\,\alpha] = t_2[\gamma\,\alpha]$

$t_1[\gamma] = t_2[\gamma]$	$\gamma \subseteq \gamma\,\alpha$
$t_1[\alpha] = t_2[\alpha]$	$\alpha \subseteq \gamma\,\alpha$
$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma\,\beta] = t_2[\gamma\,\beta]$	$\gamma\,\beta = \gamma \cup \beta$
$\gamma\,\alpha \rightarrow \gamma\,\beta$	definition of FD

Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma] = t_2[\gamma]$	definition of $\beta \rightarrow \gamma$
$\alpha \rightarrow \gamma$	definition of FD

8.25 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

Answer: Consider the following rule: if $A \rightarrow B$ and $C \rightarrow B$, then $A \rightarrow C$. That is, $\alpha = A$, $\beta = B$, $\gamma = C$. The following relation r is a counterexample to the rule.

r:

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2

Note: $A \rightarrow B$ and $C \rightarrow B$, (since no 2 tuples have the same C value, $C \rightarrow B$ is true trivially). However, it is not the case that $A \rightarrow C$ since the same A value is in two tuples, but the C value in those tuples disagree.

8.26 Use Armstrong's axioms to prove the soundness of the decomposition rule.

Answer: The decomposition rule, and its derivation from Armstrong's axioms are given below:
if $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

$\alpha \rightarrow \beta\gamma$	given
$\beta\gamma \rightarrow \beta$	reflexivity rule
$\alpha \rightarrow \beta$	transitivity rule
$\beta\gamma \rightarrow \gamma$	reflexive rule
$\alpha \rightarrow \gamma$	transitive rule

8.27 Using the functional dependencies of Practice Exercise 8.6, compute B^+.

Answer: Computing B^+ by the algorithm in Figure 8.8 we start with $result = \{B\}$. Considering FDs of the form $\beta \rightarrow \gamma$ in F, we find that the only dependencies satisfying $\beta \subseteq result$ are $B \rightarrow B$ and $B \rightarrow D$. Therefore $result = \{B, D\}$. No more dependencies in F apply now. Therefore $B^+ = \{B, D\}$

8.28 Show that the following decomposition of the schema R of Practice Exercise 8.1 is not a lossless-join decomposition:

(A, B, C)
(C, D, E).

Hint: Give an example of a relation r on schema R such that

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

Answer: Following the hint, use the following example of r:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

With $R_1 = (A, B, C)$, $R_2 = (C, D, E)$:

a. $\Pi_{R_1}(r)$ would be:

A	B	C
a_1	b_1	c_1
a_2	b_2	c_1

b. $\Pi_{R_2}(r)$ would be:

C	D	E
c_1	d_1	e_1
c_1	d_2	e_2

c. $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$ would be:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_2	b_2	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

Clearly, $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \neq r$. Therefore, this is a lossy join.

8.29 Consider the following set F of functional dependencies on the relation schema $r(A, B, C, D, E, F)$:

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. Compute B^+.

b. Prove (using Armstrong's axioms) that AF is a superkey.

c. Compute a canonical cover for the above set of functional dependencies F; give each step of your derivation with an explanation.

d. Give a 3NF decomposition of r based on the canonical cover.

e. Give a BCNF decomposition of r using the original set of functional dependencies.

f. Can you get the same BCNF decomposition of r as above, using the canonical cover?

Answer:

a. $B \rightarrow BD$ (third dependency)
$BD \rightarrow ABD$ (fourth dependency)
$ABD \rightarrow ABCD$ (first dependency)
$ABCD \rightarrow ABCDE$ *(second dependency)*

Thus, $B^+ = ABCDE$

b. Prove (using Armstrong's axioms) that AF is a superkey.

$A \rightarrow BCD$ (Given)
$A \rightarrow ABCD$ (Augmentation with A)
$BC \rightarrow DE$ (Given)
$ABCD \rightarrow ABCDE$ (Augmentation with ABCD)
$A \rightarrow ABCDE$ (Transitivity)
$AF \rightarrow ABCDEF$ (Augmentation with F)

c. We see that D is extraneous in dep. 1 and 2, because of dep. 3. Removing these two, we get the new set of rules

$$A \rightarrow BC$$
$$BC \rightarrow E$$
$$B \rightarrow D$$
$$D \rightarrow A$$

Now notice that B^+ is $ABCDE$, and in particular, the FD $B \rightarrow E$ can be determined from this set. Thus, the attribute C is extraneous in

the third dependency. Removing this attribute, and combining with the third FD, we get the final canonical cover as :

$$A \rightarrow BC$$
$$B \rightarrow DE$$
$$D \rightarrow A$$

Here, no attribute is extraneous in any FD.

d. We see that there is no FD in the canonical cover such that the set of attributes is a subset of any other FD in the canonical cover. Thus, each each FD gives rise to its own relation, giving

$$r_1(A, B, C)$$
$$r_2(B, D, E)$$
$$r_3(D, A)$$

Now the attribute F is not dependent on any attribute. Thus, it must be a part of every superkey. Also, none of the relations in the above schema have F, and hence, none of them have a superkey. Thus, we need to add a new relation with a superkey.

$$r_4(A, F)$$

e. We start with

$$r(A, B, C, D, E, F)$$

We see that the relation is not in BCNF because of the first FD. Hence, we decompose it accordingly to get

$$r_1(A, B, C, D)\ r_2(A, E, F)$$

Now we notice that $A \rightarrow E$ is an FD in F^+, and causes r_2 to violate BCNF. Once again, decomposing r_2 gives

$$r_1(A, B, C, D)\ r_2(A, F)\ r_3(A, E)$$

This schema is now in BCNF.

f. Can you get the same BCNF decomposition of r as above, using the canonical cover?
If we use the functional dependencies in the preceding canonical cover directly, we cannot get the above decomposition. However, we can infer the original dependencies from the canonical cover, and if we use those for BCNF decomposition, we would be able to get the same decomposition.

8.30 List the three design goals for relational databases, and explain why each is desirable.

Answer: The three design goals are lossless-join decompositions, dependency preserving decompositions, and minimization of repetition of

information. They are desirable so we can maintain an accurate database, check correctness of updates quickly, and use the smallest amount of space possible.

8.31 In designing a relational database, why might we choose a non-BCNF design?

Answer: BCNF is not always dependency preserving. Therefore, we may want to choose another normal form (specifically, 3NF) in order to make checking dependencies easier during updates. This would avoid joins to check dependencies and increase system performance.

8.32 Given the three goals of relational-database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Practice Exercise 8.17 for the definition of 2NF.)

Answer: The three design goals of relational databases are to avoid

- Repetition of information
- Inability to represent information
- Loss of information.

2NF does not prohibit as much repetition of information since the schema (A, B, C) with dependencies $A \rightarrow B$ and $B \rightarrow C$ is allowed under 2NF, although the same (B, C) pair could be associated with many A values, needlessly duplicating C values. To avoid this we must go to 3NF. Repetition of information is allowed in 3NF in some but not all of the cases where it is allowed in 2NF. Thus, in general, 3NF reduces repetition of information. Since we can always achieve a lossless join 3NF decomposition, there is no loss of information needed in going from 2NF to 3NF.

Note that the decomposition $\{(A, B), (B, C)\}$ is a dependency-preserving and lossless-loin 3NF decomposition of the schema (A, B, C). However, in case we choose this decomposition, retrieving information about the relationship between A, B and C requires a join of two relations, which is avoided in the corresponding 2NF decomposition.

Thus, the decision of which normal form to choose depends upon how the cost of dependency checking compares with the cost of the joins. Usually, the 3NF would be preferred. Dependency checks need to be made with *every* insert or update to the instances of a 2NF schema, whereas, only some queries will require the join of instances of a 3NF schema.

8.33 Given a relational schema $r(A, B, C, D)$, does $A \twoheadrightarrow BC$ logically imply $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$? If yes prove it, else give a counter example.

Answer: $A \twoheadrightarrow BC$ holds on the following table:

r :

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_1	c_1	d_2
a_1	b_2	c_2	d_1

If $A \twoheadrightarrow B$, then we know that there exists t_1 and t_3 such that $t_1[B] = t_3[B]$. Thus, we must choose one of the following for t_1 and t_3:

- $t_1 = r_1$ and $t_3 = r_3$, or $t_1 = r_3$ and $t_3 = r_1$:
  Choosing either $t_2 = r_2$ or $t_2 = r_4$, $t_3[C] \neq t_2[C]$.
- $t_1 = r_2$ and $t_3 = r_4$, or $t_1 = r_4$ and $t_3 = r_2$:
  Choosing either $t_2 = r_1$ or $t_2 = r_3$, $t_3[C] \neq t_2[C]$.

Therefore, the condition $t_3[C] = t_2[C]$ can not be satisfied, so the conjecture is false.

8.34 Explain why 4NF is a normal form more desirable than BCNF.

Answer: 4NF is more desirable than BCNF because it reduces the repetition of information. If we consider a BCNF schema not in 4NF (see Practice Exercise 7.16), we observe that decomposition into 4NF does not lose information provided that a lossless join decomposition is used, yet redundancy is reduced.