CHAPTER 1

Introduction

Practice Exercises

1.1 This chapter has described several major advantages of a database system. What are two disadvantages?

Answer:
Two disadvantages associated with database systems are listed below.

a. Setup of the database system requires more knowledge, money, skills, and time.

b. The complexity of the database may result in poor performance.

1.2 List five ways in which the type declaration system of a language such as Java or C++ differs from the data definition language used in a database.

Answer:

a. Executing an action in the DDL results in the creation of an object in the database; in contrast, a programming language type declaration is simply an abstraction used in the program.

b. Database DDLs allow consistency constraints to be specified, which programming language type systems generally do not allow. These include domain constraints and referential integrity constraints.

c. Database DDLs support authorization, giving different access rights to different users. Programming language type systems do not provide such protection (at best, they protect attributes in a class from being accessed by methods in another class).

d. Programming language type systems are usually much richer than the SQL type system. Most databases support only basic types such as different types of numbers and strings, although some databases do support some complex types such as arrays and objects.

e. A database DDL is focused on specifying types of attributes of relations; in contrast, a programming language allows objects and collections of objects to be created.

1.3 List six major steps that you would take in setting up a database for a particular enterprise.

Answer:
Six major steps in setting up a database for a particular enterprise are:

- Define the high-level requirements of the enterprise (this step generates a document known as the system requirements specification.)
- Define a model containing all appropriate types of data and data relationships.
- Define the integrity constraints on the data.
- Define the physical level.
- For each known problem to be solved on a regular basis (e.g., tasks to be carried out by clerks or web users), define a user interface to carry out the task, and write the necessary application programs to implement the user interface.
- Create/initialize the database.

1.4 Suppose you want to build a video site similar to YouTube. Consider each of the points listed in Section 1.2 as disadvantages of keeping data in a file-processing system. Discuss the relevance of each of these points to the storage of actual video data, and to metadata about the video, such as title, the user who uploaded it, tags, and which users viewed it.

Answer:

- **Data redundancy and inconsistency**. This would be relevant to metadata to some extent, although not to the actual video data, which are not updated. There are very few relationships here, and none of them can lead to redundancy.
- **Difficulty in accessing data**. If video data are only accessed through a few predefined interfaces, as is done in video sharing sites today, this will not be a problem. However, if an organization needs to find video data based on specific search conditions (beyond simple keyword queries), if metadata were stored in files it would be hard to find relevant data without writing application programs. Using a database would be important for the task of finding data.
- **Data isolation**. Since data are not usually updated, but instead newly created, data isolation is not a major issue. Even the task of keeping track of

who has viewed what videos is (conceptually) append only, again making isolation not a major issue. However, if authorization is added, there may be some issues of concurrent updates to authorization information.

- **Integrity problems**. It seems unlikely there are significant integrity constraints in this application, except for primary keys. If the data are distributed, there may be issues in enforcing primary key constraints. Integrity problems are probably not a major issue.
- **Atomicity problems**. When a video is uploaded, metadata about the video and the video should be added atomically, otherwise there would be an inconsistency in the data. An underlying recovery mechanism would be required to ensure atomicity in the event of failures.
- **Concurrent-access anomalies**. Since data are not updated, concurrent access anomalies would be unlikely to occur.
- **Security problems**. These would be an issue if the system supported authorization.

1.5 Keyword queries used in web search are quite different from database queries. List key differences between the two, in terms of the way the queries are specified and in terms of what is the result of a query.

Answer:
Queries used in the web are specified by providing a list of keywords with no specific syntax. The result is typically an ordered list of URLs, along with snippets of information about the content of the URLs. In contrast, database queries have a specific syntax allowing complex queries to be specified. And in the relational world the result of a query is always a table.

CHAPTER 2

Introduction to the Relational Model

Practice Exercises

2.1 Consider the employee database of Figure 2.17. What are the appropriate primary keys?

Answer:

The appropriate primary keys are shown below:

employee ($\underline{\textit{person\_name}}$, *street*, *city*)
works ($\underline{\textit{person\_name}}$, *company_name*, *salary*)
company ($\underline{\textit{company\_name}}$, *city*)

2.2 Consider the foreign-key constraint from the *dept_name* attribute of *instructor* to the *department* relation. Give examples of inserts and deletes to these relations that can cause a violation of the foreign-key constraint.

Answer:

- Inserting a tuple:

  (10111, Ostrom, Economics, 110000)

---

employee ($\underline{\textit{ID}}$, *person_name*, *street*, *city*)
works ($\underline{\textit{ID}}$, *company_name*, *salary*)
company ($\underline{\textit{company\_name}}$, *city*)

Figure 2.17 Employee database.

into the *instructor* table, where the *department* table does not have the department Economics, would violate the foreign-key constraint.

- Deleting the tuple:

(Biology, Watson, 90000)

from the *department* table, where at least one student or instructor tuple has *dept_name* as Biology, would violate the foreign-key constraint.

2.3 Consider the *time_slot* relation. Given that a particular time slot can meet more than once in a week, explain why *day* and *start_time* are part of the primary key of this relation, while *end_time* is not.

Answer:
The attributes *day* and *start_time* are part of the primary key since a particular class will most likely meet on several different days and may even meet more than once in a day. However, *end_time* is not part of the primary key since a particular class that starts at a particular time on a particular day cannot end at more than one time.

2.4 In the instance of *instructor* shown in Figure 2.1, no two instructors have the same name. From this, can we conclude that *name* can be used as a superkey (or primary key) of *instructor*?

Answer:
No. For this possible instance of the instructor table the names are unique, but in general this may not always be the case (unless the university has a rule that two instructors cannot have the same name, which is a rather unlikey scenario).

2.5 What is the result of first performing the Cartesian product of *student* and *advisor*, and then performing a selection operation on the result with the predicate *s_id* = ID? (Using the symbolic notation of relational algebra, this query can be written as $\sigma_{s\_id=ID}(student \times advisor)$.)

Answer:
The result attributes include all attribute values of *student* followed by all attributes of *advisor*. The tuples in the result are as follows: For each student who has an advisor, the result has a row containing that student's attributes, followed by an *s_id* attribute identical to the student's ID attribute, followed by the *i_id* attribute containing the ID of the students advisor.

Students who do not have an advisor will not appear in the result. A student who has more than one advisor will appear a corresponding number of times in the result.

2.6 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the name of each employee who lives in city "Miami".

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

Figure 2.18 Bank database.

b. Find the name of each employee whose salary is greater than $100000.

c. Find the name of each employee who lives in "Miami" and whose salary is greater than $100000.

Answer:

a. $\Pi_{person\_name}\ (\sigma_{city\,=\,\text{"Miami"}}\ (employee))$

b. $\Pi_{person\_name}\ (\sigma_{salary\,>\,100000}\ (employee \bowtie works))$

c. $\Pi_{person\_name}\ (\sigma_{city\,=\,\text{"Miami"}\,\wedge\,salary>100000}\ (employee \bowtie works))$

2.7 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find the name of each branch located in "Chicago".

b. Find the ID of each borrower who has a loan in branch "Downtown".

Answer:

a. $\Pi_{branch\_name}\ (\sigma_{branch\_city\,=\,\text{"Chicago"}}\ (branch))$

b. $\Pi_{ID}\ (\sigma_{branch\_name\,=\,\text{"Downtown"}}\ (borrower \bowtie_{borrower.loan\_number=loan.loan\_number} loan))$.

2.8 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who does not work for "BigBank".

b. Find the ID and name of each employee who earns at least as much as every employee in the database.

Answer:

a. To find employees who do not work for BigBank, we first find all those who *do* work for BigBank. Those are exactly the employees *not* part of the desired result. We then use set difference to find the set of all employees minus those employees that should not be in the result.

$$\Pi_{ID,person\_name}(employee) -$$
$$\Pi_{ID,person\_name}$$
$$(employee \bowtie_{employee.ID=works.ID} (\sigma_{company\_name=}\}\}\text{BigBank}''(works)))$$

b. We use the same approach as in part *a* by first finding those employess who do not earn the highest salary, or, said differently, for whom some other employee earns more. Since this involves comparing two employee salary values, we need to reference the *employee* relation twice and therefore use renaming.

$$\Pi_{ID,person\_name}(employee) -$$
$$\Pi_{A.ID,A.person\_name}(\rho_A(employee) \bowtie_{A.salary<B.salary} \rho_B(employee))$$

2.9 The **division operator** of relational algebra, "÷", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Given a tuple t, let $t[S]$ denote the projection of tuple t on the attributes in S. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

  a. $t_r[S] = t_s[S]$

  b. $t_r[R - S] = t$

Given the above definition:

a. Write a relational algebra expression using the division operator to find the IDs of all students who have taken all Comp. Sci. courses. (Hint: project *takes* to just ID and *course_id*, and generate the set of all Comp. Sci. *course_id*s using a select expression, before doing the division.)

b. Show how to write the above query in relational algebra, without using division. (By doing so, you would have shown how to define the division operation using the other relational algebra operations.)

Answer:

a. $\Pi_{ID}(\Pi_{ID,course\_id}(takes) \div \Pi_{course\_id}(\sigma_{dept\_name=\text{'Comp. Sci'}}(course))$

b. The required expression is as follows:

$$r \leftarrow \Pi_{ID,course\_id}(takes)$$

$$s \leftarrow \Pi_{course\_id}(\sigma_{dept\_name=\text{'Comp. Sci'}}(course))$$

$$\Pi_{ID}\ (takes)\ -\ \Pi_{ID}\ ((\Pi_{ID}\ (takes)\ \times\ s)\ -\ r)$$

In general, let $r(R)$ and $s(S)$ be given, with $S \subseteq R$. Then we can express the division operation using basic relational algebra operations as follows:

$$r \div s = \Pi_{R-S}\ (r)\ -\ \Pi_{R-S}\ ((\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r))$$

To see that this expression is true, we observe that $\Pi_{R-S}\ (r)$ gives us all tuples t that satisfy the first condition of the definition of division. The expression on the right side of the set difference operator

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r))$$

serves to eliminate those tuples that fail to satisfy the second condition of the definition of division. Let us see how it does so. Consider $\Pi_{R-S}\ (r) \times s$. This relation is on schema R, and pairs every tuple in $\Pi_{R-S}\ (r)$ with every tuple in s. The expression $\Pi_{R-S,S}(r)$ merely reorders the attributes of r.

Thus, $(\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r)$ gives us those pairs of tuples from $\Pi_{R-S}\ (r)$ and s that do not appear in r. If a tuple t_j is in

$$\Pi_{R-S}\ ((\Pi_{R-S}\ (r)\ \times\ s)\ -\ \Pi_{R-S,S}(r))$$

then there is some tuple t_s in s that does not combine with tuple t_j to form a tuple in r. Thus, t_j holds a value for attributes $R - S$ that does not appear in $r \div s$. It is these values that we eliminate from $\Pi_{R-S}\ (r)$.

CHAPTER 3

Introduction to SQL

Practice Exercises

3.1 Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above web site.)

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

c. Find the highest salary of any instructor.

d. Find all instructors earning the highest salary (there may be more than one with the same salary).

e. Find the enrollment of each section that was offered in Fall 2017.

f. Find the maximum enrollment, across all sections, in Fall 2017.

g. Find the sections that had the maximum enrollment in Fall 2017.

Answer:

a. Find the titles of courses in the Comp. Sci. department that have 3 credits.

```
select  title
from    course
where   dept_name = 'Comp. Sci.' and credits = 3
```

b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.
This query can be answered in several different ways. One way is as follows.

```
select  distinct takes.ID
from    takes, instructor, teaches
where   takes.course_id = teaches.course_id and
        takes.sec_id = teaches.sec_id and
        takes.semester = teaches.semester and
        takes.year = teaches.year and
        teaches.id = instructor.id and
        instructor.name = 'Einstein'
```

c. Find the highest salary of any instructor.

```
select max(salary)
from   instructor
```

d. Find all instructors earning the highest salary (there may be more than one with the same salary).

```
select  ID, name
from    instructor
where   salary = (select max(salary) from instructor)
```

e. Find the enrollment of each section that was offered in Fall 2017.

```
select  course_id, sec_id,
        (select count(ID)
        from    takes
        where   takes.year = section.year
                and takes.semester = section.semester
                and takes.course_id = section.course_id
                and takes.sec_id = section.sec_id)
        as enrollment
from    section
where   semester = 'Fall'
and     year = 2017
```

Note that if the result of the subquery is empty, the aggregate function **count** returns a value of 0.

One way of writing the query might appear to be:

```
select    takes.course_id, takes.sec_id, count(ID)
from      section, takes
where     takes.course_id = section.course_id
          and takes.sec_id = section.sec_id
          and takes.semester = section.semester
          and takes.year = section.year
          and takes.semester = 'Fall'
          and takes.year = 2017
group by takes.course_id, takes.sec_id
```

But note that if a section does not have any students taking it, it would not appear in the result. One way of ensuring such a section appears with a count of 0 is to use the **outer join** operation, covered in Chapter 4.

f. Find the maximum enrollment, across all sections, in Fall 2017.
One way of writing this query is as follows:

```
select  max(enrollment)
from    (select  count(ID) as enrollment
         from    section, takes
         where   takes.year = section.year
                 and takes.semester = section.semester
                 and takes.course_id = section.course_id
                 and takes.sec_id = section.sec_id
                 and takes.semester = 'Fall'
                 and takes.year = 2017
                 group by takes.course_id, takes.sec_id)
```

As an alternative to using a nested subquery in the **from** clause, it is possible to use a **with** clause, as illustrated in the answer to the next part of this question.

A subtle issue in the above query is that if no section had any enrollment, the answer would be empty, not 0. We can use the alternative using a subquery, from the previous part of this question, to ensure the count is 0 in this case.

g. Find the sections that had the maximum enrollment in Fall 2017.
The following answer uses a **with** clause, simplifying the query.

```
with sec_enrollment as (
    select   takes.course_id, takes.sec_id, count(ID) as enrollment
    from     section, takes
    where    takes.year = section.year
             and takes.semester = section.semester
             and takes.course_id = section.course_id
             and takes.sec_id = section.sec_id
             and takes.semester = 'Fall'
             and takes.year = 2017
    group by takes.course_id, takes.sec_id)
select  course_id, sec_id
from    sec_enrollment
where   enrollment = (select max(enrollment) from sec_enrollment)
```

It is also possible to write the query without the **with** clause, but the subquery to find enrollment would get repeated twice in the query.
While not incorrect to add **distinct** in the **count**, it is not necessary in light of the primary key constraint on *takes*.

3.2 Suppose you are given a relation *grade_points*(grade, *points*) that provides a conversion from letter grades in the *takes* relation to numeric scores; for example, an "A" grade could be specified to correspond to 4 points, an "A−" to 3.7 points, a "B+" to 3.3 points, a "B" to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received.

Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no *takes* tuple has the *null* value for *grade*.

a. Find the total grade points earned by the student with ID '12345', across all courses taken by the student.

b. Find the grade point average (*GPA*) for the above student, that is, the total grade points divided by the total credits for the associated courses.

c. Find the ID and the grade-point average of each student.

d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly.

Answer:

a. Find the total grade-points earned by the student with ID '12345', across all courses taken by the student.

```
select sum(credits * points)
from takes, course, grade_points
where takes.grade = grade_points.grade
    and takes.course_id = course.course_id
    and ID = '12345'
```

In the above query, a student who has not taken any course would not have any tuples, whereas we would expect to get 0 as the answer. One way of fixing this problem is to use the **outer join** operation, which we study later in Chapter 4. Another way to ensure that we get 0 as the answer is via the following query:

```
(select  sum(credits * points)
from     takes, course, grade_points
where    takes.grade = grade_points.grade
         and takes.course_id = course.course_id
         and ID= '12345')
union
(select  0
from     student
where    ID = '12345' and
         not exists ( select * from takes where ID = '12345'))
```

b. Find the grade point average (*GPA*) for the above student, that is, the total grade-points divided by the total credits for the associated courses.

```
select   sum(credits * points)/sum(credits) as GPA
from     takes, course, grade_points
where    takes.grade = grade_points.grade
         and takes.course_id = course.course_id
         and ID= '12345'
```

As before, a student who has not taken any course would not appear in the above result; we can ensure that such a student appears in the result by using the modified query from the previous part of this question. However, an additional issue in this case is that the sum of credits would also be 0, resulting in a divide-by-zero condition. In fact, the only meaningful way of defining the *GPA* in this case is to define it as *null*. We can ensure that such a student appears in the result with a null *GPA* by adding the following **union** clause to the above query.

```
union
(select null as GPA
from    student
where   ID = '12345' and
        not exists ( select * from takes where ID = '12345'))
```

c. Find the ID and the grade-point average of each student.

```
select    ID, sum(credits * points)/sum(credits) as GPA
from      takes, course, grade_points
where     takes.grade = grade_points.grade
          and takes.course_id = course.course_id
group by ID
```

Again, to handle students who have not taken any course, we would have to add the following **union** clause:

```
union
(select ID, null as GPA
from    student
where   not exists ( select * from takes where takes.ID = student.ID))
```

d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly. The queries listed above all include a test of equality on *grade* between *grade_points* and *takes*. Thus, for any *takes* tuple with a *null* grade, that student's course would be eliminated from the rest of the computation of the result. As a result, the credits of such courses would be eliminated also, and thus the queries would return the correct answer even if some grades are null.

3.3 Write the following inserts, deletes, or updates in SQL, using the university schema.

a. Increase the salary of each instructor in the Comp. Sci. department by 10%.

b. Delete all courses that have never been offered (i.e., do not occur in the *section* relation).

c. Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

Answer:

a. Increase the salary of each instructor in the Comp. Sci. department by 10%.

```
update instructor
set    salary = salary * 1.10
where  dept_name = 'Comp. Sci.'
```

b. Delete all courses that have never been offered (that is, do not occur in the *section* relation).

person (<u>*driver_id*</u>, *name*, *address*)
car (<u>*license_plate*</u>, *model*, *year*)
accident (<u>*report_number*</u>, *year*, *location*)
owns (<u>*driver_id*</u>, <u>*license_plate*</u>)
participated (<u>*report_number*</u>, <u>*license_plate*</u>, *driver_id*, *damage_amount*)

Figure 3.17 Insurance database

```
delete  from course
where   course_id not in
        (select course_id from section)
```

c. Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of $10,000.

```
insert into instructor
select  ID, name, dept_name, 10000
from    student
where   tot_cred > 100
```

3.4 Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the total number of people who owned cars that were involved in accidents in 2017.

b. Delete all year-2010 cars belonging to the person whose ID is '12345'.

Answer:

a. Find the total number of people who owned cars that were involved in accidents in 2017.
Note: This is not the same as the total number of accidents in 2017. We must count people with several accidents only once. Furthermore, note that the question asks for owners, and it might be that the owner of the car was not the driver actually involved in the accident.

```
select  count (distinct person.driver_id)
from    accident, participated, person, owns
where   accident.report_number = participated.report_number
        and owns.driver_id = person.driver_id
        and owns.license_plate = participated.license_plate
        and year = 2017
```

b. Delete all year-2010 cars belonging to the person whose ID is '12345'.

```
delete car
where year = 2010 and license_plate in
    (select license_plate
     from owns o
     where o.driver_id = '12345')
```

Note: The *owns*, *accident* and *participated* records associated with the deleted cars still exist.

3.5 Suppose that we have a relation *marks*(*ID*, *score*) and we wish to assign grades to students based on the score as follows: grade F if $score < 40$, grade C if $40 \le score < 60$, grade B if $60 \le score < 80$, and grade A if $80 \le score$. Write SQL queries to do the following:

a. Display the grade for each student, based on the *marks* relation.

b. Find the number of students with each grade.

Answer:

a. Display the grade for each student, based on the *marks* relation.

```
select ID,
       case
           when score < 40 then 'F'
           when score < 60 then 'C'
           when score < 80 then 'B'
           else 'A'
       end
from   marks
```

b. Find the number of students with each grade.

```
with     grades as
(
select   ID,
         case
             when score < 40 then 'F'
             when score < 60 then 'C'
             when score < 80 then 'B'
             else 'A'
         end as grade
from     marks
)
select   grade, count(ID)
from     grades
group by grade
```

As an alternative, the **with** clause can be removed, and instead the definition of *grades* can be made a subquery of the main query.

3.6 The SQL **like** operator is case sensitive (in most systems), but the lower() function on strings can be used to perform case-insensitive matching. To show how, write a query that finds departments whose names contain the string “sci” as a substring, regardless of the case.

Answer:

```
select   dept_name
from     department
where    lower(dept_name) like '%sci%'
```

3.7 Consider the SQL query

```
select p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1
```

Under what conditions does the preceding query select values of $p.a1$ that are either in $r1$ or in $r2$? Examine carefully the cases where either $r1$ or $r2$ may be empty.

Answer:
The query selects those values of *p.a1* that are equal to some value of *r1.a1* or *r2.a1* if and only if both *r1* and *r2* are non-empty. If one or both of *r1* and *r2* are empty, the Cartesian product of *p, r1* and *r2* is empty, hence the result of the query is empty. If *p* itself is empty, the result is empty.

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

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

Figure 3.18 Banking database.

a. Find the ID of each customer of the bank who has an account but not a loan.

b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

Answer:

a. Find the ID of each customer of the bank who has an account but not a loan.

```
(select ID
from    depositor)
except
(select ID
from    borrower)
```

b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

```
select  F.ID
from    customer as F, customer as S
where   F.customer_street = S.customer_street
        and F.customer_city = S.customer_city
        and S.customer_id  = '12345'
```

c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

select **distinct** *branch_name*
from *account, depositor, customer*
where *customer.id* = *depositor.id*
and *depositor.account_number* = *account.account_number*
and *customer_city* = 'Harrison'

3.9 Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation".

b. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation" and earns more than $10000.

c. Find the ID of each employee who does not work for "First Bank Corporation".

d. Find the ID of each employee who earns more than every employee of "Small Bank Corporation".

e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which "Small Bank Corporation" is located.

f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).

g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

Answer:

a. Find the ID, name, and city of residence of each employee who works for "First Bank Corporation".

employee (<u>*ID*</u>, *person_name*, *street*, *city*)
works (<u>*ID*</u>, *company_name*, *salary*)
company (<u>*company_name*</u>, *city*)
manages (<u>*ID*</u>, *manager_id*)

Figure 3.19 Employee database.

```
select e.ID, e.person_name, city
from employee as e, works as w
where w.company_name = 'First Bank Corporation' and
      w.ID = e.ID
```

b. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation” and earns more than $10000.

```
select *
from employee
where ID in
   (select ID
    from works
    where company_name = 'First Bank Corporation' and salary > 10000)
```

This could be written also in the style of the answer to part *a*.

c. Find the ID of each employee who does not work for “First Bank Corporation”.

```
select ID
from works
where company_name <> 'First Bank Corporation'
```

If one allows people to appear in *employee* without appearing also in *works*, the solution is slightly more complicated. An outer join as discussed in Chapter 4 could be used as well.

```
select ID
from employee
where ID not in
   (select ID
    from works
    where company_name = 'First Bank Corporation')
```

d. Find the ID of each employee who earns more than every employee of “Small Bank Corporation”.

```
select ID
from works
where salary > all
   (select salary
    from works
    where company_name = 'Small Bank Corporation')
```

If people may work for several companies and we wish to consider the *total* earnings of each person, the problem is more complex. But note that the fact that ID is the primary key for *works* implies that this cannot be the case.

e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which "Small Bank Corporation" is located.

```
select S.company_name
from company as S
where not exists ((select city
                   from company
                   where company_name = 'Small Bank Corporation')
                  except
                  (select city
                   from company as T
                   where S.company_name = T.company_name))
```

f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).

```
select company_name
from works
group by company_name
having count (distinct ID) >= all
    (select count (distinct ID)
     from works
     group by company_name)
```

g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

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

3.10 Consider the relational database of Figure 3.19. Give an expression in SQL for each of the following:

a. Modify the database so that the employee whose ID is '12345' now lives in "Newtown".

b. Give each manager of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than $100000; in such cases, give only a 3 percent raise.

Answer:

a. Modify the database so that the employee whose ID is '12345' now lives in "Newtown".

```
update employee
set city = 'Newtown'
where ID = '12345'
```

b. Give each manager of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than $100000; in such cases, give only a 3 percent raise.

```
update works T
set T.salary = T.salary * 1.03
where T.ID in (select manager_id
               from manages)
      and T.salary * 1.1 >  100000
      and T.company_name = 'First Bank Corporation'

update works T
set T.salary = T.salary * 1.1
where T.ID in (select manager_id
               from manages)
      and T.salary * 1.1 <= 100000
      and T.company_name = 'First Bank Corporation'
```

The above updates would give different results if executed in the opposite order. We give below a safer solution using the **case** statement.

```
update works T
set T.salary = T.salary *
    (case
        when (T.salary * 1.1 > 100000) then 1.03
        else 1.1
    end)
where T.ID in (select manager_id
               from manages) and
      T.company_name = 'First Bank Corporation'
```

CHAPTER 4

Intermediate SQL

Practice Exercises

4.1 Consider the following SQL query that seeks to find a list of titles of all courses taught in Spring 2017 along with the name of the instructor.

> **select** name, title
> **from** *instructor* **natural join** *teaches* **natural join** *section* **natural join** *course*
> **where** *semester* = 'Spring' **and** year = 2017

What is wrong with this query?

Answer:
Although the query is syntactically correct, it does not compute the expected answer because *dept_name* is an attribute of both *course* and *instructor*. As a result of the natural join, results are shown only when an instructor teaches a course in her or his own department.

4.2 Write the following queries in SQL:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "—".

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

Answer:

a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

```
select ID, count(sec_id) as Number_of_sections
from instructor natural left outer join teaches
group by ID
```

The above query should not be written using count(*) since that would count null values also. It could be written using any attribute from *teaches* which does not occur in *instructor*, which would be correct although it may be confusing to the reader. (Attributes that occur in *instructor* would not be null even if the instructor has not taught any section.)

b. Write the same query as above, but using a scalar subquery, and not using outerjoin.

```
select ID,
    (select count(*) as Number_of_sections
    from teaches T where T.id = I.id)
from instructor I
```

c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to "—".

```
select course_id, sec_id, ID,
    decode(name, null, '—', name) as name
from (section natural left outer join teaches)
    natural left outer join instructor
where semester='Spring' and year= 2018
```

The query may also be written using the **coalesce** operator, by replacing **decode**(..) with **coalesce**(*name*, '—'). A more complex version of the query can be written using union of join result with another query that uses a subquery to find courses that do not match; refer to Exercise 4.3.

d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

> **select** *dept_name*, **count**(*ID*)
> **from** *department* **natural left outer join** *instructor*
> **group by** *dept_name*

4.3 Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.

a. **select** * **from** *student* **natural left outer join** *takes*

b. **select** * **from** *student* **natural full outer join** *takes*

Answer:

a. **select** * **from** *student* **natural left outer join** *takes*
can be rewritten as:

> **select** * **from** *student* **natural join** *takes*
> **union**
> **select** *ID*, *name*, *dept_name*, *tot_cred*, *null*, *null*, *null*, *null*, *null*
> **from** *student S1* **where not exists**
> (**select** *ID* **from** *takes T1* **where** *T1.id* = *S1.id*)

b. **select** * **from** *student* **natural full outer join** *takes*
can be rewritten as:

> (**select** * **from** *student* **natural join** *takes*)
> **union**
> (**select** *ID*, *name*, *dept_name*, *tot_cred*, *null*, *null*, *null*, *null*, *null*
> **from** *student S1*
> **where not exists**
> (**select** *ID* **from** *takes T1* **where** *T1.id* = *S1.id*))
> **union**
> (**select** *ID*, *null*, *null*, *null*, *course_id*, *sec_id*, *semester*, *year*, *grade*
> **from** *takes T1*
> **where not exists**
> (**select** *ID* **from** *student S1* **where***T1.id* = *S1.id*))

4.4 Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**.

a. Give instances of relations r, s, and t such that in the result of
(r **natural left outer join** s) **natural left outer join** t
attribute C has a null value but attribute D has a non-null value.

b. Are there instances of r, s, and t such that the result of
r **natural left outer join** (s **natural left outer join** t)
has a null value for C but a non-null value for D? Explain why or why not.

Answer:

a. Consider $r = (a, b)$, $s = (b1, c1)$, $t = (b, d)$. The second expression would give $(a, b, null, d)$.

b. Since s **natural left outer join** t is computed first, the absence of nulls is both s and t implies that each tuple of the result can have D null, but C can never be null.

4.5 **Testing SQL queries**: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

a. In Section 4.1.1 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result it unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.

b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation for each foreign key. Explain why, using an example query on the university database.

c. When creating test databases, it is important to create tuples with null values for foreign-key attributes, provided the attribute is nullable (SQL allows foreign-key attributes to take on null values, as long as they are not part of the primary key and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: Use the queries from Exercise 4.2.

Answer:

a. Consider the case where a professor in the Physics department teaches an Elec. Eng. course. Even though there is a valid corresponding entry in *teaches*, it is lost in the natural join of *instructor*, *teaches* and *course*, since the instructor's department name does not match the department name of the course. A dataset corresponding to the same is:

instructor = {('12345','Gauss', 'Physics', 10000)}
teaches = {('12345', 'EE321', 1, 'Spring', 2017)}
course = {('EE321', 'Magnetism', 'Elec. Eng.', 6)}

b. The query in question 4.2(a) is a good example for this. Instructors who have not taught a single course should have number of sections as 0 in the query result. (Many other similar examples are possible.)

c. Consider the query

select * **from** *teaches* **natural join** *instructor*;

In this query, we would lose some sections if *teaches*.ID is allowed to be *null* and such tuples exist. If, just because *teaches*.ID is a foreign key to *instructor*, we did not create such a tuple, the error in the above query would not be detected.

4.6 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade_points*(grade, *points*) to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.

Answer:
We should not add credits for courses with a null grade; further, to correctly handle the case where a student has not completed any course, we should make sure we don't divide by zero, and should instead return a null value.

We break the query into a subquery that finds sum of credits and sum of credit-grade-points, taking null grades into account The outer query divides the above to get the average, taking care of divide by zero.

```
create view student_grades(ID, GPA) as
    select ID, credit_points / decode(credit_sum, 0, null, credit_sum)
    from ((select ID, sum(decode(grade, null, 0, credits)) as credit_sum,
              sum(decode(grade, null, 0, credits*points)) as credit_points
              from(takes natural join course) natural left outer join grade_points
              group by ID)
    union
    select  ID, null, null
    from    student
    where   ID not in (select ID from takes))
```

The view defined above takes care of *null* grades by considering the credit points to be 0 and not adding the corresponding credits in *credit_sum*.

employee (<u>*ID*</u>, *person_name*, *street*, *city*)
works (<u>*ID*</u>, *company_name*, *salary*)
company (<u>*company_name*</u>, *city*)
manages (<u>*ID*</u>, *manager_id*)

Figure 4.12 Employee database.

The query above ensures that a student who has not taken any course with non-null credits, and has *credit_sum* = 0 gets a GPA of *null*. This avoids the division by zero, which would otherwise have resulted.

In systems that do note support **decode**, an alternative is the **case** construct. Using **case**, the solution would be written as follows:

```
create view student_grades(ID, GPA) as
    select ID, credit_points / (case when credit_sum = 0 then null
                                else credit_sum end)
    from ((select ID, sum (case when grade is null then 0
                                else credits end) as credit_sum,
                                sum (case when grade is null then 0
                                else credits*points end) as credit_points
            from(takes natural join course) natural left outer join grade_points
            group by ID)
    union
    select  ID, null, null
    from    student
    where   ID not in (select ID from takes))
```

An alternative way of writing the above query would be to use *student* **natural left outer join** *gpa*, in order to consider students who have not taken any course.

4.7 Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer:
Plese see **??**.

Note that alternative data types are possible. Other choices for **not null** attributes may be acceptable.

4.8 As discussed in Section 4.4.8, we expect the constraint "an instructor cannot teach sections in two different classrooms in a semester in the same time slot" to hold.

```
create table employee
   (ID            numeric(6,0),
    person_name   char(20),
    street        char(30),
    city          char(30),
    primary key (ID))
```

```
create table works
   (ID            numeric(6,0),
    company_name  char(15),
    salary        integer,
    primary key (ID),
    foreign key (ID) references employee,
    foreign key (company_name) references company)
```

```
create table company
   (company_name  char(15),
    city          char(30),
    primary key (company_name))
```

```
create table manages
   (ID            numeric(6,0),
    manager_iid   numeric(6,0),
    primary key (ID),
    foreign key (ID) references employee,
    foreign key (manager_iid) references employee(ID))
```

Figure 4.101 Figure for Exercise 4.7.

a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.

b. Write an SQL assertion to enforce this constraint (as discussed in Section 4.4.8, current generation database systems do not support such assertions, although they are part of the SQL standard).

Answer:

a. Query:

```
select    ID, name, sec_id, semester, year, time_slot_id,
          count(distinct building, room_number)
from      instructor natural join teaches natural join section
group by (ID, name, sec_id, semester, year, time_slot_id)
having    count(building, room_number) > 1
```

Note that the **distinct** keyword is required above. This is to allow two different sections to run concurrently in the same time slot and are taught by the same instructor without being reported as a constraint violation.

b. Query:

```
create assertion check not exists
    ( select ID, name, sec_id, semester, year, time_slot_id,
             count(distinct building, room_number)
    from     instructor natural join teaches natural join section
    group by (ID, name, sec_id, semester, year, time_slot_id)
    having   count(building, room_number) > 1)
```

4.9 SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
   (employee_ID    char(20),
    manager_ID     char(20),
    primary key employee_ID,
    foreign key (manager_ID) references manager(employee_ID)
                                     on delete cascade )
```

Here, *employee_ID* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

Answer:
The tuples of all employees of the manager, at all levels, get deleted as well! This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second-level employee tuples, and so on, till all direct and indirect employee tuples are deleted.

4.10 Given the relations *a*(*name, address, title*) and *b*(*name, address, salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address* and that the solution is correct even if some tuples in *a* and *b* have null values for attributes *name* or *address*.

Answer:

select coalesce(*a.name, b.name*) **as** *name*,
    coalesce(*a.address, b.address*) **as** *address*,
    a.title,
    b.salary
from *a* **full outer join** *b* **on** *a.name* = *b.name* **and**
    a.address = *b.address*

4.11 Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?

Answer: There are many reasons—we list a few here. One might wish to allow a user only to append new information without altering old information. One might wish to allow a user to access a relation but not change its schema. One might wish to limit access to aspects of the database that are not technically data access but instead impact resource utilization, such as creating an index.

4.12 Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?

Answer: Both cases give the same authorization at the time the statement is executed, but the long-term effects differ. If the grant is done based on the role, then the grant remains in effect even if the user who performed the grant leaves and that user's account is terminated. Whether that is a good or bad idea depends on the specific situation, but usually granting through a role is more consistent with a well-run enterprise.

4.13 Consider a view *v* whose definition references only relation *r*.

- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
- If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?
- Give an example of an **insert** operation on a view *v* to add a tuple *t* that is not visible in the result of **select** * **from** *v*. Explain your answer.

Answer:

- No. This allows a user to be granted access to only part of relation *r*.

- Yes. A valid update issued using view *v* must update *r* for the update to be stored in the database.
- Any tuple *t* compatible with the schema for *v* but not satisfying the **where** clause in the definition of view *v* is a valid example. One such example appears in Section 4.2.4.

CHAPTER 6

Database Design using the E-R Model

Practice Exercises

6.1 Construct an E-R diagram for a car insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents. Each insurance policy covers one or more cars and has one or more premium payments associated with it. Each payment is for a particular period of time, and has an associated due date, and the date when the payment was received.

Answer:
One possible E-R diagram is shown in Figure 6.101. Payments are modeled as weak entities since they are related to a specific policy.
Note that the participation of accident in the relationship *participated* is not total, since it is possible that there is an accident report where the participating car is unknown.

6.2 Consider a database that includes the entity sets *student*, *course*, and *section* from the university schema and that additionally records the marks that students receive in different exams of different sections.

a. Construct an E-R diagram that models exams as entities and uses a ternary relationship as part of the design.

b. Construct an alternative E-R diagram that uses only a binary relationship between *student* and *section*. Make sure that only one relationship exists between a particular *student* and *section* pair, yet you can represent the marks that a student gets in different exams.

Answer:

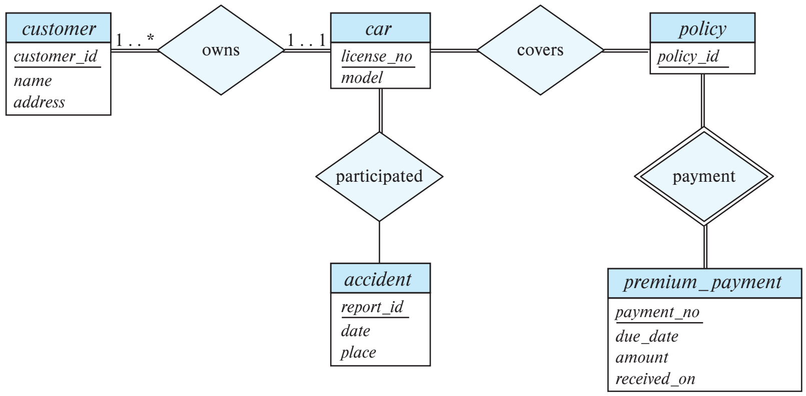


Figure 6.101 E-R diagram for a car insurance company.

a. The E-R diagram is shown in Figure 6.102. Note that an alternative is to model examinations as weak entities related to a section, rather than as strong entities. The marks relationship would then be a binary relationship between *student* and *exam*, without directly involving *section*.

b. The E-R diagram is shown in Figure 6.103. Note that here we have not modeled the name, place, and time of the exam as part of the relationship attributes. Doing so would result in duplication of the information, once per student, and we would not be able to record this information without an associated student. If we wish to represent this information, we need to retain a separate entity corresponding to each exam.

6.3 Design an E-R diagram for keeping track of the scoring statistics of your favorite sports team. You should store the matches played, the scores in each match, the players in each match, and individual player scoring statistics for each match.

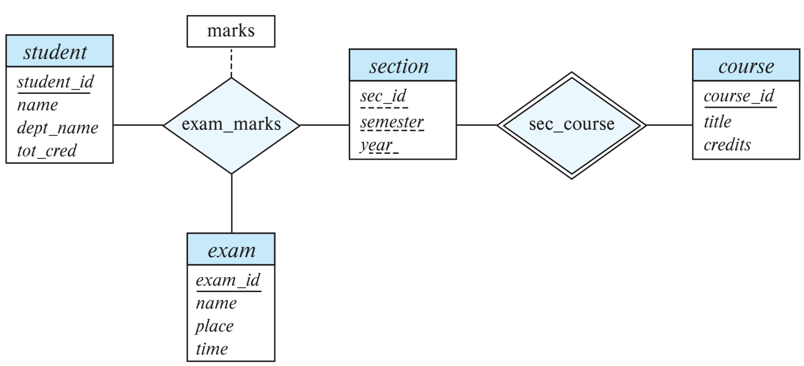


Figure 6.102 E-R diagram for marks database.

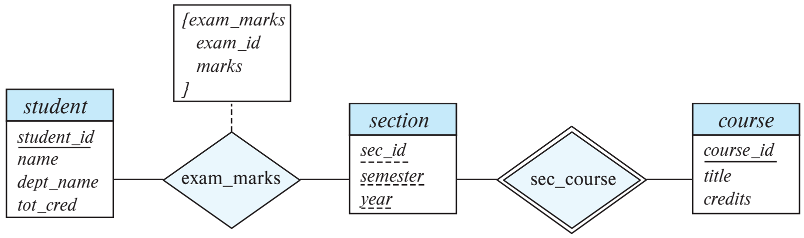


Figure 6.103 Another E-R diagram for marks database.

Summary statistics should be modeled as derived attributes with an explanation as to how they are computed.

Answer:

The diagram is shown in Figure 6.104. The derived attribute *season_score* is computed by summing the score values associated with the *player* entity set via the *played* relationship set.

6.4 Consider an E-R diagram in which the same entity set appears several times, with its attributes repeated in more than one occurrence. Why is allowing this redundancy a bad practice that one should avoid?

Answer:

The reason an entity set would appear more than once is if one is drawing a diagram that spans multiple pages.

The different occurrences of an entity set may have different sets of attributes, leading to an inconsistent diagram. Instead, the attributes of an entity set should be specified only once. All other occurrences of the entity should omit attributes. Since it is not possible to have an entity set without any attributes, an occurrence of an entity set without attributes clearly indicates that the attributes are specified elsewhere.

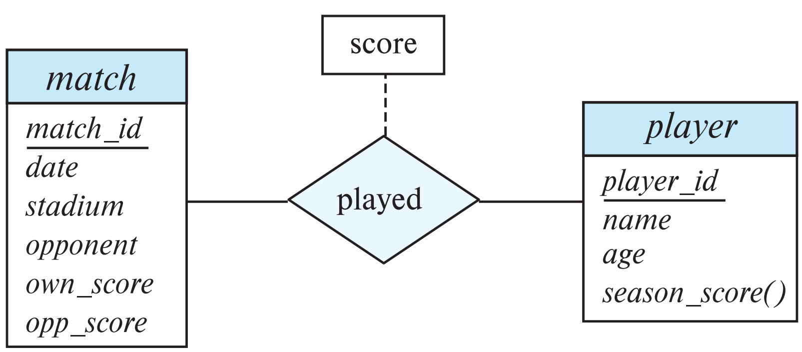


Figure 6.104 E-R diagram for favorite team statistics.

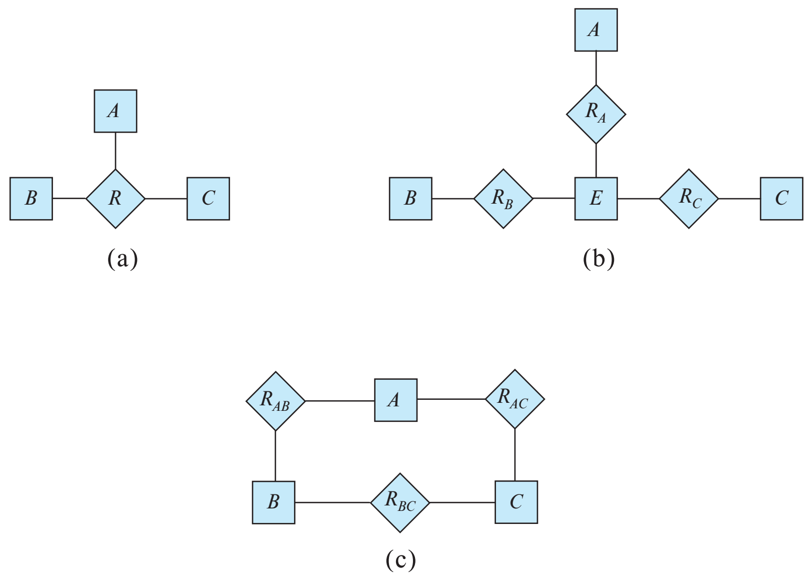


Figure 6.29 Representation of a ternary relationship using binary relationships.

6.5 An E-R diagram can be viewed as a graph. What do the following mean in terms of the structure of an enterprise schema?

a. The graph is disconnected.

b. The graph has a cycle.

Answer:

a. If a pair of entity sets are connected by a path in an E-R diagram, the entity sets are related, though perhaps indirectly. A disconnected graph implies that there are pairs of entity sets that are unrelated to each other. In an enterprise, we can say that the two parts of the enterprise are completely independent of each other. If we split the graph into connected components, we have, in effect, a separate database corresponding to each independent part of the enterprise.

b. As indicated in the answer to the previous part, a path in the graph between a pair of entity sets indicates a (possibly indirect) relationship between the two entity sets. If there is a cycle in the graph, then every pair of entity sets on the cycle are related to each other in at least two distinct ways. If the E-R diagram is acyclic, then there is a unique path between every pair of entity sets and thus a unique relationship between every pair of entity sets.

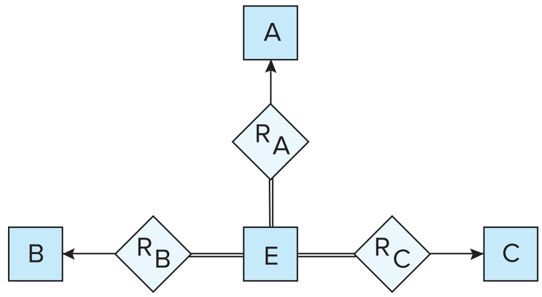


Figure 6.105 E-R diagram for Exercise Exercise 6.6b.

6.6 Consider the representation of the ternary relationship of Figure 6.29a using the binary relationships illustrated in Figure 6.29b (attributes not shown).

a. Show a simple instance of E, A, B, C, R_A, R_B, and R_C that cannot correspond to any instance of A, B, C, and R.

b. Modify the E-R diagram of Figure 6.29b to introduce constraints that will guarantee that any instance of E, A, B, C, R_A, R_B, and R_C that satisfies the constraints will correspond to an instance of A, B, C, and R.

c. Modify the preceding translation to handle total participation constraints on the ternary relationship.

Answer:

a. Let $E = \{e_1, e_2\}$, $A = \{a_1, a_2\}$, $B = \{b_1\}$, $C = \{c_1\}$, $R_A = \{(e_1, a_1), (e_2, a_2)\}$, $R_B = \{(e_1, b_1)\}$, and $R_C = \{(e_1, c_1)\}$. We see that because of the tuple (e_2, a_2), no instance of A, B, C, and R exists that corresponds to E, R_A, R_B and R_C.

b. See Figure 6.105. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B, and C.

c. Suppose A totally participates in the relationhip R, then introduce a total participation constraint between A and R_A, and similarly for B and C.

6.7 A weak entity set can always be made into a strong entity set by adding to its attributes the primary-key attributes of its identifying entity set. Outline what sort of redundancy will result if we do so.

Answer:

The primary key of a weak entity set can be inferred from its relationship with the strong entity set. If we add primary-key attributes to the weak entity set, they will be present in both the entity set, and the relationship set and they have to be the same. Hence there will be redundancy.

6.8 Consider a relation such as *sec_course*, generated from a many-to-one relationship set *sec_course*. Do the primary and foreign key constraints created on the relation enforce the many-to-one cardinality constraint? Explain why.

Answer:
In this example, the primary key of *section* consists of the attributes (*course_id*, *sec_id*, *semester*, *year*), which would also be the primary key of *sec_course*, while *course_id* is a foreign key from *sec_course* referencing *course*. These constraints ensure that a particular *section* can only correspond to one *course*, and thus the many-to-one cardinality constraint is enforced.
However, these constraints cannot enforce a total participation constraint, since a course or a section may not participate in the *sec_course* relationship.

6.9 Suppose the *advisor* relationship set were one-to-one. What extra constraints are required on the relation *advisor* to ensure that the one-to-one cardinality constraint is enforced?

Answer:
In addition to declaring *s_ID* as primary key for *advisor*, we declare *i_ID* as a superkey for *advisor* (this can be done in SQL using the **unique** constraint on *i_ID*).

6.10 Consider a many-to-one relationship R between entity sets A and B. Suppose the relation created from R is combined with the relation created from A. In SQL, attributes participating in a foreign key constraint can be null. Explain how a constraint on total participation of A in R can be enforced using **not null** constraints in SQL.

Answer:
The foreign-key attribute in R corresponding to the primary key of B should be made **not null**. This ensures that no tuple of A which is not related to any entry in B under R can come in R. For example, say **a** is a tuple in A which has no corresponding entry in R. This means when R is combined with A, it would have a foreign-key attribute corresponding to B as **null**, which is not allowed.

6.11 In SQL, foreign key constraints can reference only the primary key attributes of the referenced relation or other attributes declared to be a superkey using the **unique** constraint. As a result, total participation constraints on a many-to-many relationship set (or on the "one" side of a one-to-many relationship set) cannot be enforced on the relations created from the relationship set, using primary key, foreign key, and not null constraints on the relations.

a. Explain why.

b. Explain how to enforce total participation constraints using complex check constraints or assertions (see Section 4.4.8). (Unfortunately, these features are not supported on any widely used database currently.)

Answer:

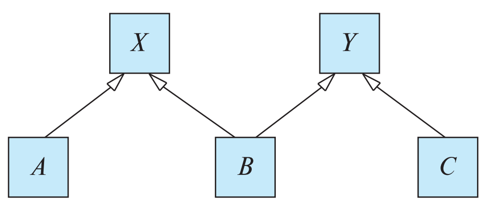
a. For the many-to-many case, the relationship set must be represented as a separate relation that cannot be combined with either participating entity. Now, there is no way in SQL to ensure that a primary-key value occurring in an entity $E1$ also occurs in a many-to-many relationship R, since the corresponding attribute in R is not unique; SQL foreign keys can only refer to the primary key or some other unique key.
Similarly, for the one-to-many case, there is no way to ensure that an attribute on the one side appears in the relation corresponding to the many side, for the same reason.

b. Let the relation R be many-to-one from entity A to entity B with a and b as their respective primary keys. We can put the following check constraints on the "one" side relation B:

constraint *total_part* **check** (b in (**select** b **from** A));
set constraints *total_part* **deferred**;

Note that the constraint should be set to deferred so that it is only checked at the end of the transaction; otherwise if we insert a b value in B before it is inserted in A, the constraint would be violated, and if we insert it in A before we insert it in B, a foreign-key violation would occur.

6.12 Consider the following lattice structure of generalization and specialization (attributes not shown).



For entity sets A, B, and C, explain how attributes are inherited from the higher-level entity sets X and Y. Discuss how to handle a case where an attribute of X has the same name as some attribute of Y.

Answer:
A inherits all the attributes of X, plus it may define its own attributes. Similarly, C inherits all the attributes of Y plus its own attributes. B inherits the attributes of both X and Y. If there is some attribute *name* which belongs to both X and Y, it may be referred to in B by the qualified name *X.name* or *Y.name*.

6.13 An E-R diagram usually models the state of an enterprise at a point in time. Suppose we wish to track *temporal changes*, that is, changes to data over time. For example, Zhang may have been a student between September 2015 and

May 2019, while Shankar may have had instructor Einstein as advisor from May 2018 to December 2018, and again from June 2019 to January 2020. Similarly, attribute values of an entity or relationship, such as *title* and *credits* of *course*, *salary*, or even *name* of *instructor*, and *tot_cred* of *student*, can change over time.

One way to model temporal changes is as follows: We define a new data type called **valid_time**, which is a time interval, or a set of time intervals. We then associate a *valid_time* attribute with each entity and relationship, recording the time periods during which the entity or relationship is valid. The end time of an interval can be infinity; for example, if Shankar became a student in September 2018, and is still a student, we can represent the end time of the *valid_time* interval as infinity for the Shankar entity. Similarly, we model attributes that can change over time as a set of values, each with its own *valid_time*.

a. Draw an E-R diagram with the *student* and *instructor* entities, and the *advisor* relationship, with the above extensions to track temporal changes.

b. Convert the E-R diagram discussed above into a set of relations.

It should be clear that the set of relations generated is rather complex, leading to difficulties in tasks such as writing queries in SQL. An alternative approach, which is used more widely, is to ignore temporal changes when designing the E-R model (in particular, temporal changes to attribute values), and to modify the relations generated from the E-R model to track temporal changes.

Answer:

.

a. The E-R diagram is shown in Figure 6.106.
The primary key attributes *student_id* and *instructor_id* are assumed to be immutable, that is, they are not allowed to change with time. All other attributes are assumed to potentially change with time.

   Note that the diagram uses multivalued composite attributes such as *valid_times* or *name*, with subattributes such as *start_time* or *value*. The *value* attribute is a subattribute of several attributes such as *name, tot_cred* and *salary*, and refers to the name, total credits or salary during a particular interval of time.

b. The generated relations are as shown below. Each multivalued attribute has turned into a relation, with the relation name consisting of the original relation name concatenated with the name of the multivalued attribute. The relation corresponding to the entity has only the primary-key attribute, and this is needed to ensure uniqueness.

student(student_id)
student_valid_times(student_id, start_time, end_time)
student_name(student_id, value, start_time, end_time
student_dept_name(student_id, value, start_time, end_time
student_tot_cred(student_id, value, start_time, end_time
instructor(instructor_id)
instructor_valid_times(instructor_id, start_time, end_time)
instructor_name(instructor_id, value, start_time, end_time
instructor_dept_name(instructor_id, value, start_time, end_time
instructor_salary(instructor_id, value, start_time, end_time
advisor(student_id, instructor_id, start_time, end_time)

The primary keys shown are derived directly from the E-R diagram. If we add the additional constraint that time intervals cannot overlap (or even the weaker condition that one start time cannot have two end times), we can remove the *end_time* from all the above primary keys.

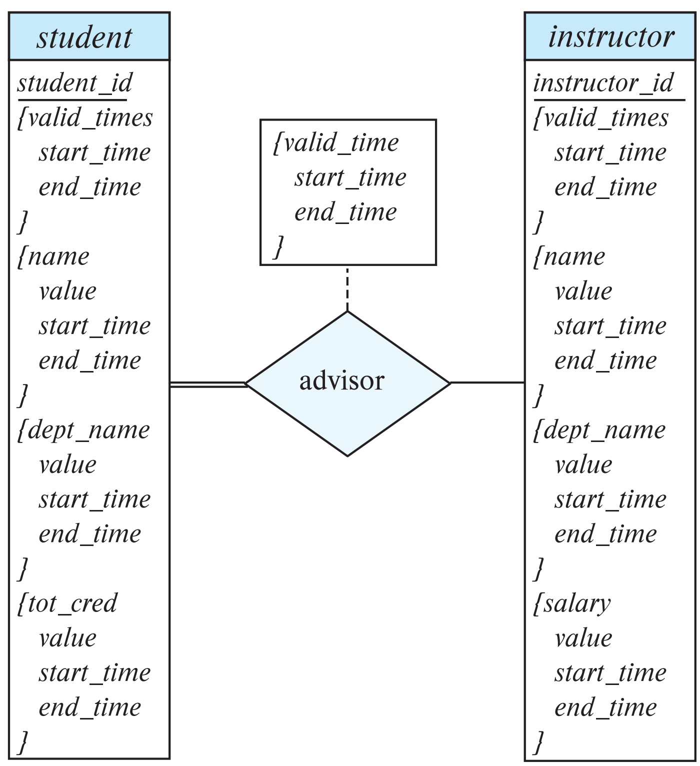


Figure 6.106 E-R diagram for Exercise 6.13

CHAPTER 7

Relational Database Design

Practice Exercises

7.1 Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

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

Show that this decomposition is a lossless decomposition if the following set F of functional dependencies holds:

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

Answer:
A decomposition $\{R_1, R_2\}$ is a lossless decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since A is a candidate key (see Practice Exercise 7.6), $R_1 \cap R_2 \rightarrow R_1$.

7.2 List all nontrivial functional dependencies satisfied by the relation of Figure 7.18.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figure 7.17 Relation of Exercise 7.2.

Answer:

The nontrivial functional dependencies are: $A \rightarrow B$ and $C \rightarrow B$, and a dependency they logically imply: $AC \rightarrow B$. C does not functionally determine A because the first and third tuples have the same C but different A values. The same tuples also show B does not functionally determine A. Likewise, A does not functionally determine C because the first two tuples have the same A value and different C values. The same tuples also show B does not functionally determine C. There are 19 trivial functional dependencies of the form $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$.

7.3 Explain how functional dependencies can be used to indicate the following:

- A one-to-one relationship set exists between entity sets *student* and *instructor*.
- A many-to-one relationship set exists between entity sets *student* and *instructor*.

Answer:

Let $Pk(r)$ denote the primary key attribute of relation r.

- The functional dependencies $Pk(student) \rightarrow Pk\ (instructor)$ and $Pk(instructor) \rightarrow Pk(student)$ indicate a one-to-one relationship because any two tuples with the same value for *student* must have the same value for *instructor*, and any two tuples agreeing on *instructor* must have the same value for *student*.
- The functional dependency $Pk(student) \rightarrow Pk(instructor)$ indicates a many-to-one relationship since any student value which is repeated will have the same instructor value, but many student values may have the same instructor value.

7.4 Use Armstrong's axioms to prove the soundness of the union rule. (*Hint*: Use the augmentation rule to show that, if $\alpha \rightarrow \beta$, then $\alpha \rightarrow \alpha\beta$. Apply the augmentation rule again, using $\alpha \rightarrow \gamma$, and then apply the transitivity rule.)

Answer:

To prove that:

$$\text{if } \alpha \rightarrow \beta \text{ and } \alpha \rightarrow \gamma \text{ then } \alpha \rightarrow \beta\gamma$$

Following the hint, we derive:

$\alpha \rightarrow \beta$	given
$\alpha\alpha \rightarrow \alpha\beta$	augmentation rule
$\alpha \rightarrow \alpha\beta$	union of identical sets
$\alpha \rightarrow \gamma$	given
$\alpha\beta \rightarrow \gamma\beta$	augmentation rule
$\alpha \rightarrow \beta\gamma$	transitivity rule and set union commutativity

7.5 Use Armstrong's axioms to prove the soundness of the pseudotransitivity rule.

Answer:

Proof using Armstrong's axioms of the pseudotransitivity rule:

if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$.

$\alpha \rightarrow \beta$	given
$\alpha\gamma \rightarrow \gamma\beta$	augmentation rule and set union commutativity
$\gamma\beta \rightarrow \delta$	given
$\alpha\gamma \rightarrow \delta$	transitivity rule

7.6 Compute the closure of the following set F of functional dependencies for relation schema $R = (A, B, C, D, E)$.

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

List the candidate keys for R.

Answer:

Note: It is not reasonable to expect students to enumerate all of F^+. Some shorthand representation of the result should be acceptable as long as the nontrivial members of F^+ are found.

Starting with $A \rightarrow BC$, we can conclude: $A \rightarrow B$ and $A \rightarrow C$.

Since $A \rightarrow B$ and $B \rightarrow D$, $A \rightarrow D$	(decomposition, transitive)
Since $A \rightarrow CD$ and $CD \rightarrow E$, $A \rightarrow E$	(union, decomposition, transitive)
Since $A \rightarrow A$, we have	(reflexive)
$A \rightarrow ABCDE$ from the above steps	(union)
Since $E \rightarrow A$, $E \rightarrow ABCDE$	(transitive)
Since $CD \rightarrow E$, $CD \rightarrow ABCDE$	(transitive)
Since $B \rightarrow D$ and $BC \rightarrow CD$, $BC \rightarrow ABCDE$	(augmentative, transitive)
Also, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow D$, etc.	

Therefore, any functional dependency with A, E, BC, or CD on the left-hand side of the arrow is in F^+, no matter which other attributes appear in the FD. Allow * to represent any set of attributes in R, then F^+ is $BD \rightarrow B$, $BD \rightarrow D$, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow BD$, $B \rightarrow D$, $B \rightarrow B$, $B \rightarrow BD$, and all FDs of the form $A * \rightarrow \alpha$, $BC * \rightarrow \alpha$, $CD * \rightarrow \alpha$, $E * \rightarrow \alpha$ where α is any subset of $\{A, B, C, D, E\}$. The candidate keys are A, BC, CD, and E.

7.7 Using the functional dependencies of Exercise 7.6, compute the canonical cover F_c.

Answer:

The given set of FDs F is:-

$$
\begin{array}{l}
A \rightarrow BC \\
CD \rightarrow E \\
B \rightarrow D \\
E \rightarrow A
\end{array}
$$

The left side of each FD in F is unique. Also, none of the attributes in the left side or right side of any of the FDs is extraneous. Therefore the canonical cover F_c is equal to F.

7.8 Consider the algorithm in Figure 7.19 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 7.8 (Section 7.4.2) and that it computes α^+ correctly.

Answer:

The algorithm is correct because:

- If A is added to *result* then there is a proof that $\alpha \rightarrow A$. To see this, observe that $\alpha \rightarrow \alpha$ trivially, so α is correctly part of *result*. If $A \notin \alpha$ is added to *result*, there must be some FD $\beta \rightarrow \gamma$ such that $A \in \gamma$ and β is already a subset of *result*. (Otherwise *fdcount* would be nonzero and the **if** condition would be false.) A full proof can be given by induction on the depth of recursion for an execution of **addin**, but such a proof can be expected only from students with a good mathematical background.

- If $A \in \alpha^+$, then A is eventually added to *result*. We prove this by induction on the length of the proof of $\alpha \rightarrow A$ using Armstrong's axioms. First observe that if procedure **addin** is called with some argument β, all the attributes in β will be added to *result*. Also if a particular FD's *fdcount* becomes 0, all the attributes in its tail will definitely be added to *result*. The base case of the proof, $A \in \alpha \Rightarrow A \in \alpha^+$, is obviously true because the first call to **addin** has the argument α. The inductive hypothesis is that if $\alpha \rightarrow A$ can be proved in n steps or less, then $A \in$ *result*. If there is a proof in $n + 1$

```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
     let β → γ denote the ith FD;
     fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
     appears [A] := NIL;
     for i := 1 to |F| do
       begin
         let β → γ denote the ith FD;
         if A ∈ β then add i to appears [A];
       end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
     if A ∉ result then
       begin
         result := result ∪ {A};
         for each element i of appears[A] do
           begin
             fdcount [i] := fdcount [i] − 1;
             if fdcount [i] := 0 then
               begin
                 let β → γ denote the ith FD;
                 addin (γ);
               end
           end
       end
   end
```

Figure 7.18 An algorithm to compute α^+.

steps that $\alpha \rightarrow A$, then the last step was an application of either reflexivity, augmentation, or transitivity on a fact $\alpha \rightarrow \beta$ proved in n or fewer steps. If reflexivity or augmentation was used in the $(n+1)^{st}$ step, A must have been in *result* by the end of the n^{th} step itself. Otherwise, by the inductive hypothesis, $\beta \subseteq$ *result*. Therefore, the dependency used in proving $\beta \rightarrow \gamma$, $A \in \gamma$, will have *fdcount* set to 0 by the end of the n^{th} step. Hence A will be added to *result*.

To see that this algorithm is more efficient than the one presented in the chapter, note that we scan each FD once in the main program. The resulting array *appears* has size proportional to the size of the given FDs. The recursive calls to **addin** result in processing linear in the size of *appears*. Hence the algorithm has time complexity which is linear in the size of the given FDs. On the other hand, the algorithm given in the text has quadratic time complexity, as it may perform the loop as many times as the number of FDs, in each loop scanning all of them once.

7.9 Given the database schema $R(A, B, C)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $B \rightarrow C$ holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)

Answer:

a. The query is given below. Its result is non-empty if and only if $B \rightarrow C$ does not hold on r.

```
select B
from r
group by B
having count(distinct C) > 1
```

b.

```
create assertion b_to_c check
   (not exists
       (select B
        from r
        group by B
        having count(distinct C) > 1
       )
   )
```

7.10 Our discussion of lossless decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?

Answer:
The natural join operator is defined in terms of the Cartesian product and the selection operator. The selection operator gives *unknown* for any query on a null value. Thus, the natural join excludes all tuples with null values on the common attributes from the final result. Thus, the decomposition would be lossy (in a manner different from the usual case of lossy decomposition), if null values occur in the left-hand side of the functional dependency used to decompose the relation. (Null values in attributes that occur only in the right-hand side of the functional dependency do not cause any problems.)

7.11 In the BCNF decomposition algorithm, suppose you use a functional dependency $\alpha \rightarrow \beta$ to decompose a relation schema $r(\alpha, \beta, \gamma)$ into $r_1(\alpha, \beta)$ and $r_2(\alpha, \gamma)$.

a. What primary and foreign-key constraint do you expect to hold on the decomposed relations?

b. Give an example of an inconsistency that can arise due to an erroneous update, if the foreign-key constraint were not enforced on the decomposed relations above.

c. When a relation schema is decomposed into 3NF using the algorithm in Section 7.5.2, what primary and foreign-key dependencies would you expect to hold on the decomposed schema?

Answer:

a. α should be a primary key for r_1, and α should be the foreign key from r_2, referencing r_1.

b. If the foreign key constraint is not enforced, then a deletion of a tuple from r_1 would not have a corresponding deletion from the referencing tuples in r_2. Instead of deleting a tuple from r, this would amount to simply setting the value of α to null in some tuples.

c. For every schema $r_i(\alpha\beta)$ added to the decomposition because of a functional dependency $\alpha \rightarrow \beta$, α should be made the primary key. Also, a candidate key γ for the original relation is located in some newly created relation r_k and is a primary key for that relation.
Foreign-key constraints are created as follows: for each relation r_i created above, if the primary key attributes of r_i also occur in any other relation r_j, then a foreign-key constraint is created from those attributes in r_j, referencing (the primary key of) r_i.

7.12 Let R_1, $R_2, \ldots, R_n$ be a decomposition of schema U. Let $u(U)$ be a relation, and let $r_i = \Pi_{R_I}(u)$. Show that

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

Answer:

Consider some tuple t in u.

Note that $r_i = \Pi_{R_i}(u)$ implies that $t[R_i] \in r_i$, $1 \leq i \leq n$. Thus,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] \in r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

By the definition of natural join,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] = \Pi_\alpha(\sigma_\beta(t[R_1] \times t[R_2] \times \ldots \times t[R_n]))$$

where the condition β is satisfied if values of attributes with the same name in a tuple are equal and where $\alpha = U$. The Cartesian product of single tuples generates one tuple. The selection process is satisfied because all attributes with the same name must have the same value since they are projections from the same tuple. Finally, the projection clause removes duplicate attribute names.

By the definition of decomposition, $U = R_1 \cup R_2 \cup \ldots \cup R_n$, which means that all attributes of t are in $t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n]$. That is, t is equal to the result of this join.

Since t is any arbitrary tuple in u,

$$u \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

7.13 Show that the decomposition in Exercise 7.1 is not a dependency-preserving decomposition.

Answer:

Therer are several functional dependencies that are not preserved. We discuss one example here. The dependency $B \rightarrow D$ is not preserved. F_1, the restriction of F to (A, B, C) is $A \rightarrow ABC$, $A \rightarrow AB$, $A \rightarrow AC$, $A \rightarrow BC$, $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$, $AB \rightarrow AC$, $AB \rightarrow ABC$, $AB \rightarrow BC$, $AB \rightarrow AB$, $AB \rightarrow A$, $AB \rightarrow B$, $AB \rightarrow C$, AC (same as AB), BC (same as AB), ABC (same as AB). F_2, the restriction of F to (C, D, E) is $A \rightarrow ADE$, $A \rightarrow AD$, $A \rightarrow AE$, $A \rightarrow DE$, $A \rightarrow A$, $A \rightarrow D$, $A \rightarrow E$, $D \rightarrow D$, E (same as A), AD, AE, DE, ADE (same as A). $(F_1 \cup F_2)^+$ is easily seen not to contain $B \rightarrow D$ since the only FD in $F_1 \cup F_2$ with B as the left side is $B \rightarrow B$, a trivial FD. Thus $B \rightarrow D$ is not preserved.

A simpler argument is as follows: F_1 contains no dependencies with D on the right side of the arrow. F_2 contains no dependencies with B on the left side of the arrow. Therefore for $B \rightarrow D$ to be preserved there must be a functional dependency $B \rightarrow \alpha$ in F_1^+ and $\alpha \rightarrow D$ in F_2^+ (so $B \rightarrow D$ would follow by transitivity). Since the intersection of the two schemes is A, $\alpha = A$. Observe that $B \rightarrow A$ is not in F_1^+ since $B^+ = BD$.

7.14 Show that there can be more than one canonical cover for a given set of functional dependencies, using the following set of dependencies:

$$X \rightarrow YZ,\ Y \rightarrow XZ, \text{ and } Z \rightarrow XY.$$

Answer: Consider the first functional dependency. We can verify that Z is extraneous in $X \rightarrow YZ$ and delete it. Subsequently, we can similarly check that X is extraneous in $Y \rightarrow XZ$ and delete it, and that Y is extraneous in $Z \rightarrow XY$ and delete it, resulting in a canonical cover $X \rightarrow Y, Y \rightarrow Z, Z \rightarrow X$.
However, we can also verify that Y is extraneous in $X \rightarrow YZ$ and delete it. Subsequently, we can similarly check that Z is extraneous in $Y \rightarrow XZ$ and delete it, and that X is extraneous in $Z \rightarrow XY$ and delete it, resulting in a canonical cover $X \rightarrow Z, Y \rightarrow X, Z \rightarrow Y$.

7.15 The algorithm to generate a canonical cover only removes one extraneous attribute at a time. Use the functional dependencies from Exercise 7.14 to show what can go wrong if two attributes inferred to be extraneous are deleted at once.

Answer: In $X \rightarrow YZ$, one can infer that Y is extraneous, and so is Z. But deleting both will result in a set of dependencies from which $X \rightarrow YZ$ can no longer be inferred. Deleting Y results in Z no longer being extraneous, and deleting Z results in Y no longer being extraneous. The canonical cover algorithm only deletes one attribute at a time, avoiding the problem that could occur if two attributes are deleted at the same time.

7.16 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

Answer:
Let F be a set of functional dependencies that hold on a schema R. Let $\sigma = \{R_1, R_2, \ldots, R_n\}$ be a dependency-preserving 3NF decomposition of R. Let X be a candidate key for R.
Consider a legal instance r of R. Let $j = \Pi_X(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$. We want to prove that $r = j$.

We claim that if t_1 and t_2 are two tuples in j such that $t_1[X] = t_2[X]$, then $t_1 = t_2$. To prove this claim, we use the following inductive argument:
Let $F' = F_1 \cup F_2 \cup \ldots \cup F_n$, where each F_i is the restriction of F to the schema R_i in σ. Consider the use of the algorithm given in Figure 7.8 to compute the

closure of X under F'. We use induction on the number of times that the *for* loop in this algorithm is executed.

- *Basis*: In the first step of the algorithm, *result* is assigned to X, and hence given that $t_1[X] = t_2[X]$, we know that $t_1[result] = t_2[result]$ is true.
- *Induction Step*: Let $t_1[result] = t_2[result]$ be true at the end of the k th execution of the *for* loop.

  Suppose the functional dependency considered in the $k+1$ th execution of the *for* loop is $\beta \rightarrow \gamma$, and that $\beta \subseteq result$. $\beta \subseteq result$ implies that $t_1[\beta] = t_2[\beta]$ is true. The facts that $\beta \rightarrow \gamma$ holds for some attribute set R_i in σ and that $t_1[R_i]$ and $t_2[R_i]$ are in $\Pi_{R_i}(r)$ imply that $t_1[\gamma] = t_2[\gamma]$ is also true. Since γ is now added to *result* by the algorithm, we know that $t_1[result] = t_2[result]$ is true at the end of the $k+1$ th execution of the *for* loop.

Since σ is dependency-preserving and X is a key for R, all attributes in R are in *result* when the algorithm terminates. Thus, $t_1[R] = t_2[R]$ is true, that is, $t_1 = t_2$ – as claimed earlier.

Our claim implies that the size of $\Pi_X(j)$ is equal to the size of j. Note also that $\Pi_X(j) = \Pi_X(r) = r$ (since X is a key for R). Thus we have proved that the size of j equals that of r. Using the result of Exercise 7.12, we know that $r \subseteq j$. Hence we conclude that $r = j$.

Note that since X is trivially in 3NF, $\sigma \cup \{X\}$ is a dependency-preserving lossless decomposition into 3NF.

7.17 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless decompositions of R' into BCNF.

Answer:

Given the relation $R' = (A, B, C, D)$ the set of functional dependencies $F' = A \rightarrow B, C \rightarrow D, B \rightarrow C$ allows three distinct BCNF decompositions.

$$R_1 = \{(A, B), (C, D), (B, C)\}$$

is in BCNF as is

$$R_2 = \{(A, B), (C, D), (A, C)\}$$

$$R_3 = \{(B, C), (A, D), (A, B)\}$$

7.18 Let a **prime** attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that A is **transitively dependent** on α. We can restate the definition of 3NF as follows: A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R. Show that this new definition is equivalent to the original one.

Answer:
Suppose R is in 3NF according to the textbook definition. We show that it is in 3NF according to the definition in the exercise. Let A be a nonprime attribute in R that is transitively dependent on a key α for R. Then there exists $\beta \subseteq R$ such that $\beta \rightarrow A$, $\alpha \rightarrow \beta$, $A \notin \alpha$, $A \notin \beta$, and $\beta \rightarrow \alpha$ does not hold. But then $\beta \rightarrow A$ violates the textbook definition of 3NF since

- $A \notin \beta$ implies $\beta \rightarrow A$ is nontrivial
- Since $\beta \rightarrow \alpha$ does not hold, β is not a superkey
- A is not any candidate key, since A is nonprime

Now we show that if R is in 3NF according to the exercise definition, it is in 3NF according to the textbook definition. Suppose R is not in 3NF according to the the textbook definition. Then there is an FD $\alpha \rightarrow \beta$ that fails all three conditions. Thus

- $\alpha \rightarrow \beta$ is nontrivial.
- α is not a superkey for R.
- Some A in $\beta - \alpha$ is not in any candidate key.

This implies that A is nonprime and $\alpha \rightarrow A$. Let γ be a candidate key for R. Then $\gamma \rightarrow \alpha$, $\alpha \rightarrow \gamma$ does not hold (since α is not a superkey), $A \notin \alpha$, and $A \notin \gamma$ (since A is nonprime). Thus A is transitively dependent on γ, violating the exercise definition.

7.19 A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$; we say that β is *partially dependent* on α. A relation schema R is in **second normal form** (**2NF**) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

Answer:
Referring to the definitions in Exercise 7.18, a relation schema R is said to be in 3NF if there is no nonprime attribute A in R for which A is transitively dependent on a key for R.

We can also rewrite the definition of 2NF given here as:
"A relation schema R is in 2NF if no nonprime attribute A is partially dependent on any candidate key for R."

To prove that every 3NF schema is in 2NF, it suffices to show that if a nonprime attribute A is partially dependent on a candidate key α, then A is also transitively dependent on the key α.

Let A be a nonprime attribute in R. Let α be a candidate key for R. Suppose A is partially dependent on α.

- From the definition of a partial dependency, we know that for some proper subset β of α, $\beta \rightarrow A$.
- Since $\beta \subset \alpha$, $\alpha \rightarrow \beta$. Also, $\beta \rightarrow \alpha$ does not hold, since α is a candidate key.
- Finally, since A is nonprime, it cannot be in either β or α.

Thus we conclude that $\alpha \rightarrow A$ is a transitive dependency. Hence we have proved that every 3NF schema is also in 2NF.

7.20 Give an example of a relation schema R and a set of dependencies such that R is in BCNF but is not in 4NF.

Answer:
There are, of course, an infinite number of such examples. We show the simplest one here.
Let R be the schema (A, B, C) with the only nontrivial dependency being $A \rightarrow\rightarrow B$