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 allows 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 focussed 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 List at least 3 different types of information that a university would maintain, beyond those listed in Section 1.6.2.
Answer:

- Information about people who are employees of the university but who are not instructors.
- Library information, including books in the library, and who has issued books.
- Accounting information including fee payment, scholarships, salaries, and all other kinds of receipts and payments of the university.

1.5 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 is not updated. There are very few relationships here, and none of them can lead to redundancy.
- **Difficulty in accessing data**. If video data is 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 meta data 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 is 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 is 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 is not updated, concurrent access anomalies would be unlikely to occur.
- **Security problems**. These would be a issue if the system supported authorization.

1.6 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 relational database of Figure **??**. What are the appropriate primary keys?
Answer: The answer is shown in Figure 2.1, with primary keys underlined.

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, which can cause a violation of the foreign key constraint.
Answer:

- Inserting a tuple:

  (10111, Ostrom, Economics, 110,000)

  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.

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

Figure 2.1 Relational database for Practice Exercise 2.1.

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 **??**, 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 be always 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 cross 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 students attributes, followed by an *s_id* attribute identical to the students 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 following expressions, which use the result of a relational algebra operation as the input to another operation. For each expression, explain in words what the expression does.

a. $\sigma_{year \geq 2009}(takes) \bowtie student$

b. $\sigma_{year \geq 2009}(takes \bowtie student)$

c. $\Pi_{ID, name, course\_id}(student \bowtie takes)$

Answer:

a. For each student who takes at least one course in 2009, display the students information along with the information about what courses the student took. The attributes in the result are:

   ID, *name*, *dept_name*, *tot_cred*, *course_id*, *section_id*, *semester*, *year*, *grade*

b. Same as (a); selection can be done before the join operation.

c. Provide a list of consisting of

$ID, name, course\_id$

of all students who took any course in the university.

2.7 Consider the relational database of Figure **??**. Give an expression in the relational algebra to express each of the following queries:

a. Find the names of all employees who live in city "Miami".

b. Find the names of all employees whose salary is greater than $100,000.

c. Find the names of all employees who live in "Miami" and whose salary is greater than $100,000.

Answer:

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

b. $\Pi_{name}\ (\sigma_{salary > 100000}\ (employee))$

c. $\Pi_{name}\ (\sigma_{city = \text{"Miami"} \wedge salary > 100000}\ (employee))$

2.8 Consider the bank database of Figure **??**. Give an expression in the relational algebra for each of the following queries.

a. Find the names of all branches located in "Chicago".

b. Find the names of all borrowers who have a loan in branch "Downtown".

Answer:

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

b. $\Pi_{customer\_name}\ (\sigma_{branch\_name = \text{"Downtown"}}\ (borrower \bowtie loan))$

CHAPTER 3

Introduction to SQL

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 Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009.

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 student.ID
from     (student join takes using(ID))
         join (instructor join teaches using(ID))
         using(course_id, sec_id, semester, year)
where    instructor.name = 'Einstein'
```

As an alternative to th **join .. using** syntax above the query can be written by enumerating relations in the **from** clause, and adding the corresponding join predicates on *ID*, *course_id*, *section_id*, *semester*, and *year* to the **where** clause.
Note that using natural join in place of **join .. using** would result in equating student *ID* with instructor *ID*, which is incorrect.

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 Autumn 2009.
One way of writing the query is as follows.

```
select     course_id, sec_id, count(ID)
from       section natural join takes
where      semester = 'Autumn'
and        year = 2009
group by course_id, sec_id
```

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 replace **natural join** by the **natural left outer join** operation, covered later in Chapter 4. Another way is to use a subquery in the **select** clause, as follows.

```
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.section_id = section.section_id)
        from   section
where   semester = 'Autumn'
and     year = 2009
```

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

f. Find the maximum enrollment, across all sections, in Autumn 2009. One way of writing this query is as follows:

```
select  max(enrollment)
from    (select    count(ID) as enrollment
         from      section natural join takes
         where     semester = 'Autumn'
         and       year = 2009
         group by course_id, 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 Autumn 2009. The following answer uses a **with** clause to create a temporary view, simplifying the query.

```
with  sec_enrollment as (
        select    course_id, sec_id, count(ID) as enrollment
        from      section natural join takes
        where     semester = 'Autumn'
        and       year = 2009
        group by course_id, 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.

3.2 Suppose you are given a relation *grade_points*(*grade*, *points*), which 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 above relation, and our university schema, write each of the following queries in SQL. You can 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 every student.

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 natural join course) natural join grade_points
whereID = '12345'
```

One problem with the above query is that if the student has not taken any course, the result 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 **natural left outer join** operation, which we study later in Chapter 4. Another way to ensure that we get 0 as the answer, is to the following query:

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

As usual, specifying join conditions can be specified in the **where** clause instead of using the **natural join** operation or the **join .. using** operation.

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 natural join course) natural join grade_points
where   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    takes.ID = '12345' and
         not exists ( select * from takes where takes.ID = '12345'))
```

Other ways of ensuring the above are discussed later in the solution to Exercise 4.5.

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

```
select    ID, sum(credits * points)/sum(credits) as GPA
from      (takes natural join course) natural join grade_points
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))
```

3.3

3.4 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 (that is, 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).

```
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
```

3.5 Consider the insurance database of Figure **??**, 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 1989.

b. Add a new accident to the database; assume any values for required attributes.

c. Delete the Mazda belonging to "John Smith".

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

a. Find the total number of people who owned cars that were involved in accidents in 1989.
Note: this is not the same as the total number of accidents in 1989. We must count people with several accidents only once.

```
select  count (distinct name)
from    accident, participated, person
where   accident.report_number = participated.report_number
and     participated.driver_id = person.driver_id
and     date between date '1989-00-00' and date '1989-12-31'
```

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

Figure ??. Insurance database.

b. Add a new accident to the database; assume any values for required attributes.
We assume the driver was "Jones," although it could be someone else. Also, we assume "Jones" owns one Toyota. First we must find the license of the given car. Then the *participated* and *accident* relations must be updated in order to both record the accident and tie it to the given car. We assume values "Berkeley" for *location*, '2001-09-01' for date and *date*, 4007 for *report_number* and 3000 for damage amount.

```
insert into accident
        values (4007, '2001-09-01', 'Berkeley')

insert into participated
        select o.driver_id, c.license, 4007, 3000
        from person p, owns o, car c
        where p.name = 'Jones' and p.driver_id = o.driver_id and
                o.license = c.license and c.model = 'Toyota'
```

c. Delete the Mazda belonging to "John Smith".
Since *model* is not a key of the *car* relation, we can either assume that only one of John Smith's cars is a Mazda, or delete all of John Smith's Mazdas (the query is the same). Again assume *name* is a key for *person*.

```
delete car
where model = 'Mazda' and license in
    (select license
     from person p, owns o
     where p.name = 'John Smith' and p.driver_id = o.driver_id)
```

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

3.6 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 ≤ *score* < 60, grade *B* if 60 ≤ *score* < 80, and grade *A* if 80 ≤ *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.

3.7 The SQL **like** operator is case sensitive, 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%'
```

3.8 Consider the SQL query

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

Figure 3.1 Banking database for Exercises 3.8 and 3.15.

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 one of $r1$ 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. Of course if *p* itself is empty, the result is as expected, i.e. empty.

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

a. Find all customers of the bank who have an account but not a loan.

b. Find the names of all customers who live on the same street and in the same city as "Smith".

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

Answer:

a. Find all customers of the bank who have an account but not a loan.

(**select** *customer_name*
from *depositor*)
except
(**select** *customer_name*
from *borrower*)

The above selects could optionally have **distinct** specified, without changing the result of the query.

b. Find the names of all customers who live on the same street and in the same city as "Smith".
One way of writing the query is as follows.

```
select  F.customer_name
from    customer F join customer S using(customer_street, customer_city)
where   S.customer_name = 'Smith'
```

The join condition could alternatively be specified in the **where** clause, instead of using bf join .. using.

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

```
select  distinct branch_name
from    account natural join depositor natural join customer
where   customer_city = 'Harrison'
```

As usual, the natural join operation could be replaced by specifying join conditions in the **where** clause.

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

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

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

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

d. Find all employees in the database who earn more than each employee of Small Bank Corporation.

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

f. Find the company that has the most employees.

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

Answer:

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

Figure 3.20. Employee database.

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

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

b. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.
If people may work for several companies, the following solution will only list those who earn more than $10,000 per annum from "First Bank Corporation" alone.

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

As in the solution to the previous query, we can use a join to solve this one also.

c. Find all employees in the database who do not work for First Bank Corporation.
The following solution assumes that all people work for exactly one company.

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

If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, or if people may have jobs with more than one company, the solution is slightly more complicated.

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

d. Find all employees in the database who earn more than each employee of Small Bank Corporation.

The following solution assumes that all people work for at most one company.

```
select employee_name
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. It can be solved by using a nested subquery, but we illustrate below how to solve it using the **with** clause.

```
with emp_total_salary as
   (select employee_name, sum(salary) as total_salary
    from works
    group by employee_name
   )
select employee_name
from emp_total_salary
where total_salary > all
   (select total_salary
    from emp_total_salary, works
    where works.company_name = 'Small Bank Corporation' and
          emp_total_salary.employee_name = works.employee_name
   )
```

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

The simplest solution uses the **contains** comparison which was included in the original System R Sequel language but is not present in the subsequent SQL versions.

```
select T.company_name
from company T
where (select R.city
       from company R
       where R.company_name = T.company_name)
    contains
      (select S.city
       from company S
       where S.company_name = 'Small Bank Corporation')
```

Below is a solution using standard SQL.

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

f. Find the company that has the most employees.

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

g. Find those companies 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')
```

3.11 Consider the relational database of Figure **??**. Give an expression in SQL for each of the following queries.

a. Modify the database so that Jones now lives in Newtown.

b. Give all managers of First Bank Corporation a 10 percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3 percent raise.

Answer:

a. Modify the database so that Jones now lives in Newtown.

The solution assumes that each person has only one tuple in the *employee* relation.

```
update employee
set city = 'Newton'
where person_name = 'Jones'
```

b. Give all managers of First Bank Corporation a 10-percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3-percent raise.

```
update works T
set T.salary = T.salary * 1.03
where T.employee_name in (select manager_name
                          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.employee_name in (select manager_name
                          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
    )
where T.employee_name in (select manager_name
                          from manages) and
      T.company_name = 'First Bank Corporation'
```

CHAPTER 4

Intermediate SQL

Practice Exercises

4.1 Write the following queries in SQL:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

b. Write the same query as above, but using a scalar subquery, without outerjoin.

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it 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 scalar subqueries. Make sure to correctly handle departments with no instructors.

Answer:

a. Display a list of all instructors, showing their ID, name, and the number of sections that they have taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outerjoin, and should not use scalar subqueries.

```
select ID, name,
        count(course_id, section_id, year,semester) as 'Number of sections'
from instructor natural left outer join teaches
group by ID, name
```

The above query should not be written using count(*) since count * counts null values also. It could be written using count(*section_id*), or

any other 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, without outerjoin.

```
select ID, name,
       (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 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "–".

```
select course_id, section_id, ID,
       decode(name, NULL, '–', name)
from (section natural left outer join teaches)
       natural left outer join instructor
where semester='Spring' and year= 2010
```

The query may also be written using the **coalesce** operator, by replacing **decode**(..) by **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 4.2.

d. Display the list of all departments, with the total number of instructors in each department, without using scalar subqueries. Make sure to correctly handle departments with no instructors.

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

4.2 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*, *section_id*, *semester*, *year*, *grade*
from *takes T1*
where not exists
(**select** ID **from** *student S1* **where***T1.id* = *S1.id*))

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

- r **natural left outer join** (s **natural left outer join** t), and
- (r **natural left outer join** s) **natural left outer join** t

a. Give instances of relations r, s and t such that in the result of the second expression, attribute C has a null value but attribute D has a non-null value.

b. Is the above pattern, with C null and D not null possible in the result of the first expression? 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. It is not possible for D to be not null while C is null in the result of the first expression, since in the subexpression s natural left outer join t, it is not possible for C to be null while D is not null. In the overall expression C can be null if and only if some r tuple does not have a matching B value in s. However in this case D will also be null.

4.4 **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 Section 3.3.3The Natural Joinsubsection.3.3.3 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 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 Exercise 4.1Item.138.

Answer:

a. Consider the case where a professor in 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 instructors department name does not match the department name of the course. A dataset corresponding to the same is:
instructor = {(12345,'Guass', 'Physics', 10000)}
teaches = {(12345, 'EE321', 1, 'Spring', 2009)}
course = {('EE321', 'Magnetism', 'Elec. Eng.', 6)}

b. The query in question 0.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 the above 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.

4.5 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise **??**; 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 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 0.

```
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
    from    student
    where ID not in (select ID from takes))
```

The view defined above takes care of **NULL** grades by considering the creditpoints to be 0, and not adding the corresponding credits in *credit_sum*. The query above ensures that if the student has not taken any course with non-NULL credits, and has *credit_sum* = 0 gets a gpa of **NULL**. This avoid the division by 0, which would otherwise have resulted.
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.

4.6 Complete the SQL DDL definition of the university database of Figure Figure 4.8Referential Integrityfigcnt.50 to include the relations *student*, *takes*, *advisor*, and *prereq*.
Answer:

```
create table student
    (ID             varchar (5),
     name           varchar (20) not null,
     dept_name      varchar (20),
     tot_cred       numeric (3,0) check (tot_cred >= 0),
     primary key (ID),
     foreign key (dept_name) references department
                    on delete set null);
```

```
create table takes
    (ID              varchar (5),
     course_id       varchar (8),
     section_id      varchar (8),
     semester        varchar (6),
     year            numeric (4,0),
     grade           varchar (2),
     primary key (ID, course_id, section_id, semester, year),
     foreign key (course_id, section_id, semester, year) references section
                     on delete cascade,
     foreign key (ID) references student
                     on delete cascade);
```

```
create table advisor
    (i_id            varchar (5),
     s_id            varchar (5),
     primary key (s_ID),
     foreign key (i_ID) references instructor (ID)
                     on delete set null,
     foreign key (s_ID) references student (ID)
                     on delete cascade);
```

```
create table prereq
    (course_id       varchar(8),
     prereq_id       varchar(8),
     primary key (course_id, prereq_id),
     foreign key (course_id) references course
                     on delete cascade,
     foreign key (prereq_id) references course);
```

4.7 Consider the relational database of Figure Figure 4.11figcnt.53. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer:

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

create table *works*
(*person_name* char(20),
company_name char(15),
salary integer,
primary key (*person_name*),
foreign key (*person_name*) **references** *employee*,
foreign key (*company_name*) **references** *company*)

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

pp**create table** *manages*
(*person_name* char(20),
manager_name char(20),
primary key (*person_name*),
foreign key (*person_name*) **references** *employee*,
foreign key (*manager_name*) **references** *employee*)

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

4.8 As discussed in Section Section 4.4.7Complex Check Conditions and Assertionssubsection.4.4 we expect the constraint "an instructor cannot teach sections in two different classrooms in a semester in the same time slot" to hold.

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 Section 4.4.7Complex Check Conditions and Assertionssubsection.4.4.7, current generation database systems do not support such assertions, although they are part of the SQL standard).

Answer:

a.

select ID, *name*, *section_id*, *semester*, *year*, *time_slot_id*,
count(**distinct** *building*, *room_number*)
from *instructor* **natural join** *teaches* **natural join** *section*
group by (ID, *name*, *section_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.

```
create assertion check not exists
    ( select ID, name, section_id, semester, year, time_slot_id,
                count(distinct building, room_number)
      from      instructor natural join teaches natural join section
      group by (ID, name, section_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_name   char(20),
    manager_name    char(20),
    primary key employee_name,
    foreign key (manager_name) references manager
                                  on delete cascade )
```

Here, *employee_name* 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 SQL-92 provides an *n*-ary operation called **coalesce**, which is defined as follows: **coalesce**(A_1, A_2, ..., A_n) returns the first nonnull A_i in the list A_1, A_2, ..., A_n, and returns null if all of A_1, A_2, ..., A_n are null.

Let *a* and *b* be relations with the schemas *A*(*name, address, title*) and *B*(*name, address, salary*), respectively. Show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition and 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 Some researchers have proposed the concept of *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \neq j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *instructor_info* (Section Section 4.2Viewssection.4.2). Show how you can use marked nulls to allow the insertion of the tuple (99999, "Johnson", "Music") through *instructor_info*.

Answer: To insert the tuple (99999, "("Johnson), "Music") into the view *instructor_info*, we can do the following:

$instructor \leftarrow$ (99999, "Johnson", $\perp_k$, $\perp$) $\cup$ $instructor$

$department \leftarrow$ ($\perp_k$, "$Music$", $\perp$) $\cup$ $department$

such that $\perp_k$ is a new marked null not already existing in the database.

Note: "Music" here is the name of a building and may or may not be related to Music department.

CHAPTER 5

Advanced SQL

Practice Exercises

5.1 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.

Answer: Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However not all kinds of queries can be written in SQL. Also nondeclarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.

Embedded SQL has the advantage of programs being less complicated since it avoids the clutter of the ODBC or JDBC function calls, but requires a specialized preprocessor.

5.2 Write a Java function using JDBC metadata features that takes a `ResultSet` as an input parameter, and prints out the result in tabular form, with appropriate names as column headings.

Answer:

```
public class ResultSetTable implements TabelModel {
    ResultSet result;
    ResultSetMetaData metadata;
    int num_cols;

    ResultSetTable(ResultSet result) throws SQLException {
        this.result = result;
        metadata = result.getMetaData();
        num_cols = metadata.getColumnCount();

        for(int i = 1; i <= num_cols; i++) {
            System.out.print(metadata.getColumnName(i) + '' '');
```

```
            }
            System.out.println();
            while(result.next()) {
                for(int i = 1; i <= num_cols; i++) {
                    System.out.print(result.getString(
                        metadata.getColumnName(i) + '' ''));
                }
                System.out.println();
            }
        }
}
```

5.3 Write a Java function using JDBC metadata features that prints a list of all relations in the database, displaying for each relation the names and types of its attributes.
Answer:

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getTables();
while (rs.next()) {
    System.out.println(rs.getString(''TABLE_NAME'');
    ResultSet rs1 = dbmd.getColumns(null, ''schema-name'',
                    rs.getString(''TABLE_NAME''), ''%'');
    while (rs1.next()) {
        System.out.println(rs1.getString(''COLUMN_NAME''),
                    rs.getString(''TYPE_NAME'');
    }
}
```

5.4 Show how to enforce the constraint "an instructor cannot teach in two different classrooms in a semester in the same time slot." using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).
Answer: FILL

5.5 Write triggers to enforce the referential integrity constraint from *section* to *time_slot*, on updates to *section*, and *time_slot*. Note that the ones we wrote in Figure 5.8 do not cover the **update** operation.
Answer: FILL

5.6 To maintain the *tot_cred* attribute of the *student* relation, carry out the following:

a. Modify the trigger on updates of *takes*, to handle all updates that can affect the value of *tot_cred*.

b. Write a trigger to handle inserts to the *takes* relation.

c. Under what assumptions is it reasonable not to create triggers on the *course* relation?

Answer: FILL

5.7 Consider the bank database of Figure 5.25. Let us define a view *branch_cust* as follows:

```
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number
```

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *maintain* the view, that is, to keep it up-to-date on insertions to and deletions from *depositor* or *account*. Do not bother about updates.

Answer: For inserting into the materialized view *branch_cust* we must set a database trigger on an insert into *depositor* and *account*. We assume that the database system uses *immediate* binding for rule execution. Further, assume that the current version of a relation is denoted by the relation name itself, while the set of newly inserted tuples is denoted by qualifying the relation name with the prefix – **inserted**.
The active rules for this insertion are given below –

```
define trigger insert_into_branch_cust_via_depositor
after insert on depositor
referencing new table as inserted for each statement
insert into branch_cust
    select branch_name, customer_name
    from inserted, account
    where inserted.account_number = account.account_number

define trigger insert_into_branch_cust_via_account
after insert on account
referencing new table as inserted for each statement
insert into branch_cust
    select branch_name, customer_name
    from depositor, inserted
    where depositor.account_number = inserted.account_number
```

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch_cust*.
The deletion of a tuple from *branch_cust* is similar to insertion, exce pt that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly

deleted set of tuples by qualifying the relation name with the keyword **deleted**.

```
define trigger delete_from_branch_cust_via_depositor
after delete on depositor
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from deleted, account
    where deleted.account_number = account.account_number

define trigger delete_from_branch_cust_via_account
after delete on account
referencing old table as deleted for each statement
delete from branch_cust
    select branch_name, customer_name
    from depositor, deleted
    where depositor.account_number = deleted.account_number
```

5.8 Consider the bank database of Figure 5.25. Write an SQL trigger to carry out the following action: On **delete** of an account, for each owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.
Answer:

```
create trigger check-delete-trigger after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
    ( select customer_name from depositor
    where account_number <> orow.account_number )
end
```

5.9 Show how to express **group by cube**(a, b, c, d) using **rollup**; your answer should have only one **group by** clause.
Answer:

groupby rollup(a), **rollup**(b), **rollup**(c), **rollup**(d)

5.10 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.
Answer: We assume that multiple students do not have the same marks since otherwise the question is not deterministic; the query below deterministically returns all students with the same marks as the n student, so it may return more than n students.

```
select student, sum(marks) as total,
            rank() over (order by (total) desc ) as trank
from S
groupby student
having trank ≤ n
```

5.11 Consider the *sales* relation from Section 5.6. Write an SQL query to compute the cube operation on the relation, giving the relation in Figure 5.21. Do not use the **cube** construct.

Answer:

```
(select color, size, sum(number)
 from sales
 groupby color, size
)
union
(select color, 'all', sum(number)
 from sales
 groupby color
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', size, sum(number)
 from sales
 groupby size
)
union
(select 'all', 'all', sum(number )
 from sales
)
```

CHAPTER 6

Formal Relational Query Languages

Practice Exercises

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

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 Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009

Answer:

a. $\Pi_{title}(\sigma_{dept\_name = \text{'Comp. Sci'} \wedge credits=3}(course))$

b. $\Pi_{ID}(\sigma_{ID = \text{'Einstein'}}(takes \bowtie \rho_{t1(IID,\ course\_id,\ section\_id,\ semester,\ year)}teaches))$
Assuming the set version of the relational algebra is used, there is no need to explicitly remove duplicates. If the multiset version is used, the grouping operator can be used without any agggregation to remove duplicates. For example given relation $r(A, B)$ possibly containing duplicates, ${}_{A,B}\mathcal{G}(r)$ would return a duplicate free version of the relation.

c. $\mathcal{G}_{\mathbf{max}(salary)}(instructor)$

d. $instructor \bowtie (\mathcal{G}_{\textbf{max}(salary)\ \textbf{as}\ salary}(instructor))$
Note that the above query renames the maximum salary as salary, so the subsequent natural join outputs only instructors with that salary.

e. $_{course\_id, section\_id}\mathcal{G}_{\textbf{count}(*)\ \textbf{as}\ enrollment}(\sigma_{year=2009 \wedge semester=\text{Autumn}}(takes))$

f. $t1 \leftarrow {}_{course\_id, section\_id}\mathcal{G}_{\textbf{count}(*)\ \textbf{as}\ enrollment}(\sigma_{year=2009 \wedge semester=\text{Autumn}}(takes))$
result = $\mathcal{G}_{\textbf{max}(enrollment)}(t1)$

g. $t2 \leftarrow \mathcal{G}_{\textbf{max}(enrollment)\ \textbf{as}\ enrollment}(t1)$
where $t1$ is as defined in the previous part of the question.
result = $t1 \bowtie t2$

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

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

b. Find the names of all employees in this database who do not work for "First Bank Corporation".

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

Answer:

a. $\Pi_{person\_name}((employee \bowtie manages) \bowtie_{(manager\_name = employee2.person\_name \wedge employee.street = employee2.street \wedge employee.city = employee2.city)} (\rho_{employee2}(employee)))$

b. The following solutions assume that all people work for exactly one company. If one allows people to appear in the database (e.g. in *employee*) but not appear in *works*, the problem is more complicated. We give solutions for this more realistic case later.

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

If people may not work for any company:

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

c. $\Pi_{person\_name}(works) - (\Pi_{works.person\_name}(works \bowtie_{(works.salary \leq works2.salary \wedge works2.company\_name = \text{"Small Bank Corporation"})} \rho_{works2}(works)))$

6.3 The natural outer-join operations extend the natural-join operation so that tuples from the participating relations are not lost in the result of the join.

Describe how the theta-join operation can be extended so that tuples from the left, right, or both relations are not lost from the result of a theta join.
Answer:

a. The left outer theta join of $r(R)$ and $s(S)$ ($r \mathbin{⟕}_\theta s$) can be defined as
$(r \bowtie_\theta s) \cup ((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The tuple of nulls is of size equal to the number of attributes in S.

b. The right outer theta join of $r(R)$ and $s(S)$ ($r \mathbin{⟖}_\theta s$) can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s)))$
The tuple of nulls is of size equal to the number of attributes in R.

c. The full outer theta join of $r(R)$ and $s(S)$ ($r \mathbin{⟗}_\theta s$) can be defined as
$(r \bowtie_\theta s) \cup ((null, null, \ldots, null) \times (s - \Pi_S(r \bowtie_\theta s))) \cup$
$((r - \Pi_R(r \bowtie_\theta s)) \times (null, null, \ldots, null))$
The first tuple of nulls is of size equal to the number of attributes in R, and the second one is of size equal to the number of attributes in S.

6.4 (**Division operation**): The division operator of relational algebra, "$\div$", 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. 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 $\Pi_{R-S}(r)$.

6.5 Let the following relation schemas be given:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Let relations $r(R)$ and $s(S)$ be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Answer:

a. $\{t \mid \exists q \in r\ (q[A] = t[A])\}$

b. $\{t \mid t \in r \wedge t[B] = 17\}$

c. $\{t \mid \exists p \in r\ \exists q \in s\ (t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = q[D] \wedge t[E] = q[E] \wedge t[F] = q[F])\}$

d. $\{t \mid \exists\, p \in r\, \exists q \in s\, (t[A] = p[A] \wedge t[F] = q[F] \wedge p[C] = q[D]\}$

6.6 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in the domain relational calculus that is equivalent to each of the following:

a. $\Pi_A(r_1)$

b. $\sigma_{B=17}(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

Answer:

a. $\{< t > \mid \exists\, p, q\, (< t, p, q > \in r_1)\}$

b. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge b = 17\}$

c. $\{< a, b, c > \mid < a, b, c > \in r_1 \vee < a, b, c > \in r_2\}$

d. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge < a, b, c > \in r_2\}$

e. $\{< a, b, c > \mid < a, b, c > \in r_1 \wedge < a, b, c > \notin r_2\}$

f. $\{< a, b, c > \mid \exists\, p, q\, (< a, b, p > \in r_1 \wedge < q, b, c > \in r_2)\}$

6.7 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in relational algebra for each of the following queries:

a. $\{< a > \mid \exists\, b\, (< a, b > \in r \wedge b = 7)\}$

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

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

Answer:

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

b. $r \bowtie s$

c. $\Pi_A(s \bowtie (\Pi_{r.A}(\sigma_{r.b>d.b}(r \times \rho_d(r)))))$

6.8 Consider the relational database of Figure 6.22 where the primary keys are underlined. Give an expression in tuple relational calculus for each of the following queries:

a. Find all employees who work directly for "Jones."

b. Find all cities of residence of all employees who work directly for "Jones."

c. Find the name of the manager of the manager of "Jones."

d. Find those employees who earn more than all employees living in the city "Mumbai."

Answer:

a.

$$\{t \mid \exists\, m \in manages\ (t[person\_name] = m[person\_name] \wedge m[manager\_name] = \text{'Jones'})\}$$

b.

$$\{t \mid \exists\, m \in manages\ \exists e \in employee(e[person\_name] = m[person\_name] \wedge m[manager\_name] = \text{'Jones'} \wedge t[city] = e[city])\}$$

c.

$$\{t \mid \exists\, m1 \in manages\ \exists m2 \in manages(m1[manager\_name] = m2[person\_name] \wedge m1[person\_name] = \text{'Jones'} \wedge t[manager\_name] = m2[manager\_name])\}$$

d.

$$\{t \mid \exists\, w1 \in works\ \neg\exists w2 \in works(w1[salary] < w2[salary] \exists e2 \in employee\ (w2[person\_name] = e2[person\_name] \wedge e2[city] = \text{'Mumbai'}))\}$$

6.9 Describe how to translate join expressions in SQL to relational algebra.
Answer: A query of the form

select $A1, A2, \ldots, An$
from $R1, R2, \ldots, Rm$
where P

can be translated into relational algebra as follows:
$\Pi_{A1, A2, \ldots, An}(\sigma_P(R1 \times R2 \times \ldots \times Rm))$
An SQL join expression of the form

$R1$ **natural join** $R2$

can be written as $R1 \bowtie R2$.
An SQL join expression of the form

$R1$ **join** $R2$ **on** (P)

can be written as $R1 \bowtie_P R2$.

An SQL join expression of the form

$$R1 \textbf{ join } R2 \textbf{ using } (A1, A2, \ldots, An)$$

can be written as $\Pi_S(R1 \bowtie_{R1.A1=R2.A1 \wedge R1.A2=R2.A2 \wedge \ldots R1.An=R2.An} R2)$
where S is $A1, A2, \ldots, An$ followed by all attributes of $R1$ other than $R1.A1, R1.A2, \ldots, R1.An$, followed by all attributes of $R2$ other than $R2.A1, R2.A2, \ldots, R2.An$,
The outer join versions of the SQL join expressions can be similarly written by using $\supset\!\bowtie$, $\bowtie\!\subset$ and $\supset\!\bowtie\!\subset$ in place of $\bowtie$.[1]
The most direct way to handle subqueries is to extend the relational algebra. To handle where clause subqueries, we need to allow selection predicates to contain nested relational algebra expressions, which can reference correlation attributes from outer level relations. Scalar subqueries can be similarly translated by allowing nested relational algebra expressions to appear in scalar expressions. An alternative approach to handling such subqueries used in some database systems, such as Microsoft SQL Server, introduces a new relational algebra operator called the Apply operator; see Chapter 30, page 1230-1231 for details. Without such extensions, translating subqueries into standard relational algebra can be rather complicated.

[1]The case of outer joins with the **using** clause is a little more complicated; with a right outer join it is possible that $R1.A1$ is null, but $R2.A1$ is not, and the output should contain the non-null value. The SQL **coalesce** function can be used, replacing S by **coalesce**($R1.A1$, $R2.A1$), **coalesce**($R1.A2$, $R2.A2$), ... **coalesce**($R1.An$, $R2.An$), followed by the other attributes of $R1$ and $R2$.

CHAPTER 7

Database Design and the E-R Model

Practice Exercises

7.1 **Answer:** The E-R diagram is shown in Figure 7.1. 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.

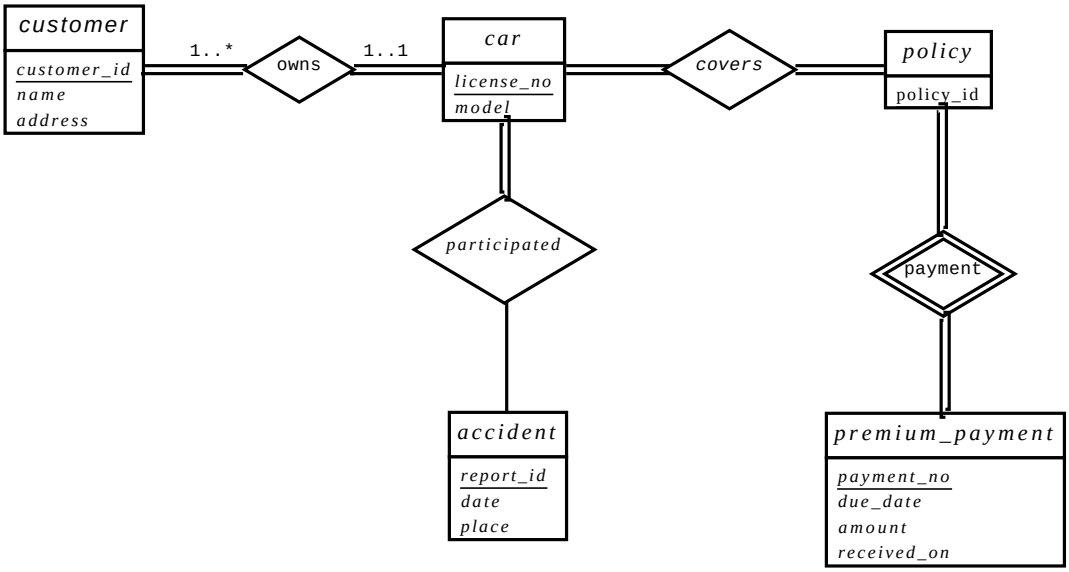


Figure 7.1 E-R diagram for a car insurance company.

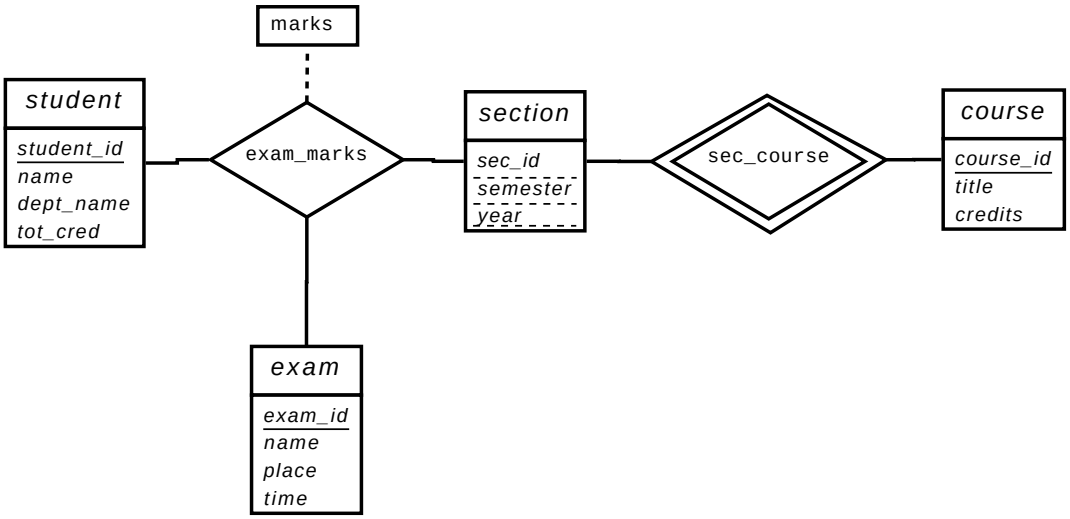


Figure 7.2 E-R diagram for marks database.

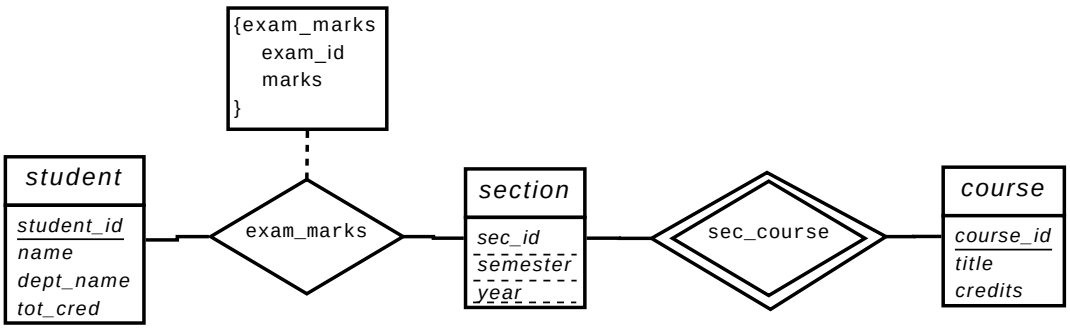


Figure 7.3 Another E-R diagram for marks database.

7.2 **Answer:** Note: the name of the relationship "course offering" needs to be changed to "section".

a. The E-R diagram is shown in Figure 7.2. Note that an alterantive is to model examinations as weak entities related to a section, rather than as a strong entity. 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 7.3. 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 would need to retain a separate entity corresponding to each exam.

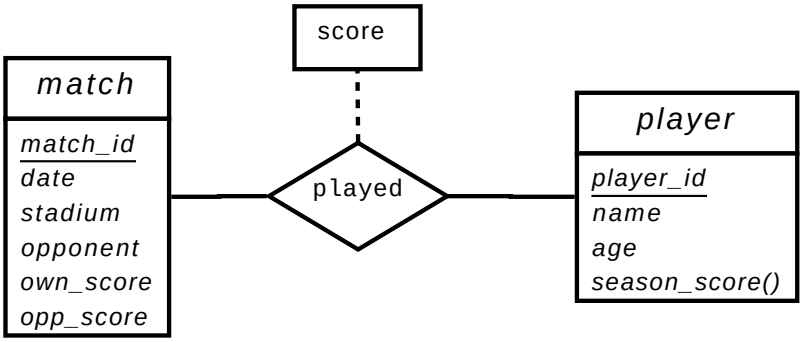


Figure 7.4 E-R diagram for favourite team statistics.

7.3 **Answer:** The diagram is shown in Figure 7.4.

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

7.5 **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 departments are completely independent of each other. If we split the graph into connected components, we have, in effect, a separate database corresponding to each connected component.

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.

7.6 **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 R exists which corresponds to E, R_A, R_B and R_C.

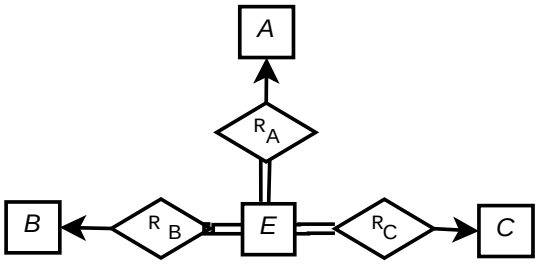


Figure 7.5 E-R diagram for Exercise 7.6b.

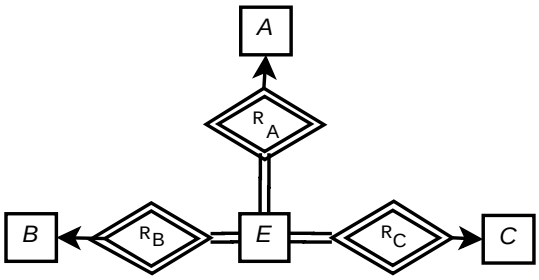


Figure 7.6 E-R diagram for Exercise 7.6d.

b. See Figure 7.5. 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.

d. Consider E as a weak entity set and R_A, R_B and R_C as its identifying relationship sets. See Figure 7.6.

7.7 **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.

7.8 **Answer:** In this example, the primary key of *section* consists of the attributes (*course_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.

7.9 **Answer:**

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

7.10 **Answer:** The foreign key attribute in R corresponding to 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 foreign key attribute corresponding to B as **null** which is not allowed.

7.11 **Answer:**

a. For the many-to-many case, the relationship must be represented as a separate relation which 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, repectively. 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 above constraint would be violated, and if we insert it in A before we insert it in B, a foreign key violation would occur.

7.12 **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 $Y.name$.

7.13 **Answer:**

a. The E-R diagram is shown in Figure 7.7. 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 sub attributes such as *start_time* or *value*.

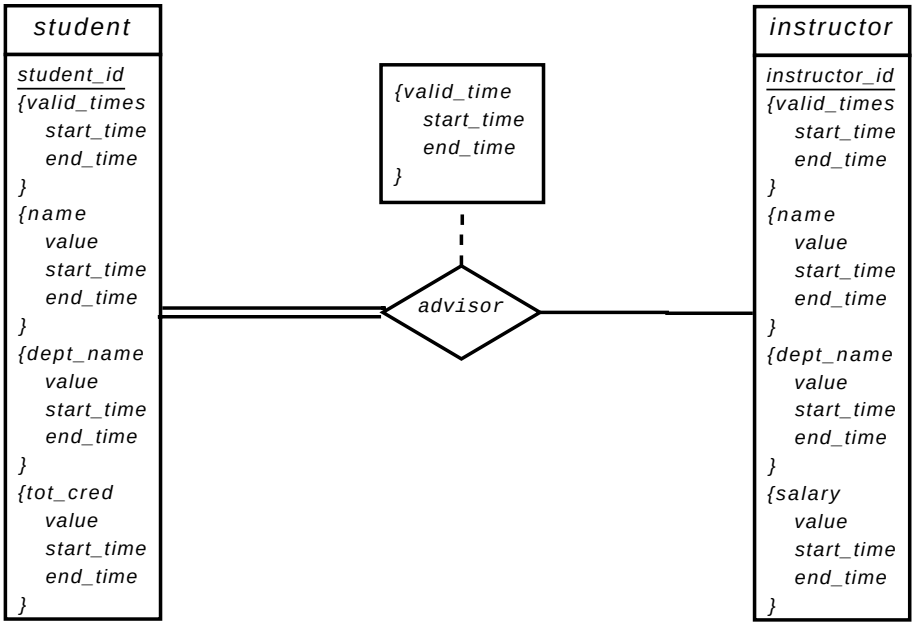


Figure 7.7 E-R diagram for Exercise 7.13

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.

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, intructor_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.

CHAPTER 8

Relational Database Design

Exercises

8.1 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-join 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-join 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 8.6), Therefore $R_1 \cap R_2 \rightarrow R_1$.

8.2 List all functional dependencies satisfied by the relation of Figure 8.17.
Answer: The nontrivial functional dependencies are: $A \rightarrow B$ and $C \rightarrow B$, and a dependency they logically imply: $AC \rightarrow B$. There are 19 trivial functional dependencies of the form $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$. 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.

8.3 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.

8.4 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

8.5 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

8.6 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.

8.7 Using the functional dependencies of Practice Exercise 8.6, compute the canonical
cover F_c.

Answer: The given set of FDs F is:-

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

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.

8.8 Consider the algorithm in Figure 8.18 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 8.8 (Section 8.4.2) 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 hypotheses is that if $\alpha \rightarrow A$ can be proved in n steps or less then $A \in result$. If there is a proof in $n+1$ 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.

8.9 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
        )
    )
```

8.10 Our discussion of lossless-join 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.)

8.11 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 is decomposed into 3NF using the algorithm in Section 8.5.2, what primary and foreign key dependencies would you expect will 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 schema because of a rule $\alpha \rightarrow \beta$, 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.

8.12 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,

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

8.13 Show that the decomposition in Practice Exercise 8.1 is not a dependency-preserving decomposition.
Answer: 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. We shall see in Practice Exercise 8.15 that $B \rightarrow D$ is indeed in F^+. Thus $B \rightarrow D$ is not preserved. Note that $CD \rightarrow ABCDE$ is also 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 an FD $B \rightarrow \alpha$ 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 $B^+ = BD$.

8.14 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless-join 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 8.8 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 Practice Exercise 8.12, 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-join decomposition into 3NF.

8.15 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless-join 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

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

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

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

8.16 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 our 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 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.

8.17 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 Practice Exercise 8.16, a relation schema R is said to be in 3NF if there is no non-prime 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 non-prime 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 non-prime attribute A is partially dependent on a candidate key α, then A is also transitively dependent on the key α.

Let A be a non-prime 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 non-prime, 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.

8.18 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:

$R(A, B, C)$
$A \twoheadrightarrow B$

```
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 8.18. An algorithm to compute α^+.

CHAPTER 9

Application Design and Development

Practice Exercises

9.1 What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs?

Answer: The CGI interface starts a new process to service each request, which has a significant operating system overhead. On the other hand, servelets are run as threads of an existing process, avoiding this overhead. Further, the process running threads could be the Web server process itself, avoiding interprocess communication which can be expensive. Thus, for small to moderate sized tasks, the overhead of Java is less than the overheads saved by avoiding process creating and communication.

For tasks involving a lot of CPU activity, this may not be the case, and using CGI with a C or C++ program may give better performance.

9.2 List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.

Answer: Most computers have limits on the number of simultaneous connections they can accept. With connectionless protocols, connections are broken as soon as the request is satisfied, and therefore other clients can open connections. Thus more clients can be served at the same time. A request can be routed to any one of a number of different servers to balance load, and if a server crashes another can take over without the client noticing any problem.

The drawback of connectionless protocols is that a connection has to be reestablished every time a request is sent. Also, session information has to be sent each time in form of cookies or hidden fields. This makes them slower than the protocols which maintain connections in case state information is required.

9.3 Consider a carelessly written Web application for an online-shopping site, which stores the price of each item as a hidden form variable in the Web page sent to the customer; when the customer submits the form, the information from the hidden form variable is used to compute the bill for the customer. What is the loophole in this scheme? (There was a real instance where the loophole was exploited by some customers of an online-shopping site, before the problem was detected and fixed.)

Answer: A hacker can edit the HTML source code of the Web page, and replace the value of the hidden variable price with whatever value they want, and use the modified Web page to place an order. The Web application would then use the user-modified value as the price of the product.

9.4 Consider another carelessly written Web application, which uses a servlet that checks if there was an active session, but does not check if the user is authorized to access that page, instead depending on the fact that a link to the page is shown only to authorized users. What is the risk with this scheme? (There was a real instance where applicants to a college admissions site could, after logging into the Web site, exploit this loophole and view information they were not authorized to see; the unauthorized access was however detected, and those who accessed the information were punished by being denied admission.)

Answer: Although the link to the page is shown only to authorized users, an unauthorized user may somehow come to know of the existence of the link (for example, from an unauthorized user, or via Web proxy logs). The user may then login to the system, and access the unauthorized page by entering its URL in the browser. If the check for user authorization was inadvertantly left out from that page, the user will be able to see the result of the page.

The HTTP referer attribute can be used to block a naive attempt to exploit such loopholes, by ensuring the referer value is from a valid page of the Web site. However, the referer attribute is set by the browser, and can be spoofed, so a malicious user can easily work around the referer check.

9.5 List three ways in which caching can be used to speed up Web server performance.

Answer: Caching can be used to improve performance by exploiting the commonalities between transactions.

a. If the application code for servicing each request needs to open a connection to the database, which is time consuming, then a pool of open connections may be created before hand, and each request uses one from those.

b. The results of a query generated by a request can be cached. If same request comes agian, or generates the same query, then the cached result can be used instead of connecting to database again.

c. The final webpage generated in response to a request can be cached. If the same request comes again, then the cached page can be outputed.

9.6 The netstat command (available on Linux and on Windows) shows the active network connections on a computer. Explain how this command can be used to find out if a particular Web page is not closing connections that it opened, or if connection pooling is used, not returning connections to the connection pool. You should account for the fact that with connection pooling, the connection may not get closed immediately.

Answer: The tester should run netstat to find all connections open to the machine/socket used by the database. (If the application server is separate from the database server, the command may be executed at either of the machines). Then, the Web page being tested should be accessed repeatedly (this can be automated by using tools such as JMeter to generate page accesses). The number of connections to the database would go from 0 to some value (depending on the number of connections retained in the pool), but after some time the number of connections should stop increasing. If the number keeps increasing, the code underlying the Web page is clearly not closing connections or returning the connection to the pool.

9.7 Testing for SQL-injection vulnerability:

a. Suggest an approach for testing an application to find if it is vulnerable to SQL injection attacks on text input.

b. Can SQL injection occur with other forms of input? If so, how would you test for vulnerability?

Answer:

a. One approach is to enter a string containing a single quote in each of the input text boxes of each of the forms provided by the application, to see if the application correctly saves the value. If it does not save the value correctly, and/or gives an error message, it is vulnerable to SQL injection.

b. Yes, SQL injection can even occur with selection inputs such as drop-down menus, by modifying the value sent back to the server when the input value is chosen, for example by editing the page directly, or in the browser's DOM tree. A test tool should be able to modify the values sent to the application, inserting a single quote into the value; the test tool should also spoof (modify) the HTTP refer attribute to be a valid value, to bypass any attempt by the application to check the refer attribute.

9.8 A database relation may have the values of certain attributes encrypted for security. Why do database systems not support indexing on encrypted attributes? Using your answer to this question, explain why database systems do not allow encryption of primary-key attributes.

Answer: It is not possible in general to index on an encrypted value, unless all occurrences of the value encrypt to the same value (and even in this case, only equality predicates would be supported). However, mapping all occurrences of a value to the same encrypted value is risky, since statistical analysis can be used to reveal common values, even without decryption; techniques based on adding random "salt" bits are used to prevent such analysis, but they make indexing impossible. One possible workaround is to store the index unencrypted, but then the index can be used to leak values. Another option is to keep the index encrypted, but then the database system should know the decryption key, to decrypt required parts of the index on the fly. Since this required modifying large parts of the database system code, databases typically do not support this option.
The primary-key constraint has to be checked by the database when tuples are inserted, and if the values are encrypted as above, the database system will not be able to detect primary key violations. Therefore database systems that support encryption of specified attributes do not allow primary-key attributes, or for that matter foreign-key attributes, to be encrypted.

9.9 Exercise 9.8 addresses the problem of encryption of certain attributes. However, some database systems support encryption of entire databases. Explain how the problems raised in Exercise 9.8 are avoided when the entire database is encrypted.

Answer: When the entire database is encrypted, it is easy for the database to perform decryption as data is fetched from disk into memory, so in-memory storage is unencrypted. With this option, everything in the database, including indices, is encrypted when on disk, but un-encrypted in memory. As a result, only the data access layer of the database system code needs to be modified to perform encryption, leaving other layers untouched. Thus, indices can be used unchanged, and primary-key and foreign-key constraints enforced without any change to the corresponding layers of the database system code.

9.10 Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?

Answer: The key problem with digital certificates (when used offline, without contacting the certificate issuer) is that there is no way to withdraw them.

For instance (this actually happened, but names of the parties have been changed) person C claims to be an employee of company X and get a new public key certified by the certifying authority A. Suppose the authority A incorrectly believed that C was acting on behalf of company X, it gives C a certificate $cert$. Now, C can communicate with person Y, who checks the certificate $cert$ presenetd by C, and believes the public key contained in $cert$ really belongs to X. Now C would communicate with Y using the public key, and Y trusts the communication is from company X.

Person Y may now reveal confidential information to C, or accept purchase order from C, or execute programs certified by C, based on the public key, thinking he is actually communicating with company X. In each case there is potential for harm to Y.

Even if A detects the impersonation, as long as Y does not check with A (the protocol does not require this check), there is no way for Y to find out that the certificate is forged.

If X was a certification authority itself, further levels of fake certificates can be created. But certificates that are not part of this chain would not be affected.

9.11 Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Answer: A scheme for storing passwords would be to encrypt each password (after adding randomly generated "salt" bits to prevent dictionary attacks), and then use a hash index on the user-id to store/access the encrypted password. The password being used in a login attempt is then encrypted (if randomly generated "salt" bits were used initially these bits should be stored with the user-id, and used when encrypting the user-supplied password). The encrypted value is then compared with the stored encrypted value of the correct password. An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist. Thus, "one-way" encryption functions, such as secure hashing functions, which do not support decryption can be used for this task. The secure hashing algorithm SHA-1 is widely used for such one-way encryption.

CHAPTER 10

Storage and File Structure

Practice Exercises

10.1 **Answer:** This arrangement has the problem that P_i and B_{4i-3} are on the same disk. So if that disk fails, reconstruction of B_{4i-3} is not possible, since data and parity are both lost.

10.2 **Answer:**

a. It is stored as an array containing physical page numbers, indexed by logical page numbers. This representation gives an overhead equal to the size of the page address for each page.

b. It takes 32 bits for every page or every 4096 bytes of storage. Hence, it takes 64 megabytes for the 64 gigabyte of flash storage.

c. If the mapping is such that, every p consecutive logical page numbers are mapped to p consecutive physical pages, we can store the mapping of the first page for every p pages. This reduces the in memory structure by a factor of p. Further, if p is an exponent of 2, we can avoid some of the least significant digits of the addresses stored.

10.3 **Answer:**

a. To ensure atomicity, a block write operation is carried out as follows:

   i. Write the information onto the first physical block.

   ii. When the first write completes successfully, write the same information onto the second physical block.

   iii. The output is declared completed only after the second write completes successfully.

   During recovery, each pair of physical blocks is examined. If both are identical and there is no detectable partial-write, then no further actions are necessary. If one block has been partially rewritten, then we replace its contents with the contents of the other block. If there

has been no partial-write, but they differ in content, then we replace the contents of the first block with the contents of the second, or vice versa. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates both copies) or results in no change.
The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

b. The idea is similar here. For any block write, the information block is written first followed by the corresponding parity block. At the time of recovery, each set consisting of the n^{th} block of each of the disks is considered. If none of the blocks in the set have been partially-written, and the parity block contents are consistent with the contents of the information blocks, then no further action need be taken. If any block has been partially-written, it's contents are reconstructed using the other blocks. If no block has been partially-written, but the parity block contents do not agree with the information block contents, the parity block's contents are reconstructed.

10.4 Answer:

a. Although moving record 6 to the space for 5, and moving record 7 to the space for 6, is the most straightforward approach, it requires moving the most records, and involves the most accesses.

b. Moving record 7 to the space for 5 moves fewer records, but destroys any ordering in the file.

c. Marking the space for 5 as deleted preserves ordering and moves no records, but requires additional overhead to keep track of all of the free space in the file. This method may lead to too many "holes" in the file, which if not compacted from time to time, will affect performance because of reduced availability of contiguous free records.

10.5 Answer: (We use "↑ i" to denote a pointer to record "i".)
The original file of Figure 10.7.

header				↑ 1
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				↑ 4
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

a. The file after **insert** (24556, Turnamian, Finance, 98000).

header				↑ 4
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

b. The file after **delete** record 2.

header				↑ 2
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2				↑ 4
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

The free record chain could have alternatively been from the header to 4, from 4 to 2, and finally from 2 to 6.

c. The file after **insert** (34556, Thompson, Music, 67000).

header				↑ 4
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2	34556	Thompson	Music	67000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

10.6 Answer:

The relation *section* with three tuples is as follows.

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-347	1	Fall	2009	Taylor	3128	C

The relation *takes* with five students for each section is as follows.

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-347	1	Fall	2009	A
12345	CS-101	1	Fall	2009	C
17968	BIO-301	1	Summer	2010	null
23856	CS-347	1	Fall	2009	A
45678	CS-101	1	Fall	2009	F
54321	CS-101	1	Fall	2009	A-
54321	CS-347	1	Fall	2009	A
59762	BIO-301	1	Summer	2010	null
76543	CS-101	1	Fall	2009	A
76543	CS-347	1	Fall	2009	A
78546	BIO-301	1	Summer	2010	null
89729	BIO-301	1	Summer	2010	null
98988	BIO-301	1	Summer	2010	null

The multitable clustering for the above two instances can be taken as:

BIO-301	1	Summer	2010	Painter	514	A
17968	BIO-301	1	Summer	2010	null	
59762	BIO-301	1	Summer	2010	null	
78546	BIO-301	1	Summer	2010	null	
89729	BIO-301	1	Summer	2010	null	
98988	BIO-301	1	Summer	2010	null	
CS-101	1	Fall	2009	Packard	101	H
00128	CS-101	1	Fall	2009	A	
12345	CS-101	1	Fall	2009	C	
45678	CS-101	1	Fall	2009	F	
54321	CS-101	1	Fall	2009	A-	
76543	CS-101	1	Fall	2009	A	
CS-347	1	Fall	2009	Taylor	3128	C
00128	CS-347	1	Fall	2009	A-	
12345	CS-347	1	Fall	2009	A	
23856	CS-347	1	Fall	2009	A	
54321	CS-347	1	Fall	2009	A	
76543	CS-347	1	Fall	2009	A	

10.7 **Answer:**

a. Everytime a record is inserted/deleted, check if the usage of the block has changed levels. In that case, update the corresponding

bits. Note that we don't need to access the bitmaps at all unless the usage crosses a boundary, so in most of the cases there is no overhead.

b. When free space for a large record or a set of records is sought, then multiple free list entries may have to be scanned before finding a proper sized one, so overheads are much higher. With bitmaps, one page of bitmap can store free info for many pages, so I/O spent for finding free space is minimal. Similarly, when a whole block or a large part of it is deleted, bitmap technique is more convenient for updating free space information.

10.8 **Answer:** Hash table is the common option for large database buffers. The hash function helps in locating the appropriate bucket, on which linear search is performed.

10.9 **Answer:**

a. MRU is preferable to LRU where $R_1 \bowtie R_2$ is computed by using a nested-loop processing strategy where each tuple in R_2 must be compared to each block in R_1. After the first tuple of R_2 is processed, the next needed block is the first one in R_1. However, since it is the least recently used, the LRU buffer management strategy would replace that block if a new block was needed by the system.

b. LRU is preferable to MRU where $R_1 \bowtie R_2$ is computed by sorting the relations by join values and then comparing the values by proceeding through the relations. Due to duplicate join values, it may be necessary to "back-up" in one of the relations. This "backing-up" could cross a block boundary into the most recently used block, which would have been replaced by a system using MRU buffer management, if a new block was needed.

Under MRU, some unused blocks may remain in memory forever. In practice, MRU can be used only in special situations like that of the nested-loop strategy discussed in Exercise 10.9a.

CHAPTER 11

Indexing and Hashing

Practice Exercises

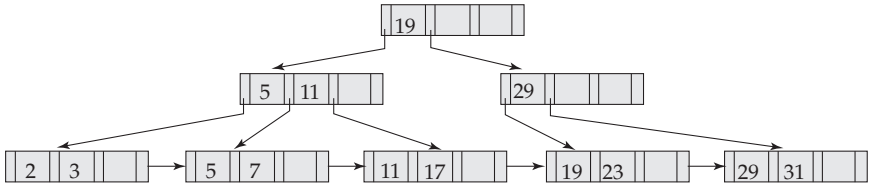
11.1 **Answer:** Reasons for not keeping indices on every attribute include:

- Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
- Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).
- Each extra index requires additional storage space.
- For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist.

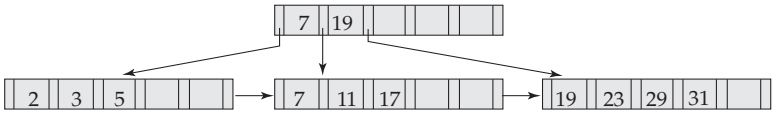
11.2 **Answer:** In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

11.3 **Answer:** The following were generated by inserting values into the B^+-tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil n/2 \rceil$ values/pointers.

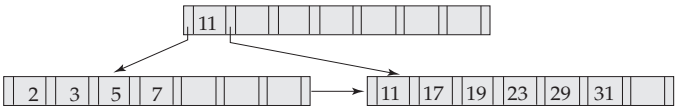
a.



b.



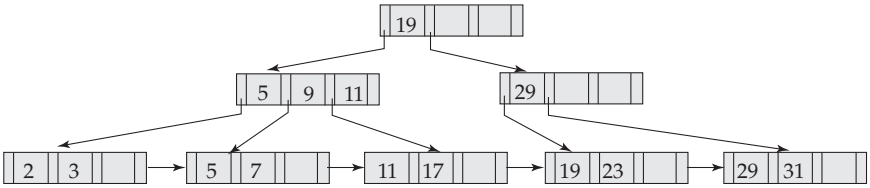
c.



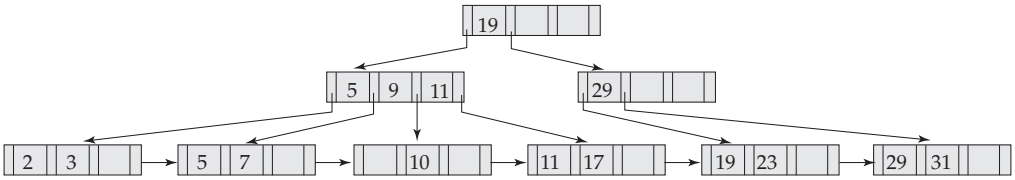
11.4 Answer:

- With structure 11.3.a:

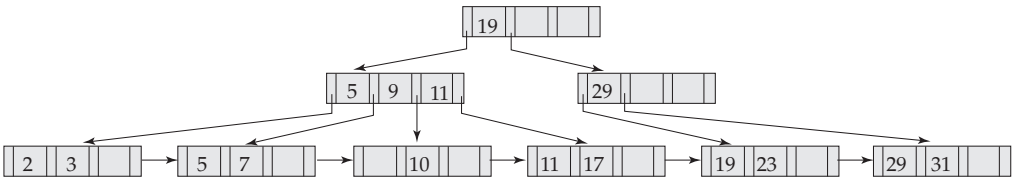
  Insert 9:



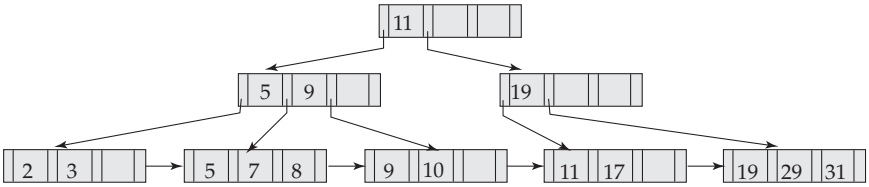
  Insert 10:



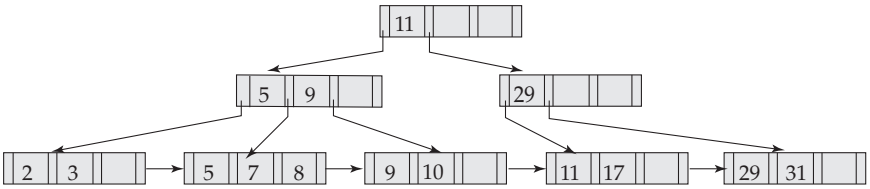
  Insert 8:



Delete 23:

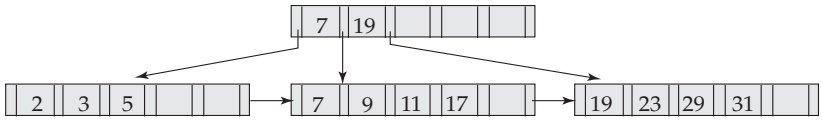


Delete 19:

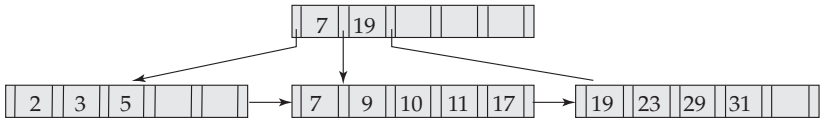


- With structure 11.3.b:

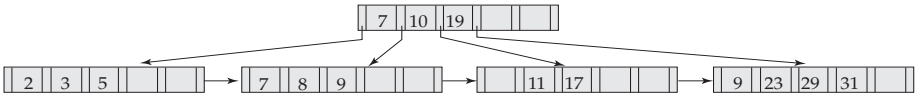
  Insert 9:



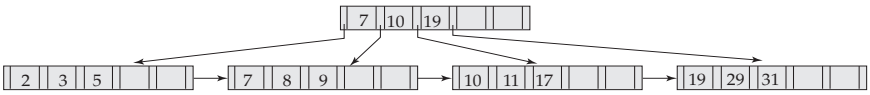
  Insert 10:



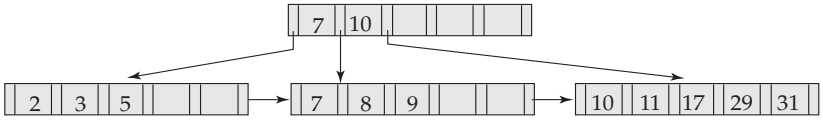
  Insert 8:



Delete 23:

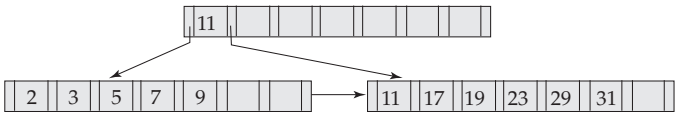


Delete 19:

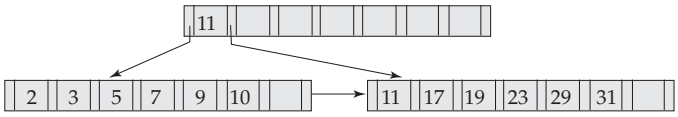


- With structure 11.3.c:

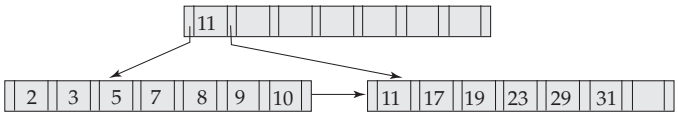
  Insert 9:



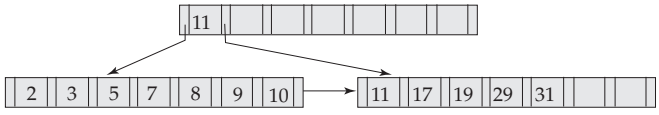
  Insert 10:



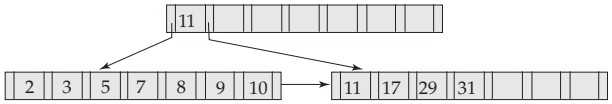
  Insert 8:



  Delete 23:

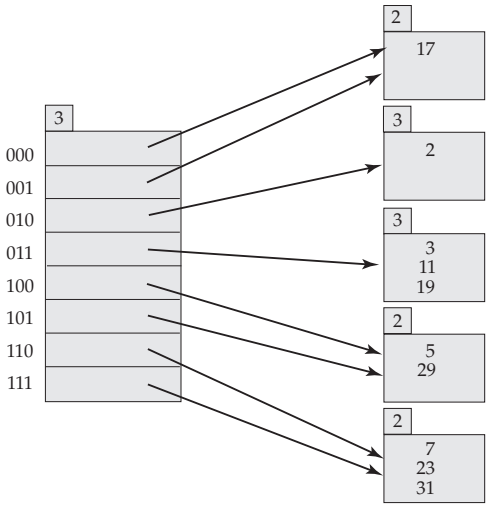


  Delete 19:



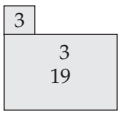
11.5 **Answer:** If there are K search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $log_{\lfloor (m-1)n/m \rfloor}(K)$

11.6 **Answer:** Extendable hash structure



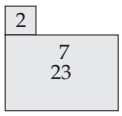
11.7 **Answer:**

a. Delete 11: From the answer to Exercise 11.6, change the third bucket to:

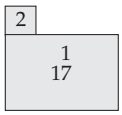


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

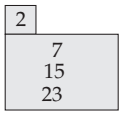
b. Delete 31: From the answer to 11.6, change the last bucket to:



c. Insert 1: From the answer to 11.6, change the first bucket to:



d. Insert 15: From the answer to 11.6, change the last bucket to:



11.8 **Answer:** The pseudocode is shown in Figure 11.1.

11.9 **Answer:** Let i denote the number of bits of the hash value used in the hash table. Let **bsize** denote the maximum capacity of each bucket. The pseudocode is shown in Figure 11.2.
Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket j differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

When merge is successful, further merging may be possible, which is handled by a recursive call to *coalesce* at the end of the function.

11.10 **Answer:** If the hash table is currently using i bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly i.
Consider a bucket j with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to i, then reset the count to 1. If the bucket is being split and i_j is one less that i, then increase the count by 1. It the bucket if being coalesced, and i_j is equal to i then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.
However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e. a count for each value of

```
function findIterator(value V) {
/* Returns an iterator for the search on the value V */
    Iterator iter();
    Set iter.value = V;
    Set C = root node
    while (C is not a leaf node) begin
        Let i = samllest number such that V <= C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m;
        end
        else Set C = C.P_i;
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i then begin
        Set iter.index = i;
        Set iter.page = C;
        Set iter.active = TRUE;
    end
    else if (V is the greater than the largest value in the leaf) then begin
        if (C.P_n.K_1 = V) then begin
            Set iter.page = C.P_n;
            Set iter.index = 1;
            Set iter.active = TRUE;
        end
        else Set iter.active = FALSE;
    end
    else Set iter.active = FALSE;
    return (iter)
}
Class Iterator {
    variables:
        value V /* The value on which the index is searched */
        boolean active /* Stores the current state of the iterator (TRUE or FALSE)*/
        int index /* Index of the next matching entry (if active is TRUE) */
        PageID page /* Page Number of the next matching entry (if active is TRUE)*/

    function next() {
        if (active) then begin
            Set retPage = page;
            Set retIndex = index;
            if (index + 1 = page.size) then begin
                page = page.P_n
                index = 0
            end
            else index = index + 1;
            if (page.K_index ≠ V)
                then active=FALSE;
            return(retPage, retIndex)
        end
        else return null;
    }
}
```

Figure 11.1 Pseudocode for findIterator and the Iterator class

```
delete(value K_l)
begin
    j = first i high-order bits of h(K_l);
    delete value K_l from bucket j;
    coalesce(bucket j);
end

coalesce(bucket j)
begin
    i_j = bits used in bucket j;
    k = any bucket with first (i_j − 1) bits same as that
        of bucket j while the bit i_j is reversed;
    i_k = bits used in bucket k;
    if(i_j ≠ i_k)
        return; /* buckets cannot be merged */
    if(entries in j + entries in k > bsize)
        return; /* buckets cannot be merged */
    move entries of bucket k into bucket j;

    decrease the value of i_j by 1;
    make all the bucket-address-table entries,
    which pointed to bucket k, point to j;

    coalesce(bucket j);
end
```

Figure 11.2 Pseudocode for deletion

common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when sufficient number of entries at the end of count array become 0.

11.11 **Answer:** We reproduce the instructor relation below.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

a. Bitmap for *salary*, with S_1, S_2, S_3 and S_4 representing the given intervals in the same order

S_1	0	0	1	0	0	0	0	0	0	0	0	0
S_2	0	0	0	0	0	0	0	0	0	0	0	0
S_3	1	0	0	0	1	0	0	1	0	0	0	0
S_4	0	1	0	1	0	1	1	0	1	1	1	1

b. The question is a bit trivial if there is no bitmap on the *dept_name* attribute. The bitmap for the *dept_name* attribute is:

Comp. Sci	1	0	0	0	0	0	1	0	0	0	1	0
Finance	0	1	0	0	0	0	0	0	1	0	0	0
Music	0	0	1	0	0	0	0	0	0	0	0	0
Physics	0	0	0	1	0	1	0	0	0	0	0	0
History	0	0	0	0	1	0	0	1	0	0	0	0
Biology	0	0	0	0	0	0	0	0	0	1	0	0
Elec. Eng.	0	0	0	0	0	0	0	0	0	0	0	1

To find all instructors in the Finance department with salary of 80000 or more, we first find the intersection of the Finance department bitmap and S_4 bitmap of *salary* and then scan on these records for salary of 80000 or more.
Intersection of Finance department bitmap and S_4 bitmap of *salary*.

S_4	0	1	0	1	0	1	1	0	1	1	1	1
Finance	0	1	0	0	0	0	0	0	1	0	0	0
$S_4 \cap$ Finance	0	1	0	0	0	0	0	0	1	0	0	0

Scan on these records with salary 80000 or more gives Wu and Singh as the instructors who satisfy the given query.

11.12 **Answer:** If the index entries are inserted in ascending order, the new entries get directed to the last leaf node. When this leaf node gets filled, it is split into two. Of the two nodes generated by the split, the left node is left untouched and the insertions takes place on the right node. This makes the occupancy of the leaf nodes to about 50 percent, except the last leaf.

If keys that are inserted are sorted in descending order, the above situation would still occur, but symmetrically, with the right node of a split never getting touched again, and occupancy would again be 50 percent for all nodes other than the first leaf.

11.13 **Answer:**

a. The cost to locate the page number of the required leaf page for an insertion is negligible since the non-leaf nodes are in memory. On the leaf level it takes one random disk access to read and one random disk access to update it along with the cost to write one page. Insertions which lead to splitting of leaf nodes require an additional page write. Hence to build a B^+-tree with n_r entries it takes a maximum of $2 * n_r$ random disk accesses and $n_r + 2 * (n_r/f)$ page writes. The second part of the cost comes from the fact that in the worst case each leaf is half filled, so the number of splits that occur is twice n_r/f.

   The above formula ignores the cost of writing non-leaf nodes, since we assume they are in memory, but in reality they would also be written eventually. This cost is closely approximated by $2*(n_r/f)/f$, which is the number of internal nodes just above the leaf; we can add further terms to account for higher levels of nodes, but these are much smaller than the number of leaves and can be ignored.

b. Substituting the values in the above formula and neglecting the cost for page writes, it takes about $10,000,000 * 20$ milliseconds, or 56 hours, since each insertion costs 20 milliseconds.

c.

function insert_in_leaf(value K, pointer P)
  if(tree is empty) create an empty leaf node L, which is also the root
  else Find the last leaf node in the leaf nodes chain L
  if (L has less than $n-1$ key values)
    then insert (K,P) at the first available location in L
  else begin
    Create leaf node $L1$
    Set $L.P_n = L1$;
    Set $K1$ = last value from page L
    insert_in_parent(1, L, $K1$, $L1$)
    insert (K,P) at the first location in $L1$
  end

function insert_in_parent(level l, pointer P, value K, pointer $P1$)
  if (level l is empty) **then begin**
    Create an empty non-leaf node N, which is also the root
    insert(P, K, $P1$) at the starting of the node N
    return
  else begin
    Find the right most node N at level l
    if (N has less than n pointers)
      then insert(K, $P1$) at the first available location in N
    else begin
      Create a new non-leaf page $N1$
      insert ($P1$) at the starting of the node N
      insert_in_parent($l+1$, pointer N, value K, pointer $N1$)
    end
  end

The insert_in_leaf function is called for each of the value, pointer pairs in ascending order. Similar function can also be build for descending order. The search for the last leaf or non-leaf node at any level can be avoided by storing the current last page details in an array.
The last node in each level might be less than half filled. To make this index structure meet the requirements of a B^+-tree, we can redistribute the keys of the last two pages at each level. Since the last but one node is always full, redistribution makes sure that both of then are at least half filled.

11.14 **Answer:** In a B^+-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leafs nodes that are contiguous in the tree. As insertions and deletions occur

on the tree, sequentiality is increasingly lost, and sequential access has to wait for disk seeks increasingly often.

a. One way to solve this problem is to rebuild the index to restore sequentiality.

b. i. In the worst case each n-block unit and each node of the B^+-tree is half filled. This gives the worst case occupancy as 25 percent.

ii. No. While splitting the n-block unit the first $n/2$ leaf pages are placed in one n-block unit, and the remaining in the second n-block unit. That is, every n-block split maintains the order. Hence, the nodes in the n-block units are consecutive.

iii. In the regular B^+-tree construction, the leaf pages might not be sequential and hence in the worst case, it takes one seek per leaf page. Using the block at a time method, for each n-node block, we will have at least $n/2$ leaf nodes in it. Each n-node block can be read using one seek. Hence the worst case seeks comes down by a factor of $n/2$.

iv. Allowing redistribution among the nodes of the same block, does not require additional seeks, where as, in regular B^+-tree we require as many seeks as the number of leaf pages involved in the redistribution. This makes redistribution for leaf blocks efficient with this scheme. Also the worst case occupancy comes back to nearly 50 percent. (Splitting of leaf nodes is preferred when the participating leaf nodes are nearly full. Hence nearly 50 percent instead of exact 50 percent)

CHAPTER 12

Query Processing

Practice Exercises

12.1 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most 3 blocks. Show the runs created on each pass of the sort-merge algorithm, when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

Answer: We will refer to the tuples (kangaroo, 17) through (baboon, 12) using tuple numbers t_1 through t_{12}. We refer to the j^{th} run used by the i^{th} pass, as r_{ij}. The initial sorted runs have three blocks each. They are:

$$
\begin{aligned}
r_{11} &= \{t_3, t_1, t_2\} \\
r_{12} &= \{t_6, t_5, t_4\} \\
r_{13} &= \{t_9, t_7, t_8\} \\
r_{14} &= \{t_{12}, t_{11}, t_{10}\}
\end{aligned}
$$

Each pass merges three runs. Therefore the runs after the end of the first pass are:

$$
\begin{aligned}
r_{21} &= \{t_3, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\} \\
r_{22} &= \{t_{12}, t_{11}, t_{10}\}
\end{aligned}
$$

At the end of the second pass, the tuples are completely sorted into one run:

$$
r_{31} = \{t_{12}, t_3, t_{11}, t_{10}, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}
$$

12.2 Consider the bank database of Figure 12.13, where the primary keys are underlined, and the following SQL query:

select *T.branch_name*
from *branch T*, *branch S*
where *T.assets* > *S.assets* **and** *S.branch_city* = "Brooklyn"

Write an efficient relational-algebra expression that is equivalent to this query. Justify your choice.

Answer:

Query:

$$\Pi_{T.branch\_name}((\Pi_{branch\_name,\, assets}(\rho_T(branch))) \bowtie_{T.assets\, >\, S.assets} (\Pi_{assets}\ (\sigma_{(branch\_city\, =\, \text{'Brooklyn'})}(\rho_S(branch)))))$$

This expression performs the theta join on the smallest amount of data possible. It does this by restricting the right hand side operand of the join to only those branches in Brooklyn, and also eliminating the unneeded attributes from both the operands.

12.3 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block transfers and seeks required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join.

b. Block nested-loop join.

c. Merge join.

d. Hash join.

Answer:

r_1 needs 800 blocks, and r_2 needs 1500 blocks. Let us assume M pages of memory. If $M > 800$, the join can easily be done in 1500 + 800 disk accesses, using even plain nested-loop join. So we consider only the case where $M \leq 800$ pages.

a. Nested-loop join:
Using r_1 as the outer relation we need $20000 * 1500 + 800 = 30,000,800$ disk accesses, if r_2 is the outer relation we need $45000 * 800 + 1500 = 36,001,500$ disk accesses.

b. Block nested-loop join:
If r_1 is the outer relation, we need $\lceil \frac{800}{M-1} \rceil * 1500 + 800$ disk accesses, if r_2 is the outer relation we need $\lceil \frac{1500}{M-1} \rceil * 800 + 1500$ disk accesses.

c. Merge-join:
Assuming that r_1 and r_2 are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil log_{M-1}(1500/M) \rceil +$

2) + 800(2⌈$log_{M-1}(800/M)$⌉ + 2) disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

d. Hash-join:
We assume no overflow occurs. Since r_1 is smaller, we use it as the build relation and r_2 as the probe relation. If $M > 800/M$, i.e. no need for recursive partitioning, then the cost is $3(1500+800) = 6900$ disk accesses, else the cost is $2(1500+800)\lceil log_{M-1}(800) - 1\rceil + 1500 + 800$ disk accesses.

12.4 The indexed nested-loop join algorithm described in Section 12.5.3 can be inefficient if the index is a secondary index, and there are multiple tuples with the same value for the join attributes. Why is it inefficient? Describe a way, using sorting, to reduce the cost of retrieving tuples of the inner relation. Under what conditions would this algorithm be more efficient than hybrid merge join?

Answer:
If there are multiple tuples in the inner relation with the same value for the join attributes, we may have to access that many blocks of the inner relation for each tuple of the outer relation. That is why it is inefficient. To reduce this cost we can perform a join of the outer relation tuples with just the secondary index leaf entries, postponing the inner relation tuple retrieval. The result file obtained is then sorted on the inner relation addresses, allowing an efficient physical order scan to complete the join.
Hybrid merge−join requires the outer relation to be sorted. The above algorithm does not have this requirement, but for each tuple in the outer relation it needs to perform an index lookup on the inner relation. If the outer relation is much larger than the inner relation, this index lookup cost will be less than the sorting cost, thus this algorithm will be more efficient.

12.5 Let r and s be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest-cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

Answer:
We can store the entire smaller relation in memory, read the larger relation block by block and perform nested loop join using the larger one as the outer relation. The number of I/O operations is equal to $b_r + b_s$, and memory requirement is $min(b_r, b_s) + 2$ pages.

12.6 Consider the bank database of Figure 12.13, where the primary keys are underlined. Suppose that a B^+-tree index on *branch_city* is available on relation *branch*, and that no other index is available. List different ways to handle the following selections that involve negation:

a. $\sigma_{\neg(branch\_city < \text{“Brooklyn”})}(branch)$

b. $\sigma_{\neg(branch\_city = \text{“Brooklyn”})}(branch)$

c. $\sigma_{\neg(branch\_city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

Answer:

a. Use the index to locate the first tuple whose *branch_city* field has value “Brooklyn”. From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch_city* field is anything other than “Brooklyn”.

c. This query is equivalent to the query

$$\sigma_{(branch\_city \geq \text{'Brooklyn'} \wedge assets < 5000)}(branch)$$

Using the *branch-city* index, we can retrieve all tuples with *branch-city* value greater than or equal to “Brooklyn” by following the pointer chains from the first “Brooklyn” tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

12.7 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Your pseudocode must define the standard iterator functions *open*(), *next*(), and *close*(). Show what state information the iterator must maintain between calls.

Answer: Let *outer* be the iterator which returns successive tuples from the pipelined outer relation. Let *inner* be the iterator which returns successive tuples of the inner relation having a given value at the join attributes. The *inner* iterator returns these tuples by performing an index lookup. The functions **IndexedNLJoin::open**, **IndexedNLJoin::close** and **IndexedNLJoin::next** to implement the indexed nested-loop join iterator are given below. The two iterators *outer* and *inner*, the value of the last read outer relation tuple t_r and a flag $done_r$ indicating whether the end of the outer relation scan has been reached are the state information which need to be remembered by **IndexedNLJoin** between calls.

```
IndexedNLJoin::open()
begin
    outer.open();
    inner.open();
    done_r := false;
    if(outer.next() ≠ false)
        move tuple from outer's output buffer to t_r;
    else
        done_r := true;
end
```

```
IndexedNLJoin::close()
begin
    outer.close();
    inner.close();
end
```

```
boolean IndexedNLJoin::next()
begin
    while(¬done_r)
    begin
        if(inner.next(t_r[JoinAttrs]) ≠ false)
        begin
            move tuple from inner's output buffer to t_s;
            compute t_r ⋈ t_s and place it in output buffer;
            return true;
        end
        else
            if(outer.next() ≠ false)
            begin
                move tuple from outer's output buffer to t_r;
                rewind inner to first tuple of s;
            end
            else
                done_r := true;
    end
    return false;
end
```

12.8 Design sort-based and hash-based algorithms for computing the relational division operation (see Practise Exercises of Chapter 6 for a definition of the division operation).

Answer: Suppose $r(T \cup S)$ and $s(S)$ be two relations and $r \div s$ has to be computed.

For sorting based algorithm, sort relation s on S. Sort relation r on (T, S). Now, start scanning r and look at the T attribute values of the first tuple. Scan r till tuples have same value of T. Also scan s simultaneously and check whether every tuple of s also occurs as the S attribute of r, in a fashion similar to merge join. If this is the case, output that value of T and proceed with the next value of T. Relation s may have to be scanned multiple times but r will only be scanned once. Total disk accesses, after

sorting both the relations, will be $|r| + N * |s|$, where N is the number of distinct values of T in r.
We assume that for any value of T, all tuples in r with that T value fit in memory, and consider the general case at the end. Partition the relation r on attributes in T such that each partition fits in memory (always possible because of our assumption). Consider partitions one at a time. Build a hash table on the tuples, at the same time collecting all distinct T values in a separate hash table. For each value of T, Now, for each value V_T of T, each value s of S, probe the hash table on (V_T, s). If any of the values is absent, discard the value V_T, else output the value V_T.
In the case that not all r tuples with one value for T fit in memory, partition r and s on the S attributes such that the condition is satisfied, run the algorithm on each corresponding pair of partitions r_i and s_i. Output the intersection of the T values generated in each partition.

12.9 What is the effect on the cost of merging runs if the number of buffer blocks per run is increased, while keeping overall memory available for buffering runs fixed?
Answer: Seek overhead is reduced, but the the number of runs that can be merged in a pass decreases potentially leading to more passes. A value of b_b that minimizes overall cost should be chosen.

CHAPTER 13

Query Optimization

Practice Exercises

13.1 Show that the following equivalences hold. Explain how you can apply them to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$, where θ uses only attributes from E_1.

Answer:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
Let us rename $(E_1 \bowtie_\theta (E_2 - E_3))$ as R_1, $(E_1 \bowtie_\theta E_2)$ as R_2 and $(E_1 \bowtie_\theta E_3)$ as R_3. It is clear that if a tuple t belongs to R_1, it will also belong to R_2. If a tuple t belongs to R_3, $t[E_3$'s attributes] will belong to E_3, hence t cannot belong to R_1. From these two we can say that

$$\forall t,\ t \in R_1 \ \Rightarrow\ t \in (R_2 - R_3)$$

It is clear that if a tuple t belongs to $R_2 - R_3$, then $t[R_2$'s attributes] $\in E_2$ and $t[R_2$'s attributes] $\notin E_3$. Therefore:

$$\forall t,\ t \in (R_2 - R_3) \ \Rightarrow\ t \in R_1$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side join will produce many tuples which will finally be removed from the result. The left hand side expression can be evaluated more efficiently.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.
θ uses only attributes from A. Therefore if any tuple t in the output of ${}_A\mathcal{G}_F(E)$ is filtered out by the selection of the left hand side, all the tuples in E whose value in A is equal to $t[A]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E)) \ \Rightarrow\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E))$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E)) \ \Rightarrow\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E))$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids performing the aggregation on groups which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$ where θ uses only attributes from E_1.
θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 \supset\!\bowtie E_2)$ is filtered out by the selection of the left hand side, all the tuples in E_1 whose value is equal to $t[E_1]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2) \ \Rightarrow\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2 \ \Rightarrow\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2)$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids producing many output tuples which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

13.2 For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent.

a. $\Pi_A(R - S)$ and $\Pi_A(R) - \Pi_A(S)$.

b. $\sigma_{B<4}({}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(R))$ and ${}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(\sigma_{B<4}(R))$.

c. In the preceding expressions, if both occurrences of max were replaced by min would the expressions be equivalent?

d. $(R \supset\!\bowtie S) \supset\!\bowtie T$ and $R \supset\!\bowtie (S \supset\!\bowtie T)$
In other words, the natural left outer join is not associative. (Hint: Assume that the schemas of the three relations are $R(a, b1)$, $S(a, b2)$, and $T(a, b3)$, respectively.)

e. $\sigma_\theta(E_1 \supset\!\bowtie E_2)$ and $E_1 \supset\!\bowtie \sigma_\theta(E_2)$, where θ uses only attributes from E_2.

Answer:

a. $R = \{(1, 2)\},\ S = \{(1, 3)\}$
The result of the left hand side expression is $\{(1)\}$, whereas the result of the right hand side expression is empty.

b. $R = \{(1, 2), (1, 5)\}$
The left hand side expression has an empty result, whereas the right hand side one has the result $\{(1, 2)\}$.

c. Yes, on replacing the max by the min, the expressions will become equivalent. Any tuple that the selection in the rhs eliminates would not pass the selection on the lhs if it were the minimum value, and would be eliminated anyway if it were not the minimum value.

d. $R = \{(1, 2)\}$, $S = \{(2, 3)\}$, $T = \{(1, 4)\}$. The left hand expression gives $\{(1, 2, null, 4)\}$ whereas the the right hand expression gives $\{(1, 2, 3, null)\}$.

e. Let R be of the schema (A, B) and S of (A, C). Let $R = \{(1, 2)\}$, $S = \{(2, 3)\}$ and let θ be the expression $C = 1$. The left side expression's result is empty, whereas the right side expression results in $\{(1, 2, null)\}$.

13.3 SQL allows relations with duplicates (Chapter 3).

a. Define versions of the basic relational-algebra operations σ, Π, $\times$, $\bowtie$, $-$, $\cup$, and $\cap$ that work on relations with duplicates, in a way consistent with SQL.

b. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational-algebra defined in part a.

Answer:

a. We define the multiset versions of the relational-algebra operators here. Given multiset relations r_1 and r_2,

i. σ
Let there be c_1 copies of tuple t_1 in r_1. If t_1 satisfies the selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$, otherwise there are none.

ii. Π
For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

iii. $\times$
If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, then there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

iv. $\bowtie$
The output will be the same as a cross product followed by a selection.

v. $-$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 - c_2$ copies of t in $r_1 - r_2$, provided that $c_1 - c_2$ is positive.

vi. $\cup$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 + c_2$ copies of t in $r_1 \cup r_2$.

vii. $\cap$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $min(c_1, c_2)$ copies of t in $r_1 \cap r_2$.

b. All the equivalence rules 1 through 7.b of section 13.2.1 hold for the multiset version of the relational-algebra defined in the first part. There exist equivalence rules which hold for the ordinary relational-algebra, but do not hold for the multiset version. For example consider the rule :-

$$A \cap B \quad = \quad A \cup B \; - \; (A - B) \; - \; (B - A)$$

This is clearly valid in plain relational-algebra. Consider a multiset instance in which a tuple t occurs 4 times in A and 3 times in B. t will occur 3 times in the output of the left hand side expression, but 6 times in the output of the right hand side expression. The reason for this rule to not hold in the multiset version is the asymmetry in the semantics of multiset union and intersection.

13.4 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys A, C, and E, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

Answer:

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because E is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute C for relation r_2 and on E for r_3. Then for each tuple in r_1, we do the following:
  a. Use the index for r_2 to look up at most one tuple which matches the C value of r_1.
  b. Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

13.5 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 13.4. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750

tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$ and give an efficient strategy for computing the join.

Answer: The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r_1, $1500/V(C, r_2)$ = 15/11 tuples (on the average) of r_2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r_3 to yield a result of approximately 10,227 tuples ($15000/11 \times 750/100 = 10227$). A good strategy should join r_1 and r_2 first, since the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

13.6 Suppose that a B^+-tree index on *building* is available on relation *department*, and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(building < \text{"Watson"})}(department)$

b. $\sigma_{\neg(building = \text{"Watson"})}(department)$

c. $\sigma_{\neg(building < \text{"Watson"} \vee budget < 50000)}(department)$

Answer:

a. Use the index to locate the first tuple whose *building* field has value "Watson". From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *building* field is anything other than "Watson".

c. This query is equivalent to the query:

$$\sigma_{building \geq \text{'Watson'} \wedge budget < 5000)}(department).$$

Using the *building* index, we can retrieve all tuples with *building* value greater than or equal to "Watson" by following the pointer chains from the first "Watson" tuple. We also apply the additional criteria of $budget < 5000$ on every tuple.

13.7 Consider the query:

```
select *
from r, s
where upper(r.A) = upper(s.A);
```

where "upper" is a function that returns its input argument with all lowercase letters replaced by the corresponding uppercase letters.

a. Find out what plan is generated for this query on the database system you use.

b. Some database systems would use a (block) nested-loop join for this query, which can be very inefficient. Briefly explain how hash-join or merge-join can be used for this query.

Answer:

a. First create relations r and s, and add some tuples to the two relations, before finding the plan chosen; or use existing relations in place of r and s. Compare the chosen plan with the plan chosen for a query directly equating $r.A = s.B$. Check the estimated statistics too. Some databases may give the same plan, but with vastly different statistics.
(On PostgreSQL, we found that the optimizer used the merge join plan described in the answer to the next part of this question.)

b. To use hash join, hashing should be done after applying the upper() function to $r.A$ and $s.A$. Similarly, for merge join, the relations should be sorted on the result of applying the upper() function on $r.A$ and $s.A$. The hash or merge join algorithms can then be used unchanged.

13.8 Give conditions under which the following expressions are equivalent

$$_{A,B}\mathcal{G}_{agg(C)}(E_1 \bowtie E_2) \quad \text{and} \quad (_{A}\mathcal{G}_{agg(C)}(E_1)) \bowtie E_2$$

where *agg* denotes any aggregation operation. How can the above conditions be relaxed if *agg* is one of **min** or **max**?
Answer: The above expressions are equivalent provided E_2 contains only attributes A and B, with A as the primary key (so there are no duplicates). It is OK if E_2 does not contain some A values that exist in the result of E_1, since such values will get filtered out in either expression. However, if there are duplicate values in $E_2.A$, the aggregate results in the two cases would be different.
If the aggregate function is min or max, duplicate A values do not have any effect. However, there should be no duplicates on (A, B); the first expression removes such duplicates, while the second does not.

13.9 Consider the issue of interesting orders in optimization. Suppose you are given a query that computes the natural join of a set of relations S. Given a subset $S1$ of S, what are the interesting orders of $S1$?
Answer: The interesting orders are all orders on subsets of attributes that can potentially participate in join conditions in further joins. Thus, let T be the set of all attributes of $S1$ that also occur in any relation in $S - S1$. Then every ordering of every subset of T is an interesting order.

13.10 Show that, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. *Hint:* A **complete binary tree** is one where every internal node has exactly two children. Use the fact that the number of different complete

binary trees with n leaf nodes is:

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

If you wish, you can derive the formula for the number of complete binary trees with n nodes from the formula for the number of binary trees with n nodes. The number of binary trees with n nodes is:

$$\frac{1}{n+1}\binom{2n}{n}$$

This number is known as the **Catalan number**, and its derivation can be found in any standard textbook on data structures or algorithms.

Answer: Each join order is a complete binary tree (every non-leaf node has exactly two children) with the relations as the leaves. The number of different complete binary trees with n leaf nodes is $\frac{1}{n}\binom{2(n-1)}{(n-1)}$. This is because there is a bijection between the number of complete binary trees with n leaves and number of binary trees with $n-1$ nodes. Any complete binary tree with n leaves has $n-1$ internal nodes. Removing all the leaf nodes, we get a binary tree with $n-1$ nodes. Conversely, given any binary tree with $n-1$ nodes, it can be converted to a complete binary tree by adding n leaves in a unique way. The number of binary trees with $n-1$ nodes is given by $\frac{1}{n}\binom{2(n-1)}{(n-1)}$, known as the Catalan number. Multiplying this by $n!$ for the number of permutations of the n leaves, we get the desired result.

13.11 Show that the lowest-cost join order can be computed in time $O(3^n)$. Assume that you can store and look up information about a set of relations (such as the optimal join order for the set, and the cost of that join order) in constant time. (If you find this exercise difficult, at least show the looser time bound of $O(2^{2n})$.)

Answer: Consider the dynamic programming algorithm given in Section 13.4. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x=2$. Thus the total cost is less than 3^n.

13.12 Show that, if only left-deep join trees are considered, as in the System R optimizer, the time taken to find the most efficient join order is around $n2^n$. Assume that there is only one interesting sort order.

Answer: The derivation of time taken is similar to the general case, except that instead of considering $2^{k+1}-2$ subsets of size less than or equal to

k for A, we only need to consider $k+1$ subsets of size exactly equal to k. That is because the right hand operand of the topmost join has to be a single relation. Therefore the total cost for finding the best join order for all subsets of size $k+1$ is $\binom{n}{k+1}(k+1)$, which is equal to $n\binom{n-1}{k}$. Summing over all k from 1 to $n-1$ using the binomial expansion of $(1+x)^{n-1}$ with $x=1$, gives a total cost of less than $n2^{n-1}$.

13.13 Consider the bank database of Figure 13.9, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Write a nested query on the relation *account* to find, for each branch with name starting with B, all accounts with the maximum balance at the branch.

b. Rewrite the preceding query, without using a nested subquery; in other words, decorrelate the query.

c. Give a procedure (similar to that described in Section 13.4.4) for decorrelating such queries.

Answer:

a. The nested query is as follows:

```
select  S.acount_number
from    account S
where   S.branch_name like 'B%' and
        S.balance =
        (select max(T.balance)
        from account T
        where T.branch_name = S.branch_name)
```

b. The decorrelated query is as follows:

```
create table t1 as
        select branch_name, max(balance)
        from account
        group by branch_name
select  account_number
from    account, t1
where   account.branch_name like 'B%' and
        account.branch_name = t1.branch_name and
        account.balance = t1.balance
```

c. In general, consider the queries of the form:

select $\cdots$
from L_1
where P_1 **and**
A_1 **op**
(**select f**(A_2)
from L_2
where P_2)

where, f is some aggregate function on attributes A_2, and *op* is some boolean binary operator. It can be rewritten as

create table t_1 **as**
select f(A_2),V
from L_2
where P_2^1
group by V
select $\cdots$
from L_1, t_1
where P_1 **and** P_2^2 **and**
A_1 **op** $t_1.A_2$

where P_2^1 contains predicates in P_2 without selections involving correlation variables, and P_2^2 introduces the selections involving the correlation variables. V contains all the attributes that are used in the selections involving correlation variables in the nested query.

13.14 The set version of the semijoin operator $\ltimes$ is defined as follows:

$$r \ltimes_\theta s = \Pi_R(r \bowtie_\theta s)$$

where R is the set of attributes in the schema of r. The multiset version of the semijoin operation returns the same set of tuples, but each tuple has exactly as many copies as it had in r.
Consider the nested query we saw in Section 13.4.4 which finds the names of all instructors who taught a course in 2007. Write the query in relational algebra using the multiset semjoin operation, ensuring that the number of duplicates of each name is the same as in the SQL query. (The semijoin operation is widely used for decorrelation of nested queries.)
Answer: The query can be written as follows:
$instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{year=2007}(teaches))$

CHAPTER 14

Transactions

Practice Exercises

14.1 **Answer:** Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

14.2 **Answer:** There are several steps in the creation of a file. A storage area is assigned to the file in the file system, a unique i-number is given to the file and an i-node entry is inserted into the i-list. Deletion of file involves exactly opposite steps.
For the file system user in UNIX, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor though, many of the internal file system actions need to have transaction semantics. All the steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

14.3 **Answer:** Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured. In contrast, most users of file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

14.4 **Answer:** If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average responce time will increase. Also when the transaction is reading data from disk, CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur.

14.5 **Answer:** Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practice are based on conflict serializability—they actually permit only a subset of

conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

14.6 **Answer:** There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, T_1, T_2, T_3, T_4, T_5.

14.7 **Answer:** A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.

14.8 **Answer:**

a. A schedule showing the Lost Update Anomaly:

T_1	T_2
read(A)	
	read(A)
	write(A)
write(A)	

In the above schedule, the value written by the transaction T_2 is lost because of the write of the transaction T_1.

b. Lost Update Anomaly in Read Committed Isolation Level

T_1	T_2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
write(A)	
unlock(A)	
commit	

The locking in the above schedule ensures the Read Committed isolation level. The value written by transaction T_2 is lost due to T_1's write.

c. Lost Update Anomaly is not possible in Repeatable Read isolation level. In repeatable read isolation level, a transaction T_1 reading a

data item X, holds a shared lock on X till the end. This makes it impossible for a newer transaction T_2 to write the value of X (which requires X-lock) until T_1 finishes. This forces the serialization order T_1, T_2 and thus the value written by T_2 is not lost.

14.9 **Answer:**

Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200 respectively.

Suppose that transaction T_1 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances. Suppose that concurrent transaction T_2 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances.

Since each of the transactions checks the integrity constraints on its own snapshot, if they run concurrently each will believe that the sum of the balances after the withdrawal is \$100 and therefore its withdrawal does not violate the integrity constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both of them can commit. This is a non-serializable execution which results into a serious problem of withdrawal of more money.

14.10 **Answer:**

Consider a web-based airline reservation system. There could be many concurrent requests to see the list of available flights and available seats in each flight and to book tickets. Suppose, there are two users A and B concurrently accessing this web application, and only one seat is left on a flight.

Suppose that both user A and user B execute transactions to book a seat on the flight, and suppose that each transaction checks the total number of seats booked on the flight, and inserts a new booking record if there are enough seats left. Let T_3 and T_4 be their respective booking transactions, which run concurrently. Now T_3 and T_4 will see from their snapshots that one ticket is available and insert new booking records. Since the two transactions do not update any common data item (tuple), snapshot isolation allows both transactions to commit. This results in an extra booking, beyond the number of seats available on the flight.

However, this situation is usually not very serious since cancellations often resolve the conflict; even if the conflict is present at the time the flight is to leave, the airline can arrange a different flight for one of the passengers on the flight, giving incentives to accept the change. Using snapshot isolation improves the overall performance in this case since the booking transactions read the data from their snapshots only and do not block other concurrent transactions.

14.11 **Answer:**

The given situation will not cause any problem for the definition of conflict serializability since the ordering of operations on each data item is necessary for conflict serializability, whereas the ordering of operations on different data items is not important.

T_1	T_2
read(A)	
	read(B)
write(B)	

For the above schedule to be conflict serializable, the only ordering requirement is **read**(B) -> **write**(B). **read**(A) and **read**(B) can be in any order. Therefore, as long as the operations on a data item can be totally ordered, the definition of conflict serializability should hold on the given multiprocessor system.

CHAPTER 15

Concurrency Control

Practice Exercises

15.1 **Answer:** Suppose two-phase locking does not ensure serializability. Then there exists a set of transactions T_0, T_1 ... T_{n-1} which obey 2PL and which produce a nonserializable schedule. A non-serializable schedule implies a cycle in the precedence graph, and we shall show that 2PL cannot produce such cycles. Without loss of generality, assume the following cycle exists in the precedence graph: $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{n-1} \rightarrow T_0$. Let α_i be the time at which T_i obtains its last lock (i.e. T_i's lock point). Then for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$. Then for the cycle we have

$$\alpha_0 < \alpha_1 < \alpha_2 < ... < \alpha_{n-1} < \alpha_0$$

Since $\alpha_0 < \alpha_0$ is a contradiction, no such cycle can exist. Hence 2PL cannot produce non-serializable schedules. Because of the property that for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$, the lock point ordering of the transactions is also a topological sort ordering of the precedence graph. Thus transactions can be serialized according to their lock points.

15.2 **Answer:**

a. Lock and unlock instructions:

T_{34}:
lock-S(A)
read(A)
lock-X(B)
read(B)
if $A = 0$
then $B := B + 1$
write(B)
unlock(A)
unlock(B)

T_{35}: **lock-S**(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

b. Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T_{31}	T_{32}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

The transactions are now deadlocked.

15.3 **Answer:** Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

15.4 **Answer:**
Consider two nodes A and B, where A is a parent of B. Let dummy vertex D be added between A and B. Consider a case where transaction T_2 has a lock on B, and T_1, which has a lock on A wishes to lock B, and T_3 wishes to lock A. With the original tree, T_1 cannot release the lock on A until it gets the lock on B. With the modified tree, T_1 can get a lock on D, and release the lock on A, which allows T_3 to proceed while T_1 waits for T_2. Thus, the protocol allows locks on vertices to be released earlier to other transactions, instead of holding them when waiting for a lock on a child. A generalization of idea based on edge locks is described in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

15.5 **Answer:** Consider the tree-structured database graph given below.

Schedule possible under tree protocol but not under 2PL:

T_1	T_2
lock *(A)*	
lock *(B)*	
unlock *(A)*	
	lock *(A)*
lock *(C)*	
unlock *(B)*	
	lock *(B)*
	unlock *(A)*
	unlock *(B)*
unlock *(C)*	

Schedule possible under 2PL but not under tree protocol:

T_1	T_2
lock *(A)*	
	lock *(B)*
lock *(C)*	
	unlock *(B)*
unlock *(A)*	
unlock *(C)*	

15.6 **Answer:**

The proof is in Kedem and Silberschatz, "Locking Protocols: From Exclusive to Shared Locks," JACM Vol. 30, 4, 1983. (The proof of serializability and deadlock freedom of the original tree protocol may be found in Silberschatz and Kedem, "Consistency in Hierarchical Database Systems", JACM Vol. 27, 1, Jan 1980.)

It is worth noting that if the protocol were modified to allow update transactions to lock any data item first, non-serializable executions can occur. Intuitively, a long running readonly transaction T_0 may precede an update transaction T_1 on node a, but continues to holds a lock on child node b. After T_1 updates a and commits, readonly transaction T_2 reads a, and thus T_1 precedes T_2. However, T_2 now overtakes T_1, share locking b and its child c (not possible with only exclusive locks), and reads c and

commits. Subsequently, transaction T_3 updates c and commits, after which T_0 share locks and reads c. Thus, there is a cycle

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_0$$

In effect, two readonly transactions see two different orderings of the same two update transactions. By requiring update transactions to always start at the root, the protocol ensures that the above situation cannot occur. In the above example T_3 would be forced to start from the root, and thus T_0 cannot be serialized after T_3.
The formal proof is by induction on the number of read-only transactions (and, we must admit, not very intuitive). We present it after introducing some notation first:
Q^X : the set of update transactions.
Q^S :the set of Read-only transactions.
$L(T_i)$: the set of data items locked by transaction T_i
Let Q the new protocol. Suppose T_i and T_k are update transactions, and T_j a read only transaction; then if T_j overlaps with T_i and T_k in the set of data items it accesses, since T_i and T_k both start from the root, T_i and T_k must also overlap on the set of data items accessed. Formally,
$\forall T_i \in Q^X \forall T_j \in Q^S \forall T_k \in Q^X$
$[((L(T_i) \bigcap L(T_j)) \neq \phi \wedge (L(T_j) \bigcap L(T_k)) \neq \phi) \Rightarrow (L(T_i) \bigcap L(T_k)) \neq \phi]$
Consider an arbitrary schedule of transactions in protocol Q. Let k be the number of Read-only transactions which participate in this schedule. We show by induction that there exist no minimal cycles in $(T, \rightarrow)$, where $T_i \rightarrow T_j$ denotes that T_i precedes T_j due to a conflict on some data item.
Let the cycle without loss of generality be,
$T_0 \rightarrow T_1 ... \rightarrow T_{m-1} \rightarrow T_0$
Here, we know that $m \geq 2k$, as there must be an Update transaction on each side of a Read-only transaction, otherwise the cycle will not be formed.

- $k = 0$: In this case each T_i is in Q^X, so Q becomes the original tree protocol which ensures the serializability and deadlock freedom.
- $k = 1$: Let T_i be the unique transaction in Q^S. We can replace T_i by T_i^X. The resulting schedule follows the original tree protocol which ensures serializability and deadlock freedom.
- $k > 1$: Let $T_i \wedge Q^S$, then $\{T_{i-1}, T_{i+1}\} \in Q^X$. As, $m \geq 2k \geq 4, i-1 \neq i+1$ (modulo m). By the assumption of the given protocol's definition,
  $L(T_{i-1}) \bigcap L(T_{i+1}) \neq \phi$
  Thus, $T_{i-1} \rightarrow T_{i+1}$ or $T_{i+1} \rightarrow T_{i-1}$; in either case, the original cycle was not minimal (in the first case, T_i can be deleted from the cycle, in the second case there is a cycle involving T_{i-i}, T_i and T_{i+1}.

By contradiction, there can be no such cycle, and thus the new protocol ensures serializability.

To show deadlock freedom, consider the waits for graph. If there is any edge from an update transaction T_j to another update transaction T_i, it is easy to see that T_i locks the root before T_j. Similarly if there is an edge from an update txn T_j to any readonly transaction T_k, which in turn can only wait for an update transaction T_i, it can be seen that T_i must have locked the root before T_j. In other words, an older update transaction cannot wait (directly or indirectly) on a younger update transaction, and thus, there can be no cycle in the waits for graph.

15.7 **Answer:**
The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979. The proof is quite non-trivial, and you may skip the details if you wish.
Consider,

- $G(V, A)$: the directed acyclic graph of the data items.
- T_0, T_1, T_2,.....,T_m are the participating transactions.
- $E(T_i)$ is the first vertex locked by transaction T_i.
- $\eta : V \rightarrow \{1,2,3,.... \}$ such that for each $u, w \in V$ if $< u, w > \in A$ then $\eta(u) < \eta(w)$
- $F(T_i, T_j)$ is that $v \in L(T_i) \bigcap L(T_j)$ for which $\eta(v)$ is minimal
- For each $e \in V$, define a relation $\omega_e \subseteq T \times T$ such that $T_i \omega_e T_j$ for $e \in L(T_i) \bigcap L(T_j)$ iff T_i successfully locked e and T_j either never (successfully) locked e or locked e, only after T_i unlocked it.
- Define also, $T_i \omega T_j = \exists e[T_i \omega_e T_j]$

a. **Lemma 1** We first show that, for the given protocol,
If $L(T_i) \bigcap L(T_j) \neq \phi$ then $F(T_i, T_j) \in \{E(T_i), E(T_j)\}$
Assume by contradiction that,
$F(T_i, T_j) \notin \{E(T_i), E(T_j)\}$
But then, by the locking protocol (as both T_i and T_j) had to lock more than **half** of the predecessors of $F(T_i, T_j)$), it follows that some predecessor of $F(T_i, T_j)$ is in $L(T_i) \bigcap L(T_j)$ contradicting the definition of $F(T_i, T_j)$.

b. **Lemma 2** Now we show that,
$T_i \omega_u T_j$ for all $u \in L(T_i) \bigcap L(T_j)$ iff $T_i \omega_{F(T_i,T_j)} T_j$.

If $u = F(T_i, T_j)$ then $T_i \omega_{F(T_i,T_j)} T_j$. is trivially true. If $u \neq F(T_i, T_j)$, then $u \notin \{E(T_i), E(T_j)\}$. It thus follows that some predecessor of w of u was successfully locked by both T_i and T_j and this u was

locked by T_i and T_j when they issued the instructions to lock u. Thus, $T_i \omega_w T_j \Leftrightarrow T_i \omega_u T_j$ (and $T_j \omega_w T_i \Leftrightarrow T_j \omega_u T_i$).
Now by induction: $T_i \omega_{F(T_i,T_j)} T_j \Leftrightarrow T_i \omega_w T_j$ and the result follows.

Now, we prove that the given protocol ensures serializability and deadlock freedom by induction on the length of minimal cycle.

a. $m = 2$: The protocol ensures no minimal cycles as shown in the above Lemma 2

b. $m > 2$: Assume by contradiction that
$T_0 \omega T_1 \omega T_2 \ldots\ldots \omega T_{m-1} \omega T_0$ is a minimal cycle of length m. We will consider two cases:

i. $F(T_i, T_{i+1})$'s are not all distinct. It follows that,

$$L(T_i) \bigcap L(T_j) \bigcap L(T_k) \neq \phi \text{ for } 0 \leq i \leq j \leq k \leq m-1$$

A. Assume $< i, j, k > = < i, i+1, i+2 >$. Then it easily follows that $T_i \omega T_{i+2}$ and (*) is not a minimal cycle.

B. Assume $< i, j, k > \neq < i, i+1, i+2 >$. If say $|j - i| > 1$ then as either $T_i \omega T_j$ or $T_j \omega T_i$ and (*) is not a minimal cycle. If $|j - i| = 1$ and $|k - j| > 1$ then the proof is analogous..

ii. All $F(T_i, T_{i+1})$'s are distinct. Then for some i

$$\eta(F(T_i, T_{i+1})) < \eta(F(T_{i+1}, T_{i+2})) \qquad (**)$$

$$\text{and } \eta(F(T_{i+1}, T_{i+2})) > \eta(F(T_{i+2}, T_{i+3})) \qquad (***)$$

By (**), $E(T_{i+1}) \neq F(T_{i+1}, T_{i+2})$ and by (***), $E(T_{i+2}) \neq F(T_{i+1}, T_{i+2})$.

Thus, $F(T_{i+1}, T_{i+2}) \notin \{E(T_{i+1}), E(T_{i+2})\}$, a contradiction to Lemma 1.

We have thus shown that the given protocol ensures serializability and deadlock freedom.

15.8 Answer:
The proof is Silberschatz and Kedem, "A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs", IEEE Trans. on Software Engg. Vol. SE-8, No. 6, Nov 1982.
The proof is rather complex; we omit details, which may be found in the above paper.

15.9 **Answer:** The access protection mechanism can be used to implement page level locking. Consider reads first. A process is allowed to read a page only after it read-locks the page. This is implemented by using `mprotect` to initially turn off read permissions to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler associated with protection violation then requests a read lock on the page, and after the lock is acquired, it uses `mprotect` to allow read access to the page by the process, and finally allows the process to continue. Write access is handled similarly.

15.10 **Answer:**

a. Serializability can be shown by observing that if two transactions have an I mode lock on the same item, the increment operations can be swapped, just like read operations. However, any pair of conflicting operations must be serialized in the order of the lock points of the corresponding transactions, as shown in Exercise 15.1.

b. The **increment** lock mode being compatible with itself allows multiple incrementing transactions to take the lock simultaneously thereby improving the concurrency of the protocol. In the absence of this mode, an **exclusive** mode will have to be taken on a data item by each transaction that wants to increment the value of this data item. An exclusive lock being incompatible with itself adds to the lock waiting time and obstructs the overall progress of the concurrent schedule.
In general, increasing the **true** entries in the compatibility matrix increases the concurrency and improves the throughput.

The proof is in Korth, "Locking Primitives in a Database System," JACM Vol. 30, 1983.

15.11 **Answer:** It would make no difference. The write protocol is such that the most recent transaction to write an item is also the one with the largest timestamp to have done so.

15.12 **Answer:** If a transaction needs to access a large a set of items, multiple granularity locking requires fewer locks, whereas if only one item needs to be accessed, the single lock granularity system allows this with just one lock. Because all the desired data items are locked and unlocked together in the multiple granularity scheme, the locking overhead is low, but concurrency is also reduced.

15.13 **Answer:** In the concurrency control scheme of Section 15.5 choosing **Start**(T_i) as the timestamp of T_i gives a subset of the schedules allowed by choosing **Validation**(T_i) as the timestamp. Using **Start**(T_i) means that whoever started first must finish first. Clearly transactions could enter the validation phase in the same order in which they began executing,

but this is overly restrictive. Since choosing **Validation**(T_i) causes fewer nonconflicting transactions to restart, it gives the better response times.

15.14 **Answer:**

- Two-phase locking: Use for simple applications where a single granularity is acceptable. If there are large read-only transactions, multiversion protocols would do better. Also, if deadlocks must be avoided at all costs, the tree protocol would be preferable.
- Two-phase locking with multiple granularity locking: Use for an application mix where some applications access individual records and others access whole relations or substantial parts thereof. The drawbacks of 2PL mentioned above also apply to this one.
- The tree protocol: Use if all applications tend to access data items in an order consistent with a particular partial order. This protocol is free of deadlocks, but transactions will often have to lock unwanted nodes in order to access the desired nodes.
- Timestamp ordering: Use if the application demands a concurrent execution that is equivalent to a particular serial ordering (say, the order of arrival), rather than *any* serial ordering. But conflicts are handled by roll-back of transactions rather than waiting, and schedules are not recoverable. To make them recoverable, additional overheads and increased response time have to be tolerated. Not suitable if there are long read-only transactions, since they will starve. Deadlocks are absent.
- Validation: If the probability that two concurrently executing transactions conflict is low, this protocol can be used advantageously to get better concurrency and good response times with low overheads. Not suitable under high contention, when a lot of wasted work will be done.
- Multiversion timestamp ordering: Use if timestamp ordering is appropriate but it is desirable for read requests to never wait. Shares the other disadvantages of the timestamp ordering protocol.
- Multiversion two-phase locking: This protocol allows read-only transactions to always commit without ever waiting. Update transactions follow 2PL, thus allowing recoverable schedules with conflicts solved by waiting rather than roll-back. But the problem of deadlocks comes back, though read-only transactions cannot get involved in them. Keeping multiple versions adds space and time overheads though, therefore plain 2PL may be preferable in low conflict situations.

15.15 **Answer:** A transaction waits on *a.* disk I/O and *b.* lock acquisition. Transactions generally wait on disk reads and not on disk writes as disk

writes are handled by the buffering mechanism in asynchronous fashion and transactions update only the in-memory copy of the disk blocks.

The technique proposed essentially separates the waiting times into two phases. The first phase – where transaction is executed without acquiring any locks and without performing any writes to the database – accounts for almost all the waiting time on disk I/O as it reads all the data blocks it needs from disk if they are not already in memory. The second phase—the transaction re-execution with strict two-phase locking—accounts for all the waiting time on acquiring locks. The second phase may, though rarely, involve a small waiting time on disk I/O if a disk block that the transaction needs is flushed to memory (by buffer manager) before the second phase starts.

The technique may increase concurrency as transactions spend almost no time on disk I/O with locks held and hence locks are held for shorter time. In the first phase the transaction reads all the data items required —and not already in memory—from disk. The locks are acquired in the second phase and the transaction does almost no disk I/O in this phase. Thus the transaction avoids spending time in disk I/O with locks held.

The technique may even increase disk throughput as the disk I/O is not stalled for want of a lock. Consider the following scenario with strict two-phase locking protocol: A transaction is waiting for a lock, the disk is idle and there are some item to be read from disk. In such a situation disk bandwidth is wasted. But in the proposed technique, the transaction will read all the required item from the disk without acquiring any lock and the disk bandwidth may be properly utilized.

Note that the proposed technique is most useful if the computation involved in the transactions is less and most of the time is spent in disk I/O and waiting on locks, as is usually the case in disk-resident databases. If the transaction is computation intensive, there may be wasted work. An optimization is to save the updates of transactions in a temporary buffer, and instead of reexecuting the transaction, to compare the data values of items when they are locked with the values used earlier. If the two values are the same for all items, then the buffered updates of the transaction are executed, instead of reexecuting the entire transaction.

15.16 **Answer:** Consider two transactions T_1 and T_2 shown below.

T_1	T_2
write(p)	
	read(p)
	read(q)
write(q)	

Let TS(T_1) < TS(T_2) and let the timestamp test at each operation except write(q) be successful. When transaction T_1 does the timestamp test for write(q) it finds that TS(T_1) < R-timestamp(q), since TS(T_1) < TS(T_2) and R-timestamp(q) = TS(T_2). Hence the writeoperation fails and transaction T_1 rolls back. The cascading results in transaction T_2 also being rolled back as it uses the value for item p that is written by transaction T_1.
If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

15.17 **Answer:** In the text, we considered two approaches to dealing with the phantom phenomenon by means of locking. The coarser granularity approach obviously works for timestamps as well. The B^+-tree index based approach can be adapted to timestamping by treating index buckets as data items with timestamps associated with them, and requiring that all read accesses use an index. We now show that this simple method works. Suppose a transaction T_i wants to access all tuples with a particular range of search-key values, using a B^+-tree index on that search-key. T_i will need to read all the buckets in that index which have key values in that range. It can be seen that any delete or insert of a tuple with a key-value in the same range will need to write one of the index buckets read by T_i. Thus the logical conflict is converted to a conflict on an index bucket, and the phantom phenomenon is avoided.

15.18 **Answer:** Note: The tree-protocol of Section 15.1.5 which is referred to in this question, is different from the multigranularity protocol of Section 15.3 and the B^+-tree concurrency protocol of Section 15.10.

One strategy for early lock releasing is given here. Going down the tree from the root, if the currently visited node's child is not full, release locks held on all nodes except the current node, request an X-lock on the child node, after getting it release the lock on the current node, and then descend to the child. On the other hand, if the child is full, retain all locks held, request an X-lock on the child, and descend to it after getting the lock. On reaching the leaf node, start the insertion procedure. This strategy results in holding locks only on the full index tree nodes from the leaf upwards, until and including the first non-full node.

An optimization to the above strategy is possible. Even if the current node's child is full, we can still release the locks on all nodes but the current one. But after getting the X-lock on the child node, we split it right away. Releasing the lock on the current node and retaining just the lock on the appropriate split child, we descend into it making it the current node. With this optimization, at any time at most two locks are held, of a parent and a child node.

15.19 **Answer:**

a. validation test for first-committer-wins scheme: Let Start(T_i), Commit(T_i) and be the timestamps associated with a transaction T_i and the update set for T_i be update_set(T_i). Then for all transactions T_k with

Commit(T_k) < Commit(T_i), one of the following two conditions must hold:

- If Commit(T_k) < Start(T_k), T_k completes its execution before T_i started, the serializability is maintained.
- If Start(T_i) < Commit(T_k) < Commit(T_i) and update_set(T_i) and update_set(T_k) do not intersect

b. Validation test for first-committer-wins scheme with W-timestamps for data items: If a transaction T_i writes a data item Q, then the W-timestamp(Q) is set to Commit(T_i). For the validation test of a transaction T_i to pass, following condition must hold:

- For each data item Q written by T_i, W-timestamp(Q) < Start(T_i)

c. First-updater-wins scheme:

i. For a data item Q written by T_i, the W-timestamp is assigned the timestamp when the write occurred in T_i

ii. Since the validation is done after acquiring the exclusive locks and the exclusive locks are held till the end of the transaction, the data item cannot be modified inbetween the lock acquisition and commit time. So, the result of validation test for a transaction would be the same at the commit time as that at the update time.

iii. Because of the exclusive locking, at the most one transaction can acquire the lock on a data item at a time and do the validation testing. Thus, two or more transactions can not do validation testing for the same data item simultaneously.

CHAPTER 16

Recovery System

Practice Exercises

16.1 Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.

Answer: Within a single transaction in undo-list, suppose a data item is updated more than once, say from 1 to 2, and then from 2 to 3. If the undo log records are processed in forward order, the final value of the data item would be incorrectly set to 2, whereas by processing them in reverse order, the value is set to 1. The same logic also holds for data items updated by more than one transaction on undo-list.

Using the same example as above, but assuming the transaction committed, it is easy to see that if redo processing processes the records in forward order, the final value is set correctly to 3, but if done in reverse order, the final value would be set incorrectly to 2.

16.2 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a media (disk) failure?

Answer: Checkpointing is done with log-based recovery schemes to reduce the time required for recovery after a crash. If there is no checkpointing, then the entire log must be searched after a crash, and all transactions undone/redone from the log. If checkpointing had been performed, then most of the log-records prior to the checkpoint can be ignored at the time of recovery.

Another reason to perform checkpoints is to clear log-records from stable storage as it gets full.

Since checkpoints cause some loss in performance while they are being taken, their frequency should be reduced if fast recovery is not critical. If we need fast recovery checkpointing frequency should be increased. If

the amount of stable storage available is less, frequent checkpointing is unavoidable.
Checkpoints have no effect on recovery from a disk crash; archival dumps are the equivalent of checkpoints for recovery from disk crashes.

16.3 Some database systems allow the administrator to choose between two forms of logging: *normal logging*, used to recover from system crashes, and *archival logging*, used to recover from media (disk) failure. When can a log record be deleted, in each of these cases, using the recovery algorithm of Section 16.4?
Answer: **Normal logging**: The following log records cannot be deleted, since they may be required for recovery:

a. Any log record corresponding to a transaction which was active during the most recent checkpoint (i.e. which is part of the <checkpoint L> entry)

b. Any log record corresponding to transactions started after the recent checkpoint.

All other log records can be deleted. After each checkpoint, more records become candidates for deletion as per the above rule.
Deleting a log record while retaining an earlier log record would result in gaps in the log, and would require more complex log processing. Therefore in practise, systems find a point in the log such that all earlier log records can be deleted, and delete that part of the log. Often, the log is broken up into multiple files, and a file is deleted when all log records in the file can be deleted.

Archival logging: Archival logging retains log records that may be needed for recovery from media failure (such as disk crashes). Archival dumps are the equivalent of checkpoints for recovery from media failure. The rules for deletion above can be used for archival logs, but based on the last archival dump instead of the last checkpoint. The frequency of archival dumps would be lesser than checkpointing, since a lot of data has to be written. Thus more log records would need to be retained with archival logging.

16.4 Describe how to modify the recovery algorithm of Section 16.4 to implement savepoints, and to perform rollback to a savepoint. (Savepoints are described in Section 16.8.3.)
Answer: A savepoint can be performed as follows:

a. Output onto stable storage all log records for that transaction which are currently in main memory.

b. Output onto stable storage a log record of the form <savepoint T_i>, where T_I is the transaction identifier.

To roll back a currently executing transaction partially till a particular savepoint, execute undo processing for that transaction, till the savepoint is reached. Redo log records are generated as usual during the undo phase above.
It is possible to perform repeated undo to a single savepoint by writing a fresh savepoint record after rolling back to that savepoint. The above algorithm can be extended to support multiple savepoints for a single transaction, by giving each savepoint a name. However, once undo has rolled back past a savepoint, it is no longer possible to undo upto that savepoint.

16.5 Suppose the deferred modification technique is used in a database.

a. Is the old-value part of an update log record required any more? Why or why not?

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo-phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions, and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique, if transactions perform a large number of updates?

Answer:

a. The old-value part of an update log record is not required. If the transaction has committed, then the old value is no longer necessary as there would be no need to undo the transaction. And if the transaction was active when the system crashed, the old values are still safe in the stable storage as they haven't been modified yet.

b. During the redo phase, the undo list need not be maintained any more, since the stable storage does not reflect updates due to any uncommitted transaction.

c. A data item read will first issue a read request on the local memory of the transaction. If it is found there, it is returned. Otherwise, the item is loaded from the database buffer into the local memory of the transaction and then returned.

d. If a single transaction performs a large number of updates, there is a possibility of the transaction running out of memory to store the local copies of the data items.

16.6 The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a B^+-tree.

a. Suggest how to share as many nodes as possible between the new copy and the shadow-copy of the B^+-tree, assuming that updates are made only to leaf entries, with no insertions and deletions.

b. Even with the above optimization, logging is much cheaper than a shadow-copy scheme, for transactions that perform small updates. Explain why.

Answer:

a. To begin with, we start with the copy of just the root node pointing to the shadow-copy. As modifications are made, the leaf entry where the modification is made and all the nodes in the path from that leaf node till the root, are copied and updated. All other nodes are shared.

b. For transactions that perform small updates, the shadow-paging scheme, would copy multiple pages for a single update, even with the above optimization. Logging, on the other hand just requires small records to be created for every update; the log records are physically together in one page or a few pages, and thus only a few log page I/O operations are required to commit a transaction. Furthermore, the log pages written out across subsequent transaction commits are likely to be adjacent physically on disk, minimizng disk arm movement.

16.7 Suppose we (incorrectly) modify the recovery algorithm of Section 16.4 to not log actions taken during transaction rollback. When recovering from a system crash, transactions that were rolled back earlier would then be included in undo-list, and rolled back again. Give an example to show how actions taken during the undo phase of recovery could result in an incorrect database state. (Hint: Consider a data item updated by an aborted transaction, and then updated by a transaction that commits.)
Answer: Consider the following log records generated with the (incorrectly) modified recovery algorithm:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_2 start>
4. <T_2, A, 1000, 2000>
5. <T_2 commit>

A rollback actually happened between steps 2 and 3, but there are no log records reflecting the same. Now, this log data is processed by the recovery algorithm. At the end of the redo phase, T_1 would get added to the undo-list, and the value of A would be 2000. During the undo phase,

since T_1 is present in the undo-list, the recovery algorithm does an undo of statement 2 and A takes the value 1000. The update made by T_2, though commited, is lost.
The correct sequence of logs is as follows:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_1, A, 1000>
4. <T_1 abort>
5. <T_2 start>
6. <T_2, A, 1000, 2000>
7. <T_2 commit>

This would make sure that T_1 would not get added to the undo-list after the redo phase.

16.8 Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.

Answer: If a transaction allocates a page to a relation, even if the transaction is rolled back, the page allocation should not be undone because other transactions may have stored records in the same page. Such operations that should not be undone are called nested top actions in ARIES. They can be modeled as operations whose undo action does nothing. In ARIES such operations are implemented by creating a dummy CLR whose UndoNextLSN is set such that the transaction rollback skips the log records generated by the operation.

16.9 Suppose a transaction deletes a record, and the free space generated thus is allocated to a record inserted by another transaction, even before the first transaction commits.

a. What problem can occur if the first transaction needs to be rolled back?

b. Would this problem be an issue if page-level locking is used instead of tuple-level locking?

c. Suggest how to solve this problem while supporting tuple-level locking, by logging post-commit actions in special log records, and executing them after commit. Make sure your scheme ensures that such actions are performed exactly once.

Answer:

a. If the first transaction needs to be rolled back, the tuple deleted by that transaction will have to be restored. If undo is performed in the usual physical manner using the old values of data items, the space allocated to the new tuple would get overwritten by the transaction

undo, damaging the new tuples, and associated data structures on the disk block. This means that a logical undo operation has to be performed i.e., an insert has to be performed to undo the delete, which complicates recovery.
On related note, if the second transaction inserts with the same key, integrity constraints might be violated on rollback.

b. If page level locking is used, the free space generated by the first transaction is not allocated to another transaction till the first one commits. So this problem will not be an issue if page level locking is used.

c. The problem can be solved by deferring freeing of space till after the transaction commits. To ensure that space will be freed even if there is a system crash immediately after commit, the commit log record can be modified to contain information about freeing of space (and other similar operations) which must be performed after commit. The execution of these operations can be performed as a transaction and log records generated, following by a post-commit log record which indicates that post commit processing has been completed for the transaction.
During recovery, if a commit log record is found with post-commit actions, but no post-commit log record is found, the effects of any partial execution of post-commit operations are rolled back during recovery, and the post commit operations are reexecuted at the end of recovery. If the post-commit log record is found, the post-commit actions are not reexecuted. Thus, the actions are guaranteed to be executed exactly once.
The problem of clashes on primary key values can be solved by holding key-level locks so that no other transaction can use the key till the first transaction completes.

16.10 Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)
Answer: Interactive transactions are more difficult to recover from than batch transactions because some actions may be irrevocable. For example, an output (write) statement may have fired a missile, or caused a bank machine to give money to a customer. The best way to deal with this is to try to do all output statements at the end of the transaction. That way if the transaction aborts in the middle, no harm will be have been done.
Output operations should ideally be done atomically; for example, ATM machines often count out notes, and deliver all the notes together instead of delivering notes one-at-a-time. If output operations cannot be done atomically, a physical log of output operations, such as a disk log of events, or even a video log of what happened in the physical world can be

maintained, to allow perform recovery to be performed manually later, for example by crediting cash back to a customers account.

16.11 Sometimes a transaction has to be undone after it has committed because it was erroneously executed, for example because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.

Suggest a modification to the recovery algorithm of Section 16.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be re-executed logically, if the updates are available in the form of SQL but cannot be re-executed using their log records. Why?

Answer:

a. Consider the a bank account A with balance \$100. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. The log records corresponding to the updates of A by transactions T_1 and T_2 would be $< T_1, A, 100, 110 >$ and $< T_2, A, 110, 120 >$ resp.
Say, we wish to undo transaction T_1. The normal transaction undo mechanism will replaces the value in question—A in this example—by the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism the resulting balance would be \$100 and we would, in effect, undo both transactions, whereas we intend to undo only transaction T_1.

b. Let the erroneous transaction be T_e.

- Identify the latest archival dump, say D, before the log record $< T_e, START >$. Restore the database using the dump.
- Redo all log records starting from the dump D till the log record $< T_e, COMMIT >$. Some transaction—apart from transaction T_e—would be active at the commit time of transaction T_e. Let S_1 be the set of such transactions.
- Rollback T_e and the transactions in the set S_1. This completes point-in-time recovery.
In case logical redo is possible, later transactions can be rexecuted logically assuming log records containing logical redo

information were written for every transaction. To perform logical redo of later transactions, scan the log further starting from the log record $< T_e$, *COMMIT*$>$ till the end of the log. Note the transactions that were started after the commit point of T_e. Let the set of such transactions be S_2. Re-execute the transactions in set S_1 and S_2 logically.

c. Consider again an example from the first item. Let us assume that both transactions are undone and the balance is reverted back to the original value \$100.

Now we wish to redo transaction T_2. If we redo the log record $< T_2, A, 110, 120 >$ corresponding to transaction T_2 the balance would become \$120 and we would, in effect, redo both transactions, whereas we intend to redo only transaction T_2.

CHAPTER 17

Database System Architectures

Practice Exercises

17.1 Instead of storing shared structures in shared memory, an alternative architecture would be to store them in the local memory of a special process, and access the shared data by interprocess communication with the process. What would be the drawback of such an architecture?

Answer: The drawbacks would be that two interprocess messages would be required to acquire locks, one for the request and one to confirm grant. Interprocess communication is much more expensive than memory access, so the cost of locking would increase. The process storing the shared structures could also become a bottleneck.

The benefit of this alternative is that the lock table is protected better from erroneous updates since only one process can access it.

17.2 In typical client–server systems the server machine is much more powerful than the clients; that is, its processor is faster, it may have multiple processors, and it has more memory and disk capacity. Consider instead a scenario where client and server machines have exactly the same power. Would it make sense to build a client–server system in such a scenario? Why? Which scenario would be better suited to a data-server architecture?

Answer: With powerful clients, it still makes sense to have a client-server system, rather than a fully centralized system. If the data-server architecture is used, the powerful clients can off-load all the long and compute intensive transaction processing work from the server, freeing it to perform only the work of satisfying read-write requests. even if the transaction-server model is used, the clients still take care of the user-interface work, which is typically very compute-intensive.

A fully distributed system might seem attractive in the presence of powerful clients, but client-server systems still have the advantage of simpler concurrency control and recovery schemes to be implemented

on the server alone, instead of having these actions distributed in all the machines.

17.3 Consider a database system based on a client–server architecture, with the server acting as a data server.

a. What is the effect of the speed of the interconnection between the client and the server on the choice between tuple and page shipping?

b. If page shipping is used, the cache of data at the client can be organized either as a tuple cache or a page cache. The page cache stores data in units of a page, while the tuple cache stores data in units of tuples. Assume tuples are smaller than pages. Describe one benefit of a tuple cache over a page cache.

Answer:

a. We assume that tuples are smaller than a page and fit in a page. If the interconnection link is slow it is better to choose tuple shipping, as in page shipping a lot of time will be wasted in shipping tuples that might never be needed. With a fast interconnection though, the communication overheads and latencies, not the actual volume of data to be shipped, becomes the bottle neck. In this scenario page shipping would be preferable.

b. Two benefits of an having a tuple-cache rather than a page-cache, even if page shipping is used, are:

   i. When a client runs out of cache space, it can replace objects without replacing entire pages. The reduced caching granularity might result in better cache-hit ratios.

   ii. It is possible for the server to ask clients to return some of the locks which they hold, but don't need (lock de-escalation). Thus there is scope for greater concurrency. If page caching is used, this is not possible.

17.4 Suppose a transaction is written in C with embedded SQL, and about 80 percent of the time is spent in the SQL code, with the remaining 20 percent spent in C code. How much speedup can one hope to attain if parallelism is used only for the SQL code? Explain.
Answer: Since the part which cannot be parallelized takes 20% of the total running time, the best speedup we can hope for has to be less than 5.

17.5 Some database operations such as joins can see a significant difference in speed when data (for example, one of the relations involved in a join) fits in memory as compared to the situation where the data does not fit in memory. Show how this fact can explain the phenomenon of **superlinear speedup**, where an application sees a speedup greater than the amount of resources allocated to it.
Answer: FILL

17.6 Parallel systems often have a network structure where sets of n processors connect to a single Ethernet switch, and the Ethernet switches themselves connect to another Ethernet switch. Does this architecture correspond to a bus, mesh or hypercube architecture? If not, how would you describe this interconnection architecture?
Answer: FILL

CHAPTER 18

Parallel Databases

Practice Exercises

18.1 In a range selection on a range-partitioned attribute, it is possible that only one disk may need to be accessed. Describe the benefits and drawbacks of this property.

Answer: If there are few tuples in the queried range, then each query can be processed quickly on a single disk. This allows parallel execution of queries with reduced overhead of initiating queries on multiple disks.

On the other hand, if there are many tuples in the queried range, each query takes a long time to execute as there is no parallelism within its execution. Also, some of the disks can become hot-spots, further increasing response time.

Hybrid range partitioning, in which small ranges (a few blocks each) are partitioned in a round-robin fashion, provides the benefits of range partitioning without its drawbacks.

18.2 What form of parallelism (interquery, interoperation, or intraoperation) is likely to be the most important for each of the following tasks?

a. Increasing the throughput of a system with many small queries

b. Increasing the throughput of a system with a few large queries, when the number of disks and processors is large

Answer:

a. When there are many small queries, inter-query parallelism gives good throughput. Parallelizing each of these small queries would increase the initiation overhead, without any significant reduction in response time.

b. With a few large queries, intra-query parallelism is essential to get fast response times. Given that there are large number of processors and disks, only intra-operation parallelism can take advantage of the parallel hardware – for queries typically have

few operations, but each one needs to process a large number of tuples.

18.3 With pipelined parallelism, it is often a good idea to perform several operations in a pipeline on a single processor, even when many processors are available.

a. Explain why.

b. Would the arguments you advanced in part *a* hold if the machine has a shared-memory architecture? Explain why or why not.

c. Would the arguments in part *a* hold with independent parallelism? (That is, are there cases where, even if the operations are not pipelined and there are many processors available, it is still a good idea to perform several operations on the same processor?)

Answer:

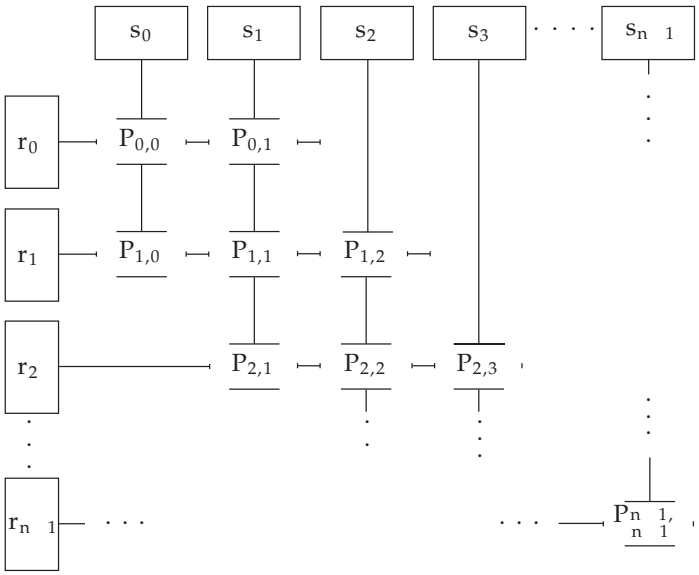
a. The speed-up obtained by parallelizing the operations would be offset by the data transfer overhead, as each tuple produced by an operator would have to be transferred to its consumer, which is running on a different processor.

b. In a shared-memory architecture, transferring the tuples is very efficient. So the above argument does not hold to any significant degree.

c. Even if two operations are independent, it may be that they both supply their outputs to a common third operator. In that case, running all three on the same processor may be better than transferring tuples across processors.

18.4 Consider join processing using symmetric fragment and replicate with range partitioning. How can you optimize the evaluation if the join condition is of the form $| r.A - s.B | \leq k$, where k is a small constant? Here, $| x |$ denotes the absolute value of x. A join with such a join condition is called a **band join**.

Answer: Relation r is partitioned into n partitions, $r_0, r_1, \ldots, r_{n-1}$, and s is also partitioned into n partitions, $s_0, s_1, \ldots, s_{n-1}$. The partitions are replicated and assigned to processors as shown below.



Each fragment is replicated on 3 processors only, unlike in the general case where it is replicated on n processors. The number of processors required is now approximately $3n$, instead of n^2 in the general case. Therefore given the same number of processors, we can partition the relations into more fragments with this optimization, thus making each local join faster.

18.5 Recall that histograms are used for constructing load-balanced range partitions.

a. Suppose you have a histogram where values are between 1 and 100, and are partitioned into 10 ranges, 1–10, 11–20, . . . , 91–100, with frequencies 15, 5, 20, 10, 10, 5, 5, 20, 5, and 5, respectively. Give a load-balanced range partitioning function to divide the values into 5 partitions.

b. Write an algorithm for computing a balanced range partition with p partitions, given a histogram of frequency distributions containing n ranges.

Answer:

a. A partitioning vector which gives 5 partitions with 20 tuples in each partition is: [21, 31, 51, 76]. The 5 partitions obtained are $1-20, 21-30, 31-50, 51-75$ and $76-100$. The assumption made in arriving at this partitioning vector is that within a histogram range, each value is equally likely.

b. Let the histogram ranges be called $h_1, h_2, \ldots, h_h$, and the partitions $p_1, p_2, \ldots, p_p$. Let the frequencies of the histogram ranges be

$n_1, n_2, \ldots, n_h$. Each partition should contain N/p tuples, where $N = \Sigma_{i=1}^{h} n_i$.
To construct the load balanced partitioning vector, we need to determine the value of the k_1^{th} tuple, the value of the k_2^{th} tuple and so on, where $k_1 = N/p, k_2 = 2N/p$ etc, until k_{p-1}. The partitioning vector will then be $[k_1, k_2, \ldots, k_{p-1}]$. The value of the k_i^{th} tuple is determined as follows. First determine the histogram range h_j in which it falls. Assuming all values in a range are equally likely, the k_i^{th} value will be

$$s_j + (e_j - s_j) * \frac{k_{ij}}{n_j}$$

where

s_j	:	first value in h_j
e_j	:	last value in h_j
k_{ij}	:	$k_i - \Sigma_{l=1}^{j-1} n_l$

18.6 Large-scale parallel database systems store an extra copy of each data item on disks attached to a different processor, to avoid loss of data if one of the processors fails.

a. Instead of keeping the extra copy of data items from a processor at a single backup processor, it is a good idea to partition the copies of the data items of a processor across multiple processors. Explain why.

b. Explain how virtual-processor partitioning can be used to efficiently implement the partitioning of the copies as described above.

c. What are the benefits and drawbacks of using RAID storage instead of storing an extra copy of each data item?

Answer: FILL

18.7 Suppose we wish to index a large relation that is partitioned. Can the idea of partitioning (including virtual processor partitioning) be applied to indices? Explain your answer, considering the following two cases (assuming for simplicity that partitioning as well as indexing are on single attributes):

a. Where the index is on the partitioning attribute of the relation.

b. Where the index is on an attribute other than the partitioning attribute of the relation.

Answer: FILL

18.8 Suppose a well-balanced range-partitioning vector had been chosen for a relation, but the relation is subsequently updated, making the partitioning unbalanced. Even if virtual-processor partitioning is used,

a particular virtual processor may end up with a very large number of tuples after the update, and repartitioning would then be required.

a. Suppose a virtual processor has a significant excess of tuples (say, twice the average). Explain how repartitioning can be done by splitting the partition, thereby increasing the number of virtual processors.

b. If, instead of round-robin allocation of virtual processors, virtual partitions can be allocated to processors in an arbitrary fashion, with a mapping table tracking the allocation. If a particular node has excess load (compared to the others), explain how load can be balanced.

c. Assuming there are no updates, does query processing have to be stopped while repartitioning, or reallocation of virtual processors, is carried out? Explain your answer.

Answer: FILL

CHAPTER 19

Distributed Databases

Practice Exercises

19.1 How might a distributed database designed for a local-area network differ from one designed for a wide-area network?

Answer: Data transfer on a local-area network (LAN) is much faster than on a wide-area network (WAN). Thus replication and fragmentation will not increase throughput and speed-up on a LAN, as much as in a WAN. But even in a LAN, replication has its uses in increasing reliability and availability.

19.2 To build a highly available distributed system, you must know what kinds of failures can occur.

a. List possible types of failure in a distributed system.

b. Which items in your list from part a are also applicable to a centralized system?

Answer:

a. The types of failure that can occur in a distributed system include
   i. Site failure.
   ii. Disk failure.
   iii. Communication failure, leading to disconnection of one or more sites from the network.

b. The first two failure types can also occur on centralized systems.

19.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Practice Exercise 19.2a, explain how 2PC ensures transaction atomicity despite the failure.

Answer: A proof that 2PC guarantees atomic commits/aborts inspite of site and link failures, follows. The main idea is that after all sites reply with a <**ready** T> message, only the co-ordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a

site can happen only after it ascertains the co-ordinator's decision, either directly from the co-ordinator, or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

a. A site can abort a transaction T (by writing an <**abort** T> log record) only under the following circumstances:

   i. It has not yet written a <**ready** T> log-record. In this case, the co-ordinator could not have got, and will not get a <**ready** T> or <**commit** T> message from this site. Therefore only an abort decision can be made by the co-ordinator.

   ii. It has written the <**ready** T> log record, but on inquiry it found out that some other site has an <**abort** T> log record. In this case it is correct for it to abort, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually aborting.

   iii. It is itself the co-ordinator. In this case also no site could have committed, or will commit in the future, because commit decisions can be made only by the co-ordinator.

b. A site can commit a transaction T (by writing an <**commit** T> log record) only under the following circumstances:

   i. It has written the <**ready** T> log record, and on inquiry it found out that some other site has a <**commit** T> log record. In this case it is correct for it to commit, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually committing.

   ii. It is itself the co-ordinator. In this case no other participating site can abort/ would have aborted, because abort decisions are made only by the co-ordinator.

19.4 Consider a distributed system with two sites, A and B. Can site A distinguish among the following?

- B goes down.
- The link between A and B goes down.
- B is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Answer:

Site A cannot distinguish between the three cases until communication has resumed with site B. The action which it performs while B is inaccessible must be correct irrespective of which of these situations has actually

occurred, and must be such that B can re-integrate consistently into the distributed system once communication is restored.

19.5 The persistent messaging scheme described in this chapter depends on timestamps combined with discarding of received messages if they are too old. Suggest an alternative scheme based on sequence numbers instead of timestamps.

Answer: We can have a scheme based on sequence numbers similar to the scheme based on timestamps. We tag each message with a sequence number that is unique for the (sending site, receiving site) pair. The number is increased by 1 for each new message sent from the sending site to the receiving site.

The receiving site stores and acknowledges a received message only if it has received all lower numbered messages also; the message is stored in the *received-messages* relation.

The sending site retransmits a message until it has received an ack from the receiving site containing the sequence number of the transmitted message, or a higher sequence number. Once the acknowledgment is received, it can delete the message from its send queue.

The receiving site discards all messages it receives that have a lower sequence number than the latest stored message from the sending site. The receiving site discards from *received-messages* all but the (number of the) most recent message from each sending site (message can be discarded only after being processed locally).

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of messages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

19.6 Give an example where the read one, write all available approach leads to an erroneous state.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Let one of the sites, say s, be down when T_1 is executed and transaction t_2 reads the balance from site s. One can see that the balance at the primary site would be \$110 at the end.

19.7 Explain the difference between data replication in a distributed system and the maintenance of a remote backup site.

Answer: In remote backup systems all transactions are performed at the primary site and the data is replicated at the remote backup site. The

remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.
The distributed systems offer greater availability by having multiple copies of the data at different sites whereas the remote backup systems offer lesser availability at lower cost and execution overhead.
In a distributed system, transaction code runs at all the sites whereas in a remote backup system it runs only at the primary site. The distributed system transactions follow two-phase commit to have the data in consistent state whereas a remote backup system does not follow two-phase commit and avoids related overhead.

19.8 Give an example where lazy replication can lead to an inconsistent database state even when updates get an exclusive lock on the primary (master) copy.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Suppose the copy of the balance at one of the sites, say s, is not consistent – due to lazy replication strategy – with the primary copy after transaction T_1 is executed and let transaction T_2 read this copy of the balance. One can see that the balance at the primary site would be \$110 at the end.

19.9 Consider the following deadlock-detection algorithm. When transaction T_i, at site S_1, requests a resource from T_j, at site S_3, a request message with timestamp n is sent. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_1. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_3 only if T_j has received the request message and cannot immediately grant the requested resource. A request from T_i to T_j in the same site is handled in the usual manner; no timestamps are associated with the edge (T_i, T_j). A central coordinator invokes the detection algorithm by sending an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. Note that such a graph contains all the local information that the site has about the state of the real graph. The wait-for graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.

When the controller has received a reply from each site, it constructs a graph as follows:

- The graph contains a vertex for every transaction in the system.
- The graph has an edge (T_i, T_j) if and only if:
  - There is an edge (T_i, T_j) in one of the wait-for graphs.

- An edge (T_i, T_j, n) (for some n) appears in more than one wait-for graph.

Show that, if there is a cycle in the constructed graph, then the system is in a deadlock state, and that, if there is no cycle in the constructed graph, then the system was not in a deadlock state when the execution of the algorithm began.

Answer: Let us say a cycle $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_m \rightarrow T_i$ exists in the graph built by the controller. The edges in the graph will either be local edges of the from (T_k, T_l) or distributed edges of the form (T_k, T_l, n). Each local edge (T_k, T_l) definitely implies that T_k is waiting for T_l. Since a distributed edge (T_k, T_l, n) is inserted into the graph only if T_k's request has reached T_l and T_l cannot immediately release the lock, T_k is indeed waiting for T_l. Therefore every edge in the cycle indeed represents a transaction waiting for another. For a detailed proof that this imlies a deadlock refer to Stuart et al. [1984].

We now prove the converse implication. As soon as it is discovered that T_k is waiting for T_l:

a. a local edge (T_k, T_l) is added if both are on the same site.

b. The edge (T_k, T_l, n) is added in both the sites, if T_k and T_l are on different sites.

Therefore, if the algorithm were able to collect all the local wait-for graphs at the same instant, it would definitely discover a cycle in the constructed graph, in case there is a circular wait at that instant. If there is a circular wait at the instant when the algorithm began execution, none of the edges participating in that cycle can disappear until the algorithm finishes. Therefore, even though the algorithm cannot collect all the local graphs at the same instant, any cycle which existed just before it started will anyway be detected.

19.10 Consider a relation that is fragmented horizontally by *plant_number*:

employee (*name, address, salary, plant_number*)

Assume that each fragment has two replicas: one stored at the New York site and one stored locally at the plant site. Describe a good processing strategy for the following queries entered at the San Jose site.

a. Find all employees at the Boca plant.

b. Find the average salary of all employees.

c. Find the highest-paid employee at each of the following sites: Toronto, Edmonton, Vancouver, Montreal.

d. Find the lowest-paid employee in the company.

Answer:

a. i. Send the query $\Pi_{name}(employee)$ to the Boca plant.
   ii. Have the Boca location send back the answer.

b. i. Compute average at New York.
   ii. Send answer to San Jose.

c. i. Send the query to find the highest salaried employee to Toronto, Edmonton, Vancouver, and Montreal.
   ii. Compute the queries at those sites.
   iii. Return answers to San Jose.

d. i. Send the query to find the lowest salaried employee to New York.
   ii. Compute the query at New York.
   iii. Send answer to San Jose.

19.11 Compute $r \ltimes s$ for the relations of Figure 19.9.
Answer: The result is as follows.

$r \ltimes s =$

A	B	C
1	2	3
5	3	2

19.12 Give an example of an application ideally suited for the cloud and another that would be hard to implement successfully in the cloud. Explain your answer.
Answer: Any application that is easy to partition, and does not need strong guarantees of consistency across partitions, is ideally suited to the cloud. For example, Web-based document storage systems (like Google docs), and Web based email systems (like Hotmail, Yahoo! mail or GMail), are ideally suited to the cloud. The cloud is also ideally suited to certain kinds of data analysis tasks where the data is already on the cloud; for example, the Google Map-Reduce framework, and Yahoo! Hadoop are widely used for data analysis of Web logs such as logs of URLs clicked by users.
Any database application that needs transactional consistency would be hard to implement successfully in the cloud; examples include bank records, academic records of students, and many other types of organizational records.

19.13 Given that the LDAP functionality can be implemented on top of a database system, what is the need for the LDAP standard?
Answer: The reasons are:

a. Directory access protocols are simplified protocols that cater to a limited type of access to data.

b. Directory systems provide a simple mechanism to name objects in a hierarchical fashion which can be used in a distributed directory system to specify what information is stored in each of the directory servers. The directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

19.14 Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.

b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

Answer:

a. We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the co-ordinator, and requiring that the lock be requested on the data item which resides on the currently elected co-ordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

19.15 Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.

b. Show how you could use a ticket scheme to ensure global serializability.

Answer:

a. The same system as in the answer to Exercise 19.14 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.

CHAPTER 20

Data Analysis and Mining

Practice Exercises

20.1 Describe benefits and drawbacks of a source-driven architecture for gathering of data at a data warehouse, as compared to a destination-driven architecture.

Answer: In a destination-driven architecture for gathering data, data transfers from the data sources to the data-warehouse are based on demand from the warehouse, whereas in a source-driven architecture, the transfers are initiated by each source.

The benefits of a source-driven architecture are

- Data can be propagated to the destination as soon as it becomes available. For a destination-driven architecture to collect data as soon as it is available, the warehouse would have to probe the sources frequently, leading to a high overhead.
- The source does not have to keep historical information. As soon as data is updated, the source can send an update message to the destination and forget the history of the updates. In contrast, in a destination-driven architecture, each source has to maintain a history of data which have not yet been collected by the data warehouse. Thus storage requirements at the source are lower for a source-driven architecture.

On the other hand, a destination-driven architecture has the following advantages.

- In a source-driven architecture, the source has to be active and must handle error conditions such as not being able to contact the warehouse for some time. It is easier to implement passive sources, and a single active warehouse. In a destination-driven architecure, each source is required to provide only a basic functionality of executing queries.
- The warehouse has more control on when to carry out data gathering activities, and when to process user queries; it is not a good

idea to perform both simultaneously, since they may conflict on locks.

20.2 Why is column-oriented storage potentially advantageous in a database system that supports a data warehouse?
Answer: No Answer

20.3 Suppose that there are two classification rules, one that says that people with salaries between $10,000 and $20,000 have a credit rating of *good*, and another that says that people with salaries between $20,000 and $30,000 have a credit rating of *good*. Under what conditions can the rules be replaced, without any loss of information, by a single rule that says people with salaries between $10,000 and $30,000 have a credit rating of *good*?
Answer: Consider the following pair of rules and their confidence levels :

No.	Rule	Conf.
1.	$\forall$ *persons* P, $10000 < P.salary \leq 20000 \Rightarrow P.credit = good$	60%
2.	$\forall$ *persons* P, $20000 < P.salary \leq 30000 \Rightarrow P.credit = good$	90%

The new rule has to be assigned a confidence-level which is between the confidence-levels for rules 1 and 2. Replacing the original rules by the new rule will result in a loss of confidence-level information for classifying persons, since we cannot distinguish the confidence levels of perople earning between 10000 and 20000 from those of people earning between 20000 and 30000. Therefore we can combine the two rules without loss of information only if their confidences are the same.

20.4 Consider the schema depicted in Figure 20.2. Give an SQL query to summarize sales numbers and price by store and date, along with the hierarchies on store and date.
Answer: query:

```
select store-id, city, state, country,
       date, month, quarter, year,
       sum(number), sum(price)
from sales, store, date
where sales.store-id = store.store-id and
       sales.date = date.date
groupby rollup(country, state, city, store-id),
       rollup(year, quarter, month, date)
```

20.5 Consider a classification problem where the classifier predicts whether a person has a particular disease. Suppose that 95% of the people tested do not suffer from the disease. (That is, *pos* corresponds to 5% and *neg* to 95% of the test cases.) Consider the following classifiers:

- Classifier C_1 which always predicts negative (a rather useless classifier of course).
- Classifier C_2 which predicts positive in 80% of the cases where the person actually has the disease, but also predicts positive in 5% of the cases where the person does not have the disease.
- Classifier C_3 which predicts positive in 95% of the cases where the person actually has the disease, but also predicts positive in 20% of the cases where the person does not have the disease.

Given the above classifiers, answer the following questions.

a. For each of the above classifiers, compute the accuracy, precision, recall and specificity.

b. If you intend to use the results of classification to perform further screening for the disease, how would you choose between the classifiers.

c. On the other hand, if you intend to use the result of classification to start medication, where the medication could have harmful effects if given to someone who does not have the disease, how would you choose between the classifiers?

Answer: No Answer

CHAPTER 21

Information Retrieval

Practice Exercises

21.1 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the Practice Exercises in this chapter to the query "SQL relation".

Answer: We do not consider the questions containing neither of the keywords as their relevance to the keywords is zero. The number of words in a question include stop words. We use the equations given in Section 21.2 to compute relevance; the log term in the equation is assumed to be to the base 2.

Q#	#wo--rds	# "SQL"	#"rela--tion"	"SQL" term freq.	"relation" term freq.	"SQL" relv.	"relation" relv.	Tota relv.
1	84	1	1	0.0170	0.0170	0.0002	0.0002	0.0004
4	22	0	1	0.0000	0.0641	0.0000	0.0029	0.0029
5	46	1	1	0.0310	0.0310	0.0006	0.0006	0.0013
6	22	1	0	0.0641	0.0000	0.0029	0.0000	0.0029
7	33	1	1	0.0430	0.0430	0.0013	0.0013	0.0026
8	32	1	3	0.0443	0.1292	0.0013	0.0040	0.0054
9	77	0	1	0.0000	0.0186	0.0000	0.0002	0.0002
14	30	1	0	0.0473	0.0000	0.0015	0.0000	0.0015
15	26	1	1	0.0544	0.0544	0.0020	0.0020	0.0041

21.2 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

Answer: Let S be a set of n keywords. An algorithm to find all documents that contain at least k of these keywords is given below :

This algorithm calculates a reference count for each document identifier. A reference count of i for a document identifier d means that at least i of the keywords in S occur in the document identified by d. The algorithm maintains a list of records, each having two fields – a document identifier, and the reference count for this identifier. This list is maintained sorted on the document identifier field.

```
initialize the list L to the empty list;
for (each keyword c in S) do
begin
    D := the list of documents identifiers corresponding to c;
    for (each document identifier d in D) do
        if (a record R with document identifier as d is on list L) then
                R.reference_count := R.reference_count + 1;
        else begin
                make a new record R;
                R.document_id := d;
                R.reference_count := 1;
                add R to L;
        end;
end;
for (each record R in L) do
    if (R.reference_count >= k) then
        output R;
```

Note that execution of the second *for* statement causes the list D to "merge" with the list L. Since the lists L and D are sorted, the time taken for this merge is proportional to the sum of the lengths of the two lists. Thus the algorithm runs in time (at most) proportional to n times the sum total of the number of document identifiers corresponding to each keyword in S.

21.3 Suggest how to implement the iterative technique for computing PageRank given that the T matrix (even in adjacency list representation) does not fit in memory.
Answer: No answer

21.4 Suggest how a document containing a word (such as "leopard") can be indexed such that it is efficiently retrieved by queries using a more general concept (such as "carnivore" or "mammal"). You can assume that the concept hierarchy is not very deep, so each concept has only a few generalizations (a concept can, however, have a large number of specializations). You can also assume that you are provided with a function that returns the concept for each word in a document. Also suggest how a query using a specialized concept can retrieve documents using a more general concept.
Answer: Add doc to index lists for more general concepts also.

21.5 Suppose inverted lists are maintained in blocks, with each block noting the largest popularity rank and TF-IDF scores of documents in the remaining blocks in the list. Suggest how merging of inverted lists can stop early if the user wants only the top K answers.

Answer: For all documents whose scores are not complete use upper bounds to compute best possible score. If K th largest completed score is greater than the largest upper bound among incomplete scores output the K top answers.No answer

CHAPTER 22

Object-Based Databases

Practice Exercises

22.1 A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity.
- Sports cars: horsepower, renter age requirement.
- Vans: number of passengers.
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive).

Construct an SQL schema definition for this database. Use inheritance where appropriate.

Answer: For this problem, we use table inheritance. We assume that **MyDate**, **Color** and **DriveTrainType** are pre-defined types.

create type *Vehicle*
(*vehicle_id* **integer**,
license_number **char(15)**,
manufacturer **char(30)**,
model **char(30)**,
purchase_date **MyDate**,
color **Color**)

create table *vehicle* **of type** *Vehicle*

create table *truck*
(*cargo_capacity* **integer**)
under *vehicle*

create table *sportsCar*

```
    (horsepower integer
     renter_age_requirement integer)
    under vehicle

create table van
    (num_passengers integer)
    under vehicle

create table offRoadVehicle
    (ground_clearance real
     driveTrain DriveTrainType)
    under vehicle
```

22.2 Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

Emp = (ename, ChildrenSet **multiset***(Children), SkillSet* **multiset***(Skills))*
Children = (name, birthday)
Skills = (type, ExamSet **setof***(Exams))*
Exams = (year, city)

a. Define the above schema in SQL, with appropriate types for each attribute.

b. Using the above schema, write the following queries in SQL.

   i. Find the names of all employees who have a child born on or after January 1, 2000.

   ii. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

   iii. List all skill types in the relation *Emp*.

Answer:

a. No Answer.

b. Queries in SQL.

   i. Program:

```
select ename
from emp as e, e.ChildrenSet as c
where 'March' in
        (select birthday.month
         from c
        )
```

   ii. Program:

```
select e.ename
from emp as e, e.SkillSet as s, s.ExamSet as x
where s.type = 'typing' and x.city = 'Dayton'
```

iii. Program:

```
select distinct s.type
from emp as e, e.SkillSet as s
```

22.3 Consider the E-R diagram in Figure 22.5, which contains composite, multivalued, and derived attributes.

a. Give an SQL schema definition corresponding to the E-R diagram.

b. Give constructors for each of the structured types defined above.

Answer:

a. The corresponding SQL:1999 schema definition is given below. Note that the derived attribute *age* has been translated into a method.

```
create type Name
    (first_name varchar(15),
     middle_initial char,
     last_name varchar(15))
create type Street
    (street_name varchar(15),
     street_number varchar(4),
     apartment_number varchar(7))
create type Address
    (street Street,
     city varchar(15),
     state varchar(15),
     zip_code char(6))
create table customer
    (name Name,
     customer_id varchar(10),
     address Adress,
     phones  char(7)  array[10],
     dob date)
method integer age()
```

b.
```
create function Name (f varchar(15), m char, l varchar(15))
returns Name
begin
    set first_name = f;
    set middle_initial = m;
    set last_name = l;
end
create function Street (sname varchar(15), sno varchar(4), ano varchar(7))
```

```
returns Street
begin
    set street_name = sname;
    set street_number = sno;
    set apartment_number =ano;
end
create function Address (s Street, c varchar(15), sta varchar(15), zip varchar(6))
returns Address
begin
    set street = s;
    set city = c;
    set state =sta;
    set zip_code =zip;
end
```

22.4 Consider the relational schema shown in Figure 22.6.

a. Give a schema definition in SQLcorresponding to the relational schema, but using references to express foreign-key relationships.

b. Write each of the queries given in Exercise 6.13 on the above schema, using SQL.

Answer:

a. The schema definition is given below. Note that backward references can be addedbut they are not so important as in OODBS because queries can be written in SQL and joins can take care of integrity constraints.

```
create type Employee
    (person_name varchar(30),
     street varchar(15),
     city varchar(15))
create type Company
    (company_name varchar(15),
    (city varchar(15))
create table employee of Employee
create table company of Company
create type Works
    (person ref(Employee) scope employee,
    comp ref(Company) scope company,
    salary int)
create table works of Works
create type Manages
    (person ref(Employee) scope employee,
    (manager ref(Employee) scope employee)
create table manages of Manages
```

b. i. **select** *comp*− >*name*

from *works*
group by *comp*
having count(*person*) $\geq$ **all(select count(**$person$**)**
from *works*
group by *comp*)

ii. **select** $comp->name$
from *works*
group by *comp*
having sum(*salary*) $\leq$ **all(select sum(**$salary$**)**
from *works*
group by *comp*)

iii. **select** $comp->name$
from *works*
group by *comp*
having avg(*salary*) $>$ **(select avg(**$salary$**)**
from *works*
where $comp->company\_name$="First Bank Corporation")

22.5 Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language–based OODB, object relational; do not specify a commercial product) you would recommend. Justify your recommendation.

a. A computer-aided design system for a manufacturer of airplanes.

b. A system to track contributions made to candidates for public office.

c. An information system to support the making of movies.

Answer:

a. A computer-aided design system for a manufacturer of airplanes:
An OODB system would be suitable for this. That is because CAD requires complex data types, and being computation oriented, CAD tools are typically used in a programming language environment needing to access the database.

b. A system to track contributions made to candidates for public office:
A relational system would be apt for this, as data types are expected to be simple, and a powerful querying mechanism is essential.

c. An information system to support the making of movies:
Here there will be extensive use of multimedia and other complex data types. But queries are probably simple, and thus an object relational system is suitable.

22.6 How does the concept of an object in the object-oriented model differ from the concept of an entity in the entity-relationship model?

Answer: An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it, and these messages are implemented by the methods.

CHAPTER 23

XML

Practice Exercises

23.1 Give an alternative representation of university information containing the same data as in Figure 23.1, but using attributes instead of subelements. Also give the DTD or XML Schema for this representation.
Answer:

a. The XML representation of data using attributes is shown in Figure 23.100.

b. The DTD for the bank is shown in Figure 23.101.

```
<university>
    <department dept_name="Comp. Sci." building="Taylor"
            budget="100000">
    </department>
    <department dept_name="Biology" building="Watson"
            budget="90000">
    </department>
    <course course_id="CS-101" title="Intro. to Computer Science"
            dept_name="Comp. Sci." credits="4">
    </course>
    <course course_id="BIO-301" title="Genetics"
            dept_name="Biology." credits="4">
    </course>
    <instructor IID="10101" name="Srinivasan"
            dept_name="Comp. Sci." salary="65000">
    </instructor>
    <instructor IID="83821" name="Brandt"
            dept_name="Comp. Sci" salary="92000">
    </instructor>
    <instructor IID="76766" name="Crick"
            dept_name="Biology" salary="72000">
    </instructor>
    <teaches IID="10101" course_id="CS-101">
    </teaches>
    <teaches IID="83821" course_id="CS-101">
    </teaches>
    <teaches IID="76766" course_id="BIO-301">
    </teaches>
</university>
```

Figure 23.100 XML representation.

23.2 Give the DTD or XML Schema for an XML representation of the following nested-relational schema:

Emp = (ename, ChildrenSet **setof***(Children), SkillsSet* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, ExamsSet **setof***(Exams))*
Exams = (year, city)

Answer: Query:

```
<!DOCTYPE db [
        <!ELEMENT emp (ename, children*, skills*)>
        <!ELEMENT children (name, birthday)>
        <!ELEMENT birthday (day, month, year)>
        <!ELEMENT skills (type, exams+)>
        <!ELEMENT exams (year, city)>
        <!ELEMENT ename( #PCDATA )>
        <!ELEMENT name( #PCDATA )>
        <!ELEMENT day( #PCDATA )>
        <!ELEMENT month( #PCDATA )>
        <!ELEMENT year( #PCDATA )>
        <!ELEMENT type( #PCDATA )>
        <!ELEMENT city( #PCDATA )>
] >
```

```
<!DOCTYPE university [
        <!ELEMENT department >
        <!ATTLIST department
                dept_name ID #REQUIRED
                building CDATA #REQUIRED
                budget CDATA #REQUIRED >
        <!ELEMENT instructor >
        <!ATTLIST instructor
                IID ID #REQUIRED
                name CDATA #REQUIRED
                dept_name IDREF #REQUIRED >
                salary CDATA #REQUIRED >
        <!ELEMENT course >
        <!ATTLIST course
                course_id ID #REQUIRED
                title CDATA #REQUIRED
                dept_name IDREF #REQUIRED >
                credits CDATA #REQUIRED >
        <!ELEMENT teaches >
        <!ATTLIST teaches
                IID IDREF #REQUIRED >
                course_id IDREF #REQUIRED
] >
```

Figure 23.101 The DTD for the university.

23.3 Write a query in XPath on the schema of Practice Exercise 23.2 to list all skill types in *Emp*.
Answer: Code:

```
/db/emp/skills/type
```

23.4 Write a query in XQuery on the XML representation in Figure 23.11 to find the total salary of all instructors in each department.
Answer: Query:

```
for $b in distinct (/university/department/dept_name)
return
<dept-total>
    <dept_name> $b/text() </dept_name>
    let $s := sum (/university/instructor[dept_name=$b]/salary)
    return <total-salary> $s </total-salary>
</dept-total>
```

23.5 Write a query in XQuery on the XML representation in Figure 23.1 to compute the left outer join of department elements with course elements. (Hint: Use universal quantification.)
Answer: Query:

```
<lojoin>
for $d in /university/department,
    $c in /university/course
where $c/dept_name = $d/dept_name
return <dept-course> $d $c </dept-course>
|
for $d in /university/department,
where every $c in /university/course satisfies
(not ($c/dept_name =$d/dept_name))
return <dept-course > $c </dept-course >
</lojoin>
```

23.6 Write queries in XQuery to output course elements with associated instructor elements nested within the course elements, given the university information representation using ID and IDREFS in Figure 23.11.

Answer: The answer in XQuery is

```
<university-2>
    for $c in /university/course
    return
        <course>
            <course_id> $c/* </course_id>
            for $a in $c/id(@instructors)
            return $a
        </course>
</university-2>
```

23.7 Give a relational schema to represent bibliographical information specified according to the DTD fragment in Figure 23.16. The relational schema must keep track of the order of author elements. You can assume that only books and articles appear as top-level elements in XML documents.

Answer: Relation schema:

book (bid, title, year, publisher, place)
article (artid, title, journal, year, number, volume, pages)
book_author (bid, first_name,last_name, order)
article_author (artid, first_name,last_name, order)

23.8 Show the tree representation of the XML data in Figure 23.1, and the representation of the tree using *nodes* and *child* relations described in Section 23.6.2.

Answer: The answer is shown in Figure 23.102.

nodes(1,element,university,–)
nodes(2,element,department,–)
nodes(3,element,department,–)
nodes(4,element,course,–)
nodes(5,element,course,–)
nodes(6,element,instructor,–)
nodes(7,element,instructor,–)
nodes(8,element,instructor,–)
nodes(9,element,teaches,–)
nodes(10,element,teaches,–)
nodes(11,element,teaches,–)
child(2,1) child(3,1) child(4,1)
child(5,1) child(6,1)
child(7,1) child(8,1) child(9,1)

Continued in Figure 23.103

Figure 23.102 Relational Representation of XML Data as Trees.

23.9 Consider the following recursive DTD:

```
<!DOCTYPE parts [
    <!ELEMENT part (name, subpartinfo*)>
    <!ELEMENT subpartinfo (part, quantity)>
    <!ELEMENT name ( #PCDATA )>
    <!ELEMENT quantity ( #PCDATA )>
] >
```

a. Give a small example of data corresponding to this DTD.

b. Show how to map this DTD to a relational schema. You can assume that part names are unique; that is, wherever a part appears, its subpart structure will be the same.

c. Create a schema in XML Schema corresponding to this DTD.

Answer:

a. The answer is shown in Figure 23.104.

b. Show how to map this DTD to a relational schema.

part(partid,name)
subpartinfo(partid, subpartid, qty)

Attributes partid and subpartid of subpartinfo are foreign keys to part.

c. The XML Schema for the DTD is as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="parts" type="partsType" />
<xs:complexType name="partType">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="subpartinfo" type="subpartinfoType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
<xs:complexType name="subpartinfoType"/>
    <xs:sequence>
        <xs:element name="part" type="partType"/>
        <xs:element name="quantity" type="xs:string"/>
    </xs:sequence>
</xs:schema>
```

child(10,1) child(11,1)
nodes(12,element,dept_name,Comp. Sci.)
nodes(13,element,building,Taylor)
nodes(14,element,budget,100000)
child(12,2) child(13,2) child(14,2)
nodes(15,element,dept_name,Biology)
nodes(16,element,building,Watson)
nodes(17,element,budget,90000)
child(15,3) child(16,3) child(17,3)
nodes(18,element,course_id,CS-101)
nodes(19,element,title,Intro. to Computer Science)
nodes(20,element,dept_name,Comp. Sci.)
nodes(21,element,credits,4)
child(18,4) child(19,4) child(20,4)child(21,4)
nodes(22,element,course_id,BIO-301)
nodes(23,element,title,Genetics)
nodes(24,element,dept_name,Biology)
nodes(25,element,credits,4)
child(22,5) child(23,5) child(24,5)child(25,5)
nodes(26,element,IID,10101)
nodes(27,element,name,Srinivasan)
nodes(28,element,dept_name,Comp. Sci.)
nodes(29,element,salary,65000)
child(26,6) child(27,6) child(28,6)child(29,6)
nodes(30,element,IID,83821)
nodes(31,element,name,Brandt)
nodes(32,element,dept_name,Comp. Sci.)
nodes(33,element,salary,92000)
child(30,7 child(31,7) child(32,7)child(33,7)
nodes(34,element,IID,76766)
nodes(35,element,dept_name,Biology)
nodes(36,element,salary,72000)
child(34,8) child(35,8) child(36,8)
nodes(37,element,IID,10101)
nodes(38,element,course_id,CS-101)
child(37,9) child(38,9)
nodes(39,element,IID,83821)
nodes(40,element,course_id,CS-101)
child(39,10) child(40,10)
nodes(41,element,IID,76766)
nodes(42,element,course_id,BIO-301)
child(41,11) child(42,11)

Figure 23.103 Continuation of Figure 23.102.

```
<parts>
    <part>
        <name> bicycle </name>
        <subpartinfo>
            <part>
                <name> wheel </name>
                <subpartinfo>
                    <part>
                        <name> rim </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> spokes </name>
                    </part>
                    <qty> 40 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> tire </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> brake </name>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> gear </name>
            </part>
            <qty> 3 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> frame </name>
            </part>
            <qty> 1 </qty>
        </subpartinfo>
    </part>
</parts>
```

Figure 23.104 Example Parts Data in XML.

CHAPTER 24

Advanced Application Development

Practice Exercises

24.1 Many applications need to generate sequence numbers for each transaction.

a. If a sequence counter is locked in two-phase manner, it can become a concurrency bottleneck. Explain why this may be the case.

b. Many database systems support built-in sequence counters that are not locked in two-phase manner; when a transaction requests a sequence number, the counter is locked, incremented and unlocked.

i. Explain how such counters can improve concurrency.

ii. Explain why there may be gaps in the sequence numbers belonging to the final set of committed transactions.

Answer: If two-phase locking is used on the counter, the counter must remain locked in exclusive mode until the transaction is done acquiring locks. During that time, the counter is unavailable and no concurrent transactions can be started regardless of whether they would have data conflicts with the transaction holding the counter.
If locking is done outside the scope of a two-phase locking protocol, the counter is locked only for the brief time it takes to increment the counter. SInce there is no other operation besides increment performed on the counter, this creates no problem except when a transaction aborts. During an abort, the counter *cannot* be restored to its old value but instead should remain as it stands. This means that some counter values are unused since those values were assigned to aborted transactions. To see why we cannot restore the old counter value, assume transaction T_a has counter value 100 and then T_b is given the next value, 101. If T_a aborts and the counter were restored to 100, the value 100 would be given to some other transaction T_c and then 101 would be given to

a second transaction T_d. Now we have two non-aborted transactions with the same counter value.

24.2 Suppose you are given a relation $r(a, b, c)$.

a. Give an example of a situation under which the performance of equality selection queries on attribute a can be greatly affected by how r is clustered.

b. Suppose you also had range selection queries on attribute b. Can you cluster r in such a way that the equality selection queries on $r.a$ and the range selection queries on $r.b$ can both be answered efficiently? Explain your answer.

c. If clustering as above is not possible, suggest how both types of queries can be executed efficiently by choosing appropriate indices.

Answer:

a. If r is not clustered on a, tuples of r with a particular a-value could be scattered throughout the area in which r is stored. This would make it necessary to scan all of r. Clustering on a combined with an index, would allow tuples of r with a particular a-value to be retrieved directly.

b. This is possible only if there is a special relationship between a and b such as "if tuple t_1 has an a-value less than the a-value of tuple t_2, then t_1 must have a b-value less than that of t_2". Aside from special cases, such a clustering is not possible since the sort order of the tuples on a is different from the sort order on b.

c. The physical order of the tuples cannot always be suited to both queries. If r is clustered on a and we have a B^+-tree index on b, the first query can be executed very efficiently. For the second, we can use the usual B^+-tree range-query algorithm to obtain pointers to all the tuples in the result relation, then sort those those pointers so that any particular disk block is accessed only once.

24.3 Suppose that a database application does not appear to have a single bottleneck; that is, CPU and disk utilization are both high, and all database queues are roughly balanced. Does that mean the application cannot be tuned further? Explain your answer.
Answer: it may still be tunable. Example using a materialized view may reduce both CPU and disk utilization

24.4 Suppose a system runs three types of transactions. Transactions of type A run at the rate of 50 per second, transactions of type B run at 100 per second, and transactions of type C run at 200 per second. Suppose the mix of transactions has 25 percent of type A, 25 percent of type B, and 50 percent of type C.

a. What is the average transaction throughput of the system, assuming there is no interference between the transactions?

b. What factors may result in interference between the transactions of different types, leading to the calculated throughput being incorrect?

Answer:

a. Let there be 100 transactions in the system. The given mix of transaction types would have 25 transactions each of type A and B, and 50 transactions of type C. Thus the time taken to execute transactions only of type A is 0.5 seconds and that for transactions only of type B or only of type C is 0.25 seconds. Given that the transactions do not interfere, the total time taken to execute the 100 transactions is $0.5 + 0.25 + 0.25 = 1$ second. i.e, the average overall transaction throughput is 100 *transactions per second*.

b. One of the most important causes of transaction interference is lock contention. In the previous example, assume that transactions of type A and B are update transactions, and that those of type C are queries. Due to the speed mismatch between the processor and the disk, it is possible that a transaction of type A is holding a lock on a "hot" item of data and waiting for a disk write to complete, while another transaction (possibly of type B or C) is waiting for the lock to be released by A. In this scenario some CPU cycles are wasted. Hence, the observed throughput would be lower than the calculated throughput.
Conversely, if transactions of type A and type B are disk bound, and those of type C are CPU bound, and there is no lock contention, observed throughput may even be better than calculated.
Lock contention can also lead to deadlocks, in which case some transaction(s) will have to be aborted. Transaction aborts and restarts (which may also be used by an optimistic concurrency control scheme) contribute to the observed throughput being lower than the calculated throughput.
Factors such as the limits on the sizes of data-structures and the variance in the time taken by book-keeping functions of the transaction manager may also cause a difference in the values of the observed and calculated throughput.

24.5 List some benefits and drawbacks of an anticipatory standard compared to a reactionary standard.
Answer: In the absence of an anticipatory standard it may be difficult to reconcile between the differences among products developed by various organizations. Thus it may be hard to formulate a reactionary standard without sacrificing any of the product development effort. This problem has been faced while standardizing pointer syntax and access mechanisms for the ODMG standard.

On the other hand, a reactionary standard is usually formed after extensive product usage, and hence has an advantage over an anticipatory standard - that of built-in pragmatic experience. In practice, it has been found that some anticipatory standards tend to be over-ambitious. SQL-3 is an example of a standard that is complex and has a very large number of features. Some of these features may not be implemented for a long time on any system, and some, no doubt, will be found to be inappropriate.

CHAPTER 25

Advanced Data Types and New Applications

Practice Exercises

25.1 What are the two types of time, and how are they different? Why does it make sense to have both types of time associated with a tuple?

Answer: A temporal database models the changing states of some aspects of the real world. The time intervals related to the data stored in a temporal database may be of two types - *valid time* and *transaction time*. The valid time for a fact is the set of intervals during which the fact is true in the real world. The transaction time for a data object is the set of time intervals during which this object is part of the physical database. Only the transaction time is system dependent and is generated by the database system.

Suppose we consider our sample bank database to be bitemporal. Only the concept of valid time allows the system to answer queries such as - "What was Smith's balance two days ago?". On the other hand, queries such as - "What did we record as Smith's balance two days ago?" can be answered based on the transaction time. The difference between the two times is important. For example, suppose, three days ago the teller made a mistake in entering Smith's balance and corrected the error only yesterday. This error means that there is a difference between the results of the two queries (if both of them are executed today).

25.2 Suppose you have a relation containing the x, y coordinates and names of restaurants. Suppose also that the only queries that will be asked are of the following form: The query specifies a point, and asks if there is a restaurant exactly at that point. Which type of index would be preferable, R-tree or B-tree? Why?

Answer: The given query is not a range query, since it requires only searching for a point. This query can be efficiently answered by a B-tree index on the pair of attributes (x, y).

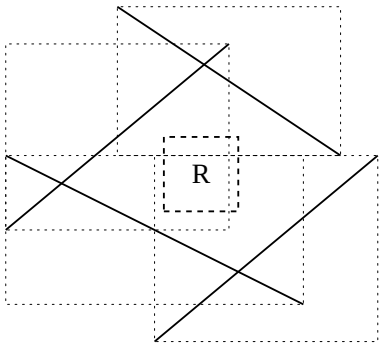
25.3 Suppose you have a spatial database that supports region queries (with circular regions) but not nearest-neighbor queries. Describe an algorithm to find the nearest neighbor by making use of multiple region queries.

Answer: Suppose that we want to search for the nearest neighbor of a point P in a database of points in the plane. The idea is to issue multiple region queries centered at P. Each region query covers a larger area of points than the previous query. The procedure stops when the result of a region query is non-empty. The distance from P to each point within this region is calculated and the set of points at the smallest distance is reported.

25.4 Suppose you want to store line segments in an R-tree. If a line segment is not parallel to the axes, the bounding box for it can be large, containing a large empty area.

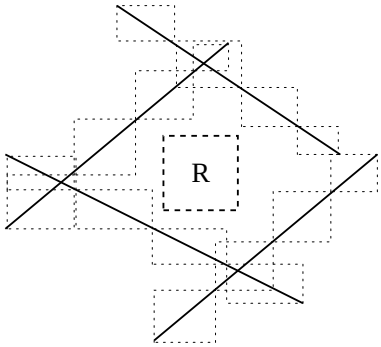
- Describe the effect on performance of having large bounding boxes on queries that ask for line segments intersecting a given region.
- Briefly describe a technique to improve performance for such queries and give an example of its benefit. Hint: You can divide segments into smaller pieces.

Answer: Large bounding boxes tend to overlap even where the region of overlap does not contain any information. The following figure:



shows a region R within which we have to locate a segment. Note that even though none of the four segments lies in R, due to the large bounding boxes, we have to check each of the four bounding boxes to confirm this.

A significant improvement is observed in the follwoing figure:



where each segment is split into multiple pieces, each with its own bounding box. In the second case, the box R is not part of the boxes indexed by the R-tree. In general, dividing a segment into smaller pieces causes the bounding boxes to be smaller and less wasteful of area.

25.5 Give a recursive procedure to efficiently compute the spatial join of two relations with R-tree indices. (Hint: Use bounding boxes to check if leaf entries under a pair of internal nodes may intersect.)

Answer: Following is a recursive procedure for computing spatial join of two R-trees.

```
SpJoin(node n1, node n2)
begin
    if(the bounding boxes of n1 and n2 do not intersect)
        return;
    if(both n1 and n2 are leaves)
        output all pairs of entries (e1, e2) such that
            e1 ∈ n1 and e2 ∈ n2, and e1 and e2 overlap;
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    for each ns1 in NS1 and ns2 in NS2;
        SpJoin(ns1, ns2);
end
```

25.6 Describe how the ideas behind the RAID organization (Section 10.3) can be used in a broadcast-data environment, where there may occasionally be noise that prevents reception of part of the data being transmitted.

Answer: The concepts of RAID can be used to improve reliability of the broadcast of data over wireless systems. Each block of data that is to be broadcast is split into *units* of equal size. A checksum value is calculated for each unit and appended to the unit. Now, parity data for these units is calculated. A checksum for the parity data is appended to it to form a parity unit. Both the data units and the parity unit are then broadcast one after the other as a single transmission.

On reception of the broadcast, the receiver uses the checksums to verify whether each unit is received without error. If one unit is found to be in error, it can be reconstructed from the other units.

The size of a unit must be chosen carefully. Small units not only require more checksums to be computed, but the chance that a burst of noise corrupts more than one unit is also higher. The problem with using large units is that the probability of noise affecting a unit increases; thus there is a tradeoff to be made.

25.7 Define a model of repeatedly broadcast data in which the broadcast medium is modeled as a virtual disk. Describe how access time and data-transfer rate for this virtual disk differ from the corresponding values for a typical hard disk.

Answer: We can distinguish two models of broadcast data. In the case of a pure broadcast medium, where the receiver cannot communicate with the broadcaster, the broadcaster transmits data with periodic cycles of retransmission of the entire data, so that new receivers can catch up with all the broadcast information. Thus, the data is broadcast in a continuous cycle. This period of the cycle can be considered akin to the worst case rotational latency in a disk drive. There is no concept of seek time here. The value for the cycle latency depends on the application, but is likely to be at least of the order of seconds, which is much higher than the latency in a disk drive.

In an alternative model, the receiver can send requests back to the broadcaster. In this model, we can also add an equivalent of disk access latency, between the receiver sending a request, and the broadcaster receiving the request and responding to it. The latency is a function of the volume of requests and the bandwidth of the broadcast medium. Further, queries may get satisfied without even sending a request, since the broadcaster happened to send the data either in a cycle or based on some other receivers request. Regardless, latency is likely to be at least of the order of seconds, again much higher than the corresponding values for a hard disk.

A typical hard disk can transfer data at the rate of 1 to 5 megabytes per second. In contrast, the bandwidth of a broadcast channel is typically only a few kilobytes per second. Total latency is likely to be of the order of seconds to hundreds or even thousands of seconds, compared to a few milliseconds for a hard disk.

25.8 Consider a database of documents in which all documents are kept in a central database. Copies of some documents are kept on mobile

computers. Suppose that mobile computer A updates a copy of document 1 while it is disconnected, and, at the same time, mobile computer B updates a copy of document 2 while it is disconnected. Show how the version-vector scheme can ensure proper updating of the central database and mobile computers when a mobile computer reconnects.

Answer: Let C be the computer onto which the central database is loaded. Each mobile computer (host) i stores, with its copy of each document d, a version-vector – that is a set of version numbers $V_{d,i,j}$, with one entry for each other host j that stores a copy of the document d, which it could possibly update.

Host A updates document 1 while it is disconnected from C. Thus, according to the version vector scheme, the version number $V_{1,A,A}$ is incremented by one.

Now, suppose host A re-connects to C. This pair exchanges version-vectors and finds that the version number $V_{1,A,A}$ is greater than $V_{1,C,A}$ by 1, (assuming that the copy of document 1 stored host A was updated most recently only by host A). Following the version-vector scheme, the version of document 1 at C is updated and the change is reflected by an increment in the version number $V_{1,C,A}$. Note that these are the only changes made by either host.

Similarly, when host B connects to host C, they exchange version-vectors, and host B finds that $V_{1,B,A}$ is one less than $V_{1,C,A}$. Thus, the version number $V_{1,B,A}$ is incremented by one, and the copy of document 1 at host B is updated.

Thus, we see that the version-vector scheme ensures proper updating of the central database for the case just considered. This argument can be very easily generalized for the case where multiple off-line updates are made to copies of document 1 at host A as well as host B and host C. The argument for off-line updates to document 2 is similar.

CHAPTER 26

Advanced Transaction Processing

Practice Exercises

26.1 Like database systems, workflow systems also require concurrency and recovery management. List three reasons why we cannot simply apply a relational database system using 2PL, physical undo logging, and 2PC.

Answer:

a. The tasks in a workflow have dependencies based on their status. For example the starting of a task may be conditional on the outcome (such as commit or abort) of some other task. All the tasks cannot execute independently and concurrently, using 2PC just for atomic commit.

b. Once a task gets over, it will have to expose its updates, so that other tasks running on the same processing entity don't have to wait for long. 2PL is too strict a form of concurrency control, and is not appropriate for workflows.

c. Workflows have their own consistency requirements; that is, failure-atomicity. An execution of a workflow must finish in an *acceptable termination state*. Because of this, and because of early exposure of uncommitted updates, the recovery procedure will be quite different. Some form of logical logging and compensation transactions will have to be used. Also to perform forward recovery of a failed workflow, the recovery routines need to restore the state information of the scheduler and tasks, not just the updated data items. Thus simple WAL cannot be used.

26.2 Consider a main-memory database system recovering from a system crash. Explain the relative merits of:

- Loading the entire database back into main memory before resuming transaction processing.
- Loading data as it is requested by transactions.

Answer:

- Loading the entire database into memory in advance can provide transactions which need high-speed or realtime data access the guarantee that once they start they will not have to wait for disk accesses to fetch data. However no transaction can run till the entire database is loaded.
- The advantage in loading on demand is that transaction processing can start rightaway; however transactions may see long and unpredictable delays in disk access until the entire database is loaded into memory.

26.3 Is a high-performance transaction system necessarily a real-time system? Why or why not?

Answer: A high-performance system is not necessarily a real-time system. In a high performance system, the main aim is to execute each transaction as quickly as possible, by having more resources and better utilization. Thus average speed and response time are the main things to be optimized. In a real-time system, speed is not the central issue. Here *each* transaction has a deadline, and taking care that it finishes within the deadline or takes as little extra time as possible, is the critical issue.

26.4 Explain why it may be impractical to require serializability for long-duration transactions.

Answer: In the presence of long-duration transactions, trying to ensure serializability has several drawbacks:-

a. With a waiting scheme for concurrency control, long-duration transactions will force long waiting times. This means that response time will be high, concurrency will be low, so throughput will suffer. The probability of deadlocks is also increased.

b. With a time-stamp based scheme, a lot of work done by a long-running transaction will be wasted if it has to abort.

c. Long duration transactions are usually interactive in nature, and it is very difficult to enforce serializability with interactiveness.

Thus the serializability requirement is impractical. Some other notion of database consistency has to be used in order to support long duration transactions.

26.5 Consider a multithreaded process that delivers messages from a durable queue of persistent messages. Different threads may run concurrently, attempting to deliver different messages. In case of a delivery failure, the

message must be restored in the queue. Model the actions that each thread carries out as a multilevel transaction, so that locks on the queue need not be held until a message is delivered.

Answer: Each thread can be modeled as a transaction T which takes a message from the queue and delivers it. We can write transaction T as a multilevel transaction with subtransactions T_1 and T_2. Subtransaction T_1 removes a message from the queue and subtransaction T_2 delivers it. Each subtransaction releases locks once it completes, allowing other transactions to access the queue. If transaction T_2 fails to deliver the message, transaction T_1 will be undone by invoking a compensating transaction which will restore the message to the queue.

26.6 Discuss the modifications that need to be made in each of the recovery schemes covered in Chapter 16 if we allow nested transactions. Also, explain any differences that result if we allow multilevel transactions.

Answer: Consider the advanced recovery algorithm of Section 16.4. The redo pass, which repeats history, is the same as before. We discuss below how the undo pass is handled.

- **Recovery with nested transactions**:
  Each subtransaction needs to have a unique TID, because a failed subtransaction might have to be independently rolled back and restarted.

  If a subtransaction fails, the recovery actions depend on whether the unfinished upper-level transaction should be aborted or continued. If it should be aborted, all finished and unfinished subtransactions are undone by a backward scan of the log (this is possible because the locks on the modified data items are not released as soon as a subtransaction finishes). If the nested transaction is going to be continued, just the failed transaction is undone, and then the upper-level transaction continues.

  In the case of a system failure, depending on the application, the entire nested-transaction may need to be aborted, or, (for e.g., in the case of long duration transactions) incomplete subtransactions aborted, and the nested transaction resumed. If the nested-transaction must be aborted, the rollback can be done in the usual manner by the recovery algorithm, during the undo pass. If the nested-transaction must be restarted, any incomplete subtransactions that need to be rolled back can be rolled back as above. To restart the nested-transaction, state information about the transaction, such as locks held and execution state, must have been noted on the log, and must restored during recovery. Mini-batch transactions (discussed in Section 24.1.10) are an example of nested transactions that must be restarted.

- **Recovery with multi-level transactions**:
  In addition to what is done in the previous case, we have to handle the problems caused by exposure of updates performed by committed subtransactions of incomplete upper-level transactions. A committed

subtransaction may have released locks that it held, so the compensating transaction has to reacquire the locks. This is straightforward in the case of transaction failure, but is more complicated in the case of system failure.

The problem is, a lower level subtransaction a of a higher level transaction A may have released locks, which have to be reacquired to compensate A during recovery. Unfortunately, there may be some other lower level subtransaction b of a higher level transaction B that started and acquired the locks released by a, before the end of A. Thus undo records for b may precede the operation commit record for A. But if b had not finished at the time of the system failure, it must first be rolled back and its locks released, to allow the compensating transaction of A to reacquire the locks.

This complicates the undo pass; it can no longer be done in one backward scan of the log. Multilevel recovery is described in detail in David Lomet, "MLR: A Recovery Method for Multi-Level Systems", ACM SIGMOD Conf. on the Management of Data 1992, San Diego.