CHAPTER 1

Introduction

Practice Exercises

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

Answer:

Two disadvantages associated with database systems are listed below.

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

CHAPTER 2

Introduction to the Relational Model

Practice Exercises

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

Answer:

The appropriate primary keys are shown below:

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

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

Answer:

- Inserting a tuple:

  (10111, Ostrom, Economics, 110000)

---

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

Figure 2.17 Employee database.

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

- Deleting the tuple:

(Biology, Watson, 90000)

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

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

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

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

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

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

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

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

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

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

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

Figure 2.18 Bank database.

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

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

Answer:

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

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

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

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

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

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

Answer:

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

Given the above definition:

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

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

Answer:

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

b. The required expression is as follows:

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 3

Introduction to SQL

Practice Exercises

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

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

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

c. Find the highest salary of any instructor.

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

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

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

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

Answer:

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

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

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

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

c. Find the highest salary of any instructor.

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

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

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

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

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

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

One way of writing the query might appear to be:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

Figure 3.17 Insurance database

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

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

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

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

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

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

Answer:

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

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

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

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

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

3.5 Suppose that we have a relation *marks*(*ID*, *score*) and we wish to assign grades to students based on the score as follows: grade *F* if *score* < 40, grade *C* if 40 ≤ *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.

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

Answer:

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

3.7 Consider the SQL query

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

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

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

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

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

Figure 3.18 Banking database.

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

---

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

Figure 3.19 Employee database.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

CHAPTER 4

Intermediate SQL

Practice Exercises

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

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

What is wrong with this query?

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

4.2 Write the following queries in SQL:

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

Hint: Use the queries from Exercise 4.2.

Answer:

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

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

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

c. Consider the query

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

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

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

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

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

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

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

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

Figure 4.12 Employee database.

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

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

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

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

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

Answer:
Plese see **??**.

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

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

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

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

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

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

Figure 4.101 Figure for Exercise 4.7.

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

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

Answer:

a. Query:

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

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

b. Query:

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

Answer:

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

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

CHAPTER 5

Advanced SQL

Practice Exercises

5.1 Consider the following relations for a company database:

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

and the Java code in Figure 5.20, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like "It finds the manager of the toy department," not a line-by-line description of what each Java statement does.)

Answer:
It prints out the manager of "dog," that manager's manager, etc., until we reach a manager who has no manager (presumably, the CEO, who most certainly is a cat). Note: If you try to run this, use your own Oracle ID and password.

5.2 Write a Java method 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:
Please see **??**

5.3 Suppose that we wish to find all courses that must be taken before some given course. That means finding not only the prerequisites of that course, but prerequisites of prerequisites, and so on. Write a complete Java program using JDBC that:

- Takes a *course_id* value from the keyboard.
- Finds prerequisites of that course using an SQL query submitted via JDBC.

```
import java.sql.*;
public class Mystery {
  public static void main(String[] args) {
     try (
        Connection con=DriverManager.getConnection(
           "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
        q = "select mname from mgr where ename = ?";
        PreparedStatement stmt=con.prepareStatement();
     )
     {
        String q;
        String empName = "dog";
        boolean more;
        ResultSet result;
        do {
           stmt.setString(1, empName);
           result = stmt.executeQuery(q);
           more = result.next();
           if (more) {
              empName = result.getString("mname");
              System.out.println (empName);
           }
        } while (more);
        s.close();
        con.close();
     }
     catch(Exception e){
        e.printStackTrace();
     }
  }
}
```

Figure 5.20 Java code for Exercise 5.1 (using Oracle JDBC).

- For each course returned, finds its prerequisites and continues this process iteratively until no new prerequisite courses are found.
- Prints out the result.

For this exercise, do not use a recursive SQL query, but rather use the iterative approach described previously. A well-developed solution will be robust to the error case where a university has accidentally created a cycle of prerequisites (that is, for example, course A is a prerequisite for course B, course B is a prerequisite for course C, and course C is a prerequisite for course A).

```
printTable(ResultSet result) throws SQLException {
    metadata = result.getMetaData();
    num_cols = metadata.getColumnCount();
    for(int i = 1; i <= num_cols; i++) {
        System.out.print(metadata.getColumnName(i) + '\t');
    }
    System.out.println();
    while(result.next()) {
        for(int i = 1; i <= num_cols; i++) {
            System.out.print(result.getString(i) + '\t'
        }
        System.out.println();
    } }
```

Figure 5.101 Java method using JDBC for Exercise 5.2.

Answer:
Please see **??**

5.4 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.

5.5 Show how to enforce the constraint "an instructor cannot teach two different sections 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:
Please see **??**

5.6 Consider the bank database of Figure 5.21. Let us define a view *branch_cust* as follows:

```
import java.sql.*;
import java.util.Scanner;
import java.util.Arrays;
public class AllCoursePrereqs {
  public static void main(String[] args) {
    try (
         Connection con=DriverManager.getConnection
           ("jdbc:oracle:thin:@edgar0.cse.lehigh.edu:1521:cse241","star","pw");
         Statement s=con.createStatement();
         ){
           String q;
           String c;
           ResultSet result;
           int maxCourse = 0;
           q = "select count(*) as C from course";
           result = s.executeQuery(q);
           if (!result.next()) System.out.println ("Unexpected empty result.");
           else maxCourse = Integer.parseInt(result.getString("C"));
           int numCourse = 0, oldNumCourse = -1;
           String[] prereqs = new String [maxCourse];
           Scanner krb = new Scanner(System.in);
           System.out.print("Input a course id (number): ");
           String course = krb.next();
           String courseString = "" + '\'' + course + '\'';
           while (numCourse != oldNumCourse) {
             for (int i = oldNumCourse + 1; i < numCourse; i++) {
               courseString += ", " + '\'' + prereqs[i] + '\'' ;
             }
             oldNumCourse = numCourse;
             q = "select prereq_id from prereq  where course_id in ("
               + courseString + ")";
             result = s.executeQuery(q);
             while (result.next()) {
               c = result.getString("prereq_id");
               boolean found = false;
               for (int i = 0; i < numCourse; i++)
                    found |= prereqs[i].equals(c);
               if (!found) prereqs[numCourse++] = c;
             }
             courseString = "" + '\'' + prereqs[oldNumCourse] + '\'';
           }
           Arrays.sort(prereqs,0,numCourse);
           System.out.print("The courses that must be taken prior to "
             + course + " are: ");
           for (int i = 0; i < numCourse; i++)
                System.out.print ((i==0?" ":", ") + prereqs[i]);
           System.out.println();
         } catch(Exception e){e.printStackTrace();
} }
```

Figure 5.102 Complete Java program using JDBC for Exercise 5.3.

```
create trigger onesec before insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id in (
   select time_slot_id
   from teaches natural join section
   where ID in (
         select ID
         from teaches natural join section
         where sec_id = nrow.sec_id and course_id = nrow.course_id and
               semester = nrow.semester and year = nrow.year
)))
begin
   rollback
end;


create trigger oneteach before insert on teaches
referencing new row as nrow
for each row
when (exists (
         select time_slot_id
         from teaches natural join section
         where ID = nrow.ID
   intersect
         select time_slot_id
         from section
         where sec_id = nrow.sec_id and course_id = nrow.course_id and
               semester = nrow.semester and year = nrow.year
))
begin
   rollback
end;
```

Figure 5.103 Trigger code for Exercise 5.5.

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

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

Figure 5.21 Banking database for Exercise 5.6.

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 *depositor* or *account*. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

Answer:
Please see **??**

5.7 Consider the bank database of Figure 5.21. Write an SQL trigger to carry out the following action: On **delete** of an account, for each customer-owner of the

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

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

Figure 5.22 Trigger code for Exercise 5.6.

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
```

5.8 Given a relation *S*(*student*, *subject*, *marks*), write a query to find the top 10 students by total marks, by using SQL ranking. Include all students tied for the final spot in the ranking, even if that results in more than 10 total students.

Answer:

```
select *
from (
    select student, total, rank() over (order by (total) desc) as t_rank
    from (
        select student, sum(marks) as total
        from S group by student
    )
)
where t_rank <= 10
```

5.9 Given a relation *nyse*(*year, month, day, shares_traded, dollar_volume*) with trading data from the New York Stock Exchange, list each trading day in order of number of shares traded, and show each day's rank.

Answer:

```
select year, month, day, shares_traded,
    rank() over (order by shares_traded desc ) as mostshares
from nyse
```

5.10 Using the relation from Exercise 5.9, write an SQL query to generate a report showing the number of shares traded, number of trades, and total dollar volume broken down by year, each month of each year, and each trading day.

Answer:

select *year, month, day*, **sum**(*shares_traded*) **as** *shares*,
    sum(*num_trades*) **as** *trades*, **sum**(*dollar_volume*) **as** *total_volume*
from *nyse*
group by rollup (*year, month, day*)

5.11 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)

CHAPTER 6

Database Design using the E-R Model

Practice Exercises

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

Answer:

One possible E-R diagram is shown in Figure 6.101. Payments are modeled as weak entities since they are related to a specific policy.

Note that the participation of accident in the relationship *participated* is not total, since it is possible that there is an accident report where the participating car is unknown.

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

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

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

Answer:

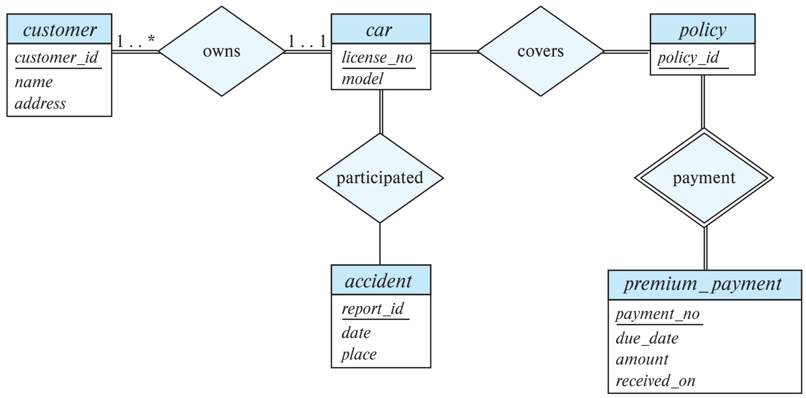


Figure 6.101 E-R diagram for a car insurance company.

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

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

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

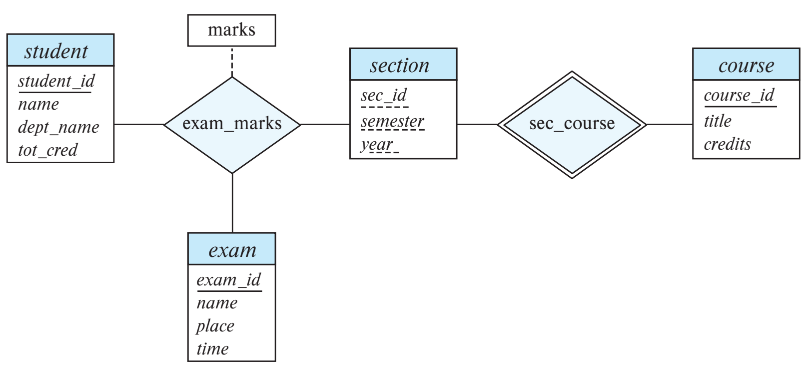


Figure 6.102 E-R diagram for marks database.

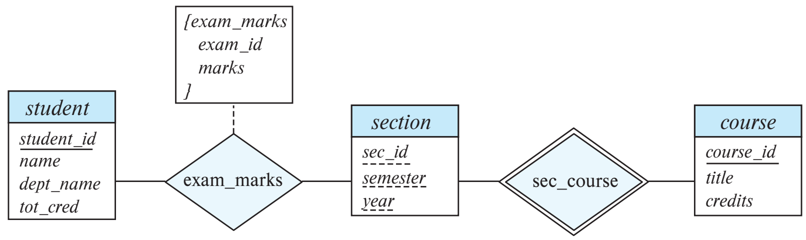


Figure 6.103 Another E-R diagram for marks database.

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

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

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

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

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

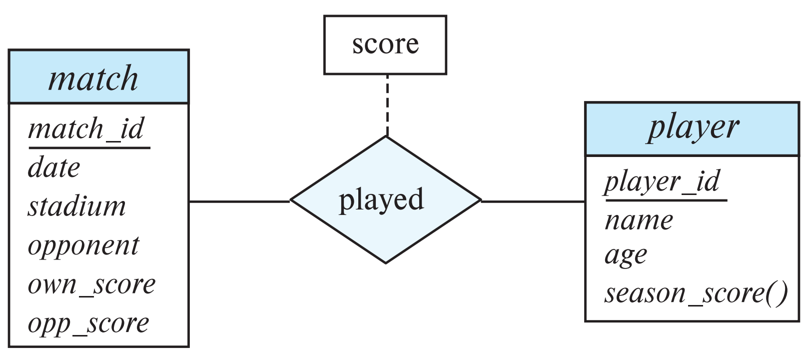


Figure 6.104 E-R diagram for favorite team statistics.

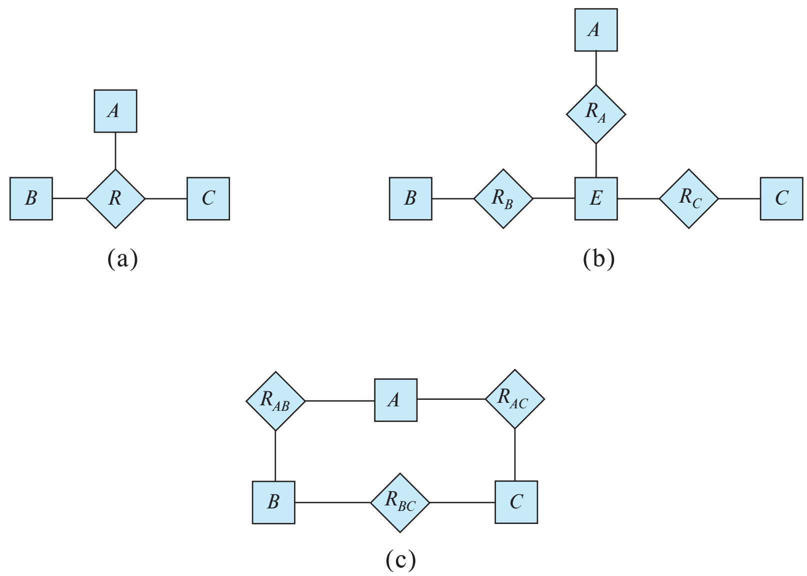


Figure 6.29 Representation of a ternary relationship using binary relationships.

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

a. The graph is disconnected.

b. The graph has a cycle.

Answer:

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

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

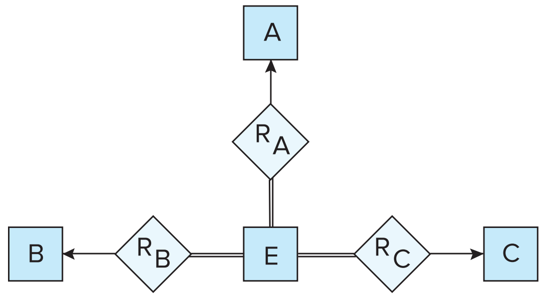


Figure 6.105 E-R diagram for Exercise Exercise 6.6b.

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

a. Explain why.

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

Answer:

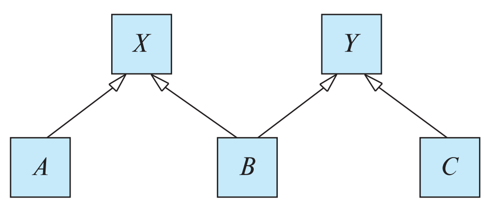
a. For the many-to-many case, the relationship set must be represented as a separate relation that cannot be combined with either participating entity. Now, there is no way in SQL to ensure that a primary-key value occurring in an entity $E1$ also occurs in a many-to-many relationship R, since the corresponding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

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

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

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

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



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

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

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

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

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

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

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

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

Answer:

.

a. The E-R diagram is shown in Figure 6.106.
The primary key attributes *student_id* and *instructor_id* are assumed to be immutable, that is, they are not allowed to change with time. All other attributes are assumed to potentially change with time.
Note that the diagram uses multivalued composite attributes such as *valid_times* or *name*, with subattributes such as *start_time* or *value*. The *value* attribute is a subattribute of several attributes such as *name, tot_cred* and *salary*, and refers to the name, total credits or salary during a particular interval of time.

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

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

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

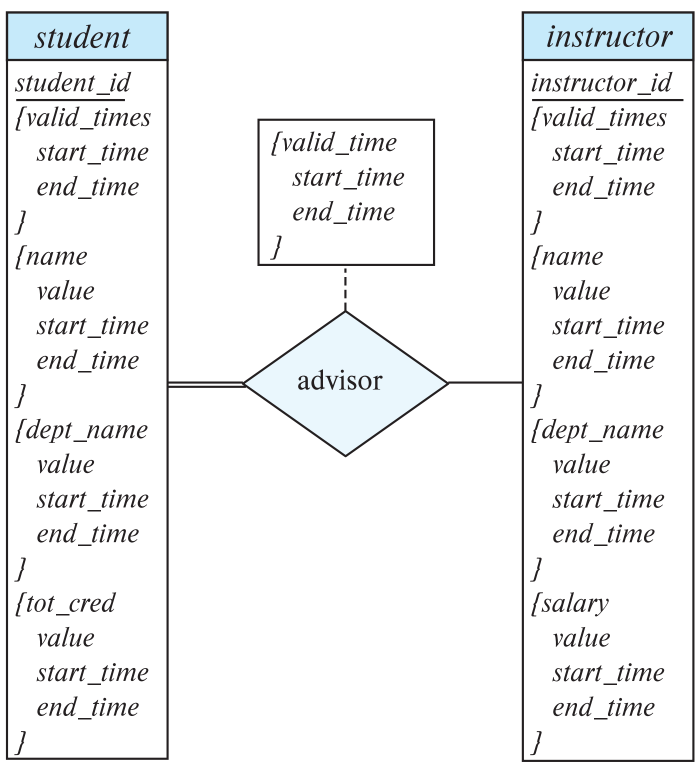


Figure 6.106 E-R diagram for Exercise 6.13

CHAPTER 7

Relational Database Design

Practice Exercises

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

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

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

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

Answer:

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

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

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

Figure 7.17 Relation of Exercise 7.2.

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

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

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

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

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

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

Answer:
To prove that:

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

Following the hint, we derive:

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

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

Answer:

Proof using Armstrong's axioms of the pseudotransitivity rule:

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

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

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

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

List the candidate keys for R.

Answer:

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

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

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

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

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

Answer:

The given set of FDs F is:-

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

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.

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

Answer:

The algorithm is correct because:

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

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

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

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

Figure 7.18 An algorithm to compute α^+.

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

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

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

Answer:

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

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

b.

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

Answer:

Consider some tuple t in u.

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

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

By the definition of natural join,

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

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

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

Since t is any arbitrary tuple in u,

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

is in BCNF as is

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 8

Complex Data Types

Practice Exercises

8.1 Provide information about the student named Shankar in our sample university database, including information from the *student* tuple corresponding to Shankar, the *takes* tuples corresponding to Shankar and the *course* tuples corresponding to these *takes* tuples, in each of the following representations:

a. Using JSON, with an appropriate nested representation.

b. Using XML, with the same nested representation.

c. Using RDF triples.

d. As an RDF graph.

Answer:

a. FILL IN

b. FILL IN

c. FILL IN

d. FILL IN

8.2 Consider the RDF representation of information from the university schema as shown in Figure 8.3. Write the following queries in SPARQL.

a. Find the titles of all courses taken by any student named Zhang.

b. Find titles of all courses such that a student named Zhang takes a section of the course that is taught by an instructor named Srinivasan.

c. Find the attribute names and values of all attributes of the instructor named Srinivasan, without enumerating the attribute names in your query.

Answer:
FILL IN

8.3 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
```

8.4 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)*

Define the above schema in SQL, using the SQL Server table type syntax from Section 8.2.1.1 to declare multiset attributes.

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
```

8.5 Consider the E-R diagram in Figure 8.7 showing entity set *instructor*. Give an SQL schema definition corresponding to the E-R diagram, treating *phone_number* as an array of 10 elements, using Oracle or PostgreSQL syntax.

Answer:
The corresponding SQL:1999 schema definition is given below. Note that the derived attribute *age* has been translated into a method.

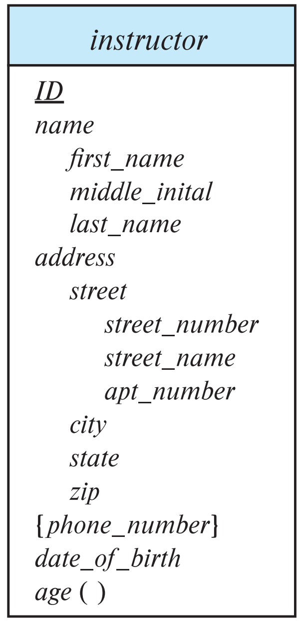


Figure 8.7 E-R diagram with composite, multivalued, and derived attributes.

```
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  varray(10) of char(7) ,
     dob date)
method integer age()
```

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

Figure 8.8 Relational database for Exercise 8.6.

The above array syntax is based on Oracle, in PostgreSQL *phones* would be declared to have type **char**(7)[ ].

8.6 Consider the relational schema shown in Figure 8.8.

a. Give a schema definition in SQL corresponding to the relational schema but using references to express foreign-key relationships.

b. Write each of the following queries on the schema, using SQL.

i. Find the company with the most employees.

ii. Find the company with the smallest payroll.

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

Answer:

a. The schema definition is given below.

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*) ≥ **all(select count(***person*)
from *works*
group by *comp*)

ii. **select** *comp*− >*name*
from *works*
group by *comp*
having sum(*salary*) ≤ **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")

8.7 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 because their relevance to the keywords is zero. The number of words in a question include stop words. We use the equations given in Section 31.2 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

8.8 Show how to represent the matrices used for computing PageRank as relations. Then write an SQL query that implements one iterative step of the iterative technique for finding PageRank; the entire algorithm can then be implemented as a loop containing the query.

Answer:
FILL

8.9 Suppose the *student* relation has an attribute named *location* of type point, and the *classroom* relation has an attribute *location* of type polygon. Write the following queries in SQL using the PostGIS spatial functions and predicates that we saw earlier:

a. Find the names of all students whose location is within the classroom Packard 101.

b. Find all classrooms that are within 100 meters or Packard 101; assume all distances are represented in units of meters.

c. Find the ID and name of student who is geographically nearest to the student with ID 12345.

d. Find the ID and names of all pairs of students whose locations are less than 200 meters apart.

Answer:
FILL

CHAPTER 9

Application 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, servlets 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 overhead saved by avoiding process creation 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 the 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 another value, 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 log in to the system and access the unauthorized page by entering its URL in the browser. If the check for user authorization was inadvertently 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 Why is it important to open JDBC connections using the try-with-resources (`try (…){ … }`) syntax?

Answer:

This ensures connections are closed properly, and you will not run out of database connections.

9.6 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 beforehand, and each request uses one from those.

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

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

9.7 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.

9.8 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 forms of HTML input other than text boxes? 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. Most modern browsers provide a way for users to edit the DOM tree. This feature can be able to modify the values sent to the application, inserting a single quote into the value.

9.9 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 requires 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.

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

Answer:
When the entire database is encrypted, it is easy for the database to perform decryption as data are 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 unencrypted 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.

9.11 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 gets 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, and it gave C a certificate *cert*. Now C can communicate with person Y, who checks the certificate *cert* presented by C and believes the public key contained in *cert* really belongs to X. C can 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 a 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 could be created. But certificates that are not part of this chain would not be affected.

9.12 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

Big Data

Practice Exercises

10.1 Suppose you need to store a very large number of small files, each of size say 2 kilobytes. If your choice is between a distributed file system and a distributed key-value store, which would you prefer, and explain why.

Answer:
The key-value store, since the distributed file system is designed to store a moderate number of large files. With each file block being multiple megabytes, kilobyte-sized files would result in a lot of wasted space in each block and poor storage performance.

10.2 Suppose you need to store data for a very large number of students in a distributed document store such as MongoDB. Suppose also that the data for each student correspond to the data in the *student* and the *takes* relations. How would you represent the above data about students, ensuring that all the data for a particular student can be accessed efficiently? Give an example of the data representation for one student.

Answer:
We would store the student data as a JSON object, with the takes tuples for the student stored as a JSON array of objects, each object corresponding to a single takes tuple. Give example ...

10.3 Suppose you wish to store utility bills for a large number of users, where each bill is identified by a customer ID and a date. How would you store the bills in a key-value store that supports range queries, if queries request the bills of a specified customer for a specified date range.

Answer:
Create a key by concatenating the customer ID and date (with date represented in the form year/month/date, e.g., 2018/02/28) and store the records indexed on this key. Now the required records can be retrieved by a range query.

10.4 Give pseudocode for computing a join $r \bowtie_{r.A=s.A} s$ using a single MapReduce step, assuming that the map() function is invoked on each tuple of r and s. Assume that the map() function can find the name of the relation using context.relname().

Answer:
With the map function, output records from both the input relations, using the join attribute value as the reduce key. The reduce function gets records from both relations with matching join attribute values and outputs all matching pairs.

10.5 What is the conceptual problem with the following snippet of Apache Spark code meant to work on very large data. Note that the collect() function returns a Java collection, and Java collections (from Java 8 onwards) support map and reduce functions.

```
JavaRDD<String< lines = sc.textFile("logDirectory");
int totalLength = lines.collect().map(s -> s.length())
                        .reduce(0,(a,b) -> a+b);
```

Answer:
The problem with the code is that the collect() function gathers the RDD data at a single node, and the map and reduce functions are then executed on that single node, not in parallel as intended.

10.6 Apache Spark:

a. How does Apache Spark perform computations in parallel?

b. Explain the statement: "Apache Spark performs transformations on RDDs in a lazy manner."

c. What are some of the benefits of lazy evaluation of operations in Apache Spark?

Answer:

a. RDDs are stored partitioned across multiple nodes. Each of the transformation operations on an RDD are executed in parallel on multiple nodes.

b. Transformations are not executed immediately but postponed until the result is required for functions such as collect() or saveAsTextFile().

c. The operations are organized into a tree, and query optimization can be applied to the tree to speed up computation. Also, answers can be pipelined from one operation to another, without being written to disk, to reduce time overheads of disk storage.

10.7 Given a collection of documents, for each word w_i, let n_i denote the number of times the word occurs in the collection. Let N be the total number of word occurrences across all documents. Next, consider all pairs of consecutive words (w_i, w_j) in the document; let $n_{i,j}$ denote the number of occurrences of the word pair (w_i, w_j) across all documents.

Write an Apache Spark program that, given a collection of documents in a directory, computes N, all pairs (w_i, n_i), and all pairs $((w_i, w_j), n_{i,j})$. Then output all word pairs such that $n_{i,j}/N \geq 10 * (n_i/N) * (n_j/N)$. These are word pairs that occur 10 times or more as frequently as they would be expected to occur if the two words occurred independently of each other.

You will find the join operation on RDDs useful for the last step, to bring related counts together. For simplicity, do not bother about word pairs that cross lines. Also assume for simplicity that words only occur in lowercase and that there are no punctuation marks.

Answer:
FILL IN ANSWER (available with SS)

10.8 Consider the following query using the tumbling window operator:

> **select** *item*, *System.Timestamp* **as** *window_end*, **sum**(*amount*)
> **from** *order* **timestamp by** *datetime*
> **group by** *itemid*, **tumblingwindow**(*hour*, 1)

Give an equivalent query using normal SQL constructs, without using the tumbling window operator. You can assume that the timestamp can be converted to an integer value that represents the number of seconds elapsed since (say) midnight, January 1, 1970, using the function *to_seconds*(*timestamp*). You can also assume that the usual arithmetic functions are available, along with the function *floor*(a) which returns the largest integer $\leq a$.

Answer:
Divide by 3600, and take floor, group by that. To output the timestamp of the window end, add 1 to hour and multiply by 3600

10.9 Suppose you wish to model the university schema as a graph. For each of the following relations, explain whether the relation would be modeled as a node or as an edge:

(i) *student*, (ii) *instructor*, (iii) *course*, (iv) *section*, (v) *takes*, (vi) *teaches*.
Does the model capture connections between sections and courses?

Answer:
Each relation corresponding to an entity (student, instructor, course, and section) would be modeled as a node. *Takes* and *teaches* would be modeled as edges. There is a further edge between *course* and *section*, which has been

merged into the *section* relation and cannot be captured with the above schema. It can be modeled if we create a separate relation that links sections to courses.

CHAPTER 11

Data Analytics

Practice Exercises

11.1 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 they become available. For a destination-driven architecture to collect data as soon as they are 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 are 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 architecture, 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.

11.2 Draw a diagram that shows how the *classroom* relation of our university example as shown in Appendix A would be stored under a column-oriented storage structure.

Answer:

The relation would be stored in three files, one per attribute, as shown below. We assume that the row number can be inferred implicitly from position, by using fixed-size space for each attribute. Otherwise, the row number would also have to be stored explicitly.

building
Packard
Painter
Taylor
Watson
Watson

room_number
101
514
3128
100
120

capacity
500
10
70
30
50

11.3 Consider the *takes* relation. Write an SQL query that computes a cross-tab that has a column for each of the years 2017 and 2018, and a column for **all**, and one row for each course, as well as a row for **all**. Each cell in the table should contain the number of students who took the corresponding course in the corresponding year, with column **all** containing the aggregate across all years, and row **all** containing the aggregate across all courses.

Answer:

11.4 Consider the data warehouse schema depicted in Figure 11.2. 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*)

11.5 Classification can be done using *classification rules*, which have a *condition*, a *class*, and a *confidence*; the confidence is the percentage of the inputs satisfying the condition that fall in the specified class.

For example, a classification rule for credit ratings may have a condition that salary is between $30,000 and $50,000, and education level is graduate, with the credit rating class of *good*, and a confidence of 80%. A second rule may have a condition that salary is between $30,000 and $50,000, and education level is high-school, with the credit rating class of *satisfactory*, and a confidence of 80%. A third rule may have a condition that salary is above $50,001, with the credit rating class of *excellent*, and confidence of 90%. Show a decision tree classifier corresponding to the above rules.

Show how the decision tree classifier can be extended to record the confidence values.

Answer:
FILL IN

11.6 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. Let *pos* denote the fraction of *true positives*, which is 5% of the test cases, and let *neg* denote the fraction of *true negatives*, which is 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.

For each classifier, let *t_pos* denote the *true positive* fraction, that is the fraction of cases where the classifier prediction was positive, and the person actually had the disease. Let *f_pos* denote the *false positive* fraction, that is the fraction of cases where the prediction was positive, but the person did not have the disease. Let *t_neg* denote *true negative* and *f_neg* denote *false negative* fractions, which are defined similarly, but for the cases where the classifier prediction was negative.

a. Compute the following metrics for each classifier:
   i. *Accuracy*, defined as (*t_pos* + *t_neg*)/(*pos*+*neg*), that is, the fraction of the time when the classifier gives the correct classification.
   ii. *Recall* (also known as *sensitivity*) defined as *t_pos*/*pos*, that is, how many of the actual positive cases are classified as positive.
   iii. *Precision*, defined as *t_pos*/(*t_pos*+*f_pos*), that is, how often the positive prediction is correct.
   iv. *Specificity*, defined as *t_neg*/*neg*.

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:
FILL

CHAPTER 12

Physical Storage Systems

Practice Exercises

12.1 SSDs can be used as a storage layer between memory and magnetic disks, with some parts of the database (e.g., some relations) stored on SSDs and the rest on magnetic disks. Alternatively, SSDs can be used as a buffer or cache for magnetic disks; frequently used blocks would reside on the SSD layer, while infrequently used blocks would reside on magnetic disk.

a. Which of the two alternatives would you choose if you need to support real-time queries that must be answered within a guaranteed short period of time? Explain why.

b. Which of the two alternatives would you choose if you had a very large *customer* relation, where only some disk blocks of the relation are accessed frequently, with other blocks rarely accessed.

Answer:
In the first case, SSD as storage layer is better since performance is guaranteed. With SSD as cache, some requests may have to read from magnetic disk, causing delays.

In the second case, since we don't know exactly which blocks are frequently accessed at a higher level, it is not possible to assign part of the relation to SSD. Since the relation is very large, it is not possible to assign all of the relation to SSD. The SSD as cache option will work better in this case.

12.2 Some databases use magnetic disks in a way that only sectors in outer tracks are used, while sectors in inner tracks are left unused. What might be the benefits of doing so?

Answer:
The disk's data-transfer rate will be greater on the outer tracks than the inner tracks. This is because the disk spins at a constant rate, so more sectors pass underneath the drive head in a given amount of time when the arm is posi-

tioned on an outer track than when on an inner track. Even more importantly, by using only outer tracks, the disk arm movement is minimized, reducing the disk access latency. This aspect is important for transaction-processing systems, where latency affects the transaction-processing rate.

12.3 Flash storage:

a. How is the flash translation table, which is used to map logical page numbers to physical page numbers, created in memory?

b. Suppose you have a 64-gigabyte flash storage system, with a 4096-byte page size. How big would the flash translation table be, assuming each page has a 32-bit address, and the table is stored as an array?

c. Suggest how to reduce the size of the translation table if very often long ranges of consecutive logical page numbers are mapped to consecutive physical page numbers.

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 gigabytes 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.

12.4 Consider the following data and parity-block arrangement on four disks:

Disk 1	Disk 2	Disk 3	Disk 4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

The B_is represent data blocks; the P_is represent parity blocks. Parity block P_i is the parity block for data blocks B_{4i-3} to B_{4i}. What, if any, problem might this arrangement present?

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.

12.5 A database administrator can choose how many disks are organized into a single RAID 5 array. What are the trade-offs between having fewer disks versus more disks, in terms of cost, reliability, performance during failure, and performance during rebuild?

Answer:

Fewer disks has higher cost, but with more disks, the chance of two disk failures, which would lead to data loss, is higher. Further, performance during failure would be poor since a block read from a failed disk would result a large number of block reads from the other disks. Similarly, the overhead for rebuilding the failed disk would also be higher, since more disks need to be read to reconstruct the data in the failed disk.

12.6 A power failure that occurs while a disk block is being written could result in the block being only partially written. Assume that partially written blocks can be detected. An atomic block write is one where either the disk block is fully written or nothing is written (i.e., there are no partial writes). Suggest schemes for getting the effect of atomic block writes with the following RAID schemes. Your schemes should involve work on recovery from failure.

a. RAID level 1 (mirroring)

b. RAID level 5 (block interleaved, distributed parity)

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, its 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.

12.7 Storing all blocks of a large file on consecutive disk blocks would minimize seeks during sequential file reads. Why is it impractical to do so? What do operating systems do instead, to minimize the number of seeks during sequential reads?

Answer:

Reading data sequentially from a large file could be done with only one seek if the entire file were stored on consecutive disk blocks. Ensuring availability of large numbers of consecutive free blocks is not easy, since files are created and deleted, resulting in fragmentation of the free blocks on disks. Operating systems allocate blocks on large but fixed-sized sequential extents instead, and only one seek is required per extent.

CHAPTER 13

Data Storage Structures

Practice Exercises

13.1 Consider the deletion of record 5 from the file of Figure 13.3. Compare the relative merits of the following techniques for implementing the deletion:

a. Move record 6 to the space occupied by record 5, and move record 7 to the space occupied by record 6.

b. Move record 7 to the space occupied by record 5.

c. Mark record 5 as deleted, and move no records.

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 it 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 the reduced availability of contiguous free records.

13.2 Show the structure of the file of Figure 13.4 after each of the following steps:

a. Insert (24556, Turnamian, Finance, 98000).

b. Delete record 2.

c. Insert (34556, Thompson, Music, 67000).

Answer:

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

Figure 13.101 The file after **insert** (24556, Turnamian, Finance, 98000).

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

Figure 13.102 The file after **delete** record 2.

We use "↑ *i*" to denote a pointer to record "*i*".

a. See **??**.

b. See **??**. Note that the free record chain could have alternatively been from the header to 4, from 4 to 2, and finally from 2 to 6.

c. See **??**.

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

Figure 13.103 The file after **insert** (34556, Thompson, Music, 67000).

13.3 Consider the relations *section* and *takes*. Give an example instance of these two relations, with three sections, each of which has five students. Give a file structure of these relations that uses multitable clustering.

Answer:
The relation *section* with three tuples is as follows:

course_id	sec_id	semester	year	building	room
BIO-301	1	Summer	2010	Painter	514
CS-101	1	Fall	2009	Packard	101
CS-347	1	Fall	2009	Taylor	3128

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

See **??**.
The multitable clustering for the above two instances can be taken as:

13.4 Consider the bitmap representation of the free-space map, where for each block in the file, two bits are maintained in the bitmap. If the block is between 0 and 30 percent full the bits are 00, between 30 and 60 percent the bits are 01, between 60 and 90 percent the bits are 10, and above 90 percent the bits are 11. Such bitmaps can be kept in memory even for quite large files.

a. Outline two benefits and one drawback to using two bits for a block, instead of one byte as described earlier in this chapter.

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

Figure 13.104 The relation *takes* with five students for each section.

b. Describe how to keep the bitmap up to date on record insertions and deletions.

c. Outline the benefit of the bitmap technique over free lists in searching for free space and in updating free space information.

Answer:

a. The space used is less with 2 bits, and the number of times the free-space map needs to be updated decreases significantly, since many inserts/deletes do not result in any change in the free-space map. However, we have only an approximate idea of the free space available, which could lead both to wasted space and/or to increased search cost for finding free space for a record.

b. Every time 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.

c. 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 a proper-sized one is found, 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

BIO-301	1	Summer	2010	Painter	5
17968	BIO-301	1	Summer	2010	n
59762	BIO-301	1	Summer	2010	n
78546	BIO-301	1	Summer	2010	n
89729	BIO-301	1	Summer	2010	n
98988	BIO-301	1	Summer	2010	n
CS-101	1	Fall	2009	Packard	1
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	3
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

Figure 13.105 The multitable clustering for the above two instances can be taken as:

deleted, bitmap technique is more convenient for updating free space information.

13.5 It is important to be able to quickly find out if a block is present in the buffer, and if so where in the buffer it resides. Given that database buffer sizes are very large, what (in-memory) data structure would you use for this task?

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.

13.6 Suppose your university has a very large number of *takes* records, accumulated over many years. Explain how table partitioning can be done on the *takes* relation, and what benefits it could offer. Explain also one potential drawback of the technique.

Answer:
The table can be partitioned on (year, semester). Old *takes* records that are no longer accessed frequently can be stored on magnetic disk, while newer records can be stored on SSD. Queries that specify a year can be answered without reading records for other years.

A drawback is that queries that fetch records corresponding to multiple years will have a higher overhead, since the records may be partitioned across different relations and disk blocks.

13.7 Give an example of a relational-algebra expression and a query-processing strategy in each of the following situations:

a. MRU is preferable to LRU.

b. LRU is preferable to MRU.

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 Section 13.8a.

13.8 PostgreSQL normally uses a small buffer, leaving it to the operating system buffer manager to manage the rest of main memory available for file system buffering. Explain (a) what is the benefit of this approach, and (b) one key limitation of this approach.

Answer:
The database system does not know what are the memory demands from other processes. By using a small buffer, PostgreSQL ensures that it does not grab too much of main memory. But at the same time, even if a block is evicted from buffer, if the file system buffer manager has enough memory allocated to it, the evicted page is likely to still be cached in the file system buffer. Thus, a database buffer miss is often not very expensive since the block is still in the file system buffer.

The drawback of this approach is that the database system may not be able to control the file system buffer replacement policy. Thus, the operating system may make suboptimal decisions on what to evict from the file system buffer.

CHAPTER 14

Indexing

Practice Exercises

14.1 Indices speed query processing, but it is usually a bad idea to create indices on every attribute, and every combination of attributes, that are potential search keys. Explain why.

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.

14.2 Is it possible in general to have two clustering indices on the same relation for different search keys? Explain your answer.

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 the 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.

14.3 Construct a B^+-tree for the following set of key values:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Assume that the tree is initially empty and values are added in ascending order. Construct B$^+$-trees for the cases where the number of pointers that will fit in one node is as follows:

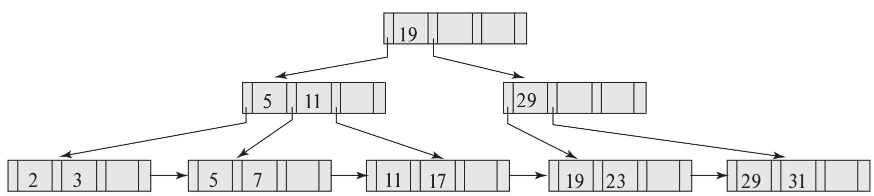
a. Four

b. Six

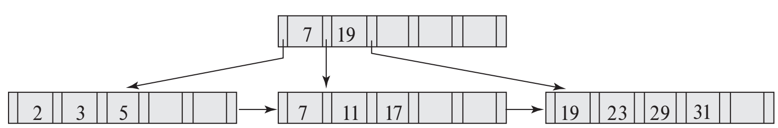
c. Eight

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.

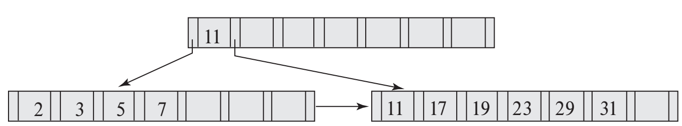
a.



b.



c.



14.4 For each B$^+$-tree of Exercise 14.3, show the form of the tree after each of the following series of operations:

a. Insert 9.

b. Insert 10.

c. Insert 8.

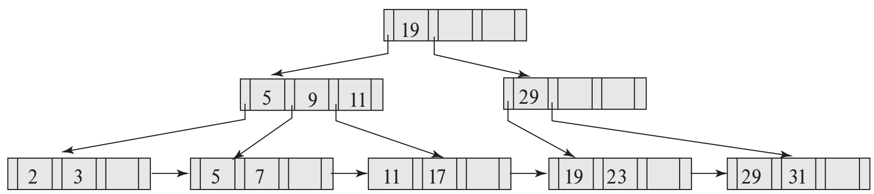
d. Delete 23.

e. Delete 19.

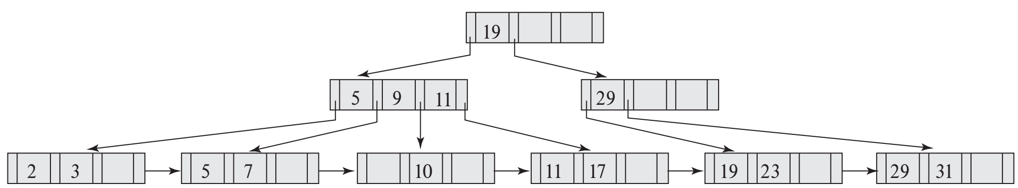
Answer:

- With structure Exercise 14.3.a:

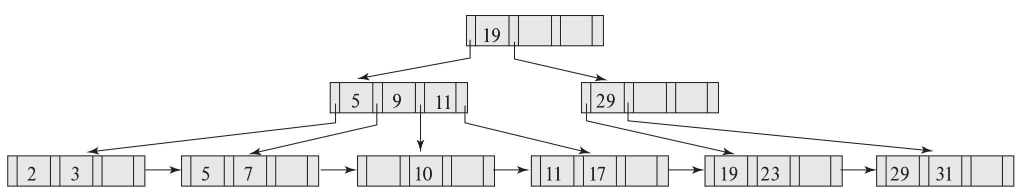
  Insert 9:



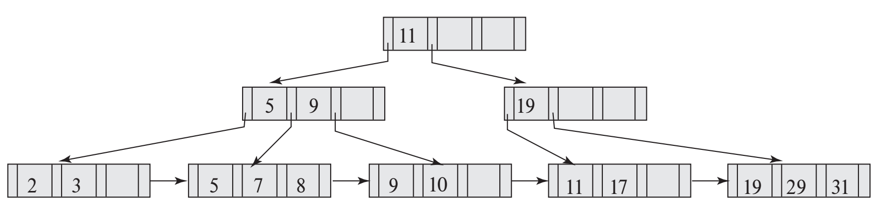
Insert 10:



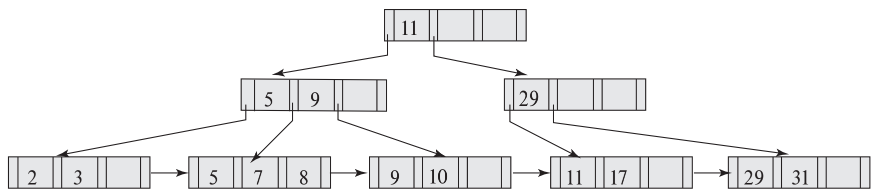
Insert 8:



Delete 23:

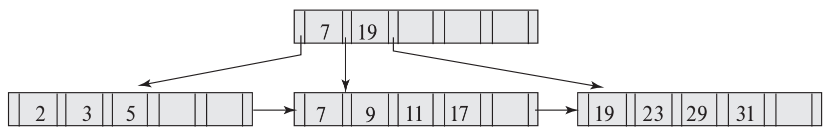


Delete 19:

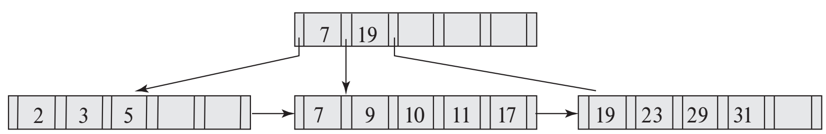


- With structure Exercise 14.3.b:

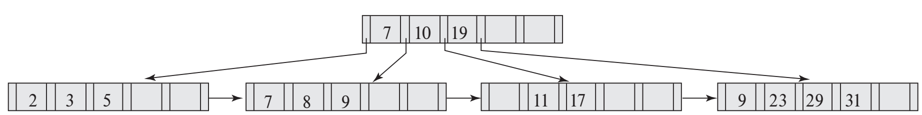
Insert 9:



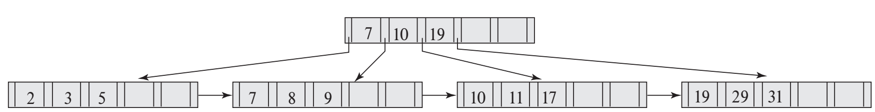
Insert 10:



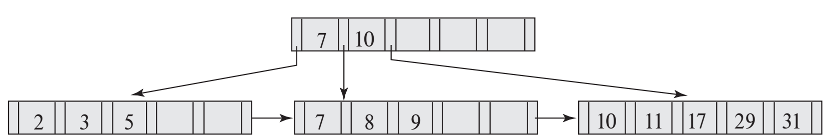
Insert 8:



Delete 23:

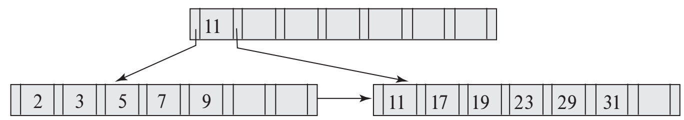


Delete 19:

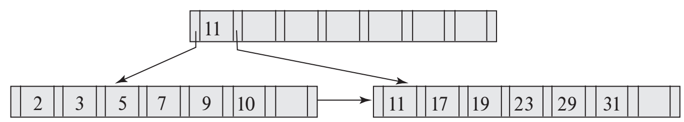


- With structure Exercise 14.3.c:

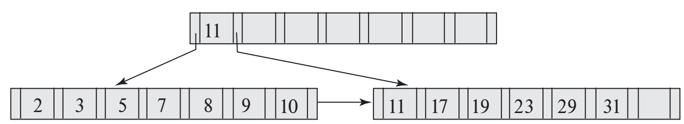
  Insert 9:



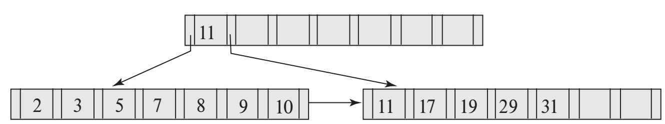
  Insert 10:



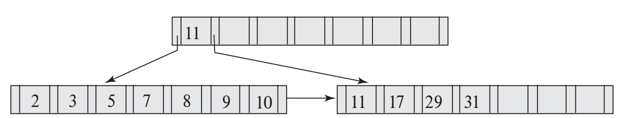
  Insert 8:



  Delete 23:



  Delete 19:



14.5 Consider the modified redistribution scheme for B$^+$-trees described on page 651. What is the expected height of the tree as a function of n?

Answer:

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

14.6 Give pseudocode for a B$^+$-tree function findRangeIterator(), which is like the function findRange(), except that it returns an iterator object, as described in Section 14.3.2. Also give pseudocode for the iterator class, including the variables in the iterator object, and the next() method.

Answer:

FILL IN

14.7 What would the occupancy of each leaf node of a B^+-tree be if index entries were inserted in sorted order? Explain why.

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 take place on the right node. This makes the occupancy of the leaf nodes about 50 percent except for 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.

14.8 Suppose you have a relation r with n_r tuples on which a secondary B^+-tree is to be constructed.

a. Give a formula for the cost of building the B^+-tree index by inserting one record at a time. Assume each block will hold an average of f entries and that all levels of the tree above the leaf are in memory.

b. Assuming a random disk access takes 10 milliseconds, what is the cost of index construction on a relation with 10 million records?

c. Write pseudocode for bottom-up construction of a B^+-tree, which was outlined in Section 14.4.4. You can assume that a function to efficiently sort a large file is available.

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 built 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 them are at least half filled.

14.9 The leaf nodes of a B^+-tree file organization may lose sequentiality after a sequence of inserts.

a. Explain why sequentiality may be lost.

b. To minimize the number of seeks in a sequential scan, many databases allocate leaf pages in extents of n blocks, for some reasonably large n. When the first leaf of a B^+-tree is allocated, only one block of an n-block unit is used, and the remaining pages are free. If a page splits, and its n-block unit has a free page, that space is used for the new page. If the n-block unit is full, another n-block unit is allocated, and the first $n/2$ leaf pages are placed in one n-block unit and the remaining one in the second n-block unit. For simplicity, assume that there are no delete operations.

   i. What is the worst-case occupancy of allocated space, assuming no delete operations, after the first n-block unit is full?

   ii. Is it possible that leaf nodes allocated to an n-node block unit are not consecutive, that is, is it possible that two leaf nodes are allocated to one n-node block, but another leaf node in between the two is allocated to a different n-node block?

   iii. Under the reasonable assumption that buffer space is sufficient to store an n-page block, how many seeks would be required for a leaf-level scan of the B^+-tree, in the worst case? Compare this number with the worst case if leaf pages are allocated a block at a time.

   iv. The technique of redistributing values to siblings to improve space utilization is likely to be more efficient when used with the preceding allocation scheme for leaf blocks. Explain why.

Answer:

a. 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.

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 pages 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 come down by a factor of $n/2$.

iv. Allowing redistribution among the nodes of the same block does not require additional seeks, whereas in regular B^+-trees 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)

14.10 Suppose you are given a database schema and some queries that are executed frequently. How would you use the above information to decide what indices to create?

Answer:

Indices on any attributes on which there are selection conditions; if there are only a few distinct values for that attribute, a bitmap index may be created, otherwise a normal B^+-tree index.

B^+-tree indices on primary-key and foreign-key attributes.

Also indices on attributes that are involved in join conditions in the queries.

14.11 In write-optimized trees such as the LSM tree or the stepped-merge index, entries in one level are merged into the next level only when the level is full. Suggest how this policy can be changed to improve read performance during periods when there are many reads but no updates.

Answer:

If there have been no updates in a while, but there are a lot of index look ups on an index, then entries at one level, say i, can be merged into the next level, even if the level is not full. The benefit is that reads would then not have to look up indices at level i, reducing the cost of reads.

14.12 What trade offs do buffer trees pose as compared to LSM trees?

Answer:

The idea of buffer trees can be used with any tree-structured index to reduce the cost of inserts and updates, including spatial indices. In contrast, LSM trees can only be used with linearly ordered data that are amenable to merging. On the other hand, buffer trees require more random I/O to perform insert operations as compared to (all variants of) LSM trees.

Write-optimized indices can significantly reduce the cost of inserts, and to a lesser extent, of updates, as compared to B^+-trees. On the other hand, the index lookup cost can be significantly higher for write-optimized indices as compared to B^+-trees.

14.13 Consider the *instructor* relation shown in Figure 14.1.

a. Construct a bitmap index on the attribute *salary*, dividing *salary* values into four ranges: below 50,000, 50,000 to below 60,000, 60,000 to below 70,000, and 70,000 and above.

b. Consider a query that requests all instructors in the Finance department with salary of 80,000 or more. Outline the steps in answering the query, and show the final and intermediate bitmaps constructed to answer the query.

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.

14.14 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:
FILL IN

14.15 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:
Start with regions with very small radius, and retry with a larger radius if a particular region does not contain any result. For example, each time the radius could be increased by a factor of (say) 1.5. The benefit is that since we do not use a very large radius compared to the minimum radius required, there will (hopefully!) not be too many points in the circular range query result.

CHAPTER 15

Query Processing

Practice Exercises

15.1 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most three 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:

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

15.2 Consider the bank database of Figure 15.14, 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.

15.3 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,

---

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 15.14 Bank database.

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 $B_s = 1500(2\lceil log_{M-1}(1500/M)\rceil + 2) + 800(2\lceil log_{M-1}(800/M)\rceil + 2)$ 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.

15.4 The indexed nested-loop join algorithm described in Section 15.5.3 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.

15.5 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 the memory requirement is $min(b_r, b_s) + 2$ pages.

15.6 Consider the bank database of Figure 15.14, 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. $\sigma_{\neg(branch\_city < \text{“Brooklyn”} \lor assets < 5000)}(branch)$

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

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

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.

15.7 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. Please see **??**

15.8 Design sort-based and hash-based algorithms for computing the relational division operation (see Practice Exercise 2.9 for a definition of the division operation).

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

For a 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 we 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, and 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.

15.9 What is the effect on the cost of merging runs if the number of buffer blocks per run is increased while overall memory available for buffering runs remains 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.

```
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
```

Figure 15.101 Answer for Exercise 15.7.

15.10 Consider the following extended relational-algebra operators. Describe how to implement each operation using sorting and using hashing.

a. **Semijoin** ($\ltimes_\theta$): The multiset semijoin operator $r \ltimes_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta$ if there is at least one tuple s_j such that r_i and s_j satisfy predicate θ; otherwise r_i does not appear in the result.

b. **Anti-semijoin** ($\overline{\ltimes}_\theta$): The multiset anti-semijoin operator $r \overline{\ltimes}_\theta s$ is defined as follows: if a tuple r_i appears n times in r, it appears n times in the result of $r \ltimes_\theta$ if there does not exist any tuple s_j in s such that r_i and s_j satisfy predicate θ; otherwise r_i does not appear in the result.

Answer:

FILL IN: CHeck for duplicate preservation

As in the case of join algorithms, semijoin and anti-semijoin can be done efficiently if the join conditions are equijoin conditions. We describe below how to efficiently handle the case of equijoin conditions using sorting and hashing. With arbitrary join conditions, sorting and hashing cannot be used; (block) nested loops join needs to be used instead.

a. **Semijoin:**

- **Semijoin using sorting:** Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge algorithm and add tuples of r to the result whenever the join attributes of the current tuples of r and s match.
- **Semijoin using hashing:** Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a value, add the current tuple of r to the result.

  Note that if r and s are large, they can be partitioned on the join attributes first and the above procedure applied on each partition. If r is small but s is large, a hash index can be built on r and probed using s; and if an s tuple matches an r tuple, the r tuple can be output and deleted from the hash index.

b. **Anti-semijoin:**

- **Anti-semijoin using sorting**: Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge algorithm and add tuples of r to the result if no tuple of s satisfies the join predicate for the corresponding tuple of r.
- **Anti-semijoin using hashing**: Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a null value, add the current tuple of r to the result.

As for semijoin, partitioning can be used if r and s are large. An index on r can be used instead of an index on s, but then when an s tuple matches an r tuple, the r tuple is deleted from the index. After processing all s tuples, all remaining r tuples in the index are output as the result of the anti-semijoin operation.

15.11 Suppose a query retrieves only the first K results of an operation and terminates after that. Which choice of demand-driven or producer-driven pipelining (with buffering) would be a good choice for such a query? Explain your answer.

Answer:
Demand driven is better, since it will only generate the top K results. Producer driven may generate a lot more answers, many of which would not get used.

15.12 Current generation CPUs include an *instruction cache*, which caches recently used instructions. A function call then has a significant overhead because the set of instructions being executed changes, resulting in cache misses on the instruction cache.

a. Explain why producer-driven pipelining with buffering is likely to result in a better instruction cache hit rate, as compared to demand-driven pipelining.

b. Explain why modifying demand-driven pipelining by generating multiple results on one call to *next*(), and returning them together, can improve the instruction cache hit rate.

Answer:
Producer-driven pipelining executes the same set of instructions to generate multiple tuples by consuming already generated tuples from the inputs. Thus instruction cache hits will be more. In comparison, demand-driven pipelining switches from the instructions of one function to another for each tuple, resulting in more misses.

By generating multiple results at one go, a *next*(() function would receive multiple tuples in its inputs and have a loop that generates multiple tuples for its output without switching execution to another function. Thus, the instruction cache hit rate can be expected to improve.

15.13 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 in **??**

```
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;
```

Figure 15.102 Answer for Exercise 15.13.

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.

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.

15.14 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.

15.15 Explain why the nested-loops join algorithm (see Section 15.5.1) would work poorly on a database stored in a column-oriented manner. Describe an alternative algorithm that would work better, and explain why your solution is better.

Answer:
If the nested-loops join algorithm is used as is, it would require tuples for each of the relations to be assembled before they are joined. Assembling tuples can be expensive in a column store, since each attribute may come from a separate area of the disk; the overhead of assembly would be particularly wasteful if many tuples do not satisfy the join condition and would be discarded. In such a situation it would be better to first find which tuples match by accessing only the join columns of the relations. Sort-merge join, hash join, or indexed nested loops join can be used for this task. After the join is performed, only tuples that get output by the join need to be assembled; assembly can be done by sorting the join result on the record identifier of one of the relations and accessing the corresponding attributes, then resorting on record identifiers of the other relation to access its attributes.

15.16 Consider the following queries. For each query, indicate if column-oriented storage is likely to be beneficial or not, and explain why.

a. Fetch ID, *name* and *dept_name* of the student with ID 12345.

b. Group the *takes* relation by *year* and *course_id*, and find the total number of students for each (*year*, *course_id*) combination.

Answer:
FILL IN AND recheck question

CHAPTER 16

Query Optimization

Practice Exercises

16.1 Download the university database schema and the large university dataset from dbbook.com. Create the university schema on your favorite database, and load the large university dataset. Use the **explain** feature described in Note 16.1 on page 746 to view the plan chosen by the database, in different cases as detailed below.

a. Write a query with an equality condition on *student.name* (which does not have an index), and view the plan chosen.

b. Create an index on the attribute *student.name*, and view the plan chosen for the above query.

c. Create simple queries joining two relations, or three relations, and view the plans chosen.

d. Create a query that computes an aggregate with grouping, and view the plan chosen.

e. Create an SQL query whose chosen plan uses a semijoin operation.

f. Create an SQL query that uses a **not in** clause, with a subquery using aggregation. Observe what plan is chosen.

g. Create a query for which the chosen plan uses correlated evaluation (the way correlated evaluation is represented varies by database, but most databases would show a filter or a project operator with a subplan or subquery).

h. Create an SQL update query that updates a single row in a relation. View the plan chosen for the update query.

i. Create an SQL update query that updates a large number of rows in a relation, using a subquery to compute the new value. View the plan chosen for the update query.

Answer:
The answer depends on the database.
FILL IN Suggested queries for each exercise as verified on some database

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

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

b. $\sigma_\theta({}_A\gamma_F(E)) \;\equiv\; {}_A\gamma_F(\sigma_\theta(E))$, where θ uses only attributes from A.

c. $\sigma_\theta(E_1 ⟕ E_2) \equiv \sigma_\theta(E_1) ⟕ E_2$, 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, $t[E_3\text{'s attributes}]$ 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 $t[R_2\text{'s attributes}] \in E_2$ and $t[R_2\text{'s attributes}] \notin E_3$. 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. $\sigma_\theta({}_A\gamma_F(E)) \;=\; {}_A\gamma_F(\sigma_\theta(E))$, where θ uses only attributes from A.

θ uses only attributes from A. Therefore if any tuple t in the output of ${}_A\gamma_F(E)$ 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:

$$\forall t,\; t \notin \sigma_\theta({}_A\gamma_F(E)) \;\Rightarrow\; t \notin {}_A\gamma_F(\sigma_\theta(E))$$

Using similar reasoning, we can also conclude that

$$\forall t,\; t \notin {}_A\gamma_F(\sigma_\theta(E)) \;\Rightarrow\; t \notin \sigma_\theta({}_A\gamma_F(E))$$

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 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. $\sigma_\theta(E_1 ⟕ E_2) \;=\; \sigma_\theta(E_1) ⟕ E_2$ where θ uses only attributes from E_1.

θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 ⟕ E_2)$ 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:

$$\forall t,\; t \notin \sigma_\theta(E_1 ⟕ E_2) \;\Rightarrow\; t \notin \sigma_\theta(E_1) ⟕ E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\; t \notin \sigma_\theta(E_1) ⟕ E_2 \;\Rightarrow\; t \notin \sigma_\theta(E_1 ⟕ E_2)$$

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 going to be removed from the result. Thus the right-hand side expression can be evaluated more efficiently than the left-hand side expression.

16.3 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. $\sigma_{B<4}(\,_A\gamma_{max(B)\ \textbf{as}\ B}(r))$ and $_A\gamma_{max(B)\ \textbf{as}\ B}(\sigma_{B<4}(r))$.

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

d. $(r ⟖ s) ⟖ t$ and $r ⟖ (s ⟖ t)$
In other words, the natural right outer join is not associative.

e. $\sigma_\theta(E_1 ⟕ E_2)$ and $E_1 ⟕ \sigma_\theta(E_2)$, 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 $\{(1,2,null)\}$.

16.4 SQL allows relations with duplicates (Chapter 3), and the multiset version of the relational algebra is defined in Note 3.1 on page 80, Note 3.2 on page 97, and Note 3.3 on page 108. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational algebra.

Answer:
All the equivalence rules 1 through 7.b of section Section 16.2.1 hold for the multiset version of the relational algebra defined in Chapter 2.
There exist equivalence rules that 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.

16.5 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.

16.6 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 16.5. 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.

16.7 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.

16.8 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.

16.9 Give conditions under which the following expressions are equivalent:

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

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.

16.10 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.

16.11 Modify the FindBestPlan(S) function to create a function FindBestPlan(S, O), where O is a desired sort order for S, and which considers interesting sort orders. A *null* order indicates that the order is not relevant. *Hints*: An algorithm A may give the desired order O; if not a sort operation may need to be added to get the desired order. If A is a merge-join, FindBestPlan must be invoked on the two inputs with the desired orders for the inputs.

Answer:
FILL IN

16.12 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.

16.13 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 16.4. 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.

16.14 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 $n2^{n-1}$.

16.15 Consider the bank database of Figure 16.9, 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, but in SQL.

c. Give a relational algebra expression using semijoin equivalent to the query.

d. Give a procedure (similar to that described in Section 16.4.4) 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 t_1 **as**
select *branch_name*, **max**(*balance*)
from *account*
group by *branch_name*
select *account_number*
from *account*, t_1
where *account.branch_name* **like** 'B%' **and**
account.branch_name = t_1.*branch_name* **and**
account.balance = t_1.*balance*

c. FILL IN

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

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 16.9 Banking database.

select ...
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

***** FILL IN **** GIVE Relational algebra version *****

create table t_1 **as**
select f(A_2),V
from L_2
where P_2^1
group by V
select ...
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.

CHAPTER 17

Transactions

Practice Exercises

17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

Answer:

Even in this case the recovery manager is needed to perform rollback of aborted transactions for cases where the transaction itself fails.

17.2 Consider a file system such as the one on your favorite operating system.

a. What are the steps involved in the creation and deletion of files and in writing data to a file?

b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.

Answer:

There are several steps in the creation of a file. A storage area is assigned to the file in the file system. (In UNIX, 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, 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 steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

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.

17.4 What class or classes of storage can be used to ensure durability? Why?

Answer:
Only stable storage ensures true durability. Even nonvolatile storage is susceptible to data loss, albeit less so than volatile storage. Stable storage is only an abstraction. It is approximated by redundant use of nonvolatile storage in which data are not only replicated but distributed phyically to reduce to near zero the chance of a single event casuing data loss.

17.5 Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

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.

17.6 Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.

Answer:

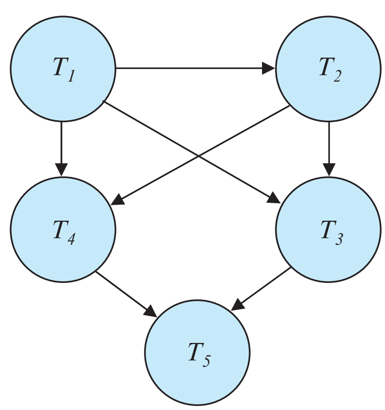


Figure 17.16 Precedence graph for Practice Exercise 17.6.

There is a serializable schedule corresponding to the precedence graph 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.

17.7 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.

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.

17.8 The **lost update** anomaly is said to occur if a transaction T_j reads a data item, then another transaction T_k writes the data item (possibly based on a previous read), after which T_j writes the data item. The update performed by T_k has been lost, since the update done by T_j ignored the value written by T_k.

a. Give an example of a schedule showing the lost update anomaly.

b. Give an example schedule to show that the lost update anomaly is possible with the **read committed** isolation level.

c. Explain why the lost update anomaly is not possible with the **repeatable read** isolation level.

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.

17.9 Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.

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 nonserializable execution which results into a serious problem.

17.10 Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.

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 will 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.

17.11 The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.

Does this situation cause any problem for the definition of conflict serializability? Explain your answer.

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 18

Concurrency Control

Practice Exercises

18.1 Show that the two-phase locking protocol ensures conflict serializability and that transactions can be serialized according to their lock points.

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 nonserializable 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 nonserializable 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.

18.2 Consider the following two transactions:

T_{34}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

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.

18.3 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

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.

18.4 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

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 the 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* .

18.5 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

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)	

18.6 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix `mprotect` command, for example). Describe how the access-protection mechanism can be used for page-level locking in a persistent programming language.

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.

18.7 Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let V be the value of data item X. The operation

increment(X) by C

sets the value of X to $V + C$ in an atomic step. The value of X is not available to the transaction unless the latter executes a read(X).
Assume that increment operations lock the item in increment mode using the compatibility matrix in Figure 18.25.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency.

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," Journal of the ACM Volume 30, (1983).

18.8 In timestamp ordering, **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(Q)

successfully. Would this change in wording make any difference? Explain your answer.

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.

18.9 Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

Answer:

If a transaction needs to access a large 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.

18.10 For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking
- Two-phase locking with multiple-granularity locking.
- The tree protocol
- Timestamp ordering
- Validation
- Multiversion timestamp ordering
- Multiversion two-phase locking

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.

18.11 Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

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 reexecution 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 a 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 items 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 items 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.

18.12 Consider the timestamp-ordering protocol, and two transactions, one that writes two data items p and q, and another that reads the same two data items. Give a schedule whereby the timestamp test for a `write` operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

Answer:

Consider two transactions T_1 and T_2 shown below.

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

Let $\text{TS}(T_1) < \text{TS}(T_2)$, 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 $\text{TS}(T_1) <$ R-timestamp(q), since $\text{TS}(T_1) < \text{TS}(T_2)$ and R-timestamp(q) $= \text{TS}(T_2)$. Hence the write operation 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.

18.13 Devise a timestamp-based protocol that avoids the phantom phenomenon.

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.

18.14 Suppose that we use the tree protocol of Section 18.1.5 to manage concurrent access to a B^+-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

Answer:

Note: The tree protocol of Section Section 18.1.5 which is referred to in this question is different from the multigranularity protocol of Section 18.3 and the B^+-tree concurrency protocol of Section 18.10.2.

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, then 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 upward, 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.

18.15 The snapshot isolation protocol uses a validation step which, before performing a write of a data item by transaction T, checks if a transaction concurrent with T has already written the data item.

a. A straightforward implementation uses a start timestamp and a commit timestamp for each transaction, in addition to an *update set*, that, is the set of data items updated by the transaction. Explain how to perform validation for the first-committer-wins scheme by using the transaction timestamps along with the update sets. You may assume that validation and other commit processing steps are executed serially, that is, for one transaction at a time,

b. Explain how the validation step can be implemented as part of commit processing for the first-committer-wins scheme, using a modification of the above scheme, where instead of using update sets, each data item has a write timestamp associated with it. Again, you may assume that validation and other commit processing steps are executed serially.

c. The first-updater-wins scheme can be implemented using timestamps as described above, except that validation is done immediately after acquiring an exclusive lock, instead of being done at commit time.

   i. Explain how to assign write timestamps to data items to implement the first-updater-wins scheme.

   ii. Show that as a result of locking, if the validation is repeated at commit time the result would not change.

   iii. Explain why there is no need to perform validation and other commit processing steps serially in this case.

Answer:

a. Validation test for first-committer-wins scheme: Let StartTS(T_i), CommitTS(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 CommitTS(T_k) < CommitTS(T_i), one of the following two conditions must hold:

- If CommitTS(T_k) < StartTS(T_k), T_k completes its execution before T_i started, the serializability is maintained.
- StartTS(T_i) < CommitTS(T_k) < CommitTS(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 CommitTS(T_i). For the validation test of a transaction T_i to pass, the following condition must hold:

- For each data item Q written by T_i, W-timestamp(Q) < StartTS(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 in between the lock acquisition and commit time. So, the result of the 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 most one transaction can acquire the lock on a data item at a time and do the validation testing. Thus, two or more transactions cannot do validation testing for the same data item simultaneously.

18.16 Consider functions *insert_latchfree*() and *delete_latchfree*(), shown in Figure 18.23.

a. Explain how the ABA problem can occur if a deleted node is reinserted.

b. Suppose that adjacent to *head* we store a counter *cnt*. Also suppose that DCAS((*head,cnt*), (*oldhead, oldcnt*), (*newhead, newcnt*)) atomically performs a compare-and-swap on the 128 bit value (*head,cnt*). Modify the *insert_latchfree*() and *delete_latchfree*() to use the DCAS operation to avoid the ABA problem.

c. Since most processors use only 48 bits of a 64 bit address to actually address memory, explain how the other 16 bits can be used to implement a counter, in case the DCAS operation is not supported.

Answer:

a. Let the head of the list be pointer $n1$, and the next three elements be $n2$ and $n3$. Suppose process $P1$ which is performing a delete, reads pointer

$n1$ as head and $n2$ as *newhead*, but before it executes CAS(*head*, $n1$, $n2$), process $P2$ deletes $n1$, then deletes $n2$ and then inserts $n1$ back at the head.

The CAS would replace $n1$ by a pointer to $n2$, since the head is still $n1$. However, node $n2$ has meanwhile been deleted and is garbage. Thus, the list is now inconsistent.

b. The following code

```
atomic_read(head, cnt) {
    repeat
        oldhead = head
        oldcnt = cnt
        result = DCAS((head, cnt), (oldhead, oldcnt), (oldhead, oldcnt))
    until (result == success)
    return (oldhead, oldcnt)
}

insert_latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        (oldhead, oldcnt) = atomic_read(head, cnt)
        node->next = oldhead
        newcnt = oldcnt+1
        result = DCAS(head, (oldhead, oldcnt), (node, newcnt))
    until (result == success)
}

delete_latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        (oldhead, oldcnt) = atomic_read(head, cnt)
        newhead = oldhead->next
        newcnt = oldcnt+1
        result = DCAS(head, (oldhead, oldcnt), (newhead, newcnt))
    until (result == success)
}
```

The *atomic_read* function ensures that the 128 bit address, counter pair is read atomically, by using the DCAS instruction to ensure that the values are still same (the DCAS instruction stores the same values back if it succeeds, so there is no change in the value). If the DCAS fails, we may have read an old pointer and a new value, or vice versa, requiring the values to be read again.

The ABA problem would be avoided by the modified code for *insert_latchfree*() and *delete_latchfree*(), since although the reinsert of the $n1$ by $P2$ would result in the head having the same pointer $n1$ as earlier, counter *cnt* would be different from *oldcnt*, resulting in the CAS operation of $P1$ failing.

c. Most processors use only the last 48 bits of a 64 bit address to access memory (which can support 256 Terabytes of memory). The first 16 bits of a 64 bit value can then be used as a counter, and the last 48 bits as the address, with the counter and the address extracted using bit-and operations before being used, and using bit-and and bit-or operations to reconstruct the 64 bit value from a pointer and a counter. If a hardware implementation does not support DCAS, this could be used as an alternative to a DCAS, although it still runs a the small risk of the counter wrapping around if there are exactly 64K other operations on the list between the read of the head and the CAS operation.

CHAPTER 19

Recovery System

Practice Exercises

19.1 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 will 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 is set incorrectly to 2.

19.2 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 must be undone/redone from the log. If checkpointing is 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.

19.3 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 19.4?

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 practice, systems find a point in the log where all earlier log records can be deleted, and they 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 preceding rules for deletion 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 less than checkpointing, since a lot of data have to be written. Thus more log records would need to be retained with archival logging.

19.4 Describe how to modify the recovery algorithm of Section 19.4 to implement savepoints and to perform rollback to a savepoint. (Savepoints are described in Section 19.9.3.)

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 to a particular savepoint, execute undo processing for that transaction until 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 up to that savepoint.

19.5 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 because 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.

19.6 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 or 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 to 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, minimizing disk arm movement.

19.7 Suppose we (incorrectly) modify the recovery algorithm of Section 19.4 to note 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.

19.8 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.

19.9 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 a 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 until 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 until the first transaction completes.

19.10 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 customer's account.

19.11 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 19.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be reexecuted logically, if the updates are available in the form of SQL but cannot be reexecuted 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 $< T_2, A, 110, 120 >$ respectively.

Say we wish to undo transaction T_1. The normal transaction undo mechanism will replace the value in question—A in this example—with the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism, the resulting balance will be

$100 and we will, 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 to 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.

- Roll back 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*$>$ to 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. Reexecute 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 will become $120 and we will, in effect, redo both transactions, whereas we intend to redo only transaction T_2.

19.12 The recovery techniques that we described assume that blocks are written atomically to disk. However, a block may be partially written when power fails, with some sectors written, and others not yet written.

a. What problems can partial block writes cause?

b. Partial block writes can be detected using techniques similar to those used to validate sector reads. Explain how.

c. Explain how RAID 1 can be used to recover from a partially written block, restoring the block to either its old value or to its new value.

Answer:

FILL IN

19.13 The Oracle database system uses undo log records to provide a snapshot view of the database under snapshot isolation. The snapshot view seen by transaction T_i reflects updates of all transactions that had committed when T_i started and the updates of T_i; updates of all other transactions are not visible to T_i.

Describe a scheme for buffer handling whereby transactions are given a snapshot view of pages in the buffer. Include details of how to use the log to generate the snapshot view. You can assume that operations as well as their undo actions affect only one page.

Answer:

First, determine if a transaction is currently modifying the buffer. If not, then return the current contents of the buffer. Otherwise, examine the records in the undo log pertaining to this buffer. Make a copy of the buffer, then for each relevant operation in the undo log, apply the operation to the buffer copy starting with the most recent operation and working backwards until the point at which the modifying transaction began. Finally, return the buffer copy as the snapshot buffer.

CHAPTER 20

Database-System Architectures

Practice Exercises

20.1 Is a multiuser system necessarily a parallel system? Why or why not?

Answer:

No. A single processor with only one core can run multiple processes to manage mutiple users. Most modern systems are parallel, however.

20.2 Atomic instructions such as compare-and-swap and test-and-set also execute a memory fence as part of the instruction on many architectures. Explain what is the motivation for executing the memory fence, from the viewpoint of data in shared memory that is protected by a mutex implemented by the atomic instruction. Also explain what a process should do before releasing a mutex.

Answer:

FILL IN MORE

The memory fence ensures that the process that gets the mutex will see all updates that happened before the instruction, as long as processes execute a fence before releasing the mutex. Thus, even if the data was updated on a different core, the process that acquires the mutex is guaranteed to see the latest value of the data.

20.3 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.

20.4 Explain the distinction between a *latch* and a *lock* as used for transactional concurrency control.

Answer:
Latches are short-duration locks that manage access to internal system data structures. Locks taken by transactions are taken on database data items and are often held for a substantial fraction of the duration of the transaction. Latch acquisition and release are not covered by the two-phase locking protocol.

20.5 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 is 5. In Amdahl's law: $\frac{1}{(1-p)+(p/n)}$, $p = 4/5$ and n is arbitrarily large. So, $1 - p = 1/5$ and p/n aproaches zero.

20.6 Consider a pair of processes in a shared memory system such that process A updates a data structure, and then sets a flag to indicate that the update is completed. Process B monitors the flag, and starts processing the data structure only after it finds the flag is set.

Explain the problems that could arise in a memory architecture where writes may be reordered, and explain how the sfence and lfence instructions can be used to ensure the problem does not occur.

Answer:
The goal here is that the consumer process B should see the data structure state after all updates have been completed. But out of order writes to main memory can result in the consumer process seeing some but not all the updates to the data structure, even after the flag has been set.

To avoid this problem, the producer process A should issue an sfence after the updates, but before setting the flag. It can optionally issue an sfence after setting the flag, to push the update to memory with minimum delay. The consumer process B should correspondingly issue an lfence after the flag has been found to be set, before accessing the datastructure.

20.7 In a shared-memory architecture, why might the time to access a memory location vary depending on the memory location being accessed?

Answer:

In a NUMA architecture, a processor can access its own memory faster than it can access shared memory associated with another processor due to the time taken to transfer data between processors.

20.8 Most operating systems for parallel machines (i) allocate memory in a local memory area when a process requests memory, and (ii) avoid moving a process from one core to another. Why are these optimizations important with a NUMA architecture?

Answer:

In a NUMA architecture, a processor can access its own memory faster that it can access shared memory associated with another processor due to the time taken to transfer data between processors. Thus, if the data of a process resides in local memory, the process execution would be faster than if the memory is non-local.

Further, if a process moves from one core to another, it may lose the benefits of local allocation of memory, and be forced to carry out many memory accesses from other cores. To avoid this problem, most operating systems avoid moving a process from one core to another wherever possible.

20.9 Some database operations such as joins can see a significant difference in speed when data (e.g., one of the relations involved in a join) fits in memory as compared to the situation where the data do 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:

We illustrate this by an example. Suppose we double the amount of main memory and that as a result, one of the relations now fits entirely in main memory. We can now use a nested-loop join with the inner-loop relation entirely in main memory and incur disk accesses for reading the input relations only one time. With the original amount of main memory, the best join strategy may have had to read a relation in from disk more than once.

20.10 What is the key distinction between homogeneous and federated distributed database systems?

Answer:

The key diference is the degree of cooperation among the systems and the degree of centralized control. Homogeneous systems share a global schema, run the same database-system software and actively cooperate on query processing. Federated systems may have distinct schemas and software, and may cooperate in only a limited manner.

20.11 Why might a client choose to subscribe only to the basic infrastructure-as-a-service model rather than to the services offered by other cloud service models?

Answer:
A client may wish to control its own applications and thus may not wish to subscribe to a software-as-a-service model; or the client might wish further to be able to choose and manage its own database system and thus not wish to subscribe to a platform-as-a-service model.

20.12 Why do cloud-computing services support traditional database systems best by using a virtual machine, instead of running directly on the service provider's actual machine, assuming that data is on external storage?

Answer:
By using a virtual machine, if a physical machine fails, virtual machines running on that physical machine can be restarted quickly on one or more other physical machines, improving availability. (Assuming of course that data remains accessible, either by storing multiple copies of data, or by storing data in an highly available external storage system.)

CHAPTER 21

Parallel and Distributed Storage

Practice Exercises

21.1 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.

21.2 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 five 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}	:	$k_i - \Sigma_{l=1}^{j-1} n_l$

21.3 Histograms are traditionally constructed on the values of a specific attribute (or set of attributes) of a relation. Such histograms are good for avoiding data distribution skew but are not very useful for avoiding execution skew. Explain why.

Now suppose you have a workload of queries that perform point lookups. Explain how you can use the queries in the workload to come up with a partitioning scheme that avoids execution skew.

Answer:
FILL

21.4 Replication:

a. Give two reasons for replicating data across geographically distributed data centers.

b. Centralized databases support replication using log records. How is the replication in centralized databases different from that in parallel/distributed databases?

Answer:

a. By replicating across data centers, even if a data center fails, for example due to a power outage or a natural disaster, the data would still be available from another data center. By keeping the data centers geographically separated, the chances of a single natural disaster such as an earthquake or a storm affecting both the data centers at the same time are minimized.

b. Centralized databases typically support only full database replication using log records (although some support logical replication allowing replication to be restricted to some relations). However, they do not support partitioning, or the ability to replicate different parts of the database at different nodes; the latter helps minimize the load increase at a replica when a node fails by spreading the load across multiple nodes.

21.5 Parallel indices:

a. Secondary indices in a centralized database store the record identifier. A global secondary index too could potentially store a partition number holding the record, and a record identifier within the partition. Why would this be a bad idea?

b. Global secondary indices are implemented in a way similar to local secondary indices that are used when records are stored in a B^+-tree file organization. Explain the similarities between the two scenarios that result in a similar implementation of the secondary indices.

Answer:

a. Any updated such as splitting or moving a partition, which is required to balance load, would require a large number of updates to secondary indices.

b. In both cases records may move (across nodes, or to a different location within the node) which would require a large number of updates to secondary indices if they stored direct pointers. The indirection through the clustering index key / partitioning key allows record movement without any updates to the secondary index.

21.6 Parallel database systems store replicas of each data item (or partition) on more than one node.

a. Why is it a good idea to distribute the copies of the data items allocated to a node across multiple other nodes, instead of storing all the copies in the same node (or set of nodes).

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

Answer:

a. The copies of the data items at a node should be partitioned across multiple other nodes, rather than stored in a single node, for the following reasons:

- To better distribute the work which should have been done by the failed node, among the remaining nodes.
- Even when there is no failure, this technique can to some extent deal with hot-spots created by read-only transactions.

b. RAID level 0 itself stores an extra copy of each data item (mirroring). Thus this is similar to mirroring performed by the database itself, except that the database system does not have to bother about the details of performing the mirroring. It just issues the write to the RAID system, which automatically performs the mirroring.

RAID level 5 is less expensive than mirroring in terms of disk space requirement, but writes are more expensive, and rebuilding a crashed disk is more expensive.

21.7 Partitioning and replication.

a. Explain why range-partitioning gives better control on tablet sizes than hash partitioning. List an analogy between this case and the case of B^+-tree indices versus hash indices.

b. Some systems first perform hashing on the key, and then use range partitioning on the hash values. What could be a motivation for this choice, and what are its drawbacks as compared to performing range partition direction on the key?

c. It is possible to horizontally partition data, and then perform vertical partitioning locally at each node. It is also possible to do the converse, where vertical partitioning is done first, and then each partition is then horizontally partitioned independently. What are are the benefits of the first option over the second one?

Answer:

a. Hash partitioning does not permit any control on individual tablet sizes, unlike range partitioning which allows overfull partitions to be split quite easily. B^+-tree indices use range partitioning, allowing a leaf node to be split if it is overfull. In contrast, it is not easy to split a hash bucket in a hash index if the bucket is overfull.

Some approaches similar to those used for dynamic hashing (such as linear hashing or extendable hashing) have been proposed to allow overfull hash buckets to be split while leaving other hash buckets untouched, but range partitioning provides a simpler solution.

b. Hashing allows keys of various types to be mapped to a single data type, simplifying the job of partitioning the data. The drawback is that range queries cannot be supported using hashing (without performing a full table scan), whereas direct range-partitioning allows efficient support for range queries.

c. The first option allows reconstruction of records at a single node if a query only accesses records at that node. With the second option, the vertical fragments corresponding to one record may potentially be residing on different nodes, requiring extra communication to get the vertical fragments together.

21.8 In order to send a request to the master replica of a data item, a node must know which replica is the master for that data item.

a. Suppose that between the time the node identifies which node is the master replica for a data item, and the time the request reaches the identified node, the mastership has changed, and a different node is now the master. How can such a situation be dealt with?

b. While the master replica could be chosen on a per-partition basis, some systems support a *per-record master replica*, where the records of a partition (or tablet) are replicated at some set of nodes, but each record's master replica can be on any of the nodes from within this set of nodes, independent of the master replica of other records. List two benefits of keeping track of master on a per-record basis.

c. Suggest how to keep track of the master replica for each record, when there are a large number of records.

Answer:

a. If a node receives a request for a data item when it is not the master, it can send an error reply with the reason for the error to the requesting node. The requesting node can then find the current master and resend the request to the current master. Alternatively, the old master can forward the message to the new master, which can reply to the requesting node.

b. Tracking mastership on a per-record basis allows the master to be located in a geographical region where most requests for the data item occur, for example the region where the user resides. Reads can then be satisfied without any communication with other regions, which is generally much slower due to speed-of-light delays. Further, writes can also be done locally, and replicated asynchronously to the other replicas.

c. Each record can have an extra hidden field that stores the master replica of that record. In case the information is outdated, all the replicas of the data item can be accessed to find the nodes listed as masters for that data item; those nodes can be contacted to find the current master.

CHAPTER 22

Parallel and Distributed Query Processing

Practice Exercises

22.1 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, interquery 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, intraquery parallelism is essential to get fast response times. Given that there are large numbers of processors and disks, only intraoperation 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.

22.2 Describe how partial aggregation can be implemented for the **count** and **avg** aggregate functions to reduce data transfer.

Answer:
FILL

22.3 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 speedup 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.

22.4 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 in **??**

Each fragment is replicated on three 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.

22.5 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the query:

$$_{r.C}\gamma_{\text{count}(s.D)}(\;(\sigma_{A>5}(r)) \bowtie_{r.B=s.B} s\;)$$

a. Give a parallel query plan using the exchange operator, for computing the subtree of the query involving only the select and join operators.

b. Now extend the above to compute the aggregate. Make sure to use pre-aggregation to minimize the data transfer.

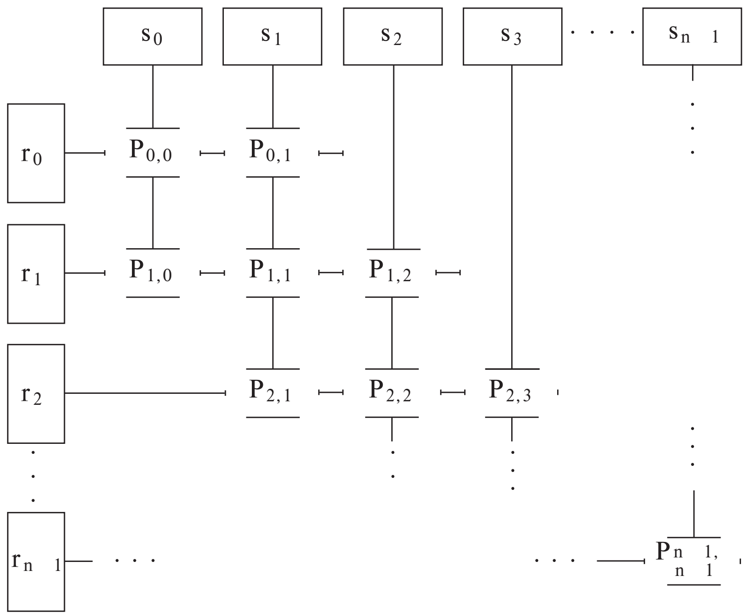


Figure 22.101 The three levels of data abstraction.

c. Skew during aggregation is a serious problem. Explain how pre-aggregation as above can also significantly reduce the effect of skew during aggregation.

Answer:

a. This is a small variant of an example from the chapter.

b. This one is very straightforward, since it is already the example in the chapter

c. Pre-aggregation can greatly reduce the size of the data sent to the final aggregation step. So even if there is skew, the absolute data sizes are smaller, resulting in significant reduction in the impact of the skew.

22.6 Suppose relation r is stored partitioned and indexed on A, and s is stored partitioned and indexed on B. Consider the join $r \bowtie_{r.B=s.B} s$. Suppose s is relatively small, but not small enough to make asymmetric fragment-and-replicate join the best choice, and r is large, with most r tuples not matching any s tuple. A hash-join can be performed but with a semijoin filter used to reduce the data transfer. Explain how semijoin filtering using Bloom filters would work in this parallel join setting.

Answer:

Since s is small, it makes sense to send a Bloom filter on $s.B$ to all partitions of r. Then we use the Bloom filter to find r tuples that may match some s tuple, and repartition the matching r tuples on $r.B$, sending them to the nodes containing s (which is already partitioned on $s.B$). Then the join can be performed at each site storing s tuples. The Bloom filter can significantly reduce the number of r tuples transferred.

Note that repartitioning s does not make sense since it is already partitioned on the join attribute, unlike r.

22.7 Suppose you want to compute $r ⟕_{r.A=s.A} s$.

a. Suppose s is a small relation, while r is stored partitioned on $r.B$. Give an efficient parallel algorithm for computing the left outer join.

b. Now suppose that r is a small relation, and s is a large relation, stored partitioned on attribute $s.B$. Give an efficient parallel algorithm for computing the above left outer join.

Answer:

a. Replicating s to all nodes, and computing the left outerjoin independently at each node would be a good option in this case.

b. The best technique in this case is to replicate r to all nodes, and compute $r \bowtie s_i$ at each node i. Then, we send back the list of r tuples that had matches at site i back to a single node, which takes the union of the returned r tuples from each node i. Tuples in r that are absent in this union are then padded with nulls and added to the output.

22.8 Suppose you want to compute ${}_{A,B}\gamma_{sum(C)}$ on a relation s which is stored partitioned on $s.B$. Explain how you would do it efficiently, minimizing/avoiding repartitioning, if the number of distinct $s.B$ values is large, and the distribution of number of tuples with each $s.B$ value is relatively uniform.

Answer:

The aggregate can be computed locally at each node, with no repartitioning at all, since partitioning on $s.B$ implies partitioning on $s.A, s.B$. To understand why, partitioning on (A, B) requires that tuples with the same value for (A, B) must be in the same partition. Partitioning on just B, ignoring A, also satisfies this requirement.

Of course not partitioning at all also satisfies the requirement, but that defeats the purpose of parallel query processing. As long as the number of distinct $s.B$ values is large enough and the number of tuples with each $s.B$ value are relatively uniform and not highly skewed, using the existing partitioning on $s.B$ will give good performance.

22.9 MapReduce implementations provide fault tolerance, where you can reexecute only failed mappers or reducers. By default, a partitioned parallel join execution would have to be rerun completely in case of even one node failure. It is possible to modify a parallel partitioned join execution to add fault tolerance in a manner similar to MapReduce, so failure of a node does not require full reexecution of the query, but only actions related to that node. Explain what needs to be done at the time of partitioning at the sending node and receiving node to do this.

Answer: This is an application of ideas from MapReduce to join processing. There are two steps: first the data is repartitioned, and then join is performed, corresponding to the map and reduce steps.

A failure during the repartition can be handled by reexecuting the work of the failed node. However, the destination must ensure that tuples are not processed twice. To do so, it can store all received tuples in local disk, and start processing only after all tuples have been received. If the sender fails meanwhile, and a new node takes over, the receivers can discard all tuples received from the failed sender, and receive them again. This part is not too expensive.

Failures during the final join computation can be handled similar to reducer failure, by getting the data again from the partitioners. However, in the MapReduce paradigm tuples to be sent to reducers are stored on disk at the mappers, so they can be resent if required. This can also be done with parallel joins, but there is now a significant extra cost of writing the tuples to disk.

Another option is to find the tuples to be sent to the failed join node by rescanning the input. But now, all partitioners have to reread their entire input, which makes the process very expensive, similar in cost to rerunning the join. As a result this is not viewed as useful.

22.10 If a parallel data-store is used to store two relations r and s and we need to join r and s, it may be useful to maintain the join as a materialized view. What are the benefits and overheads in terms of overall throughput, use of space, and response time to user queries?

Answer:

Performing a join on a cloud data-storage system can be very expensive, if either of the relations to be joined is partitioned on attributes other than the join attributes, since a very large amount of data would need to be transferred to perform the join. However, if $r \bowtie s$ is maintained as a materialized view, it can be updated at a relatively low cost each time each time either r or s is updated, instead of incurring a very large cost when the query is executed. Thus, queries are benefitted at some cost to updates.

With the materialized view, overall throughput will be much better if the join query is executed reasonably often relative to updates, but may be worse if the join is rarely used, but updates are frequent.

The materialized view will certainly require extra space, but given that disk capacities are very high relative to IO (seek) operations and transfer rates, the extra space is likely to not be an major overhead.

The materialized view will obviously be very useful to evaluate join queries, reducing time greatly by reducing data transfer across machines.

22.11 Explain how each of the following join algorithms can be implemented using the MapReduce framework:

a. Broadcast join (also known as asymmetric fragment-and-replicate join).

b. Indexed nested loop join, where the inner relation is stored in a parallel data-store.

c. Partitioned join.

Answer:
FILL

CHAPTER 23

Parallel and Distributed Transaction Processing

Practice Exercises

23.1 What are the key differences between a local-area network and a wide-area network, that affect the design of a distributed database?

Answer:
Data transfer is much faster, and communication latency is much lower on a local-area network (LAN) than on a wide-area network (WAN). Protocols that require multiple rounds of communication maybe acceptable in a local area network, but distributed databases designed for wide-area networks try to minimize the number of such rounds of communication.
Replication to a local node for reducing latency is quite important in a wide-area network, but less so in a local area network.
Network link failure and network partition are also more likely in a wide-area network than in a local area network, where systems can be designed with more redundancy to deal with failures. Protocols designed for wide-area networks should handle such failures without creating any inconsistencies in the database.

23.2 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.

23.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Exercise 23.2a, explain how 2PC ensures transaction atomicity despite the failure.

Answer:
A proof that 2PC guarantees atomic commits/aborts in spite of site and link failures follows. The main idea is that after all sites reply with a <**ready** *T*> message, only the coordinator 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 coordinator's decision, either directly from the coordinator 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 coordinator 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 coordinator.

   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 coordinator's decision (either directly or indirectly) before actually aborting.

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

b. A site can commit a transaction *T* (by writing a <**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 coordinator's decision (either directly or indirectly) before actually committing.

ii. It is itself the coordinator. In this case no other participating site can abort or would have aborted because abort decisions are made only by the coordinator.

23.4 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 it must be such that *B* can re-integrate consistently into the distributed system once communication is restored.

23.5 The persistent messaging scheme described in this chapter depends on timestamps. A drawback is that they can discard received messages only if they are too old, and may need to keep track of a large number of received messages. Suggest an alternative scheme based on sequence numbers instead of timestamps, that can discard messages more rapidly.

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.

23.6 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 entire database 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. Different data items may be replicated at different nodes.
In a distributed system, transaction code can run at all the sites, whereas in a remote backup system it runs only at the primary site. The distributed system transactions needs to follow two-phase commit or other consensus protocols to keep the data in consistent state, whereas a remote backup system does not follow two-phase commit and avoids related overhead.

23.7 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 if data were read from a node other than the master.

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.

23.8 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 edgem (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 implies 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 be detected.

23.9 Consider the chain-replication protocol, described in Section 23.4.3.2, which is a variant of the primary-copy protocol.

a. If locking is used for concurrency control, what is the earliest point when a process can release an exclusive lock after updating a data item?

b. While each data item could have its own chain, give two reasons it would be preferable to have a chain defined at a higher level, such as for each partition or tablet.

c. How can consensus protocols be used to ensure that the chain is uniquely determined at any point in time?

Answer:

a. The lock can be released only after the update has been recorded at the tail of the chain, since further reads will read the tail. Two phase locking may also have to be respected.

b. The overhead of recording chains per data item would be high. Even more so, in case of failures, chains have to be updated, which would have an even greater overhead if done per item.

c. All nodes in the chain have to agree on the chain membership and order. Consensus can be used to ensure that updates to the chain are done in a fault-tolerant manner. A fault-tolerant coordination service such as ZooKeeper or Chubby could be used to ensure this consensus, by updating metadata that is replicated using consensus; the coordination service hides the details of consensus, and allows storage and update of (a limited amount of) metadata.

23.10 If the primary copy scheme is used for replication, and the primary gets disconnected from the rest of the system, a new node may get elected as primary. But the old primary may not realize it has got disconnected, and may get reconnected subsequently without realizing that there is a new primary.

a. What problems can arise if the old primary does not realize that a new one has taken over?

b. How can leases be used to avoid these problems?

c. Would such a situation, where a participant node gets disconnected and then reconnected without realizing it was disconnected, cause any problem with the majority or quorum protocols?

Answer:

a. The old primary may receive read requests and reply to them, serving old data that is missing subsequent updates.

b. Leases can be used so that at the end of the lease, the primary knows that it if it did not successfuly renew the lease, it should stop serving requests. If it is disconnected, it would be unable to renew the lease.

c. This situation would not cause a problem with the majority protocol since the write set (or write quorum) and the read set (read quorum) must have at least one node in common, which would serve the latest value.

23.11 Consider a federated database 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 federated database 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 coordinator and requiring that the lock be requested on the data item which resides on the currently elected coordinator.

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 nonserializable.

23.12 Consider a federated database 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 23.11 is assumed, except that now both the global transactions are read-only. Consider the following schedule:

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 systemwide 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 systemwide precedence graph is eliminated.

23.13 Suppose you have a large relation $r(A, B, C)$ and a materialized view $v = {}_A\gamma_{\text{sum}(B)}(r)$. View maintenance can be performed as part of each transaction that updates r, on a parallel/distributed storage system that supports transactions across multiple nodes. Suppose the system uses two-phase commit along with a consensus protocol such as Paxos, across geographically distributed data centers.

a. Explain why it is not a good idea to perform view maintenance as part of the update transaction, if some values of attribute A are "hot" at certain points in time, that is, many updates pertain to those values of A.

b. Explain how operation locking (if supported) could solve this problem.

c. Explain the tradeoffs of using asynchronous view maintenance in this context.

Answer:

This is a very bad idea from the viewpoint of throughput. Most transactions would now update a few aggregate records, and updates would get serialized on the lock. The problem that due to Paxos delays plus 2PC delays, commit processing will take a long time (hundreds of milliseconds) and there would be very high contention on the lock. Transaction throughput would decrease to tens of transactions per second, even if transactions do not conflict on any other items.

If the storage system supported operation locking, that could be an alternative to improve concurrency, since view maintenance can be done using operation locks that do not conflict with each other. Transaction throughput would be greatly increased.

Asynchronous view maintenance would avoid the bottleneck and lead to much better throughput, but at the risk of reads of the view seeing stale data.