INSTRUCTOR'S MANUAL
TO ACCOMPANY

Database System Concepts

Seventh Edition

Abraham Silberschatz
Yale University

Henry F. Korth
Lehigh University

S. Sudarshan
Indian Institute of Technology, Bombay

April 1, 2019

Preface

This volume is an instructor's manual for the Seventh Edition of *Database System Concepts* by Abraham Silberschatz, Hank Korth, and S. Sudarshan. It consists of answers to the Exercises in the parent text.

Although we have tried to produce an instructor's manual that will aid all of the users of our book as much as possible, there can always be improvements (improved answers, additional questions, sample test questions, programming projects, alternative orders of presentation of the material, additional references, and so on). We invite you to help us in improving this manual. If you have better solutions to the exercises or other items that would be of use with *Database System Concepts*, we invite you to send them to us for consideration in later editions of this manual. All contributions will, of course, be properly credited to their contributor. Email should be addressed to db-book-authors@cs.yale.edu.

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

Contents

CHAPTER 1

Introduction

Exercises

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

Answer:

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

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

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

- Both systems contain a collection of data and a set of programs which access the data. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.
- A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, whereas data written by one program in a file-processing system may not be readable by another program.

- A database management system is designed to allow flexible access to data (i.e., queries), whereas a file-processing system is designed to allow predetermined access to data (i.e., compiled programs).
- A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file.

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

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

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

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

a. interaction with the file manager
b. integrity enforcement
c. security enforcement
d. backup and recovery
e. concurrency control

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

a. No DBMS can do without this. If there is no file manager interaction then nothing stored in the files can be retrieved.
b. Consistency constraints may not be satisfied. For example, an instructor may belong to a nonexistent department, two students may have the same ID, account balances could go below the minimum allowed, and so on.

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

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

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

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

Answer:

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

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

1.11 Assume that two students are trying to register for a course in which there is only one open seat. What component of a database system prevents both students from being given that last seat?

Answer:
The concurrency-control manager, which is part of the transaction manager, ensures that at most one student will register successfully.

1.12 Explain the difference between two-tier and three-tier application architectures. Which is better suited for web applications? Why?

Answer:
In a two-tier application architecture, the application runs on the client machine and directly communicates with the database system running on the server. In contrast, in a three-tier architecture, application code running on the client's machine communicates with an application server at the server, and it never directly communicates with the database. The three-tier archicture is better suited for web applications.

1.13 List two features developed in the 2000s and that help database systems handle data-analytics workloads.

Answer:
Traditional database systems store data row-by-row. Because data analytics often focus on only a few columns of a table, column-stores were introduced to allow faster retrieval of those columns actually being used.

Because of the high processing demands of data analytics combined with the broader availability of parallel processing, the map-reduce framework was introduced to facilitate coding parallel data-analytics applications.

1.14 Explain why NoSQL systems emerged in the 2000s, and briefly contrast their features with traditional database systems.

Answer:
NoSQL systems relax the rigidity of storing data in tables by allowing a diverse set of data types. They allow for faster initial application development. However, NoSQL systems lack traditional systems' support for strong data consistency, instead relying on a weaker concept of eventual consistency.

1.15 Describe at least three tables that might be used to store information in a social-networking system such as Facebook.

Answer:
Some possible tables are:

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

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

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

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

CHAPTER 2

Introduction to the Relational Model

The relational model remains the primary data model for commercial data-processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. It has retained this position by incorporating various new features and capabilities over its half-century of existence. Among those additions are object-relational features such as complex data types and stored procedures, support for XML data, and various tools to support semi-structured data. The relational model's independence from any specific underlying low-level data structures has allowed it to persist despite the advent of new approaches to data storage, including modern column-stores that are designed for large-scale data mining.

In this chapter, we first study the fundamentals of the relational model. A substantial theory exists for relational databases. In Chapter 6 and Chapter 7, we shall examine aspects of database theory that help in the design of relational database schemas, while in Chapter 15 and Chapter 16 we discuss aspects of the theory dealing with efficient processing of queries. In Chapter 27, we study aspects of formal relational languages beyond our basic coverage in this chapter (Section 2.6).

Exercises

2.10 Describe the differences in meaning between the terms *relation* and *relation schema*.

Answer:

A relation schema is a type definition, and a relation is an instance of that schema. For example, *student* (*ss#, name*) is a relation schema and

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

is a relation based on that schema.

2.11 Consider the *advisor* relation shown in the schema diagram in Figure 2.9, with *s_id* as the primary key of *advisor*. Suppose a student can have more than one advisor. Then, would *s_id* still be a primary key of the *advisor* relation? If not, what should the primary key of *advisor* be?

Answer:

No, *s_id* would not be a primary key, since there may be two (or more) tuples for a single student, corresponding to two (or more) advisors. The primary key should then consist of the two attributes *s_id, i_id*.

2.12 Consider the bank database of Figure 2.18. Assume that branch names and customer names uniquely identify branches and customers, but loans and accounts can be associated with more than one customer.

a. What are the appropriate primary keys?

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

Answer:

a. The primary keys of the various schemas are underlined. We allow customers to have more than one account, and more than one loan.

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*)

b. The foreign keys are as follows:

i. For *loan*: *branch_name* referencing *branch*.

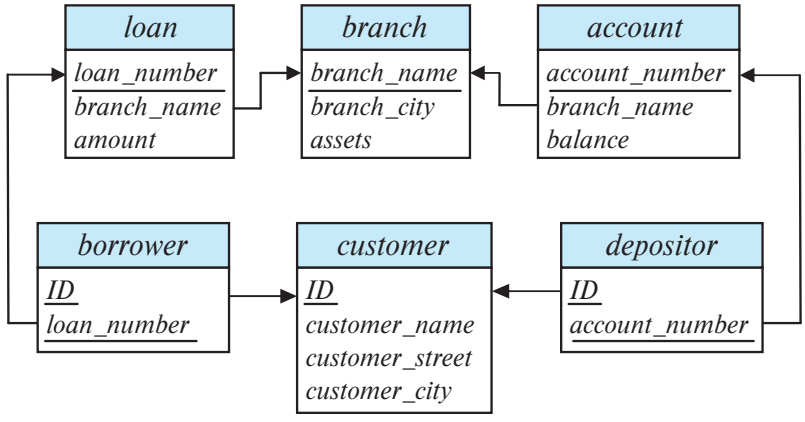
ii. For *borrower*: Attribute *ID* referencing *customer* and *loan_number* referencing *loan*

iii. For *account*: *branch_name* referencing *branch*.

iv. For *depositor*: Attribute *ID* referencing *customer* and *account_number* referencing *account*

2.13 Construct a schema diagram for the bank database of Figure 2.18.

Answer:



2.14 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 works for "BigBank".

b. Find the ID, name, and city of residence of each employee who works for "BigBank".

c. Find the ID, name, street address, and city of residence of each employee who works for "BigBank" and earns more than $10000.

d. Find the ID and name of each employee in this database who lives in the same city as the company for which she or he works.

Answer:

a. $\Pi_{ID,\ person\_name}\ (\sigma_{company\_name\ =\ \text{"BigBank"}}\ (works))$

b. $\Pi_{ID,\ person\_name,\ city}\ (employee\ \bowtie_{employee.id=works.id}\ (\sigma_{company\_name\ =\ \text{"BigBank"}}\ (works)))$

c. $\Pi_{ID,\ person\_name,\ street,\ city}\ (\sigma_{(company\_name\ =\ \text{"BigBank"}\ \wedge\ salary\ >\ 10000)}\ (works \bowtie_{employee.id=works.id}\ employee))$

d. $\Pi_{ID,person\_name}\ (\sigma_{employee.city=company.city}(employee \bowtie_{employee.ID=works.ID}\ works \bowtie_{works.company\_name=company.company\_name}\ company))$

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

a. Find each loan number with a loan amount greater than $10000.

b. Find the ID of each depositor who has an account with a balance greater than $6000.

c. Find the ID of each depositor who has an account with a balance greater than $6000 at the "Uptown" branch.

Answer:

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

b. $\Pi_{ID}\,(\sigma_{balance > 6000}\,(depositor \bowtie_{depositor.account\_number = account.account\_number} account))$

c. $\Pi_{ID}\,(\sigma_{balance > 6000 \wedge branch\_name = \text{"Uptown"}}\,(depositor \bowtie_{depositor.account\_number = account.account\_number} account))$

2.16 List two reasons why null values might be introduced into a database.

Answer:
Nulls may be introduced into the database because the actual value is either unknown or does not exist. For example, an employee whose address has changed and whose new address is not yet known should be retained with a null address.

2.17 Discuss the relative merits of imperative, functional, and declarative languages.

Answer:
Declarative languages greatly simplify the specification of queries (at least, the types of queries they are designed to handle). They free the user from having to worry about how the query is to be evaluated; not only does this reduce programming effort, but in fact in most situations the query optimizer can do a much better job of choosing the best way to evaluate a query than a programmer working by trial and error.
Both functional and imperative languages require the programmer to specify a specific set of actions. Functional languages have no side-effects, that is, there is no program state to update. This avoids bugs that result from unantipated side effects. Functional languages permit the use of algebraic equivalences in query optimization. The step-by-step execution plan for actually running a database query is usually expressed in an imperative style. Imperative programming is the style most familiar to most programmers.

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

a. Find the ID and name of each instructor in the Physics department.

b. Find the ID and name of each instructor in a department located in the building "Watson".

c. Find the ID and name of each student who has taken at least one course in the "Comp. Sci." department.

d. Find the ID and name of each student who has taken at least one course section in the year 2018.

e. Find the ID and name of each student who has not taken any course section in the year 2018.

Answer:

a. $\Pi_{ID,name}(\sigma_{dept\_name=\}\}Physics''}(instructor))$

b. $\Pi_{ID,name}(instructor \bowtie_{instructor.dept\_name=department.dept\_name} (\sigma_{building=\}\}Watson''}(department)))$

c. $\Pi_{ID,name}(student \bowtie_{student.ID=takes.ID} takes \bowtie_{takes.course\_id=course.course\_id} \sigma_{dept\_name=\}\}Comp.\ Sci.''}(course))$

d. $\Pi_{ID,name}(student \bowtie_{student.ID=takes.ID} \sigma_{year=2018}(takes))$

e. $\Pi_{ID,name}(student) - \Pi_{ID,name}(student \bowtie_{student.ID=takes.ID} (\sigma_{year=2018}(takes)))$

CHAPTER 3

Introduction to SQL

Exercises

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

a. Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

b. Find the ID and name of each student who has not taken any course offered before 2017.

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

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

Answer:

a. Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

```
select  student.ID, name
from    student, takes, course
where   course.dept_name = 'Comp. Sci.'
        and student.ID = takes.ID
        and course.course_id = takes.course_id
```

b. Find the ID and name of each student who has not taken any course offered before 2017.

```
select distinct ID, name
from student
where ID not in (
        select ID
        from takes
        where year < 2017)
```

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

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

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

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

3.12 Write the SQL statements using the university schema to perform the following operations:

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

b. Create a section of this course in Fall 2017, with *sec_id* of 1, and with the location of this section not yet specified.

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

d. Delete enrollments in the above section where the student's ID is 12345.

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

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

Answer:

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

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

b. Create a section of this course in Fall 2017, with *sec_id* of 1, and with the location of this section not yet specified.

```
insert into section
    values ('CS-001', 1, 'Fall', 2017, null, null, null)
```

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

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

```
insert into takes
    select ID, 'CS-001', 1, 'Fall', 2017, null
    from student
    where dept_name = 'Comp. Sci.'
```

d. Delete enrollments in the above section where the student's ID is 12345.

```
delete from takes
where course_id= 'CS-001' and sec_id = 1 and
    year = 2017 and semester = 'Fall' and
    ID = 12345.
```

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

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

delete from section
where course_id = 'CS-001'

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

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

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

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

3.13 Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

Answer:

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


create table car
        (license_plate varchar(50),
        model         varchar(50),
        year          integer,
        primary key (license_plate));


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

```
create table owns
        (driver_id varchar(50),
        license_plate varchar(50),
        primary key (driver_id, license_plate),
        foreign key (driver_id) references person,
        foreign key (license_plate) references car);
```

```
create table participated
        (report_number   integer,
        license_plate    varchar(50),
        driver_id        varchar(50),
        damage_amount    integer,
        primary key (report_number, license_plate),
        foreign key (license_plate) references car,
        foreign key (report_number) references accident)),
        foreign key (driver_id) references person;
```

3.14 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 number of accidents involving a car belonging to a person named "John Smith".

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

Answer:

a. Find the number of accidents involving a car belonging to a person named "John Smith".

```
select count (distinct report_number)
 from participated, owns, person
 where owns.driver_id = person.driver_id
        and person.name = 'John Smith'
        and owns.license_plate = participated.license_plate
```

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

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

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

a. Find each customer who has an account at *every* branch located in "Brooklyn".

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

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

Answer:

a. Find each customer who has an account at *every* branch located in "Brooklyn".

```
select ID, customer_name
from customer as c
where   (select count(*)
         from branch
         where branch_city = 'Brooklyn')
         =
         (select count(distinct branch.branch_name)
         from customer, depositor, account, branch
         where depositor.ID = customer.ID
         and depositor.account_number = account.account_number
         and  account.branch_name = branch.branch_name
         and branch_city = 'Brooklyn')
```

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

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

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

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

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

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

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

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

d. Find the company that has the smallest payroll.

Answer:

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

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

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

```
select e.ID, e.person_name
from employee as e, employee as m, manages
where e.ID = manages.ID
      and m.ID = manages.manager_id
      and e.street = m.street and e.city = m.city
```

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

```
select E.ID, person_name
from works as T, employee as E
where E.ID = T.ID
and salary > (select avg (salary)
              from works as S
              where T.company_name = S.company_name)
```

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

d. Find the company that has the smallest payroll.

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

3.17 Consider the employee database of Figure 3.19. Give an expression in SQL for each of the following queries.

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

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

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

Answer:

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

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

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

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

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

```
create table      employee
(ID               numeric (5,0),
 person_name      varchar(20),
 street           char(30),
 city             varchar(20),
 primary key      (ID));

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

create table      works
(ID               numeric (5,0),
 company_name     varchar(20),
 salary           numeric(8, 2),
 primary key      (ID));
 foreign key (ID) references employee
 foreign key (company_name) references company

create table      manages
(ID               numeric (5,0),
 manager_id       numeric (5,0),
 primary key      (ID)
 foreign key (ID) references employee
 foreign key (manager_id) references employee(ID)
);
```

Figure 3.101 Schema definition for the employee database of Exercise 3.18.

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

3.18 Give an SQL schema definition for the employee database of Figure 3.19. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema. Include any foreign-key constraints that might be appropriate.

Answer:

Please look at Figure 3.101.

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

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

Figure 3.20 Library database.

Answer:

- The null value signifies an unknown value.
- The null value signifies that a value does not exist.

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

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

3.21 Consider the library database of Figure 3.20. Write the following queries in SQL.

a. Find the member number and name of each member who has borrowed at least one book published by "McGraw-Hill".

b. Find the member number and name of each member who has borrowed every book published by "McGraw-Hill".

c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.

d. Find the average number of books borrowed per member. Take into account that if a member does not borrow any books, then that member does not appear in the *borrowed* relation at all, but that member still counts in the average.

Answer:

a. Find the member number and names of each member who has borrowed at least one book published by "McGraw-Hill".

```
select distinct m.member_no, name
from member as m, book as b, borrowed as l
where m.memb_no = l.memb_no
      and l.isbn = b.isbn and
          b.publisher = 'McGraw-Hill'
```

b. Find the member number and name of each member who has borrowed every book published by "McGraw-Hill".

```
select distinct member_no.name
from member
where not exists
      ((select isbn
      from book
      where publisher = 'McGraw-Hill')
      except
      (select isbn
      from borrowed
      where borrowed.memb_no = member.memb_no))
```

c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.

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

The above query could be written using the **having** clause. In that case the **where** clause would be removed and the inner subquery would add a **having** clause after the **group by** clause as follows:
having count(*isbn*) > 5

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

```
select (select count(*) from member) / (select count(*)/memcount
from borrowed)
from dual
```

Note that the above query ensures that members who have not borrowed any books are also counted. In some systems it is required not to have the "**from dual" clause, while others require it and use "dual" to represent a dummy relation that serves as a placeholder. SQL Server is an example of the former and Oracle an example of the latter.**

3.22 Rewrite the **where** clause

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

without using the **unique** construct.

Answer:

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

3.23 Consider the query:

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

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

Answer:

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

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

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

3.24 Using the university schema, write an SQL query to find the name and ID of those Accounting students advised by an instructor in the Physics department.

Answer:

```
select student.ID, student.name
from student, advisor, instructor
where student.ID = s_ID
        and instructor.ID= i_ID
        and student.dept_name= 'Accounting'
        and instructor.dept_name= 'Physics'
```

3.25 Using the university schema, write an SQL query to find the names of those departments whose budget is higher than that of Philosophy. List them in alphabetic order.

Answer:

```
select X.dept_name
from department as X, department as H
where H.dept_name = 'Philosophy' and X.budget > H.budget
order by X.dept_name
```

3.26 Using the university schema, use SQL to do the following: For each student who has retaken a course at least twice (i.e., the student has taken the course at least three times), show the course ID and the student's ID.
Please display your results in order of course ID and do not display duplicate rows.

Answer:

```
select distinct course_id, ID
from takes
group by ID, course_id
having count(*) > 2
order by course_id;
```

3.27 Using the university schema, write an SQL query to find the IDs of those students who have retaken at least three distinct courses at least once (i.e, the student has taken the course at least two times).

Answer:

```
select distinct, ID
from (
        select course_id, ID
        from takes
        group by ID, course_id
        having count(*) > 1)
group by ID
having count(course_id) > 2
```

3.28 Using the university schema, write an SQL query to find the names and IDs of those instructors who teach every course taught in his or her department (i.e., every course that appears in the *course* relation with the instructor's department name). Order result by name.

Answer:

```
select name, ID
from instructor as I
where not exists (
        select *
        from course as C
        where C.dept_name = I.dept_name
              and not exists (
                        select *
                        from teaches as T
                        where T.ID = I.ID and T.course_id = C.course_id
                        )
        )
order by name
```

3.29 Using the university schema, write an SQL query to find the name and ID of each History student whose name begins with the letter 'D' and who has *not* taken at least five Music courses.

Answer:

```
select name, ID
from student
where dept_name = 'History' and name like 'D%' and 5 <= (
          select count(distinct course_id)
          from course
          where course.dept_name= 'Music' and not exists (
                    select *
                    from takes
                    where takes.ID = student.ID
                          and takes.course_id= course.course_id
                    )
          )
```

3.30 Consider the following SQL query on the university schema:

```
select avg(salary) - (sum(salary) / count(*))
from instructor
```

We might expect that the result of this query is zero since the average of a set of numbers is defined to be the sum of the numbers divided by the number of numbers. Indeed this is true for the example *instructor* relation in Figure 2.1. However, there are other possible instances of that relation for which the result would *not* be zero. Give one such instance, and explain why the result would not be zero.

Answer:
Suppose we take the relation in Figure 2.1 and add one more tuple:

('99999', Newbie, Music, *null*)

This adds one to the count of tuples in *instructor* but does not alter the sum of the salaries since **sum** ignores the null value. As a result, **sum**(*salary*)/**count**(∗) is now smaller than before. However, **avg** does not include null values in computing the average, and so the result of **avg**(*salary*) remains the same as before.

3.31 Using the university schema, write an SQL query to find the ID and name of each instructor who has never given an A grade in any course she or he has taught. (Instructors who have never taught a course trivially satisfy this condition.)

Answer:

```
select ID, name
from instructor
except
select distinct instructor.ID, instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
        and teaches.course_id = takes.course_id
        and teaches.year = takes.year
        and teaches.semester= takes.semester
        and teaches.sec_id= takes.sec_id
        and takes.grade = 'A';
```

3.32 Rewrite the preceding query, but also ensure that you include only instructors who have given at least one other non-null grade in some course.

Answer:

```
select distinct(instructor.ID), instructor.name
from instructor, teaches, takes
where instructor.ID=teaches.ID
        and teaches.course_id=takes.course_id
        and teaches.year = takes.year
        and teaches.semester = takes.semester
        and teaches.sec_id = takes.sec_id
        and takes.grade is not null
except
select distinct instructor.ID, instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
        and teaches.course_id = takes.course_id
        and teaches.year = takes.year
        and teaches.semester = takes.semester
        and teaches.sec_id = takes.sec_id
        and takes.grade= 'A';
```

Or, without using **except**:

```
select distinct instructor.ID, instructor.name
from instructor, teaches, takes
where instructor.ID = teaches.ID
        and teaches.course_id = takes.course_id
        and teaches.year = takes.year
        and teaches.semester = takes.semester
        and teaches.sec_id = takes.sec_id
        and takes.grade is not null
        and instructor.ID not in (
                select instructor.ID
                from instructor, teaches, takes
                where instructor.ID = teaches.ID
                        and teaches.course_id = takes.course_id
                        and teaches.year = takes.year
                        and teaches.semester = takes.semester
                        and teaches.sec_id = takes.sec_id
                        and takes.grade = 'A'
        )
```

3.33 Using the university schema, write an SQL query to find the ID and title of each course in Comp. Sci. that has had at least one section with afternoon hours (i.e., ends at or after 12:00). (You should eliminate duplicates if any.)

Answer:

```
select distinct course.course_id, course.title
from course, section, time_slot
where course.course_id = section.course_id
        and section.time_slot_id = time_slot.time_slot_id
        and time_slot.end_time >= 12
        and course.dept_name = 'Comp. Sci.';
```

3.34 Using the university schema, write an SQL query to find the number of students in each section. The result columns should appear in the order "courseid, secid, year, semester, num". You do not need to output sections with 0 students.

Answer:

```
select course_id, sec_id, year, semester, count(*) as num
from takes
group by course_id, sec_id, year, semester
```

3.35 Using the university schema, write an SQL query to find section(s) with maximum enrollment. The result columns should appear in the order "courseid, secid, year, semester, num". (It may be convenient to use the *with* construct.)

Answer:

```
with enrollment as (
        select course_id, sec_id, year, semester, count(*) as num
        from takes
        group bycourse_id, sec_id, year, semester
    ),
    maxnumber as (
        select max(num) as maxnum
        from enrollment
    )
select course_id, sec_id, year, semester, num
from enrollment, maxnumber
where num = maxnum
```

CHAPTER 4

Intermediate SQL

Exercises

4.12 Consider the query

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Explain why appending **natural join** *section* in the **from** clause would not change the result.

Answer:
Adding a natural join with *section* would remove from the result each tuple in *takes* whose values for (*course_id*, *semester*, *year*, *sec_id*) do not appear in *section*. However, since *takes* has the constraint:

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

there cannot be a tuple in *takes* whose values for (*course_id*, *semester*, *year*, *sec_id*) do not appear in *section*.

4.13 Rewrite the query

```
select *
from section natural join classroom
```

without using a natural join but instead using an inner join with a **using** condition.

Answer:

```
select *
from section join classroom using (building, room_number)
```

4.14 Write an SQL query using the university schema to find the ID of each student who has never taken a course at the university. Do this using no subqueries and no set operations (use an outer join).

Answer:

```
select ID
from student left outer join takes using (ID)
where course_id is null
```

4.15 Express the following query in SQL using no subqueries and no set operations.

```
select ID
from student
except
select s_id
from advisor
where i_ID is not null
```

Answer:

```
select ID
from student left outer join advisor on ID= s_id
where i_ID is null
```

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

Answer:

a. Query:

```
select ID
from employee natural left outer join manages
where manager_id is null
```

b. Query:

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

4.17 Under what circumstances would the query

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

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

Answer:
There are two cases for which the *title* attribute is null:

- Since *course_id* is a foreign key in the *takes* table referencing the *course* table, the title attribute in any tuple obtained from the above query can be null if there is a course in *course* table that has a null title.
- If a student has not taken any course, as it is a **natural full outer join**, such a student's entry would appear in the result with a **null** *title* entry.

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

Answer:

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

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

4.19 For the view of Exercise 4.18, explain why the database system would not allow a tuple to be inserted into the database through this view.

Answer:
For any such tuple, for example (2017, 1000), there may be many ways to insert *takes* tuples to create a sum of 1000 and no way to choose any particular one. Furthermore, if the sum is already more than 1000 for year 2017, then there is no insertion possible because courses with negative credits are not allowed.

4.20 Show how to express the **coalesce** function using the **case** construct.

Answer:

```
select
        case
                when (A1 is not null) then A1
                when (A2 is not null) then A2
                .
                .
                .
                when (An is not null) then An
                else null
        end
from A
```

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

Answer:

Consider the case where the authorization is provided by the user Satoshi and not the manager role. If we revoke the authorization from Satoshi, for example because Satoshi left the company, all authorizations that Satoshi had granted would also be revoked, even if the grant was to an employee whose job has not changed.

If the grant is done by the manager role, revoking authorizations from Satoshi will not result in such cascading revocation.

In terms of the authorization graph, we can treat Satoshi and the role manager as nodes. When the grant is from the manager role, revoking the manager role from Satoshi has no effect on the grants from the manager role.

4.22 Suppose user A, who has all authorization privileges on a relation r, grants **select** on relation r to **public** with grant option. Suppose user B then grants **select** on r to A. Does this cause a cycle in the authorization graph? Explain why.

Answer:

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

4.23 Suppose a user creates a new relation $r1$ with a foreign key referencing another relation $r2$. What authorization privilege does the user need on $r2$? Why should this not simply be allowed without any such authorization?

Answer:
The user needs the **references** privilege on $r2$. The reason for this is so that once a foreign key references a relation, that creates restrictions on when a tuple can be updated or deleted from that relation. Thus the foreign key constraint that the user is creating on $r1$ actually constrains $r2$ also.

4.24 Explain the difference between integrity constraints and authorization constraints.

Answer:
Integrity constraints protect the database from data inconsistency. Authorization constraints protect the database from unauthorized access but offer no protection against any inconsistencies that might be introduced by an authorized user.

CHAPTER 5

Advanced SQL

Exercises

5.12 Write a Java program that allows university administrators to print the teaching record of an instructor.

a. Start by having the user input the login *ID* and password; then open the proper connection.

b. The user is asked next for a search substring and the system returns (ID, *name*) pairs of instructors whose names match the substring. Use the **like** ('%substring%') construct in SQL to do this. If the search comes back empty, allow continued searches until there is a nonempty result.

c. Then the user is asked to enter an ID number, which is a number between 0 and 99999. Once a valid number is entered, check if an instructor with that ID exists. If there is no instructor with the given ID, print a reasonable message and quit.

d. If the instructor has taught no courses, print a message saying that. Otherwise print the teaching record for the instructor, showing the department name, course identifier, course title, section number, semester, year, and total enrollment (and sort those by *dept_name, course_id, year, semester*).

Test carefully for bad input. Make sure your SQL queries won't throw an exception. At login, exceptions may occur since the user might type a bad password, but catch those exceptions and allow the user to try again.

Answer:
FILL

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

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

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

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

Answer:

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

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

c. The method, in XML format, is shown in Figure 5.101; other output formats are acceptable.

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

Answer:

a. Same as for JDBC.

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

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

5.15 Consider an employee database with two relations

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

where the primary keys are underlined. Write a function *avg_salary* that takes a company name as an argument and finds the average salary of employees at that company. Then, write an SQL statement, using that function, to find companies whose employees earn a higher salary, on average, than the average salary at "First Bank".

Answer:
Please see Figure 5.102.

5.16 Consider the relational schema

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

Figure 5.101 XML progrem for Exercise 5.13.

$$part(\underline{part\_id}, name, cost)$$
$$subpart(\underline{part\_id}, \underline{subpart\_id}, count)$$

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

Answer:

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

    select *
        from total_part
```

5.17 Consider the relational schema from Exercise 5.16. Write a JDBC function using nonrecursive SQL to find the total cost of part "P-100", including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.

Answer:
FILL

5.18 Redo Exercise 5.12 using the language of your database system for coding stored procedures and functions. Note that you are likely to have to consult the online

---

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

select company_name
from works
where avg_salary(company_name) > avg_salary("First Bank")
```

Figure 5.102 Function for Exercise 5.15.

documentation for your system as a reference, since most systems use syntax differing from the SQL standard version followed in the text. Specifically, write a prodedure that takes an instructor *ID* as an argument and produces printed output in the format specified in Exercise 5.12, or an appropriate message if the instructor does not exist or has taught no courses. (For a simpler version of this exercise, rather than providing printed output, assume a relation with the appropriate schema and insert your answer there without worrying about testing for erroneous argument values.)

Answer: *This solution is in Oracle's PL/SQL.*
Please see Figure 5.103.

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

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

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

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

In general, it is extremely difficult for the system statically to identify and prohibit such triggers. Hence database systems, at runtime, may put a limit on the depth of nested trigger calls. There are also limits on which tables a trigger may modify. For example, the system may disallow a trigger from modifying a table from which it read.

5.21 Modify the recursive query in Figure 5.16 to define a relation

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

where the attribute *depth* indicates how many levels of intermediate prerequisites there are between the course and the prerequisite. Direct prerequisites have a depth of 0. Note that a prerequisite course may have multiple depths and thus may appear more than once.

Answer:

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

select *
from prereq_depth
```

5.22 Given relation $s(a, b, c)$, write an SQL statement to generate a histogram showing the sum of c values versus a, dividing a into 20 equal-sized partitions (i.e., where each partition contains 5 percent of the tuples in s, sorted by a).

Answer:

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

5.23 Consider the *nyse* relation of Exercise 5.9. For each month of each year, show the total monthly dollar volume and the average monthly dollar volume for that month and the two prior months. (*Hint*: First write a query to find the total dollar volume for each month of each year. Once that is right, put that in the from clause of the outer query that solves the full problem. That outer query will need windowing. The subquery does not.)

Answer:

```
select year, month,
        sum(DV) over (order by year, month rows 2 preceding) as totaldv,
        avg(DV) over (order by year, month rows 2 preceding) as avgdv
from (
        select year, month, sum(dollar_volume) as DV
        from nyse group by year, month)
```

5.24 Consider the relation, r, shown in Figure 5.22. Give the result of the following query:

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

Answer:

Garfield	359	A	1
Garfield	359	B	1
Garfield	359	*null*	2
Garfield	*null*	*null*	2
Painter	705	D	2
Painter	705	*null*	2
Painter	*null*	*null*	2
Saucon	550	C	1
Saucon	550	*null*	1
Saucon	651	A	1
Saucon	651	*null*	1
Saucon	*null*	*null*	2

```
create or replace procedure teaching (arg_id in instructor.id%type)
as
        dept course.dept_name%type;
        crs course.course_id%type;
        secid section.sec_id%type;
        title course.title%type;
        yr section.year%type;
        sem section.semester%type;
        i integer;
        cursor c (iid in instructor.id%type) is
        select dept_name, course_id, title, sec_id, year,
                semester,count(*) as enrollment
        from takes join
            (select dept_name, course_id, sec_id, title, year, semester from course
             natural join section natural join teaches
             where id = iid) T using (course_id, sec_id, semester, year)
        group by dept_name, course_id, title, sec_id, year, semester
        order by dept_name, course_id, year, semester desc;
        begin
            select count(*) into i from instructor where id = arg_id;
            if i = 0 then
                dbms_output.put_line ('Instructor ' || arg_id || ' does not exist');
                return;
            end if;
            open c (arg_id);
            loop
                fetch c into dept, crs, title, secid, yr, sem, i;
                exit when c%notfound;
                dbms_output.put_line(rpad(dept,20,' ') || rpad(crs,5,' ') ||
                    rpad(title,30,' ') || rpad (secid,4,' ') ||rpad(sem,10,' ')||
                    rpad(yr,7,' ') || i);
            end loop;
            if c%rowcount = 0 then
                dbms_output.put_line('Instructor ' || arg_id ||
                    ' has not taught any courses');
            end if;
        end;
```

Figure 5.103 Java program for Exercise 5.18.

building	*room_number*	*time_slot_id*	*course_id*	*sec_id*
Garfield	359	A	BIO-101	1
Garfield	359	B	BIO-101	2
Saucon	651	A	CS-101	2
Saucon	550	C	CS-319	1
Painter	705	D	MU-199	1
Painter	403	D	FIN-201	1

Figure 5.22 The relation *r* for Exercise 5.24.

CHAPTER 6

Database Design using the E-R Model

Exercises

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

Answer:

A *superkey* is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. A superkey may contain extraneous attributes. If K is a superkey, then so is any superset of K. A superkey for which no proper subset is also a superkey is called a *candidate key*. It is possible that several distinct sets of attributes could serve as candidate keys. The *primary key* is one of the candidate keys that is chosen by the database designer as the principal means of identifying entities within an entity set.

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

Answer:

Answers may vary. We show a simple E-R diagram for the hospital in Figure 6.201. Although the diagram meets the specifications of the question, a real-world hospital would have many more requirements, such as tracking patient admissions and visits, including which doctor sees a patient on each visit, recording results of tests in a more structured manner, and so on.

6.16 Extend the E-R diagram of Exercise 6.3 to track the same information for all teams in a league.

Answer:

See Figure 6.202. Note that we assume a player can play in only one team; if a player may switch teams, we would have to track for each match the team

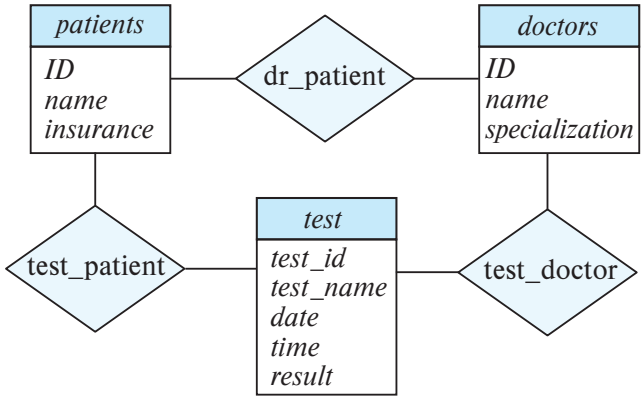


Figure 6.201 E-R diagram for a hospital.

the player was on, which we could do by turning the relationship *played* into a ternary relationship.

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

Answer:

A strong entity set has a primary key. All entities in the set are distinguishable by that key. A weak entity set has no primary key unless attributes of the strong entity set on which it depends are included. Entities in a weak entity set are partitioned according to their relationship with entities in a strong entity set.

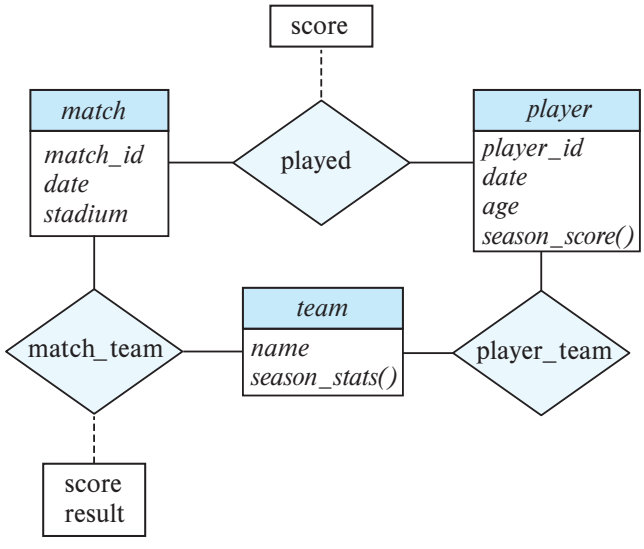


Figure 6.202 E-R diagram for all team statistics.

Entities within each partition are distinguishable by a discriminator, which is a set of attributes.

6.18 Consider two entity sets A and B that both have the attribute X (among others whose names are not relevant to this question).

a. If the two Xs are completely unrelated, how should the design be improved?

b. If the two Xs represent the same property and it is one that applies both to A and to B, how should the design be improved? Consider three subcases:

- X is the primary key for A but not B
- X is the primary key for both A and B
- X is not the primary key for A nor for B

Answer:

a. Give them distinct names

b. Solution for each case:

- This means that X in B is, in effect, a foreign key referencing A. That's fine in the relational model but a bad E-R design. Remove X from B and use a relationship set to connect A and B, making it m-1 from B to A.
- This means that A and B are specializations of the same entity set. Design an appropriate generalization/specialization hierarchy.
- In this case, both A and B have the attribute X, but it does not define any specific relationship between an A entity and a B entity. So there is no improvement to be made.

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

Answer:

We have weak entities for several reasons:

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

6.20 Construct appropriate relation schemas for each of the E-R diagrams in:

a. Exercise 6.1.

b. Exercise 6.2.

c. Exercise 6.3.

d. Exercise 6.15.

Answer:

a. Car insurance tables:

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

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

b. Student exam tables:

i. Ternary relationship:

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

ii. Binary relationship:

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

c. Player match tables:

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

d. Hospital tables:

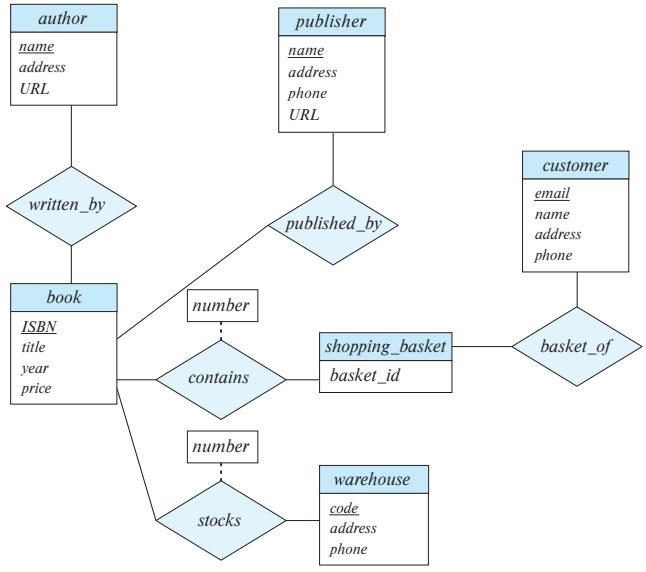


Figure 6.30 E-R diagram for modeling an online bookstore.

patients (patient-id, name, insurance, date-admitted, date-checked-out)
doctors (doctor-id, name, specialization)
test (testid, testname, date, time, result)
doctor-patient (patient-id, doctor-id)
test-log (testid, patient-id) performed-by (testid, doctor-id)

6.21 Consider the E-R diagram in Figure 6.30, which models an online bookstore.

a. Suppose the bookstore adds Blu-ray discs and downloadable video to its collection. The same item may be present in one or both formats, with differing prices. Draw the part of the E-R diagram that models this addition, showing just the parts related to video.

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

Answer:

A good way to design this is to model first that the bookstore adds videos, and then model the fact that they may be in Blu-ray disk format or in downloadable format; the same video may be present in both formats.

a. An ER-diagram portion related to videos is shown in Figure 6.203.

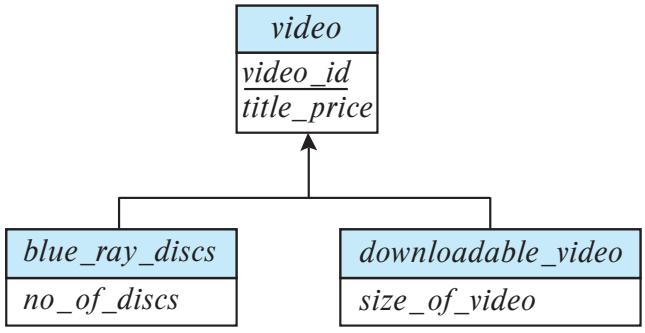


Figure 6.203 E-R diagram for Exercise 6.21(a)

b. The E-R diagram shown in Figure 6.204 should be added to the E-R diagram of Figure 6.30. Entities that are shown already in Figure 6.30 are shown with only their names, omitting the attributes. The *contains* relationship in Figure 6.30 should be replaced by the version in Figure 6.204. All other parts of Figure 6.30 remain unchanged.

.

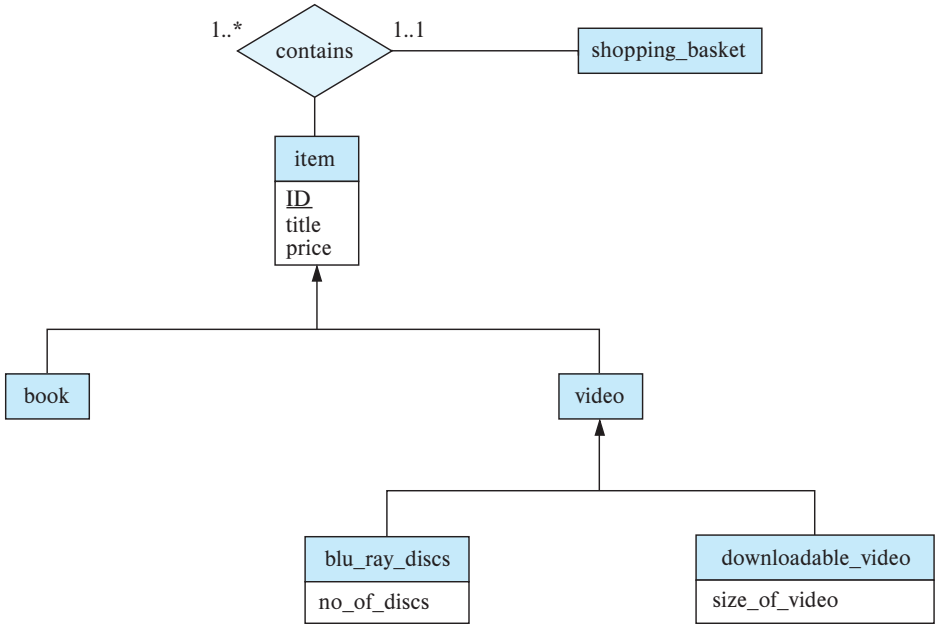


Figure 6.204 E-R diagram for Exercise 6.21(b)

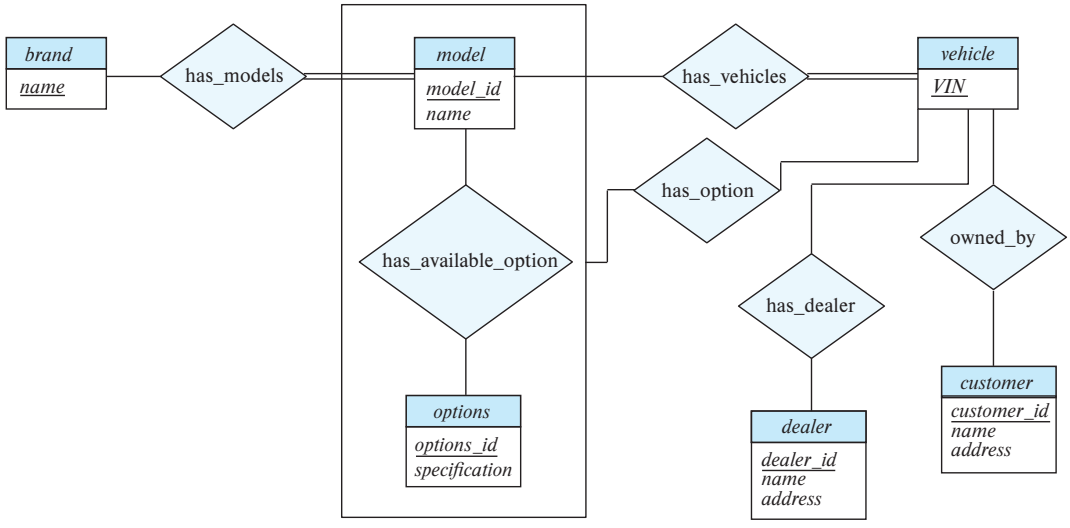


Figure 6.205 E-R diagram for Exercise 6.22.

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

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

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

Answer:

The E-R diagram is shown in Figure 6.205. Note that the *has_option* relationship links a vehicle to an aggregation on the relationship *has_available_option*, instead of directly to the entity set *options*, to ensure that a particular vehicle instance cannot get an option that does not correspond to its model. The alternative of directly linking to *options* is acceptable if ensuring the above integrity constraint is not critical.

The relational schema, along with primary-key and foreign-key constraints, is shown in Figure 6.206.

6.23 Design a database for a worldwide package delivery company (e.g., DHL or FedEx). The database must be able to keep track of customers who ship items

brand(brand_name),
model(model_id, model_name)
vehicle(VIN, dealer_id, customer_id)
option(option_id, specification)
customer(customer_id, customer_name, address)
dealer(dealer_id, dealer_name, address)
has_model(brand_name, model_id ,
foreign key *brand_name* **references** *brand,*
foreign key *model_id* **references** *model)*
has_vehicle(model_id, VIN,
foreign key *VIN* **references** *vehicle,*
foreign key *model_id* **references** *model)*
has_available_option(model_id, option_id,
foreign key *option_id* **references** *option,*
foreign key *model_id* **references** *model)*
has_option(VIN, model_id, option_id,
foreign key *VIN* **references** *vehicle,*
foreign key *(model_id, option_id)* **references** *available_option)*
has_dealer(VIN, dealer_id ,
foreign key *dealer_id* **references** *dealer,*
foreign key *VIN* **references** *vehicle)*
owned_by(VIN, customer_id,
foreign key *customer_id* **references** *customer,*
foreign key *VIN* **references** *vehicle)*

Figure 6.206 The relational schema for Exercise 6.22.

and customers who receive items; some customers may do both. Each package must be identifiable and trackable, so the database must be able to store the location of the package and its history of locations. Locations include trucks, planes, airports, and warehouses.

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

Answer:

An E-R diagram for a worldwide package delivery company is shown in Figure 6.207.

The relational schema, including primary-key and foreign-key constraints, is shown in Figure 6.208.

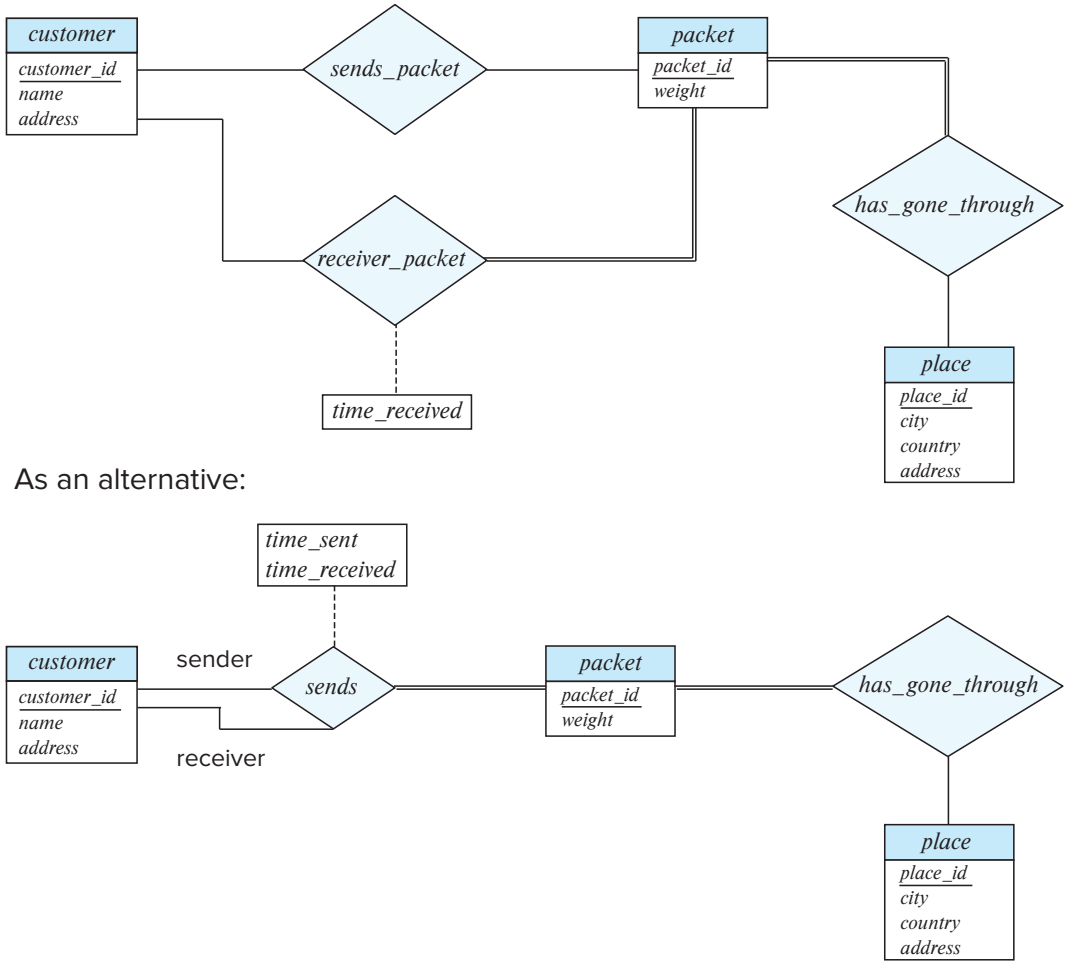


Figure 6.207 An E-R diagam for Exercise 6.23

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

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

Answer:

There are many valid answers. An E-R diagram is shown in Figure 6.209. We assume that the schedule of a flight is fixed, and we allow specification of the days on which a flight is scheduled. For a particular instance of a flight, we record actual times of departure and arrival. The relational schema is shown in Figure 6.210.

customer(customer_id, customer_name, address)
package(package_id, weight, contents)
location(loc_id)
truck(loc_id, VIN)
plane(loc_id, type, mfg)
airport(loc_id, city, code)
warehouse(loc_id, address)
at(package_id, loc_id, time_in, time_out,
foreign key *package_id* **references** *package,*
foreign key *loc_id* **references** *location)*
receive(customer_id, package_id, time,
foreign key *customer_id* **references** *customer,*
foreign key *package_id* **references** *package)*
send(customer_id, package_id, time,
foreign key *customer_id* **references** *customer,*
foreign key *package_id* **references** *package)*

Figure 6.208 A relational schema for Exercise 6.23

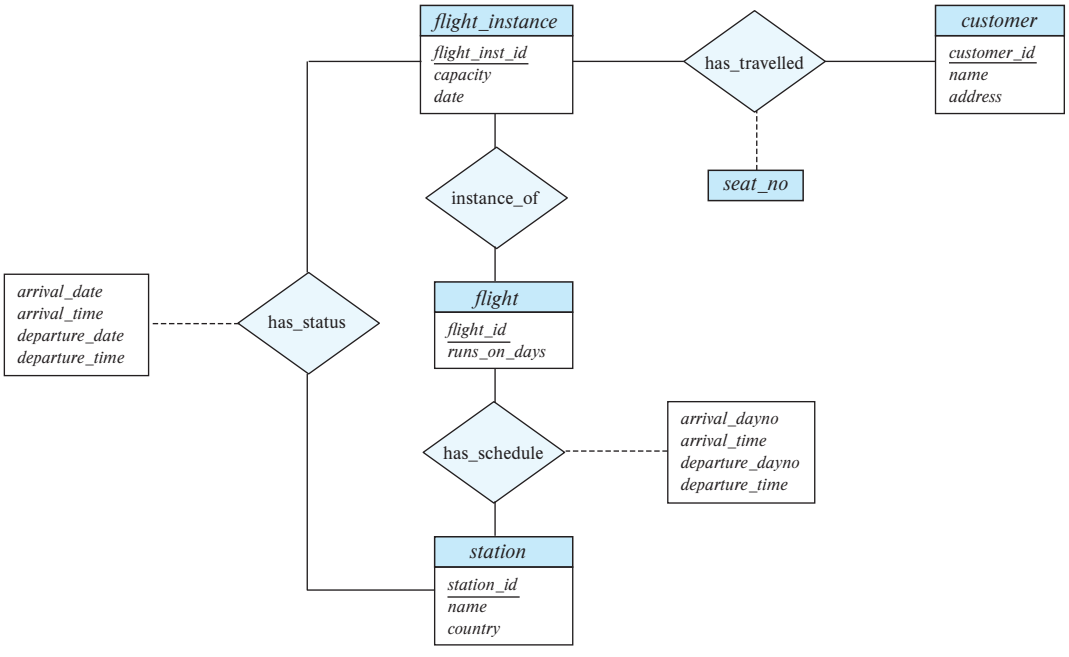


Figure 6.209 E-R diagram for airline databases.

flight_instance(flight_inst_id, capacity, date)
customer(customer_id, customer_name, address)
flight(flight_id, runs_on_days)
airport(airport_id, name, country)
has_traveled(flight_inst_id, customer_id, seat_number,
foreign key *flight_inst_id* **references** *flight_instance,*
foreign key *customer_id* **references** *customer)*
instance_of(flight_inst_id, flight_id,
foreign key *flight_inst_id* **references** *flight_instance,*
foreign key *flight_id* **references** *flight)*
has_schedule(flight_id, airport_id, arrival_time, departure_time,
foreign key *flight_id* **references** *flight,*
foreign key *airport_id* **references** *airport)*
has_status(flight_inst_id, airport_id, arrival_time, departure_time
foreign key *flight_inst_id* **references** *flight_instance,*
foreign key *airport_id* **references** *airport)*

Figure 6.210 A relational schema for airline databases.

6.25 In Section 6.9.4, we represented a ternary relationship (repeated in Figure 6.29a) using binary relationships, as shown in Figure 6.29b. Consider the alternative shown in Figure 6.29c. Discuss the relative merits of these two alternative representations of a ternary relationship by binary relationships.

Answer:
In the model of Figure 6.29b, there can be instances where E, A, B, C, R_A, R_B and R_C cannot correspond to any instance of A, B, C and R.
The model of Figure 6.29c will not be able to represent all ternary relationships. Consider the ABC relationship set below.

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

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

6.26 Design a generalization – specialization hierarchy for a motor vehicle sales company. The company sells motorcycles, passenger cars, vans, and buses. Justify

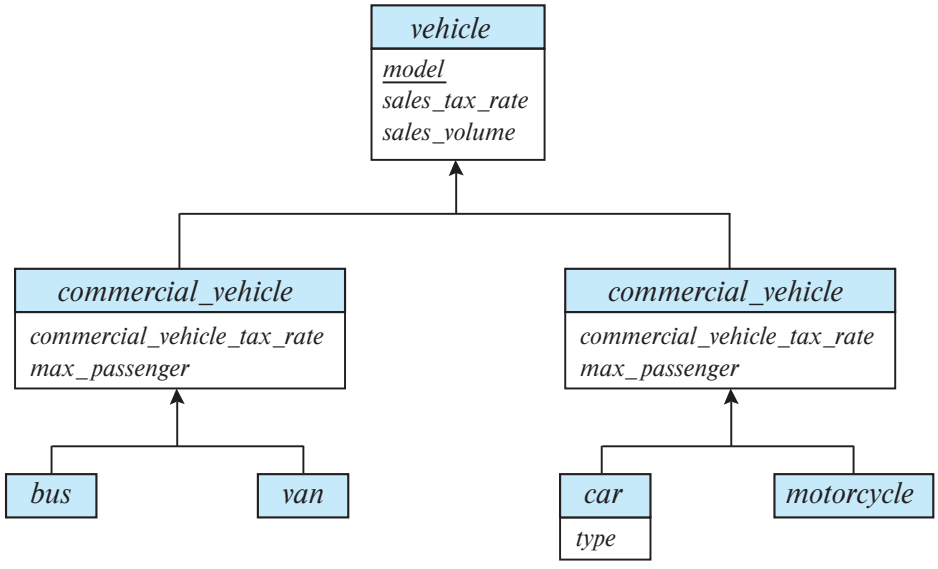


Figure 6.211 E-R diagram of motor vehicle sales company.

your placement of attributes at each level of the hierarchy. Explain why they should not be placed at a higher or lower level.

Answer:

Figure 6.211 gives one possible hierarchy; note that there could be many alternative solutions. The generalization – specialization hierarchy for the motor vehicle company is given in the figure. *model*, *sales-tax-rate*, and *sales-volume* are attributes necessary for all types of vehicles. Commercial vehicles attract commercial vehicle tax, and each kind of commercial vehicle has a passenger-carrying capacity specified for it. Some kinds of noncommercial vehicles attract luxury vehicle tax. Cars alone can be of several types, such as sports car, sedan, wagon, etc., hence the attribute *type*.

6.27 Explain the distinction between disjoint and overlapping constraints.

Answer:

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

6.28 Explain the distinction between total and partial constraints.

Answer:

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

CHAPTER 7

Relational Database Design

Exercises

7.19 Give a lossless decomposition into BCNF of schema R of Exercise 7.1.

Answer:
From Exercise 7.6, we know that the candidate keys are A, BC, CD, and E and that $B \rightarrow D$ is nontrivial and the left-hand side is not a superkey. By the algorithm of Figure 7.11 we derive the relations $\{(A, B, C, E), (B, D)\}$. This is in BCNF.

7.20 Give a lossless, dependency-preserving decomposition into 3NF of schema R of Exercise 7.1.

Answer:
First we note that the dependencies given in Exercise 7.1 form a canonical cover. Generating the schema from the algorithm of Figure 7.12 we get

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

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

Note that the original schema $R = (A, B, C, D, E)$ is already in 4NF. Thus, it was not necessary to apply the algorithm as we have done above. The single original schema is trivially a lossless, dependency-preserving decomposition.

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

Answer:

- Repetition of information is a condition in a relational database where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation.

This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult. It can lead to inconsistent data if updates are done to one copy of the value but not to another.

- Inability to represent information is a condition where there is a relationship that exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values; otherwise a tuple without the unrelated information cannot be inserted into the relation.
- Inability to represent information can also occur because of loss of information that results from the decomposition of one relation into two relations that cannot be combined to re-create the original relation. Such a lossy decomposition may happen implicitly, even without explicitly carrying out decomposition, if the initial relational schema itself corresponds to the decomposition.

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

Answer:

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

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

Answer:

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

Reflexivity rule: If α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.

Assume $\exists\, t_1$ and t_2 such that $t_1[\alpha] = t_2[\alpha]$

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

Augmentation rule: If $\alpha \rightarrow \beta$, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$.

Assume $\exists\, t_1,\ t_2$ such that $t_1[\gamma\alpha] = t_2[\gamma\alpha]$

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

Transitivity rule: If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

Assume $\exists\ t_1,\ t_2$ such that $t_1[\alpha] = t_2[\alpha]$

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

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

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

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2

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

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

Answer:
The decomposition rule and its derivation from Armstrong's axioms are given below:

If $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

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

7.26 Using the functional dependencies of Exercise 7.6, compute B^+.

Answer:
Computing B^+ by the algorithm in Figure 7.8 we start with *result* $= \{B\}$. Considering FDs of the form $\beta \rightarrow \gamma$ in F, we find that the only dependencies satisfying $\beta \subseteq$ *result* are $B \rightarrow B$ and $B \rightarrow D$. Therefore *result* $= \{B, D\}$. No more dependencies in F apply now. Therefore $B^+ = \{B, D\}$.

7.27 Show that the following decomposition of the schema R of Exercise 7.1 is not a lossless decomposition:

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

Hint: Give an example of a relation $r(R)$ such that $\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$

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

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

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

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

A	B	C
a_1	b_1	c_1
a_2	b_2	c_1

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

C	D	E
c_1	d_1	e_1
c_1	d_2	e_2

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

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

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

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

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

a. Compute B^+.

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

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

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

e. Give a BCNF decomposition of the given schema using the original set F of functional dependencies.

Answer:

a. Dependencies:

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

Thus, $B^+ = ABCDE$

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

$A \rightarrow BCD$ (Given)
$A \rightarrow ABCD$ (Augmentation with A)
$BC \rightarrow DE$ (Given)
$ABCD \rightarrow ABCDE$ (Augmentation with ABCD)
$A \rightarrow ABCDE$ (Transitivity)
$AG \rightarrow ABCDEG$ (Augmentation with G)

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

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

Now notice that B^+ is $ABCDE$, and in particular, the FD $B \rightarrow E$ can be determined from this set. Thus, the attribute C is extraneous in the third dependency. Removing this attribute, and combining with the third FD, we get the final canonical cover as :

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

Here, no attribute is extraneous in any FD.

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

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

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

$$r_4(A, G)$$

e. We start with

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

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

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

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

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

This schema is now in BCNF.

7.29 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$B \rightarrow D$$
$$DE \rightarrow B$$
$$DEG \rightarrow AB$$
$$AC \rightarrow DE$$

R is not in BCNF for many reasons, one of which arises from the functional dependency $AB \rightarrow CD$. Explain why $AB \rightarrow CD$ shows that R is not in BCNF and then use the BCNF decomposition algorithm starting with $AB \rightarrow CD$ to generate a BCNF decomposition of R. Once that is done, determine whether your result is or is not dependency preserving, and explain your reasoning.

Answer:

AB is not a superkey for R, nor is $AB \rightarrow CD$ trivial. We start using $AB \rightarrow CD$ and replace R with:

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

It turns out that neither of those is in BCNF. R_1 is not in BCNF because $B \rightarrow D$ holds on R_1 and is nontrivial, and B is not a superkey for R_1. So we replace R_1 with:

$$R_3 = (B, D)$$
$$R_4 = (A, B, C)$$

R_3 is in BCNF because all two-attribute schemas are in BCNF (see Exercise 7.36). So it is part of our answer. If R_4 is not in BCNF, it must be because of a functional dependency with just one attribute on the left side (anything else is either trivial or has a superkey on the left side). It is then easy to see that R_4 is also in BCNF and therefore part of the answer.
Back now to R_2, which is not in BCNF because $AB \rightarrow E$ is nontrivial and the left side is not a superkey. The reason this dependency holds on R_2 is that $AB \rightarrow CD$ so $AB \rightarrow C$ so $AB \rightarrow AC$ and then, since $AC \rightarrow DE$, we get $AB \rightarrow DE$, then, $AB \rightarrow E$. So we decompose R_2 to:

$$R_5 = (A, B, E)$$
$$R_6 = (A, B, G)$$

both of which are in BCNF. The answer is R_3, R_4, R_5, R_6

This is not dependency preserving. It suffices to show one dependency that is not preserved. Consider $DE \rightarrow B$. DE determines only itself and B, and there is no schema in the decomposition with all of B, D, E.

7.30 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$A \rightarrow BC$$
$$BD \rightarrow E$$
$$CD \rightarrow AB$$

a. Find a nontrivial functional dependency containing no extraneous attributes that is logically implied by the above three dependencies and explain how you found it.

b. Use the BCNF decomposition algorithm to find a BCNF decomposition of R. Start with $A \rightarrow BC$. Explain your steps.

c. For your decomposition, state whether it is lossless and explain why.

d. For your decomposition, state whether it is dependency preserving and explain why.

Answer:

a. $AD \rightarrow E$ since $AD \rightarrow BCD \rightarrow BCDE \rightarrow E$
Another is $CD \rightarrow E$ since $CD \rightarrow AB \rightarrow ABCD \rightarrow BD \rightarrow E$

b. First use $A \rightarrow BC$ to decompose R into $R_1 = (A, B, C)$ and $R_2 = (A, D, E, G)$. Then use the inferred $AD \rightarrow E$ to decompose R_2 into $R_3 = (A, D, E)$ and $R_4 = (A, D, G)$. The resulting decomposition is $(A, B, C), (A, D, E), (A, D, G)$

c. It is lossless since when the algorithm decomposes a schema using $\alpha \rightarrow \beta$, the attributes in α appear in both resulting schemas ($\alpha \cap \beta = \emptyset$) and can be used to join two schemas without introducing more tuples, since $\alpha \rightarrow \alpha\beta$.

d. It is not dependency preserving. It preserves $A \rightarrow BC$, $AD \rightarrow E$, but not $BD \rightarrow E$.

7.31 Consider the schema $R = (A, B, C, D, E, G)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$ADE \rightarrow GDE$$
$$B \rightarrow GC$$
$$G \rightarrow DE$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F, along with an explanation of the steps you took to generate it

c. The remaining steps of the algorithm, with explanation

d. The final decomposition

Answer:

a. AB is the only candidate key.

b. D and E are extraneous on the right side of $ADE \rightarrow GDE$.
A on the left side of $AB \rightarrow CD$ is extraneous since $(B)^+ = \{B, C, D, E, G\}$ using the last two FDs.
At this point we have:

$$B \rightarrow CD$$
$$ADE \rightarrow G$$
$$B \rightarrow GC$$
$$G \rightarrow DE$$

But then C and D are extraneous in $B \rightarrow CD$. That results in B determining the empty set, so we take out the first functional dependency. No other extraneous attribute can be found, and we come up with the canonical cover:

$$ADE \rightarrow G$$
$$B \rightarrow GC$$
$$G \rightarrow DE$$

c. The 3NF decomposition of R at this point is:

$$R_1 = (A, D, E, G)$$
$$R_2 = (B, C, G)$$
$$R_3 = (D, E, G)$$

R_3 is subsumed by R_1. We lack a schema containing a candidate key for R, so we add that.

d. We have (A, B), R_1 and R_2 as the final decomposition:
$(A, B), (B, C, G), (A, D, E, G)$.

7.32 Consider the schema $R = (A, B, C, D, E, G, H)$ and the set F of functional dependencies:

$$AB \rightarrow CD$$
$$D \rightarrow C$$
$$DE \rightarrow B$$
$$DEH \rightarrow AB$$
$$AC \rightarrow DC$$

Use the 3NF decomposition algorithm to generate a 3NF decomposition of R, and show your work. This means:

a. A list of all candidate keys

b. A canonical cover for F

c. The steps of the algorithm, with explanation

d. The final decomposition

Answer:

a. Candidate keys: Observe that E, G, H do not appear on the right side of any functional dependency, and so they must all be a part of any candidate key. The candidate keys are: $ABEGH, ACEGH, DEGH$.

b. Canonical cover:
C is extraneous on the right side of $AC \rightarrow DC$.
B is extraneous on the right side of $DEH \rightarrow AB$.
C is extraneous on the right side of $AB \rightarrow CD$.
So we now have:

$$AB \rightarrow D$$
$$D \rightarrow C$$
$$DE \rightarrow B$$
$$DEH \rightarrow A$$
$$AC \rightarrow D$$

So our initial design is $(A, B, D), (C, D), (B, D, E), (A, D, E, H), (A, C, D)$. We remove (C, D) because it is contained in (A, C, D) and we add a schema consisting of one of the candidate keys. We arbitrarily choose (D, E, G, H). $(A, B, D), (B, D, E), (A, D, E, H), (A, C, D), (D, E, G, H)$.

7.33 Although the BCNF algorithm ensures that the resulting decomposition is lossless, it is possible to have a schema and a decomposition that was not generated by the algorithm, that is in BCNF, and is not lossless. Give an example of such a schema and its decomposition.

Answer:
There are many such answers. A very simple one (though not the simplest) is a schema (A, B, C) in which only trivial functional dependencies hold. Then the decomposition $((A, B), (B, C))$ is in BCNF but is not lossless. That it is not lossless is easy to see using the two-tuple example $(1, 1, 1), (2, 1, 2)$.
An even simpler answer is the schema (A, B), decomposed into $((A), (B))$.

7.34 Show that every schema consisting of exactly two attributes must be in BCNF regardless of the given set F of functional dependencies.

Answer:
Let us denote the two-attribute schema as (A, B). Then the only possible nontrivial functional dependencies that could hold are $A \rightarrow B$ and $B \rightarrow A$. In both of these cases, the left side of the functional dependency would then be a superkey.

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

Answer:
The three design goals are lossless decompositions, dependency-preserving decompositions, and minimization of repetition of information. They are desirable so we can maintain an accurate database, check the correctness of updates quickly, and use the smallest amount of space possible.

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

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

7.37 Given the three goals of relational database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Exercise 7.19 for the definition of 2NF.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:
In *books*, we see that

isbn $\twoheadrightarrow$ *title*, *publisher*, *author*

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

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

After this, we still have

$$isbn \twoheadrightarrow author$$

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

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

Similarly, in *users*,

$$deptid \rightarrow deptname$$

and yet, *deptid* is not a superkey. Hence, we break *users* into

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

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

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

7.41 Although SQL does not support functional dependency constraints, if the database system supports constraints on materialized views, and materialized views are maintained immediately, it is possible to enforce functional dependency constraints in SQL. Given a relation $r(A, B, C)$, explain how constraints on materialized views can be used to enforce the functional dependency $B \rightarrow C$.

Answer:

create materialized view *V* **as**
select distinct *B, C*
from *r*;
alter table *V* **add constraint** *V_pk* **primary key** (*B*);

7.42 Given two relations $r(A, B, validtime)$ and $s(B, C, validtime)$, where *validtime* denotes the valid time interval, write an SQL query to compute the temporal natural join of the two relations. You can use the && operator to check if two intervals overlap and the $*$ operator to compute the intersection of two intervals.

Answer:

select *A, B, C*, *r.validtime* $*$ *s.validtime*
from *r*, *s*
where *r*.*B* = *s*.*B* **and** *r.validtime* && *s.validtime*

CHAPTER 8

Beyond Relational Data

Exercises

8.10 Redesign the database of Exercise 8.4 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first and fourth normal form schemas.

Answer:

To put the schema into first normal form, we flatten all the attributes into a single relation schema.

$$\textit{Employee-details} = (\textit{ename, cname, bday, bmonth, byear, stype, xyear, xcity})$$

We rename the attributes for the sake of clarity. *cname* is *Children.name*, and *bday, bmonth, byear* are the *Birthday* attributes. *stype* is *Skills.type*, and *xyear* and *xcity* are the *Exams* attributes. The FDs and multivalued dependencies we assume are:-

$$\begin{aligned} \textit{ename}, \textit{cname} &\rightarrow \textit{bday}, \textit{bmonth}, \textit{byear} \\ \textit{ename} &\twoheadrightarrow \textit{cname}, \textit{bday}, \textit{bmonth}, \textit{byear} \\ \textit{ename}, \textit{stype} &\twoheadrightarrow \textit{xyear}, \textit{xcity} \end{aligned}$$

The FD captures the fact that a child has a unique birthday, under the assumption that one employee cannot have two children of the same name. The MVDs capture the fact there is no relationship between the children of an employee and his or her skills information.

The redesigned schema in fourth normal form is:-

$$\begin{aligned} \textit{Employee} &= (\textit{ename}) \\ \textit{Child} &= (\textit{ename, cname, bday, bmonth, byear}) \\ \textit{Skill} &= (\textit{ename, stype, xyear, xcity}) \end{aligned}$$

ename will be the primary key of *Employee*, and (*ename, cname*) will be the primary key of *Child*. The *ename* attribute is a foreign key in *Child* and in *Skill*, referring to the *Employee* relation.

8.11 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 8.2.1.3. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

Answer:

A corresponding relational schema in third normal form is given below:

People = (*name, address*)
Students = (*name, degree, student-department*)
Teachers = (*name, salary, teacher-department*)

name is the primary key for all the three relations, and it is also a foreign key referring to *People* for both *Students* and *Teachers*.

Instead of placing only the *name* attribute of *People* in *Students* and *Teachers*, both its attributes can be included. In that case, there will be a slight change, namely (*name, address*) will become the foreign key in *Students* and *Teachers*. The primary keys will remain the same in all tables.

8.12 Consider the E-R diagram in Figure 8.9, which contains specializations, using subtypes and subtables.

a. Give an SQL schema definition of the E-R diagram.

b. Give an SQL query to find the names of all people who are not secretaries.

c. Give an SQL query to print the names of people who are neither employees nor students.

d. Can you create a person who is an employee and a student with the schema you created? Explain how, or explain why it is not possible.

Answer:

a. Schema definition:
Please see Figure 8.101.

b. SQL query:

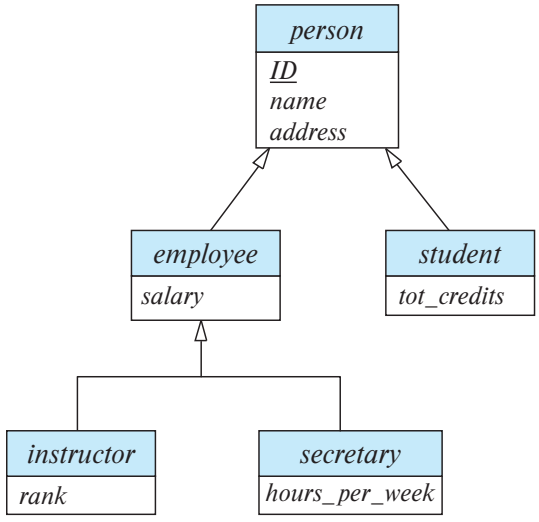


Figure 8.9 Specialization and generalization.

```
select *
from person
where ID not in
        (select ID from secretary)
```

c. SQL query:

```
select *
from only person
```

d. It is not possible to create a person who is an employee and a student, since there is no most-specific type in our schema corresponding to such a person. To do so, we would have to create a corresponding type (such as TeachingAssistant) using multiple inheritance, and a corresponding table that is under employee and student. However, SQL does not permit multiple inheritance of tables, so this is not possible in SQL.

8.13 Suppose you wish to perform keyword querying on a set of tuples in a database, where each tuple has only a few attributes, each containing only a few words. Does the concept of term frequency make sense in this context? And that of inverse document frequency? Explain your answer. Also suggest how you can define the similarity of two tuples using TF-IDF concepts.

Answer:

Term frequency is the logarithm of the number of occurrences of the term divided by the number of terms in the document. When it comes to small databases with few attributes, each containing only a few words, the concept

```
create type Person
    ID varchar(10),
    name varchar(30),
    address varchar(40))
create type Employee
    under Person
    (salary integer)
create type Student
    under Person
    (tot_credits integer)
create type Instructor
    under Employee
    (rank varchar(10))
create type Secretary
    under Employee
    (hours_per_week integer)

create table person of Person
create table employee of Employee
    under person
create table student of Student
    under person
create table instructor of Instructor
    under employee
create table secretary of Secretary
    under employee
```

Figure 8.101 Schema definition for Exercise 8.12.

of term frequency may not make sense. The relevance of a term may not depend on the number of occurrences of the term, and also when the domain is very small the logarithmic increase we used in the term frequency many not be a good indicator.

The inverse document frequency, which is the inverse of the number of documents that contain this term, may also not be very relevant in this case. For example, the primary key value and some other key may be having the inverse document frequency of 1, but we can't assume their weights to be equal.

The similarity of the two tuples can be measured by the *cosinesimilarity* metric. But one major difference is that only values that belong to the same attribute should be considered. Two different attributes from two tuples may be having the same value, but that doesn't increase the similarity factor.

8.14 Web sites that want to get some publicity can join a web ring, where they create links to other sites in the ring in exchange for other sites in the ring creating links to their site. What is the effect of such rings on popularity ranking techniques such as PageRank?

Answer:

PageRank is a measure of popularity of a page based on the popularity of the pages that link to the page. It may be noted that the pages that are pointed to from many web pages are more likely to be visited, and thus will have a higher PageRank. Similarly, pages pointed to by web pages with a high PageRank will also have a higher probability of being visited, and thus will have a higher PageRank. In the given scenario where web sites join a web ring and create links to other sites, the PageRank of all the pages increases. The number of links referencing each page increases, which only increases the PageRank.

8.15 The Google search engine provides a feature whereby web sites can display advertisements supplied by Google. The advertisements supplied are based on the contents of the page. Suggest how Google might choose which advertisements to supply for a page, given the page contents.

Answer:

Google might use the concepts in similarity-based retrieval. Here, they can give the system a document A and the set of advertisements B, and ask the system to retrieve advertisements that are similar to A. One approach is to find k terms in A with highest values of TF(A,t) * IDF(t), and to use to these k terms as a query to find relevance of other documents. The *cosine similarity* metric can also be used to determine which advertisements to supply for a page, given the page contents.

Google may also take into account the user-profile of the user when deciding what advertisements to display, and could potentially even choose to ignore the web page content when deciding what advertisement to show.

CHAPTER 9

Application Development

Exercises

9.12 Write a servlet and associated HTML code for the following very simple application: A user is allowed to submit a form containing a value, say *n*, and should get a response containing *n* "*" symbols.

Answer:

- **HTML form** – Please see Figure 9.101.
- **Servlet Code** – Please see Figure 9.102.

9.13 Write a servlet and associated HTML code for the following simple application: A user is allowed to submit a form containing a number, say *n*, and should get a response saying how many times the value *n* has been submitted previously. The number of times each value has been submitted previously should be stored in a database.

```
<html>
    <head>
        <title>DB Book Exercise 8.8 </title>
    </head>
    <form action="servlet/StarServlet" method=get>
        Enter the value for "n"
        <br>
        <input type=text size=5 name="n">
        <input type=submit value="submit">
    </form>
</html>
```

Figure 9.101 HTML form for Exercise 9.12.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class StarServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        int n = Integer.parseInt(request.getParameter("n"));
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE>Exercise 8.8</TITLE></HEAD>");
        out.println("<BODY>");
        for (int i = 0; i < n; i++) {
            out.print("*");
        }
        out.println("</BODY>");
        out.close();
    }
}
```

Figure 9.102 Servlet Code for Exercise 9.12.

Answer:

- **HTML form 9-13** – Please see Figure 9.103.
- **Schema 9-13** – Please see Figure 9.104.
- **Servlet Code 9-13** – Please see Figure 9.105.

9.14 Write a servlet that authenticates a user (based on user names and passwords stored in a database relation) and sets a session variable called *userid* after authentication.

Answer:

- **HTML form** – Please see Figure 9.106.
- **Schema** – Please see Figure 9.107.
- **Servlet Code** – Please see Figure 9.108.

9.15 What is an SQL injection attack? Explain how it works and what precautions must be taken to prevent SQL injection attacks.

```
<html>
    <head>
        <title>DB Book Exercise 9.13 </title>
    </head>
    <form action="servlet/KeepCountServlet" method=get>
        Enter the value for "n"
        <br>
        <input type=text size=5 name="n">
        <input type=submit value="submit">
    </form>
</html>
```

Figure 9.103 HTML form for Exercise 9.13.

Answer:

SQL injection attack occurs when a malicious user (attacker) manages to get an application to execute an SQL query created by the attacker. If an application constructs an SQL query string by concatenating the user-supplied parameters, the application is prone to SQL injection attacks.
For example, suppose an application constructs and executes a query to retrieve a user's password in the following way:

```
String userid = request.getParameter("userid");
executeQuery("SELECT password FROM userinfo WHERE userid= ' " + userid + " ' ");
```

Now, if a user types the value for the parameter as:

```
john' OR userid= 'admin
```

the query constructed will be:

```
SELECT password FROM userinfo WHERE userid='john' OR userid='admin';
```

This can reveal unauthorized information to the attacker.

```
CREATE TABLE SUBMISSION_COUNT (
    value integer unique,
    scount integer not null);
```

Figure 9.104 Servlet Code for Exercise 9.13.

```
import java.io.*; import java.sql.*; import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class KeepCountServlet extends HttpServlet {
    private static final String query =
        "SELECT scount FROM SUBMISSION_COUNT WHERE value=?";

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        int n = Integer.parseInt(request.getParameter("n"));
        int count = 0;
        try {
            Connection conn = getConnection();
            PreparedStatement pstmt = conn.prepareStatement(query);
            pstmt.setInt(1, n);
            ResultSet rs = pstmt.executeQuery();
            if (rs.next()) {
                count = rs.getInt(1);
            }
            pstmt.close();

            Statement stmt = conn.createStatement();
            if (count == 0) {
                stmt.executeUpdate("INSERT INTO SUBMISSION_COUNT VALUES ("
                        + n + ", 1)");
            } else {
                stmt.executeUpdate("UPDATE SUBMISSION_COUNT SET " +
                        "scount=scount+1 WHERE value=" + n);
            }
            stmt.close();
            conn.close();
        }
        catch(Exception e) {
            throw new ServletException(e.getMessage());
        }
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE>Exercise 9.13</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("The value "+n+ " has been submitted "+count+" times previously.");
        out.println("</BODY>");
        out.close();
    }
}
```

Figure 9.105 Servlet Code for Exercise 9.13.

Prevention:

Use prepared statements, with any value that is taken as user input (not just text fields, but even options in drop-down menus) passed as a parameter; user input

```
<html>
    <head>
        <title>DB Book Exercise 9.14 </title>
    </head>
    <form action="servlet/SimpleAuthServlet" method=get>
        User Name:
        <input type=text size=20 name="user">
        <BR>
        <BR>
        Password :
        <input type=password size=20 name="passwd">
        <BR>
        <input type=submit value="submit">
    </form>
</html>
```

Figure 9.106 HTML form for Exercise 9.13.

should **never** be concatenated directly into a query string. The JDBC, ODBC, ADO.NET, or other libraries that provide prepared statements ensure that special characters like quotes are escaped as appropriate for the target database, so that SQL injection attempts will fail.

9.16 Write pseudocode to manage a connection pool. Your pseudocode must include a function to create a pool (providing a database connection string, database user name, and password as parameters), a function to request a connection from the pool, a connection to release a connection to the pool, and a function to close the connection pool.

Answer:

Please see Figure 9.109.

9.17 Explain the terms CRUD and REST.

Answer:

```
CREATE TABLE USERAUTH(
        userid integer primary key,
        username varchar(100) unique,
        password varchar(20)
);
```

Figure 9.107 Servlet Code for Exercise 9.13.

```
import java.io.*; import java.sql.*; import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleAuthServlet extends HttpServlet {
    private static final String query =
            "SELECT userid, password FROM USERAUTH WHERE username=?";

    public void doGet(HttpServletRequest request, HttpServletResponse response)
                    throws ServletException, IOException
    {
        String user = request.getParameter("user");
        String passwd = request.getParameter("passwd");
        String dbPass = null;
        int userId = -1;
        try {
            Connection conn = getConnection();
            PreparedStatement pstmt = conn.prepareStatement(query);
            pstmt.setString(1, user);
            ResultSet rs = pstmt.executeQuery();
            if (rs.next()) {
                userId = rs.getInt(1);
                dbPass = rs.getString(2);
            }
            pstmt.close();
            conn.close();
        }
        catch(Exception e) {
            throw new ServletException(e.getMessage());
        }
        String message;
        if(passwd.equals(dbPass)) {
            message = "Authentication successful";
            getServletContext().setAttribute("userid", new Integer(userId));
        } else {
            message = "Authentication failed! Please check the username " +
                "and password.";
        }

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HEAD><TITLE>Exercise 9.14</TITLE></HEAD>");
        out.println("<BODY>");
        out.println(message);
        out.println("</BODY>");
        out.close();
    }
}
```

```
// This connection pool manager is NOT thread-safe.

INITIAL_POOL_SIZE = 20;
POOL_SIZE_INCREMENT = 5;
MAX_POOL_SIZE = 100;
Queue freeConnections = empty queue;
Queue activeConnections = empty queue;
String poolConnURL;
String poolConnUserId;
String poolConnPasswd;

createPool(connString, userid, password) {
      poolConnURL = connString;
      poolConnUserId = userid;
      poolConnPasswd = password;
      for (i = 0; i < INITIAL_POOL_SIZE; i++) {
            conn = createConnection(connString, userid, password);
            freeConnections.add(conn);
      }
}

Connection getConnection() {
      if(freeConnections.size() != 0){
            conn = freeConnections.remove();
            activeConnections.add(conn);
            return conn;
      }
      activeConns = activeConnections.size();
      if (activeConns == MAX_POOL_SIZE)
            ERROR("Max pool size reached");
      if (MAX_POOL_SIZE – activeConns > POOL_SIZE_INCREMENT)
            connsToCreate = POOL_SIZE_INCREMENT;
      else
            connsToCreate = MAX_POOL_SIZE – activeConns;

      for (i = 0; i < connsToCreate; i++) {
            conn = createConnection(poolConnURL, poolConnUserId,
                        poolConnPasswd);
            freeConnections.add(conn);
      }
      return getConnection();
}

releaseConnection(conn) {
      activeConnections.remove(conn);
      freeConnections.add(conn);
}
```

```
closePool() {
        if(activeConnections.size() != 0)
                WARNING("Connections active. Will force close.");
        for (i=0; i < freeConnections.size(); i++) {
                conn = freeConnections.elementAt(i);
                freeConnections.removeElementAt(i);
                conn.close();
        }

        for (i=0; i < activeConnections.size(); i++) {
                conn = activeConnections.elementAt(i);
                activeConnections.removeElementAt(i);
                conn.close();
        }
}
```

Figure 9.110 Continuation of Figure 9.109.

The term CRUD refers to simple user interfaces to a relation (or an object model), that provide the ability to Create tuples, Read tuples, Update tuples, and Delete tuples (or objects).

In Representation State Transfer (or REST), Web service function calls are executed by a standard HTTP request to a URL at an application server, with parameters sent as standard HTTP request parameters. The application server executes the request (which may involve updating the database at the server), generates and encodes the result, and returns the result as the result of the HTTP request. The server can use any encoding for a particular requested URL; XML and the JavaScript Object Notation (JSON) encoding are widely used. The requestor parses the returned page to access the returned data.

9.18 Many web sites today provide rich user interfaces using Ajax. List two features each of which reveals if a site uses Ajax, without having to look at the source code. Using the above features, find three sites which use Ajax; you can view the HTML source of the page to check if the site is actually using Ajax.

Answer:

a. A web site with a form that allows you to select a value from one menu (e.g., country), and once a value has been selected, you are allowed to select a value from another menu (e.g., state from a list of states in that country) with the list of values for the second menu (e.g., state) empty until the first value (e.g., country) is selected, probably uses Ajax. If the number of countries is small, the site may well send all country-state pairs

ahead of time and then simply use Javascript without Ajax; however, if you notice a small delay in populating the second menu, the site probably uses Ajax.

b. A web site that supports autocompletion for text that you are typing almost surely uses Ajax. For example, a search web site that suggests possible completions of your query as you type the query in, or a web-based email site that suggests possible completions of an email address as you type in the address almost surely uses Ajax to communicate with a server after you type in each character (sometimes starting after the 3rd or 4th character), and it responds with possible completions.

c. A web form that, on filling in one piece of data, such as your email address or employee code, fills in other fields such as your name and contact information, without refreshing the page, almost surely uses Ajax to retrieve required information using the information (such as the email address or employee code) provided by the user.

Popular web sites that use Ajax include almost all current generation web email interfaces (such as GMail, Yahoo! mail, or Windows Live mail), and almost all search engines which provide autocompletion. Online document management systems such as Google Docs or Office Live use Ajax extensively to push your updates to the server and to fetch concurrent updates (to different parts of the document or spreadsheet) transparently. Check your organization's web applications to find more local examples.

9.19 XSS attacks:

a. What is an XSS attack?

b. How can the referer field be used to detect some XSS attacks?

Answer:

a. In an XSS attack, a malicious user enters code written in a client-side scripting language such as JavaScript or Flash instead of entering a valid name or comment. When a different user views the entered text, the browser executes the script, which can carry out actions such as sending private cookie information back to the malicious user or executing an action on a different web site, such as a bank web site, that the user may be logged into.

b. The HTTP protocol allows a server to check the referer of a page access, that is, the URL of the page that had the link that the user clicked on to initiate the page access. By checking that the referer is valid, for example, that the referer URL is a page on the same web site (say the bank web site in the previous example), XSS attacks that originated on a different

web site accessed by the user can be prevented. The referer field is set by the browser, so a malicious or compromised browser can spoof the referer field, but a basic XSS attack can be detected and prevented.

9.20 What is multifactor authentication? How does it help safeguard against stolen passwords?

Answer:

In multifactor authentication (with two-factor authentication as a special case), multiple independent *factors* (that is, pieces of information or processes) are used to identify a user. The factors should not share a common vulnerability; for example, if a system merely required two passwords, both could be vulnerable to leakage in the same manner (by network sniffing, or by a virus on the computer used by the user, for example). In addition to secret passwords, which serve as one factor, one-time passwords sent by SMS to a mobile phone or a hardware token that generates a (time-based) sequence of one-time passwords are examples of extra factors.

9.21 Consider the Oracle Virtual Private Database (VPD) feature described in Section 9.8.5 and an application based on our university schema.

a. What predicate (using a subquery) should be generated to allow each faculty member to see only *takes* tuples corresponding to course sections that they have taught?

b. Give an SQL query such that the query with the predicate added gives a result that is a subset of the original query result without the added predicate.

c. Give an SQL query such that the query with the predicate added gives a result containing a tuple that is not in the result of the original query without the added predicate.

Answer:

a. The following predicate can be added to queries on *takes*, to ensure that each faculty member only sees *takes* tuples corresponding to course sections that they have taught; the predicate assumes that *syscontext.user_id()* returns the instructor identifier.

```
exists (select *
        from teaches
        where teaches.ID=syscontext.user_name() and
              teaches.course_id= takes.course_id and
              teaches.sec_id= takes.sec_id and
              teaches.year= takes.year and
              teaches.semester= takes.semester)
```

b. The following query retrieves a subset of the answers, due to the above predicate:

select * from *takes*;

c. The following query gives a result tuple that is not in the result of the original query:

select count(*) from *takes*;

The aggregated function above can be any of the aggregate functions, such as sum, average, min or max on any attribute; in the case of min or max the result could be the same if the person executing the query is authorized to see the tuple corresponding to the min or max value. For count, sum, and average, the values are likely to be different as long as there is more than one section.

9.22 What are two advantages of encrypting data stored in the database?

Answer:

a. Unauthorized users who gain access to the OS files in which the DBMS stores the data cannot read the data.

b. If the application encrypts the data before they reach the database, it is possible to ensure privacy for the user's data such that even privileged users like database administrators cannot access other users' data.

9.23 Suppose you wish to create an audit trail of changes to the *takes* relation.

a. Define triggers to create an audit trail, logging the information into a relation called, for example, *takes_trail*. The logged information should include the user-id (assume a function *user_id*() provides this information) and a timestamp, in addition to old and new values. You must also provide the schema of the *takes_trail* relation.

b. Can the preceding implementation guarantee that updates made by a malicious database administrator (or someone who manages to get the administrator's password) will be in the audit trail? Explain your answer.

Answer:

a. **Schema for the *takes_trail* table**

```
takes_trail(user_id integer, timestamp datetime, operation_code integer,
        new_ID, new_course_id, new_sec_id, new_year, new_sem, new_grade
        old_ID, old_course_id, old_sec_id, old_year, old_sem, old_grade
```

Trigger for INSERT

```
create trigger takes_insert after insert on takes
referencing new row as nrow
for each row
begin
      insert into takes_trail values (user_id(), systime(), 1,
            nrow.ID, nrow.course_id, nrow.sec_id, nrow.year, nrow.sem, nrow.grade
            null, null, null, null, null, null);
end
```

Trigger for UPDATE

```
create trigger takes_update after update on takes
referencing old row as orow, referencing new row as nrow
for each row
begin
      insert into takes_trail values (user_id(), systime(), 2,
            nrow.ID, nrow.course_id, nrow.sec_id, nrow.year, nrow.sem, nrow.grade
            orow.ID, orow.course_id, orow.sec_id, orow.year, orow.sem, orow.grade);
end
```

Trigger for DELETE

```
create trigger takes_delete after delete on takes
referencing old row as orow
for each row
begin
      insert into account_trail values (user_id(), systime(), 3,
            null, null, null, null, null, null);
            orow.ID, orow.course_id, orow.sec_id, orow.year, orow.sem, orow.grade);
end
```

b. No. Someone who has the administrator privileges can disable the trigger and thus bypass the trigger-based audit trail.

9.24 Hackers may be able to fool you into believing that their web site is actually a web site (such as a bank or credit card web site) that you trust. This may be done by misleading email, or even by breaking into the network infrastructure and rerouting network traffic destined for, say mybank.com, to the hacker's site. If you enter your user name and password on the hacker's site, the site can record it and use it later to break into your account at the real site. When you use a URL such as https://mybank.com, the HTTPS protocol is used to prevent such attacks. Explain how the protocol might use digital certificates to verify authenticity of the site.

Answer:

In the HTTPS protocol, a web site first sends a digital certificate to the user's browser. The browser decrypts the digital certificate using the stored public key of the trusted certification authority and displays the site's name from the decrypted message. The user can then verify if the site name matches the one he/she intended to visit (in this example mybank.com) and accept the certificate. The browser then uses the site's public key (that is part of the digital cer-

tificate) to encrypt the user's data. Note that it is possible for a malicious user to gain access to the digital certificate of mybank.com, but since the user's data (such as password) are encrypted using the public key of mybank.com, the malicious user will not be able to decrypt the data.

9.25 Explain what is a challenge – response system for authentication. Why is it more secure than a traditional password-based system?

Answer:

In a challenge – response system, a secret password is issued to the user and is also stored on the database system. When a user has to be authenticated, the database system sends a challenge string to the user. The user encrypts the challenge string using his/her secret password and returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password and checking the result with the original challenge string. The challenge-response system is more secure than a traditional password-based system since the password is not transferred over the network during authentication.

CHAPTER 10

Big Data

Exercises

10.10 Give four ways in which information in web logs pertaining to the web pages visited by a user can be used by the web site.

Answer:

a. To decide what information to present to the user

b. To decide what advertisements to show to the user

c. To decide how the web site should be structured

d. To determine user preferences and trends

10.11 One of the characteristics of Big Data is the variety of data. Explain why this characteristic has resulted in the need for languages other than SQL for processing Big Data.

Answer:
Variety of data beyond just relations cannot be handled by SQL. Need imperative code to handle full generality.

10.12 Suppose your company has built a database application that runs on a centralized database, but even with a high-end computer and appropriate indices created on the data, the system is not able to handle the transaction load, leading to slow processing of queries. What would be some of your options to allow the application to handle the transaction load?

Answer:
Here are some options.

- Use a parallel database,
- Use a cache such as redis or memcached to keep data in memory (possibly across multiple databases),

- Replicate the database and send read-only queries to the replica.

10.13 The map-reduce framework is quite useful for creating inverted indices on a set of documents. An inverted index stores for each word a list of all document IDs that it appears in (offsets in the documents are also normally stored, but we shall ignore them in this question).

For example, if the input document IDs and contents are as follows:

1: data clean
2: data base
3: clean base

then the inverted lists would

data: 1, 2
clean: 1, 3
base: 2, 3

Give pseudocode for map and reduce functions to create inverted indices on a given set of files (each file is a document). Assume the document ID is available using a function context.getDocumentID(), and the map function is invoked once per line of the document. The output inverted list for each word should be a list of document IDs separated by commas. The document IDs are normally sorted, but for the purpose of this question you do not need to bother to sort them.

Answer:
No answer

10.14 Fill in the blanks below to complete the following Apache Spark program which computes the number of occurrences of each word in a file. For simplicity we assume that words only occur in lowercase, and there are no punctuation marks.

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts =
    textFile._____(s -> Arrays.asList(s.split(" "))._____())
     .mapToPair(word -> new _____).reduceByKey((a, b) -> a + b);
```

Answer:

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaPairRDD<String, Integer> counts = textFile
.flatMap(s -> Arrays.asList(s.split(" ")).iterator())
.mapToPair(word -> new Tuple2<>(word, 1))
.reduceByKey((a, b) -> a + b);
```

10.15 Suppose a stream can deliver tuples out of order with respect to tuple timestamps. What extra information should the stream provide, so a stream query processing system can decide when all tuples in a window have been seen?

Answer:
Punctuations, which guarantee that no further tuples with a timestamp lower than some value will be delivered.

10.16 Explain how multiple operations can be executed on a stream using a publish-subscribe system such as Apache Kafka.

Answer:
Each operation subscribes to its input streams and publishes data to its output streams.

CHAPTER 11

Data Analysis

Exercises

11.7 Why is column-oriented storage potentially advantageous in a database system that supports a data warehouse?

Answer:
No answer

11.8 Consider each of the *takes* and *teaches* relations as a fact table; they do not have an explicit measure attribute, but assume each table has a measure attribute *reg_count* whose value is always 1. What would the dimension attributes and dimension tables be in each case. Would the resultant schemas be star schemas or snowflake schemas?

Answer:
The *takes* fact table has dimension tables *student*, *course*, *semester*, and *year*; *student* and *course* link further to department. Thus the scheme is a snowflake schema. The case of *takes* is similar.

11.9 Consider the star schema from Figure 11.2. Suppose an analyst finds that monthly total sales (sum of the *price* values of all *sales* tuples) have decreased, instead of growing, from April 2018 to May 2018. The analyst wishes to check if there are specific item categories, stores, or customer countries that are responsible for the decrease.

a. What are the aggregates that the analyst would start with, and what are the relevant drill-down operations that the analyst would need to execute?

b. Write an SQL query that shows the item categories that were responsible for the decrease in sales, ordered by the impact of the category on the sales decrease, with categories that had the highest impact sorted first.

Answer:

Aggregate sales by month for the range April 2019 to May 2018. Drill down (separately) on attributes store_id, customer.country, and item.category.
Write a query that computes the total sales decrease for each category, and order on that attribute.

11.10 Suppose half of all the transactions in a clothes shop purchase jeans, and one-third of all transactions in the shop purchase T-shirts. Suppose also that half of the transactions that purchase jeans also purchase T-shirts. Write down all the (nontrivial) association rules you can deduce from the above information, giving support and confidence of each rule.

Answer:
The rules are as follows. The last rule can be deduced from the previous ones.

Rule	Support	Conf.
$\forall$ *transactions* T, **true** $\Rightarrow$ *buys*$(T, jeans)$	50%	50%
$\forall$ *transactions* T, **true** $\Rightarrow$ *buys*$(T, t\text{-}shirts)$	33%	33%
$\forall$ *transactions* T, *buys*$(T, jeans) \Rightarrow$ *buys*$(T, t\text{-}shirts)$	25%	50%
$\forall$ *transactions* T, *buys*$(T, t\text{-}shirts) \Rightarrow$ *buys*$(T, jeans)$	25%	75%

11.11 The organization of parts, chapters, sections, and subsections in a book is related to clustering. Explain why, and to what form of clustering.

Answer:
The organization of a book's content is a form of **hierarchical clustering**. Contents within a single subsection are closely related, whereas different parts of a book cover a more diverse range of topics.

11.12 Suggest how predictive mining techniques can be used by a sports team, using your favorite sport as an example.

Answer:
Given the large amount of statistics collected during and about sporting events, there are many ways a sports team can make use of predictive data mining:

- Some players may be more effective in certain situations or environments, so data mining can predict when each player should be used.
- Specific strategies may be more effective against certain teams or during certain situations in the game.
- Predictive mining can estimate the outcome of a match beforehand, information which could be useful to a team before entering a tournament.

CHAPTER 12

Physical Storage Systems

Exercises

12.6 List the physical storage media available on the computers you use routinely. Give the speed with which data can be accessed on each medium.

Answer:
Your answer will be based on the computers and storage media that you use. Typical examples would be hard disk, CD and DVD disks, and flash memory in the form of USB keys, memory cards, or solid-state disks.

The following table shows the typical transfer speeds for the above mentioned storage media, as of early 2010.

Storage Media	Speed (in MB/s)
CD Drive	8
DVD Drive	20
USB Keys	30
Memory Cards	1 - 40
Hard Disk	100
Solid-State Disks	> 100

Note that speeds of flash memory cards can vary significantly, with some low-end cards giving low transfer speeds, although better ones give much higher transfer speeds.

12.7 How does the remapping of bad sectors by disk controllers affect data-retrieval rates?

Answer:
Remapping of bad sectors by disk controllers does reduce data-retrieval rates because of the loss of sequentiality among the sectors. But that is better than the loss of data in case of no remapping!

12.8 Operating systems try to ensure that consecutive blocks of a file are stored on consecutive disk blocks. Why is doing so very important with magnetic disks? If SSDs were used instead, is doing so still important, or is it irrelevant? Explain why.

Answer:
While seek times are very small on SSDs as compared to magnetic disks, data transfer rates with sequential I/O are still much faster on SSDs as compared to random I/O on SSDs. Thus, keeping file blocks sequential on disk continues to be important with SSDs, although the impact of not doing so would not be as drastic as it would be with magnetic disk.

12.9 RAID systems typically allow you to replace failed disks without stopping access to the system. Thus, the data in the failed disk must be rebuilt and written to the replacement disk while the system is in operation. Which of the RAID levels yields the least amount of interference between the rebuild and ongoing disk accesses? Explain your answer.

Answer:
RAID level 1 (mirroring) is the one which facilitates rebuilding of a failed disk with minimum interference with the ongoing disk accesses. This is because rebuilding in this case involves copying data from just the failed disk's mirror. In the other RAID levels, rebuilding involves reading the entire contents of all the other disks.

12.10 What is scrubbing, in the context of RAID systems, and why is scrubbing important?

Answer:
Successfully written sectors which are subsequently damaged, but where the damage has not been detected, are referred to as latent sector errors. In RAID systems, latent errors can lead to data loss even on a single disk failure, if the latent error exists on one of the other disks. Disk scrubbing is a background process that reads disk sectors during idle periods, with the goal of detecting latent sector errors. If a sector error is found, the sector can either be rewritten if the media have not been damaged, or remapped to a spare sector in the disk. The data in the sector can be recovered from the other disks in the RAID array.

12.11 Suppose you have data that should not be lost on disk failure, and the application is write-intensive. How would you store the data?

Answer:
A RAID array can handle the failure of a single drive (two drives in the case of RAID 6) without data loss and is relatively inexpensive. There are several RAID alternatives, each with different performance and cost implications. For write-intensive data with mostly sequential writes, RAID 1 and RAID 5 will both perform well, but with less storage overhead for RAID 5. If writes are random,

RAID 1 is preferred, since a random block write requires multiple reads and writes in RAID 5. If you wish to protect from two-disk failure, using RAID 1 but with three-way replication instead of mirroring is an option.

CHAPTER 13

Data Storage Structures

Exercises

13.9 In the variable-length record representation, a null bitmap is used to indicate if an attribute has the null value.

a. For variable-length fields, if the value is null, what would be stored in the offset and length fields?

b. In some applications, tuples have a very large number of attributes, most of which are null. Can you modify the record representation such that the only overhead for a null attribute is the single bit in the null bitmap?

Answer:

a. It does not matter on what we store in the offset and length fields since we are using a null bitmap to identify null entries. But it would make sense to set the offset and length to zero to avoid having arbitrary values.

b. We should be able to locate the null bitmap and the offset and length values of non-null attributes using the null bitmap. This can be done by storing the null bitmap at the beginning and then for non-null attributes, store the value (for fixed size attributes), or offset and length values (for variable sized attributes) in the same order as in the bitmap, followed by the values for non-null variable sized attributes. This representation is space efficient but needs extra work to retrieve the attributes.

13.10 Explain why the allocation of records to blocks affects database-system performance significantly.

Answer:

If we allocate related records to blocks, we can often retrieve most, or all, of the requested records by a query with one disk access. Disk accesses tend to be the bottlenecks in databases; since this allocation strategy reduces the number of disk accesses for a given operation, it significantly improves performance.

13.11 List two advantages and two disadvantages of each of the following strategies for storing a relational database:

a. Store each relation in one file.

b. Store multiple relations (perhaps even the entire database) in one file.

Answer:

a. Advantages of storing a relation as a file include using the file system provided by the OS , thus simplifying the DBMS, but incurs the disadvantage of restricting the ability of the DBMS to increase performance by using more sophisticated storage structures.

b. By using one file for the entire database, these complex structures can be implemented through the DBMS, but this increases the size and complexity of the DBMS.

13.12 In the sequential file organization, why is an overflow *block* used even if there is, at the moment, only one overflow record?

Answer:
An overflow block is used in sequential file organization because a block is the abbrfont space which can be read from a disk. Therefore, using any smaller region would not be useful from a performance standpoint. The space saved by allocating disk storage in record units would be overshadowed by the performance cost of allowing blocks to contain records of multiple files.

13.13 Give a normalized version of the *Index_metadata* relation, and explain why using the normalized version would result in worse performance.

Answer:
The *Index_metadata* relation can be normalized as follows:

Index_metadata(index_name, relation_name, index_type)
Index_Attrib_metadata (index-name, position, attribute-name)

The normalized version will require extra disk accesses to read *Index_Attrib_metadata* everytime an index has to be accessed. Thus, it will lead to worse performance.

13.14 Standard buffer managers assume each block is of the same size and costs the same to read. Consider a buffer manager that, instead of LRU, uses the rate of reference to objects, that is, how often an object has been accessed in the last n seconds. Suppose we want to store in the buffer objects of varying sizes, and varying read costs (such as web pages, whose read cost depends on the site from which they are fetched). Suggest how a buffer manager may choose which block to evict from the buffer.

Answer:
A good solution would make use of a *priority queue* to evict pages, where the priority (p) is ordered by the *expected cost* of re-reading a page given it's past access frequency (f) in he last n seconds, it's re-read cost (c), and its size s:

$$p = f * c/s$$

The buffer manager should choose to evict pages with the lowest value of p, until there is enough free space to read in a newly referenced object.

CHAPTER 14

Indexing

Exercises

14.16 When is it preferable to use a dense index rather than a sparse index? Explain your answer.

Answer:
It is preferable to use a dense index instead of a sparse index when the file is not sorted on the indexed field (such as when the index is a secondary index) or when the index file is small compared to the size of memory.

14.17 What is the difference between a clustering index and a secondary index?

Answer:
The clustering index is on the field which specifies the sequential order of the file. There can be only one clustering index, while there can be many secondary indices.

14.18 For each B^+-tree of Exercise 14.3, show the steps involved in the following queries:

a. Find records with a search-key value of 11.

b. Find records with a search-key value between 7 and 17, inclusive.

Answer:

With the structure provided by the solution to Practice Exercise 14.3a:

a. Find records with a value of 11
   i. Search the first-level index; follow the first pointer.
   ii. Search next level; follow the third pointer.
   iii. Search leaf node; follow first pointer to records with key value 11.

b. Find records with value between 7 and 17 (inclusive)
   i. Search top index; follow first pointer.

ii. Search next level; follow second pointer.
iii. Search third level; follow second pointer to records with key value 7, and after accessing them, return to leaf node.
iv. Follow fourth pointer to next leaf block in the chain.
v. Follow first pointer to records with key value 11, then return.
vi. Follow second pointer to records with with key value 17.

With the structure provided by the solution to Practice Exercise 14.3b:

a. Find records with a value of 11
   i. Search top level; follow second pointer.
   ii. Search next level; follow second pointer to records with key value 11.
b. Find records with value between 7 and 17 (inclusive)
   i. Search top level; follow second pointer.
   ii. Search next level; follow first pointer to records with key value 7, then return.
   iii. Follow second pointer to records with key value 11, then return.
   iv. Follow third pointer to records with key value 17.

With the structure provided by the solution to Practice Exercise 14.3c:

a. Find records with a value of 11
   i. Search top level; follow second pointer.
   ii. Search next level; follow first pointer to records with key value 11.
b. Find records with value between 7 and 17 (inclusive)
   i. Search top level; follow first pointer.
   ii. Search next level; follow fourth pointer to records with key value 7, then return.
   iii. Follow eighth pointer to next leaf block in chain.
   iv. Follow first pointer to records with key value 11, then return.
   v. Follow second pointer to records with key value 17.

14.19 The solution presented in Section 14.3.5 to deal with nonunique search keys added an extra attribute to the search key. What effect could this change have on the height of the B^+-tree?

Answer:
The resultant B-tree's extended search key is unique. This results in a greater number of nodes. A single node (which points to mutiple records with the same key) in the original tree may correspond to multiple nodes in the result

tree. Depending on how they are organized, the height of the tree may increase; it might be more than that of the original tree.

14.20 Suppose there is a relation $r(A, B, C)$, with a B^+-tree index with search key (A, B).

a. What is the worst-case cost of finding records satisfying $10 < A < 50$ using this index, in terms of the number of records retrieved n_1 and the height h of the tree?

b. What is the worst-case cost of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$ using this index, in terms of the number of records n_2 that satisfy this selection, as well as n_1 and h defined above?

c. Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$?

Answer:

a. This query does not correspond to a range query on the search key as the condition on the first attribute if the search key is a comparison condition. It looks up records which have the value of A between 10 and 50. However, each record is likely to be in a different block, because of the ordering of records in the file, leading to many I/O operations. In the worst case, for each record, it needs to traverse the whole tree (cost is h), so the total cost is $n_1 * h$.

b. This query can be answered by using an ordered index on the search key (A, B). For each value of A this is between 10 and 50, and the system located records with B value between 5 and 10. However, each record could is likely to be in a different disk block. This amounts to executing the query based on the condition on A, and this costs $n_1 * h$. Then these records are checked to see if the condition on B is satisfied. So, the total cost in the worst case is $n_1 * h$.

c. n_1 records satisfy the first condition and n_2 records satisfy the second condition. When both the conditions are queried, n_1 records are output in the first stage. So, in the case where $n_1 = n_2$, no extra records are output in the furst stage. Otherwise, the records which don't satisfy the second condition are also output with an additional cost of h each (worst case).

14.21 Suppose you have to create a B^+-tree index on a large number of names, where the maximum size of a name may be quite large (say 40 characters) and the average name is itself large (say 10 characters). Explain how prefix compression can be used to maximize the average fanout of nonleaf nodes.

Answer:

There are two problems in the given scenario. The first problem is names can be of variable length. The second problem is names can be long (maximum is 40 characters), leading to a low fanout and a correspondingly increased tree height. With variable-length search keys, different nodes can have different fanouts even if they are full. The fanout of nodes can be increased by using a technique called prefix compression. With prefix compression, the entire search key value is not stored at internal nodes, only a prefix of each search-key sufficient to distinguish between the key values in the subtrees that it separates. The full name can be stored in the leaf nodes; this way we don't lose any information, and we also maximize the average fanout of internal nodes.

14.22 Suppose a relation is stored in a B^+-tree file organization. Suppose secondary indices store record identifiers that are pointers to records on disk.

a. What would be the effect on the secondary indices if a node split happened in the file organization?

b. What would be the cost of updating all affected records in a secondary index?

c. How does using the search key of the file organization as a logical record identifier solve this problem?

d. What is the extra cost due to the use of such logical record identifiers?

Answer:

a. When a leaf page is split in a B^+-tree file organization, a number of records are moved to a new page. In such cases, all secondary indices that store pointers to the relocated records would have to be updated, even though the values in the records may not have changed.

b. Each leaf page may contain a fairly large number of records, and each of them may be in different locations on each secondary index. Thus, a leaf-page split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.

c. One solution is to store the values of the primary-index search key attributes in secondary indices, in place of pointers to the indexed records. Relocation of records because of leaf-page splits then does not require any update on any secondary index.

d. Locating a record using the secondary index now requires two steps: First we use the secondary index to find the primary index search-key values, and then we use the primary index to find the corresponding records. This approach reduces the cost of index update due to file reorganization, although it increases the cost of accessing data using a secondary index.

14.23 What trade-offs do write-optimized indices pose as compared to B^+-tree indices?

Answer:
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.

14.24 An *existence bitmap* has a bit for each record position, with the bit set to 1 if the record exists, and 0 if there is no record at that position (for example, if the record were deleted). Show how to compute the existence bitmap from other bitmaps. Make sure that your technique works even in the presence of null values by using a bitmap for the value *null*.

Answer:
The existence bitmap for a relation can be calculated by taking the union (logical-or) of all the bitmaps on that attribute, including the bitmap for value *null*.

14.25 Spatial indices that can index spatial intervals can conceptually be used to index temporal data by treating valid time as a time interval. What is the problem with doing so, and how is the problem solved?

Answer:
The problem is with time intervals whose end time is infinity; such intervals are used for facts that are currently valid and remain valid until they are updated. Spatial indices are generally designed to only handle finite intervals. A solution is to index tuples with an infinite end time separately from other tuples; an index on the start time suffices for such unbounded time tuples. Tuples that have a finite end time can be indexed using a spatial index such as an R-tree or an interval tree (if it is available).

14.26 Some attributes of relations may contain sensitive data, and may be required to be stored in an encrypted fashion. How does data encryption affect index schemes? In particular, how might it affect schemes that attempt to store data in sorted order?

Answer:
Note that indices must operate on the encrypted data or someone could gain access to the index to interpret the data. Otherwise, the index would have to be restricted so that only certain users could access it. To keep the data in sorted order, the index scheme would have to decrypt the data at each level in a tree. Note that hash systems would not be affected.

CHAPTER 15

Query Processing

Exercises

15.17 Suppose you need to sort a relation of 40 gigabytes, with 4-kilobyte blocks, using a memory size of 40 megabytes. Suppose the cost of a seek is 5 milliseconds, while the disk transfer rate is 40 megabytes per second.

a. Find the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$.

b. In each case, how many merge passes are required?

c. Suppose a flash storage device is used instead of a disk, and it has a latency of 20 microsecond and a transfer rate of 400 megabytes per second. Recompute the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$, in this setting.

Answer:

a. The number of blocks in the main memory buffer available for sorting(M) is $\frac{40\times10^6}{4\times10^3} = 10^4$. The number of blocks containing records of the given relation (b_r) is $\frac{40\times10^9}{4\times10^3} = 10^7$. Then the cost of sorting the relation is: (*Number of disk seeks* × *Disk seek cost*) + (*Number of block transfers* × *Block transfer time*). Here Disk seek cost is $5x10^{-3}$ seconds and Block transfer time is 10^{-4} seconds ($\frac{4\times10^3}{40\times10^6}$). The number of block transfers is independent of b_b and is equal to 25×10^6.

- **Case 1**: $b_b = 1$
  Using the equation in Section 15.4, the number of disk seeks is 5002×10^3. Therefore the cost of sorting the relation is: $(5002 \times 10^3) \times (5 \times 10^{-3}) + (25 \times 10^6) \times (10^{-4}) = 25 \times 10^3 + 2500 = 27500$ seconds.
- **Case 2:** $b_b = 100$

The number of disk seeks is: 52×10^3. Therefore the cost of sorting the relation is: $(52 \times 10^3) \times (5 \times 10^{-3}) + (25 \times 10^6) \times (10^{-4}) = 260 + 2500 = 2760$ seconds.

b. Disk storage The number of merge passes required is given by $\lceil log_{M-1}(\frac{b_r}{M}) \rceil$. This is independent of b_b. Substituting the values above, we get $\lceil log_{10^4-1}(\frac{10^7}{10^4}) \rceil$, which evaluates to 1.

c. Flash storage: Here latency is 20 microseconds, and block transfer time is $4000/400,000,000 = 10^{-5}$.

- **Case 1**: $b_b = 1$
  The number of disk seeks is: 5002×10^3. Therefore, the cost of sorting the relation is: $(5002 \times 10^3) \times (20 \times 10^{-6}) + (25 \times 10^6) \times (10^{-5}) = 100.04 + 250 = 350.04$ seconds.
- **Case 2**: $b_b = 100$
  The number of disk seeks is: 52×10^3. Therefore, the cost of sorting the relation is: $(52 \times 10^3) \times (20 \times 10^{-6}) + (25 \times 10^6) \times (10^{-5}) = 1.4 + 250$, which is approx $= 251.4$ seconds.

15.18 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

Answer:
In general it is not desirable to force users to choose a query- processing strategy because naive users might select an inefficient strategy. The reason users would make poor choices about processing queries is that they would not know how a relation is stored, nor about its indices. It is unreasonable to force users to be aware of these details since ease of use is a major object of database query languages. If users are aware of the costs of different strategies, they could write queries efficiently, thus helping performance. This could happen if experts were using the system.

15.19 Design a variant of the hybrid merge-join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

Answer:
We merge the leaf entries of the first sorted secondary index with the leaf entries of the second sorted secondary index. The result file contains pairs of addresses, the first address in each pair pointing to a tuple in the first relation, and the second address pointing to a tuple in the second relation.
This result file is first sorted on the first relation's addresses. The relation is then scanned in physical storage order, and addresses in the result file are replaced by the actual tuple values. Then the result file is sorted on the second

relation's addresses, allowing a scan of the second relation in physical storage order to complete the join.

15.20 Estimate the number of block transfers and seeks required by your solution to Exercise 15.19 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Exercise 15.3.

Answer:

r_1 occupies 800 blocks, and r_2 occupies 1500 blocks. Let there be n pointers per index leaf block (we assume that both the indices have leaf blocks and pointers of equal sizes). Let us assume M pages of memory, $M < 800$. r_1's index will need $B_1 = \lceil \frac{20000}{n} \rceil$ leaf blocks, and r_2's index will need $B_2 = \lceil \frac{45000}{n} \rceil$ leaf blocks. Therefore, the merge join will need $B_3 = B_1 + B_2$ accesses, without output. The number of output tuples is estimated as $n_o = \lceil \frac{20000 * 45000}{max(V(C,r_1),V(C,r_2))} \rceil$. Each output tuple will need two pointers, so the number of blocks of join output will be $B_{o1} = \lceil \frac{n_o}{n/2} \rceil$. Hence the join needs $B_j = B_3 + B_{o1}$ disk block accesses.

Now we have to replace the pointers with actual tuples. For the first sorting, $B_{s1} = B_{o1}(2\lceil log_{M-1}(B_{o1}/M) \rceil + 2)$ disk accesses are needed, including the writing of output to disk. The number of blocks of r_1 which have to be accessed in order to replace the pointers with tuple values is $min(800, n_o)$. Let n_1 pairs of the form (r_1 *tuple, pointer to* r_2) fit in one disk block. Therefore, the intermediate result after replacing the r_1 pointers will occupy $B_{o2} = \lceil (n_o/n_1) \rceil$ blocks. Hence the first pass of replacing the r_1 pointers will cost $B_f = B_{s1} + B_{o1} + min(800, n_o) + B_{o2}$ disk accesses.

The second pass for replacing the r_2 pointers has a similar analysis. Let n_2 tuples of the final join fit in one block. Then the second pass of replacing the r_2 pointers will cost $B_s = B_{s2} + B_{o2} + min(1500, n_o)$ disk accesses, where $B_{s2} = B_{o2}(2\lceil log_{M-1}(B_{o2}/M) \rceil + 2)$.

Hence the total number of disk accesses for the join is $B_j + B_f + B_s$, and the number of pages of output is $\lceil n_o/n_2 \rceil$.

15.21 The hash-join algorithm as described in Section 15.5.5 computes the natural join of two relations. Describe how to extend the hash-join algorithm to compute the natural left outer join, the natural right outer join, and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *takes* and *student* relations.

Answer:

For the probe relation tuple t_r under consideration, if no matching tuple is found in the build relation's hash partition, it is padded with nulls and included in the result. This will give us the natural left outer join $t_r \bowtie t_s$. To get the natural right outer join $t_r \bowtie t_s$, we can keep a boolean flag with each tuple in the current build relation partition s_i residing in memory, and set it whenever any probe relation tuple matches with it. When we are finished with s_i, all the

tuples in it which do not have their flag set are padded with nulls and included in the result. To get the natural full outer join, we do both the above operations together.

To try out our algorithm, we use the sample *student* and *takes* relations of Figure A.8 and Figure A.9. Let us assume that there is enough memory to hold three tuples of the build relation plus a hash index for those three tuples. We use *takes* as the build relation. We use the simple hashing function which returns the *student_ID* mod 10. Taking the partition corresponding to value 7, we get $r_1 = \{(\text{“Snow”})\}$, and $s_1 = \phi$. The tuple in the probe relation partition will have no matching tuple, so (“70557”, “Snow”, “Physics”,“0”, *null*) is outputted. Proceeding in a similar way, we process all the partitions and complete the join.

15.22 Suppose you have to compute ${}_A\gamma_{sum(C)}(r)$ as well as ${}_{A,B}\gamma_{sum(C)}(r)$. Describe how to compute these together using a single sorting of r.

Answer:
Run the sorting operation on r, grouping by (A, B), as required for the second result. When evaluating the sum aggregate, keep running totals for both the (A, B) grouping as well as for just the A grouping.

15.23 Write pseudocode for an iterator that implements a version of the sort – merge algorithm where the result of the final merge is pipelined to its consumers. Your pseudocode must define the standard iterator functions *open*(), *next*(), and *close*(). Show what state information the iterator must maintain between calls.

Answer:
Let M denote the number of blocks in the main memory buffer available for sorting. For simplicity we assume that there are less than M runs created in the run creation phase. The pseudocode for the iterator functions open, next, and close are as shown in Figure 15.101.

15.24 Explain how to split the hybrid hash-join operator into sub-operators to model pipelining. Also explain how this split is different from the split for a hash-join operator.

Answer:
There is one operator to partition the build input, which also creates the in-memory index for partition 0. The other operator partitions the probe input and also performs the probe operators for partition 0. The edge between the two operators is a blocking edge.

15.25 Suppose you need to sort relation r using sort–merge and merge–join the result with an already sorted relation s.

a. Describe how the sort operator is broken into suboperators to model the pipelining in this case.

```
SortMergeJoin::open()
begin
    repeat
        read M blocks of the relation;
        sort the in-memory part of the relation;
        write the sorted data to a run file R_i
    until the end of the relation
    read one block of each of the N run files R_i, into a
        buffer block in memory
    done_r := false;
end

SortMergeJoin::close()
begin
    clear all the N runs from main memory and disk;
end

boolean SortMergeJoin::next()
begin
    if the buffer block of any run R_i is empty and not end-of-file(R_i)
        begin
            read the next block of R_i (if any) into the buffer block;
        end
    if all buffer blocks are empty
        return false;
    choose the first tuple (in sort order) among the buffer blocks;
    write the tuple to the output buffer;
    delete the tuple from the buffer block and increment its pointer;
    return true;
end
```

Figure 15.101 Pseudocode for Exercise 15.23.

b. The same idea is applicable even if both inputs to the merge join are the outputs of sort–merge operations. However, the available memory has to be shared between the two merge operations (the merge–join algorithm itself needs very little memory). What is the effect of having to share memory on the cost of each sort-merge operation?

Answer:

a. Sort is broken into run generation and merge. The merge step is pipelined with the join operation.

b. If the sort – merge operations are run in parallel and memory is shared equally between the two, each operation will have only $M/2$ frames for its memory buffer. This may increase the number of runs required to merge the data.

CHAPTER 16

Query Optimization

Exercises

16.16 Suppose that a B$^+$-tree index on (*dept_name*, *building*) is available on relation *department*. What would be the best way to handle the following selection?

$$\sigma_{(building < \text{“Watson”}) \wedge (budget < 55000) \wedge (dept\_name = \text{“Music”})}(department)$$

Answer:
Using the index on (*dept_name*, *building*), we locate the first tuple having (*building* “Watson” and *dept_name*“Music”). We then follow the pointers retrieving successive tuples as long as *building* is less than “Watson”. From the tuples retrieved, the ones not satisfying the condition (*budget* < 55000) are rejected.

16.17 Show how to derive the following equivalences by a sequence of transformations using the equivalence rules in Section 16.2.1.

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, where θ_2 involves only attributes from E_2

Answer:

a. Using rule 1, $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E)$ becomes $\sigma_{\theta_1}(\sigma_{\theta_2 \wedge \theta_3}(E))$. On applying rule 1 again, we get $\sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$.

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2)$ on applying rule 1 becomes $\sigma_{\theta_1}(\sigma_{\theta_2}(E_1 \bowtie_{\theta_3} E_2))$. This on applying rule 7.a becomes $\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$.

16.18 Consider the two expressions $\sigma_\theta(E_1 \rtimes E_2)$ and $\sigma_\theta(E_1 \bowtie E_2)$.

a. Show using an example that the two expressions are not equivalent in general.

b. Give a simple condition on the predicate θ, which if satisfied will ensure that the two expressions are equivalent.

Answer:

a. Consider relations *dept*(*id, deptname*) and *emp*(*id,name,dept_id*) with sample data as follows:

Sample data for *dept*:

501	Finance
502	Administration
503	Marketing
504	Sales

Sample data for *emp*:

1	John 501
2	Martin 503
3	Sarah 504

Now consider the expressions

$\sigma_{deptname<'Z'}(dept \bowtie emp)$ and

$\sigma_{deptname<'Z'}(dept \rtimes emp)$.

The result of the first expression is:

501	Finance	1	John
503	Marketing	2	Martin
504	Sales	3	Sarah

whereas the result of the second expression is:

501	Finance	1	John
502	Administration	null	null
503	Marketing	2	Martin
504	Sales	3	Sarah

b. Considering the same example, if θ included the condition "*name* = 'E-instein'", the two expressions would be equivalent, that is they would always have the same result, since any tuple that is in in $dept \rtimes emp$ but not in $dept \bowtie emp$ would not satisfy the condition since its *name* attribute would be null.

16.19 A set of equivalence rules is said to be *complete* if, whenever two expressions are equivalent, one can be derived from the other by a sequence of uses of the

equivalence rules. Is the set of equivalence rules that we considered in Section 16.2.1 complete? Hint: Consider the equivalence $\sigma_{3=5}(r) \equiv \{\ \}$.

Answer:
Two relational expressions are defined to be *equivalent* when on all input relations, they give the same output. The set of equivalence rules considered in Section 16.2.1 is not complete. The expressions $\sigma_{3=5}(r)$ and $\{\ \}$ are equivalent, but this cannot be shown by using just these rules.

16.20 Explain how to use a histogram to estimate the size of a selection of the form $\sigma_{A \leq v}(r)$.

Answer:
Suppose the histogram H storing the distribution of values in r is divided into ranges $r_1, \ldots, r_n$. For each range r_i with low value $r_{i:low}$ and high value $r_{i:high}$, if $r_{i:high}$ is less than v, we add the number of tuples given by

$$H(r_i)$$

to the estimated total. If $v < r_{i:high}$ and $v >= r_{i:low}$, we assume that values within r_i are uniformly distributed, and we add

$$H(r_i) * \frac{v - r_{i:low}}{r_{i:high} - r_{i:low}}$$

to the estimated total.

16.21 Suppose two relations r and s have histograms on attributes $r.A$ and $s.A$, respectively, but with different ranges. Suggest how to use the histograms to estimate the size of $r \bowtie s$. Hint: Split the ranges of each histogram further.

Answer:
Find the largest unit u that evenly divides the range size of both histograms. Divide each histogram into ranges of size u, assuming that values within a range are uniformly distributed. Then compute the estimated join size using the technique for histograms with equal range sizes.

16.22 Consider the query

```
select A, B
from   r
where r.B < some (select B
                  from   s
                  where s.A = r.A)
```

Show how to decorrelate this query using the multiset version of the semi join operation.

Answer:

The solution can be written in relational algebra as follows: $\Pi_{A,B}(r \ltimes \theta s)$ where $\theta = (r.B < s.B \wedge s.A = r.A)$.
The SQL query corresponding to this can be written if the database provides a semi join operator, and this varies across implementations.

16.23 Describe how to incrementally maintain the results of the following operations on both insertions and deletions:

a. Union and set difference.

b. Left outer join.

Answer:

a. Given materialized view $v = r \cup s$, when a tuple is inserted in r, we check if it is present in v, and if not we add it to v. When a tuple is deleted from r, we check if it is there in s, and if not, we delete it from v. Inserts and deletes in s are handled in symmetric fashion.

For set difference, given view $v = r - s$, when a tuple is inserted in r, we check if it is present in s, and if not we add it to v. When a tuple is deleted from r, we delete it from v if present. When a tuple is inserted in s, we delete it from v if present. When a tuple is deleted from s, we check if it is present in r, and if so we add it to v.

b. Given materialized view $v = r \rtimes s$, when a set of tuples i_r is inserted in r, we add the tuples $i_r \rtimes s$ to the view. When i_r is deleted from r, we delete $i_r \rtimes s$ from the view. When a set of tuples i_s is inserted in s, we compute $r \bowtie i_s$. We find all the tuples of r which previously did not match any tuple from s (i.e., those padded with *null* in $r \rtimes s$) but which match i_s. We remove all those *null* padded entries from the view and add the tuples $r \bowtie s$ to the view. When i_s is deleted from s, we delete the tuples $r \bowtie i_s$ from the view. Also, we find all the tuples in r which match i_s but which do not match any other tuples in s. We add all those to the view, after padding them with *null* values.

16.24 Give an example of an expression defining a materialized view and two situations (sets of statistics for the input relations and the differentials) such that incremental view maintenance is better than recomputation in one situation, and recomputation is better in the other situation.

Answer:
Let r and s be two relations. Consider a materialized view on these defined by $(r \bowtie s)$. Suppose 70 percent of the tuples in r are deleted. Then recomputation is better than incremental view maintenance. This is because in incremental view maintenance, the 70 percent of the deleted tuples have to be joined with s while in recomputation, just the remaining 30 percent of the tuples in r have to be joined with s.

However, if the number of tuples in r is increased by a small percentage, for example 2%, then incremental view maintenance is likely to be better than recomputation.

16.25 Suppose you want to get answers to $r \bowtie s$ sorted on an attribute of r, and want only the top K answers for some relatively small K. Give a good way of evaluating the query:

a. When the join is on a foreign key of r referencing s, where the foreign key attribute is declared to be not null.

b. When the join is not on a foreign key.

Answer:

a. Sort r and collect the top K tuples. These tuples are guaranteed to be contained in $r \bowtie s$ since the join is on a foreign key of r referencing s.

b. Execute $r \bowtie s$ using a standard join algorithm until the first K results have been found. After K tuples have been computed in the result set, continue executing the join, but immediately discard any tuples from r that have attribute values less than all of the tuples in the result set. If a newly joined tuple t has an attribute value greater than at least one of the tuples in the result set, replace the lowest-valued tuple in the result set with t.

16.26 Consider a relation $r(A, B, C)$, with an index on attribute A. Give an example of a query that can be answered by using the index only, without looking at the tuples in the relation. (Query plans that use only the index, without accessing the actual relation, are called *index-only* plans.)

Answer:
Any query that only involves the attribute A of r can be executed by only using the index. For example, the query

select sum(A)
from r

only needs to use the values of A, and thus does not need to look at r.

16.27 Suppose you have an update query U. Give a simple sufficient condition on U that will ensure that the Halloween problem cannot occur, regardless of the execution plan chosen or the indices that exist.

Answer:
The attributes referred in the WHERE clause of the update query U should not be a part of the SET clauses of U. This will ensure that the Halloween problem cannot occur.

CHAPTER 17

Transactions

Exercises

17.12 List the ACID properties. Explain the usefulness of each.

Answer:
The ACID properties and the need for each of them are:

- **Consistency:** Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.
- **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are. Clearly lack of atomicity will lead to inconsistency in the database.
- **Isolation:** When multiple transactions execute concurrently, it should be the case that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently with it. The user view of a transaction system requires the isolation property and the property that concurrent schedules take the system from one consistent state to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency-preserving transactions are allowed.
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

17.13 During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.

Answer:
The possible sequences of states are:

a. *active* → *partially committed* → *committed*. This is the normal sequence a successful transaction will follow. After executing all its statements, it enters the *partially committed* state. After enough recovery information has been written to disk, the transaction finally enters the *committed* state.

b. *active* → *partially committed* → *failed* → *aborted*. After executing the last statement of the transaction, it enters the *partially committed* state. But before enough recovery information is written to disk, a hardware failure may occur, destroying the memory contents. In this case the changes that it made to the database are undone, and the transaction enters the *failed* state. It is then rolled back, after which it enters the *aborted* state.

c. *active* → *failed* → *aborted*. After the transaction starts, if it is discovered at some point that normal execution cannot continue (either due to internal program errors or external errors), it enters the failed state. It is then rolled back, after which it enters the *aborted* state.

17.14 Explain the distinction between the terms *serial schedule* and *serializable schedule*.

Answer:
A schedule in which all the instructions belonging to one single transaction appear together is called a *serial schedule*. A *serializable schedule* has a weaker restriction that it should be *equivalent* to some serial schedule. There are two definitions of schedule equivalence – conflict equivalence and view equivalence.

17.15 Consider the following two transactions:

T_{13}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{14}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ as the initial values.

a. Show that every serial execution involving these two transactions preserves the consistency of the database.

b. Show a concurrent execution of T_{13} and T_{14} that produces a nonserializable schedule.

c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?

Answer:

a. There are two possible executions: T_{13} T_{14} and T_{14} T_{13}.

Case 1:

	A	B
initially	0	0
after T_{13}	0	1
after T_{14}	0	1

Consistency met: $A = 0 \vee B = 0 \equiv T \vee F = T$

Case 2:

	A	B
initially	0	0
after T_{14}	1	0
after T_{13}	1	0

Consistency met: $A = 0 \vee B = 0 \equiv F \vee T = T$

b. Any interleaving of T_{13} and T_{14} results in a nonserializable schedule.

T_{13}	T_{14}
read(A)	
	read(B)
	read(A)
read(B)	
if $A = 0$ **then** $B = B + 1$	
	if $B = 0$ **then** $A = A + 1$
	write(A)
write(B)	

c. There is no parallel execution resulting in a serializable schedule. From part *a*. we know that a serializable schedule results in $A = 0 \vee B = 0$. Suppose we start with T_{13} **read**(A). Then when the schedule ends, no matter when we run the steps of T_2, $B = 1$. Now suppose we start executing T_{14} prior to completion of T_{13}. Then T_2 **read**(B) will give B a value of 0. So when T_2 completes, $A = 1$. Thus $B = 1 \wedge A = 1 \rightarrow \neg (A = 0 \vee B = 0)$. Similarly for starting with T_{14} **read**(B).

17.16 Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.

Answer:

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

As we can see, the above schedule is serializable with an equivalent serial schedule T_1, T_2. In the above schedule, T_2 commits before T_1.

Note that T_1 could have an unlock instruction right after its read instruction and that, as a result, this schedule can occur even with strict two-phase locking, where exclusive locks are held to commit, but shared locks can be released early in two-phase manner.

17.17 What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow nonrecoverable schedules? Explain your answer.

Answer:

A recoverable 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 commit operation of T_j. Recoverable schedules are desirable because failure of a transaction might otherwise bring the system into an irreversibly inconsistent state. For this reason, one could argue that nonrecoverable schedules are never acceptable. Although not mentioned in the chapter, nonrecoverable schedules may sometimes be needed when updates must be made visible early due to time constraints, even if they have not yet been committed, which may be required for very long-duration transactions.

17.18 Why do database systems support concurrent execution of transactions, despite the extra effort needed to ensure that concurrent execution does not cause any problems?

Answer:

Transaction-processing systems usually allow multiple transactions to run concurrently. It is far easier to insist that transactions run serially. However there are three good reasons for allowing concurrency:

- Improved throughput and resource utilization. A transaction may involve I/O activity and CPU activity. The CPU and the disk in a computer system can operate in parallel. This can be exploited to run multiple transactions in parallel. For example, while a read or write on behalf of one transaction

is in progress on one disk, another transaction can be running in the CPU. This increases the throughput of the system.

- Reduced waiting time. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses. It reduces unpredictable delays and the average response time.
- The parallel architecture of virtually all modern computers.

17.19 Explain why the read-committed isolation level ensures that schedules are cascade-free.

Answer:

The read-committed isolation level ensures that a transaction reads only the committed data. A transaction T_i cannot read a data item X which has been modified by a yet uncommitted concurrent transaction T_j. This makes T_i independent of the success or failure of T_j. Hence, the schedules which follow read-committed isolation level become cascade-free.

17.20 For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation but is not serializable:

a. Read uncommitted

b. Read committed

c. Repeatable read

Answer:

a. Read uncommitted:

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

In the above schedule, T_2 reads the value of A written by T_1 even before T_1 commits. This schedule is not serializable since T_1 also reads a value written by T_2, resulting in a cycle in the precedence graph.

b. Read committed:

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

In the above schedule, the first time T_1 reads A, it sees a value of A before it was written by T_2, while the second **read**(A) by T_1 sees the value written by T_2 (which has already committed). The first read results in T_1 preceding T_2, while the second read results in T_2 preceding T_1, and thus the schedule is not serializable.

c. Repeatable read:

Consider the following schedule, where T_1 reads all tuples in r satisfying predicate P; to satisfy repeatable read, it must also share-lock these tuples in a two-phase manner.

T_1	T_2
pred_read(r, P)	
	insert(t)
	write(A)
	commit
read(A)	
commit	

Suppose that the tuple t inserted by T_2 satisfies P; then the insert by T_2 causes T_2 to be serialized after T_1, since T_1 does not see t. However, the final **read**(A) operation of T_1 forces T_2 to precede T_1, causing a cycle in the precedence graph.

17.21 Suppose that in addition to the operations read and write, we allow an operation pred_read(r, P), which reads all tuples in relation r that satisfy predicate P.

a. Give an example of a schedule using the pred_read operation that exhibits the phantom phenomenon and is nonserializable as a result.

b. Give an example of a schedule where one transaction uses the pred_read operation on relation r and another concurrent transaction

deletes a tuple from r, but the schedule does not exhibit a phantom conflict. (To do so, you have to give the schema of relation r and show the attribute values of the deleted tuple.)

Answer:

a. The repeatable read schedule in the preceding question is an example of a schedule exhibiting the phantom phenomenon and is nonserializable.

b. Consider the schedule

T_1	T_2
pred_read(r, $r.A$=5)	
	delete(t)
	write(B)
	commit
read(B)	
commit	

Suppose that tuple t deleted by T_2 is from relation r but does not satisfy predicate P, for example, because its A value is 3. Then there is no phantom conflict between T_1 and T_2, and T_2 can be serialized before T_1.

CHAPTER 18

Concurrency Control

Exercises

18.17 What benefit does strict two-phase locking provide? What disadvantages result?

Answer:
Because it produces only cascadeless schedules, recovery is very easy. But the set of schedules obtainable is a subset of those obtainable from plain two-phase locking, thus concurrency is reduced.

18.18 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

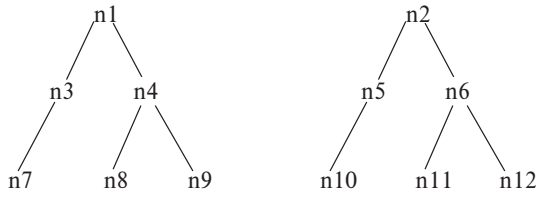
Answer:
It is relatively simple to implement, imposes low rollback overhead because of cascadeless schedules, and usually allows an acceptable level of concurrency.

18.19 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

Answer:
Take a system with two trees:



We have two transactions, T_1 and T_2. Consider the following legal schedule:

T_1	T_2
lock(*n1*)	
lock(*n3*)	
write(*n3*)	
unlock(*n3*)	
	lock(*n2*)
	lock(*n5*)
	write(*n5*)
	unlock(*n5*)
lock(*n5*)	
read(*n5*)	
unlock(*n5*)	
unlock(*n1*)	
	lock(*n3*)
	read(*n3*)
	unlock(*n3*)
	unlock(*n2*)

This schedule is not serializable.

18.20 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?

Answer:
Deadlock avoidance is preferable if the consequences of abort are serious (as in interactive transactions), and if there is high contention and a resulting high probability of deadlock.

18.21 If deadlock is avoided by deadlock-avoidance schemes, is starvation still possible? Explain your answer.

Answer:
A transaction may become the victim of deadlock-prevention rollback arbitrarily many times, thus creating a potential starvation situation.

18.22 In multiple-granularity locking, what is the difference between implicit and explicit locking?

Answer:
When a transaction *explicitly* locks a node in shared or exclusive mode, it *implicitly* locks all the descendents of that node in the same mode. The transaction need not explicitly lock the descendent nodes. There is no difference in the functionalities of these locks: the only difference is in the way they are acquired and their presence tested.

18.23 Although SIX mode is useful in multiple-granularity locking, an exclusive and intention-shared (XIS) mode is of no use. Why is it useless?

Answer:
An exclusive lock is incompatible with any other lock kind. Once a node is locked in exclusive mode, none of the descendents can be simultaneously accessed by any other transaction in any mode. Therefore an exclusive and intention-shared declaration has no meaning.

18.24 The multiple-granularity protocol rules specify that a transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?

Answer:
If T_i has locked the parent node P in S or SIX mode, then it means it has implicit S locks on all the descendent nodes of the parent node including Q. So there is no need for locking Q in S or IS mode, and the protocol does not allow doing that.

18.25 Suppose the lock hierarchy for a database consists of database, relations, and tuples.

a. If a transaction needs to read a lot of tuples from a relation r, what locks should it acquire?

b. Now suppose the transaction wants to update a few of the tuples in r after reading a lot of tuples. What locks should it acquire?

c. If at run-time the transaction finds that it needs to actually update a very large number of tuples (after acquiring locks assuming only a few tuples would be updated). What problems would this cause to the lock table, and what could the database do to avoid the problem?

Answer:

a. IS on database, S on r.

b. IX on database, SIX on r, X on required tuples.

c. The lock table in memory would get full due to a large number of locks. To avoid this problem, upgrade lock from SIX to X on the relation. Thus the individual X locks are escalated to a relation level X lock.

18.26 When a transaction is rolled-back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?

Answer:
A transaction is rolled back because a newer transaction has read or written the data which it was supposed to write. If the rolled back transaction is re-introduced with the same timestamp, the same reason for rollback is still valid, and the transaction will have be rolled back again. This will continue indefinitely.

18.27 Show that there are schedules that are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

Answer:
A schedule which is allowed in the two-phase locking protocol but not in the timestamp protocol is:

step	T_0	T_1	*Precedence marks*
1	lock-S(A)		
2	read(A)		
3		lock-X(B)	
4		write(B)	
5		unlock(B)	
6	lock-S(B)		
7	read(B)		$T_1 \rightarrow T_0$
8	unlock(A)		
9	unlock(B)		

This schedule is not allowed in the timestamp protocol because at step 7, the W-timestamp of B is 1.

A schedule which is allowed in the timestamp protocol but not in the two-phase locking protocol is:

step	T_0	T_1	T_2
1	**write**(A)		
2		**write**(A)	
3			**write**(A)
4	**write**(B)		
5		**write**(B)	

This schedule cannot have lock instructions added to make it legal under two-phase locking protocol because T_1 must unlock (A) between steps 2 and 3 and must lock (B) between steps 4 and 5.

18.28 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a `read` request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for `write` requests?

Answer:

Using the commit bit, a read request is made to wait if the transaction which wrote the data item has not yet committed. Therefore, if the writing transaction fails before commit, we can abort that transaction alone. The waiting read will then access the earlier version in case of a multiversion system, or the restored value of the data item after abort in case of a single-version system. For writes, this commit bit checking is unnecessary. That is because either the write is a "blind" write and thus independent of the old value of the data item or there was a prior read, in which case the test was already applied.

18.29 As discussed in Exercise 18.15, snapshot isolation can be implemented using a form of timestamp validation. However, unlike the multiversion timestamp-ordering scheme, which guarantees serializability, snapshot isolation does not guarantee serializability. Explain the key difference between the protocols that results in this difference.

Answer:

The timestamp validation step for the snapshot isolation level checks for the presence of common written data items between the transactions. However, write skew can occur, where a transaction T_1 updates an item A whose old version is read by T_2, while T_2 updates an item B whose old version is read by T_1, resulting in a nonserializable execution. There is no validation of reads against writes in the snapshot isolation protocol.

The multiversion timestamp-ordering protocol, on the other hand, avoids the write skew problem by rolling back a transaction that writes a data item which has been already read by a transaction with a higher timestamp.

18.30 Outline the key similarities and differences between the timestamp-based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 18.15, and the optimistic-concurrency control-without-read-validation scheme, described in Section 18.9.3.

Answer:
Both the schemes do not ensure serializability. The version number check in the optimistic-concurrency- control-without-read-validation implements the first committer-wins rule used in the snapshot isolation.

Unlike the snapshot isolation, the reads performed by a transaction in optimistic-concurrency- control-without-read-validation may not correspond to the snapshot of the database. Different reads by the same transaction may return data values corresponding to different snapshots of the database.

18.31 Consider a relation $r(A, B, C)$ and a transaction T that does the following: find the maximum A value in r, and insert a new tuple in r whose A value is 1+ the maximum A value. Assume that an index is used to find the maximum A value.

a. Suppose that the transaction locks each tuple it reads in S mode, and the tuple it creates in X mode, and performs no other locking. Now suppose two instances of T are run concurrently. Explain how the resultant execution could be non-serializable.

b. Now suppose that $r.A$ is declared as a primary key. Can the above non-serializable execution occur in this case? Explain why or why not.

Answer:
No answer

18.32 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

Answer:
The phantom phenomenon arises when, due to an insertion or deletion, two transactions logically conflict despite not locking any data items in common. The insertion case is described in the book. Deletion can also lead to this phenomenon. Suppose T_i deletes a tuple from a relation while T_j scans the relation. If T_i deletes the tuple and then T_j reads the relation, T_i should be serialized before T_j. Yet there is no tuple that both T_i and T_j conflict on.

An interpretation of 2PL as just locking the accessed tuples in a relation is incorrect. There is also an index or a relation data that has information about the tuples in the relation. This information is read by any transaction that scans the relation and modified by transactions that update, or insert into, or delete from the relation. Hence locking must also be performed on the index or relation data, and this will avoid the phantom phenomenon.

18.33 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?

Answer:
The degree-two consistency avoids cascading aborts and offers increased concurrency, but the disadvantage is that it does not guarantee serializability, and the programmer needs to ensure it.

18.34 Give example schedules to show that with key-value locking, if lookup, insert, or delete does not lock the next-key value, the phantom phenomenon could go undetected.

Answer:
In the next-key locking technique, every index lookup or insert or delete must not only the keys found within the range (or the single key, in case of a point lookup) but also the next-key value—that is, the key value greater than the last key value that was within the range. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict in the key value next to the inserted key value. The next-key value should be locked to ensure that conflicts with subsequent range lookups of other queries are detected, thereby detecting the phantom phenomenon.

18.35 Many transactions update a common item (e.g., the cash balance at a branch) and private items (e.g., individual account balances). Explain how you can increase concurrency (and throughput) by ordering the operations of the transaction.

Answer:
The private items can be updated by the individual transactions independently. They can acquire the exclusive locks for the private items (as no other transaction needs it) and update the data items. But the exclusive lock for the common item is shared among all the transactions. The common item should be locked before the transaction decides to update it. And when it holds the lock for the common item, all other transactions should wait until it is released. But in order that the common item is updated correctly, the transaction should follow a certain pattern. A transacation can update its private item as and when it requires, but before updating the private item again, the common item should be updated. So, essentially the private and the common items should be accessed alternately, otherwise the private item's update will not be reflected in the common item.

a. No possibility of deadlock and no starvation. The lock for the common item should be granted based on the time of requests.

b. The schedule is serializable.

18.36 Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher-numbered items may be locked. Locks may be released at any time. Only X-locks are used. Show by an example that this protocol does not guarantee serializability.

Answer:

We have two transactions, T_1 and T_2. Consider the following legal schedule:

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

Explanation: In the given example schedule, let's assume A is a higher numbered item than B.

a. T_i executes write(A) before T_j executes read(A). So, there's an edge T_i->T_j.

b. T_j executes write(B) before T_i executes read(A). So, there's an edge T_j->T_i.

There's a cycle in the graph, which means the given schedule is not conflict serializable.

CHAPTER 19

Recovery System

Exercises

19.14 Explain the difference between the three storage types—volatile, nonvolatile, and stable—in terms of I/O cost.

Answer:

Volatile storage is storage which fails when there is a power failure. Cache, main memory, and registers are examples of volatile storage. Nonvolatile storage is storage which retains its content despite power failures. An example is magnetic disk. Stable storage is storage which theoretically survives any kind of failure (short of a complete disaster!). This type of storage can only be approximated by replicating data.

In terms of I/O cost, volatile memory is the fastest and nonvolatile storage is typically several times slower. Stable storage is slower than nonvolatile storage because of the cost of data replication.

19.15 Stable storage cannot be implemented.

a. Explain why it cannot be.

b. Explain how database systems deal with this problem.

Answer:

a. Stable storage cannot really be implemented because all storage devices are made of hardware, and all hardware is vulnerable to mechanical or electronic device failures.

b. Database systems approximate stable storage by writing data to multiple storage devices simultaneously. Even if one of the devices crashes, the data will still be available on a different device. Thus data loss becomes extremely unlikely.

19.16 Explain how the database may become inconsistent if some log records pertaining to a block are not output to stable storage before the block is output to disk.

Answer:

Consider a banking scheme and a transaction which transfers \$50 from account A to account B. The transaction has the following steps:

a. **read**(A,a_1)

b. $a_1 := a_1 - 50$

c. **write**(A,a_1)

d. **read**(B,b_1)

e. $b_1 := b_1 + 50$

f. **write**(B,b_1)

Suppose the system crashes after the transaction commits, but before its log records are flushed to stable storage. Further assume that at the time of the crash the update of A in the third step alone had actually been propagated to disk, whereas the buffer page containing B was not yet written to disk. When the system comes up it is in an inconsistent state, but recovery is not possible because there are no log records corresponding to this transaction in stable storage.

19.17 Outline the drawbacks of the no-steal and force buffer management policies.

Answer:

Drawback of the no-steal policy: The no-steal policy does not work with transactions that perform a large number of updates, since the buffer may get filled with updated pages that cannot be evicted to disk, and the transaction cannot then proceed.

Drawback of the force policy: The force policy might slow down the commit of transactions as it forces all modified blocks to be flushed to disk before commit. Also, the number of output operations is more in the case of the force policy. This is because frequently updated blocks are output multiple times, once for each transaction. Further, the policy results in more seeks, since the blocks written out are not likely to be consecutive on disk.

19.18 Suppose two-phase locking is used, but exclusive locks are released early, that is, locking is not done in a strict two-phase manner. Give an example to show why transaction rollback can result in a wrong final state, when using the log-based recovery algorithm.

Answer:
No answer

19.19 Physiological redo logging can reduce logging overheads significantly, especially with a slotted page record organization. Explain why.

Answer:
If a slotted page record organization is used, the deletion of a record from a page may result in many other records in the page being shifted. With physical redo logging, all bytes of the page affected by the shifting of records must be logged. On the other hand, if physiological redo logging is used, only the deletion operation is logged. This results in a much smaller log record. Redo of the deletion operation would delete the record and shift other records as required.

19.20 Explain why logical undo logging is used widely, whereas logical redo logging (other than physiological redo logging) is rarely used.

Answer:
The class of operations which release locks early are called logical operations. Once such lower-level locks are released, such operations cannot be undone by using the old values of updated data items. Instead, they must be undone by executing a compensating operation called a logical undo operation. In order to allow logical undo of operations, special log records are needed to store the necessary logical undo information. Thus logical undo logging is used widely.

Redo operations are performed exclusively using physical log records. This is because the state of the database after a system failure may reflect some updates of an operation and not of other operations, depending on what buffer blocks had been written to disk before the failure. The database state on disk might not be in an *operation consistent* state, i.e., it might have partial effects of operations. Logical undo or redo operations cannot be performed on an inconsistent data structure.

19.21 Consider the log in Figure 19.5. Suppose there is a crash just before the log record $<T_0$ abort$>$ is written out. Explain what would happen during recovery.

Answer:
Recovery would happen as follows:

Redo phase:

a. Undo-list = T_0, T_1
b. Start from the checkpoint entry and perform the redo operation.
c. C = 600
d. T_1 is removed from the undo-list as there is a commit record.

e. T_2 is added to the undo-list on encountering the $< T_2$ start $>$ record.

f. A = 400

g. B = 2000

Undo phase:

a. Undo-list = T_0, T_2

b. Scan the log backwards from the end.

c. A = 500; output the redo-only record $< T_2$, A, 500 $>$

d. output $< T_2$ abort $>$

e. B = 2000; output the redo-only record $< T_0$, B, 2000 $>$

f. Output $< T_0$ abort $>$

At the end of the recovery process, the state of the system is as follows:

A = 500
B = 2000
C = 600

The log records added during recovery are:

$< T_2$, A, 500 $>$
$< T_2$ abort $>$
$< T_0$, B, 2000 $>$
$< T_0$ abort $>$

Observe that B is set to 2000 by two log records, one created during normal rollback of T_0 and the other created during recovery, when the abort of T_0 is completed. Clearly the second one is redundant, although not incorrect. Optimizations described in the ARIES algorithm (and equivalent optimizations described in Section 19.8 for the case of logical operations) can help avoid redundant operations, which create such redundant log records.

19.22 Suppose there is a transaction that has been running for a very long time but has performed very few updates.

a. What effect would the transaction have on recovery time with the recovery algorithm of Section 19.4, and with the ARIES recovery algorithm?

b. What effect would the transaction have on deletion of old log records?

Answer:

a. If a transaction has been running for a very long time, with few updates, it means that during recovery the undo phase will have to scan the log backwards to the beginning of this transaction. This will increase the recovery time in the case of the recovery algorithm of Section 19.4. However, in the case of ARIES the effect is not that bad because ARIES considers the LastLSN and PrevLSN values of transactions in the undo-list during its backward scan, allowing it to skip intermediate records belonging to completed transactions.

b. A long-running transaction implies that no log records which are written after it started can be deleted until it either commits or aborts. This might lead to a very large log file being generated, though most of the transactions in the log file have completed. This transaction becomes a bottleneck for deletion of old log records.

19.23 Consider the log in Figure 19.7. Suppose there is a crash during recovery, just before the operation abort log record is written for operation O_1. Explain what will happen when the system recovers again.

Answer:

There is no checkpoint in the log, so recovery starts from the beginning of the log and replays each action that is found in the log.

The redo phase would add the following log records:

$< T_0$, B, 2050 >
$< T_0$, C, 600 >
$< T_1$, C, 400 >
$< T_0$, C, 500 >

At the end of the redo phase, the undo list contains transactions T_0 and T_1, since their start records are found, but not their end-of-abort records.

During the undo phase, scanning backwards in the log, the following events happen:

$< T_0$, C, 400 >
$< T_1$, C, 600 > /* The operation end of $T_1.O_2$ is found, */
/* and logical undo adds +200 to the current value of C. */
/* Other log records of T_1 are skipped till $T_1.O_2$ operation */
/* begin is found. Log records of other txns would be */
/* processed, but there are none here. */
$< T_1, O_2$, operation-abort > /* On finding $T_1.O_2$ operation begin */
$< T_1$, abort > /* On finding T_1 start */
/* Next, the operation end of $T_0.O_1$ is found, and */

/* logical undo adds +100 to the current value of C.*/
< T_0, C, 700 >
/* Other operations of T_1 till $T_0.O_1$ begin are skipped */
/* And when $T_0.O_1$ operation begin is found: */
< T_0, O_1, operation-abort >
< T_0, B, 2000 >
< T_0, abort >

Finally the values of data items B and C would be 2000 and 700, which were their original values before T_0 or T_1 started.

19.24 Compare log-based recovery with the shadow-copy scheme in terms of their overheads for the case when data are being added to newly allocated disk pages (in other words, there is no old value to be restored in case the transaction aborts).

Answer:
In general, with logging, each byte that is written is actually written twice, once as part of the page to which it is written and once as part of the log record for the update. In contrast, shadow-copy schemes avoid log writes, at the expense of increased I/O operations and lower concurrency.
In the case of data added to newly allocated pages:

a. There is no old value, so there is no need to manage a shadow copy for recovery, which benefits the shadow-copy scheme.

b. Log-based recovery would unnecessarily write old values unless optimizations to skip the old value part are implemented for such newly allocated pages. Without such an optimization, logging has a higher overhead.

c. Newly allocated pages are often formatted, for example by zeroing out all bytes and then setting bytes corresponding to data structures in a slotted-page architecture. These operations also have to be logged, adding to the overhead of logging.

d. However, the benefit of shadow copy over logging is really significant only if the page is filled with a significant amount of data before it is written, since a lot of logging is avoided in this case.

19.25 In the ARIES recovery algorithm:

a. If at the beginning of the analysis pass, a page is not in the checkpoint dirty page table, will we need to apply any redo records to it? Why?

b. What is RecLSN, and how is it used to minimize unnecessary redos?

Answer:

a. If a page is not in the checkpoint dirty page table at the beginning of the analysis pass, redo records prior to the checkpoint record need not be applied to it as it means that the page has been flushed to disk and been removed from DirtyPageTable before the checkpoint. However, the page may have been updated after the checkpoint, which means it will appear in the dirty page table at the end of the analysis pass.
For pages that appear in the checkpoint dirty page table, redo records prior to the checkpoint may also need to be applied.

b. The RecLSN is an entry in DirtyPageTable, which reflects the LSN at the end of the log when the page was added to DirtyPageTable. During the redo pass of the ARIES algorithm, if the LSN of the update log record encountered is less than the RecLSN of the page in DirtyPageTable, then that record is not redone but skipped. Further, the redo pass starts at RedoLSN, which is the earliest of the RecLSNs among the entries in the checkpoint DirtyPageTable, since earlier log records would certainly not need to be redone. (If there are no dirty pages in the checkpoint, the RedoLSN is set to the LSN of the checkpoint log record.)

19.26 Explain the difference between a system crash and a "disaster."

Answer:
In a system crash, the CPU goes down, and a disk may also crash. But stable storage at the site is assumed to survive system crashes. In a "disaster," *everything* at a site is destroyed. Stable storage needs to be distributed to survive disasters.

19.27 For each of the following requirements, identify the best choice of degree of durability in a remote backup system:

a. Data loss must be avoided, but some loss of availability may be tolerated.

b. Transaction commit must be accomplished quickly, even at the cost of loss of some committed transactions in a disaster.

c. A high degree of availability and durability is required, but a longer running time for the transaction commit protocol is acceptable.

Answer:

a. Two very safe is suitable here because it guarantees durability of updates by committed transactions, though it can proceed only if both primary and backup sites are up. Availability is low, but it is mentioned that this is acceptable.

b. One safe committing is fast because it does not have to wait for the logs to reach the backup site. Since data loss can be tolerated, this is the best option.

c. With two safe committing, the probability of data loss is quite low, and also commits can proceed as long as at least the primary site is up. Thus availability is high. Commits take more time than in the one safe protocol, but that is mentioned as acceptable.

CHAPTER 20

Database-System Architectures

Exercises

20.13 Consider a bank that has a collection of sites, each running a database system. Suppose the only way the databases interact is by electronic transfer of money between themselves, using persistent messaging. Would such a system qualify as a distributed database? Why?

Answer:
In a distributed database, it should be possible to run queries across sites and to run transactions across sites using protocols such as two-phase commit. Each site provides an environment for execution of both global transactions initiated at remote sites and local transactions. The system described in the question does not have these properties, and hence it cannot qualify as a distributed database.

20.14 Assume that a growing enterprise has outgrown its current computer system and is purchasing a new parallel computer. If the growth has resulted in many more transactions per unit time, but the length of individual transactions has not changed, what measure is most relevant—speedup, batch scaleup, or transaction scaleup? Why?

Answer:
Transaction scaleup is the most relevant measure in this scenario, since the length of an individual transaction does not increase.

20.15 Database systems are typically implemented as a set of processes (or threads) accessing shared memory.

a. How is access to the shared-memory area controlled?

b. Is two-phase locking appropriate for serializing access to the data structures in shared memory? Explain your answer.

Answer:

a. A locking system is the usual means of controlling access to shared data structures. Since many database transactions only involve reading from a data structure, **shared** mode locks should be used. A process that wishes to modify the data structure needs to obtain an **exclusive** lock on the data structure which prohibits concurrent access from other reading or writing processes.

Locks used for controlling access to shared-memory data structures are typically not held in a two-phase manner (as described below), and they are often called *latches* to distinguish them from locks held in a two-phase manner.

b. Shared-memory areas are usually hot spots of contention, since every transaction needs to access the shared memory frequently. If such memory areas were locked in a two-phase manner, concurrency would be greatly reduced, reducing performance correspondingly. Instead, locks (latches) on shared-memory data structures are released after performing operations on the data structure.

Serializability at a lower level, such as the exact layout of data on a page, or the structure of a B-tree or hash-index, is not important as long as the differences caused by nonserial execution are not visible at a higher level (typically, at the relational abstraction). Locks on tuples (or higher granularity) are retained in a two-phase manner to ensure serializability at the higher level.

20.16 Is it wise to allow a user process to access the shared-memory area of a database system? Explain your answer.

Answer:

No, the shared-memory area may contain data that the user's process is not authorized to see, and so allowing direct access to shared memory is a security risk.

20.17 What are the factors that can work against linear scale up in a transaction processing system? Which of the factors are likely to be the most important in each of the following architectures: shared-memory, shared disk, and shared nothing?

Answer:

Increasing contention for shared resources prevents linear scale up with increasing parallelism. In a shared-memory system, contention for memory (which implies bus contention) will result in falling scale up with increasing parallelism. In a shared-disk system, it is contention for disk and bus access that affects scale up. In a shared-nothing system, inter-process communication overheads will be the main impeding factor. Since there is no shared mem-

ory, acquiring locks and other activities requiring message passing between processes will take more time with increased parallelism.

20.18 Memory systems today are divided into multiple modules, each of which can be serving a separate request at a given time, in contrast to earlier architectures where there was a single interface to memory. What impact has such a memory architecture have on the number of processors that can be supported in a shared-memory system?

Answer:
If all memory requests have to go through a single memory module, the memory module becomes the bottleneck as the number of processors increases. After some point, adding processors will not result in any performnace improvement. However, if the memory system is itself able to run multiple requests in parallel, a larger number of processors can be supported in a shared-memory system, providing better speedup and/or scaleup.

20.19 Assume we have data items $d_1, d_2, \ldots, d_n$ with each d_i protected by a lock stored in memory location M_i.

a. Describe the implementation of lock-X(d_i) and unlock(d_i) via the use of the test-and-set instruction.

b. Describe the implementation of lock-X(d_i) and unlock(d_i) via the use of the compare-and-swap instruction.

Answer:

a. To lock d_i, execute a *test-and-set*(M_i). If the return value is 0, then the lock was granted, otherwise some other process holds the lock.

b. To lock d_i, execute a *compare-and-swap*(M_i,0,*pid*), where *pid* is the process identifier of the process that is executing the *compare-and-swap*. If the return value is 0, then the lock was granted, otherwise some other process holds the lock.

20.20 In a shared-nothing system data access from a remote node can be done by remote procedure calls, or by sending messages. But remote direct memory access (RDMA) provides a much faster mechanism for such data access. Explain why.

Answer:
In addition to the cost of sending messages, which may involve several layers of a network stack, processing of the message at the receiver may have to wait till the recipient process gets its turn for CPU. Processing of the reply has similar latencies. In contrast, RDMA bypasses the network and process stacks, and directly fetches data from a remote machine. Thereby latency can be greatly reduced.

20.21 Suppose that a major database vendor offers its database system (e.g., Oracle, SQL Server DB2) as a cloud service. Where would this fit among the cloud-service models? Why?

Answer:
It would fit at the platform-as-a-service level. A database system is more than just a virtual machine, but it is also not an end-user application.

20.22 If an enterprise uses its own ERP application on a cloud service under the platform-as-a-service model, what restrictions would there be on when that enterprise may upgrade the ERP system to a new version?

Answer:
Since the cloud vendor controls the underlying platform, the enterprise faces two restrictions. First, it cannot install an upgrade that is not compatible with the installed version of the underlying platform software (controlled by the cloud vendor). Second, if the cloud vendor upgrades its platform software, that action may render the enterprise's version of the ERP system obsolete, thus mandating an immediate upgrade of the ERP system.

CHAPTER 21

Parallel and Distributed Storage

Exercises

21.9 For each of the three partitioning techniques, namely, round-robin, hash partitioning, and range partitioning, give an example of a query for which that partitioning technique would provide the fastest response.

Answer:

Round-robin partitioning:
When relations are large and queries read entire relations, round-robin gives good speedup and fast response time.

Hash partitioning:
For point queries on the partitioning attributes, this gives the fastest response, because each disk can process a different query simultaneously. If the hash partitioning is uniform, entire relation scans can be performed efficiently.

Range partitioning: For range queries on the partitioning attributes, which access a few tuples, range partitioning gives the fastest response.

21.10 What factors could result in skew when a relation is partitioned on one of its attributes by:

a. Hash partitioning?

b. Range partitioning?

In each case, what can be done to reduce the skew?

Answer:

a. Hash partitioning:
Too many records with the same value for the hashing attribute, or a poorly chosen hash function without the properties of randomness and

uniformity, can result in a skewed partition. To improve the situation, we should experiment with better hashing functions for that relation.

b. Range partitioning:
Non-uniform distribution of values for the partitioning attribute (including duplicate values for the partitioning attribute) which are not taken into account by a bad partitioning vector is the main reason for skewed partitions. Sorting the relation on the partitioning attribute and then dividing it into n ranges with an equal number of tuples per range will give a good partitioning vector with very low skew.

21.11 What is the motivation for storing related records together in a key-value store? Explain the idea using the notion of an entity group.

Answer:
No answer

21.12 Why is it easier for a distributed file system such as GFS or HDFS to support replication than it is for a key-value store?

Answer:
These distributed file systems do not support updates, and instead only allow appends to existing files, and moreover do not allow a file that is undergoing appends to be read until it is closed (and cannot be updated anymore). At that point, all replicas have exactly the same value, so reads can be processed from any of the replicas. Key-value stores cannot impose such restrictions on reads, writes and updates.

21.13 Joins can be expensive in a key-value store, and difficult to express if the system does not support SQL or a similar declarative query language. What can an application developer do to efficiently get results of join or aggregate queries in such a setting?

Answer:
The developer can precompute the join or aggregate query as a materialized view. Since a system that does not support declarative queries will not support view maintenance, the task of view maintenance falls on the developer. Whenever an underlying relation is updated, the view must also be updated.

Persistent messaging techniques can be used to perform view maintenance, by sending a message whenever the underlying relation is updated. The message is read by the view maintenance code which then updates the materialized view. This way, transactions that update the underlying relations are not delayed by the code for view maintenance.

CHAPTER 22

Parallel and Distributed Query Processing

Exercises

22.12 Can partitioned join be used for $r \bowtie_{r.A<s.A \wedge r.B=s.B} s$? Explain your answer.

Answer:
Here we have an equi-join condition which can be executed first, and the extra conditions can be checked independently on each tuple in the join result. Partitioned parallelism is useful to execute the equi-join, Relation r should be partitioned on attribute A and s on attribute B.

22.13 Describe a good way to parallelize each of the following:

a. The difference operation

b. Aggregation by the **count** operation

c. Aggregation by the **count distinct** operation

d. Aggregation by the **avg** operation

e. Left outer join, if the join condition involves only equality

f. Left outer join, if the join condition involves comparisons other than equality

g. Full outer join, if the join condition involves comparisons other than equality

Answer:

a. We can parallelize the difference operation by partitioning the relations on all attributes and then computing differences locally at each processor. As in aggregation, the cost of transferring tuples during partitioning

can be reduced by partially computing differences at each processor before partitioning.

b. Let us refer to the group-by attribute as attribute A and the attribute on which the aggregation function operates as attribute B. **count** is performed just like **sum** (mentioned in the book) except that a count of the number of values of attribute B for each value of attribute A is transferred to the correct destination processor, instead of a sum. After partitioning, the partial counts from all the processors are added up locally at each processor to get the final result.

c. For this, partial counts cannot be computed locally before partitioning. Each processor instead transfers all unique B values for each A value to the correct destination processor. After partitioning, each processor locally counts the number of unique tuples for each value of A and then outputs the final result.

d. This can again be implemented like **sum**, except that for each value of A, a **sum** of the B values as well as a **count** of the number of tuples in the group is transferred during partitioning. Then each processor outputs its local result by dividing the total sum by the total number of tuples for each A value assigned to its partition.

e. This can be performed just like partitioned natural join. After partitioning, each processor computes the left outer join locally using any of the strategies of Chapter 15.

f. The left outer join can be computed using an extension of the fragment-and-replicate scheme to compute non equi-joins. Consider $r ⟕ s$. The relations are partitioned, and $r \bowtie s$ is computed at each site. We also collect tuples from r that did not match any tuples from s; call the set of these dangling tuples at site i as d_i. After the above step is done at each site, for each fragment of r, we take the intersection of the d_i's from every processor in which the fragment of r was replicated. The intersections give the real set of dangling tuples; these tuples are padded with nulls and added to the result. The intersections themselves, followed by the addition of padded tuples to the result, can be done in parallel by partitioning.

g. The algorithm is basically the same as above, except that when combining results, the processing of dangling tuples must done for both relations.

22.14 Suppose you wish to handle a workload consisting of a large number of small transactions by using shared-nothing parallelism.

a. Is intraquery parallelism required in such a situation? If not, why, and what form of parallelism is appropriate?

b. What form of skew would be of significance with such a workload?

c. Suppose most transactions accessed one *account* record, which includes an *account_type* attribute, and an associated *account_type_master* record, which provides information about the account type. How would you partition and/or replicate data to speed up transactions? You may assume that the *account_type_master* relation is rarely updated.

Answer:

a. Intraquery parallelism is probably not appropriate for this situation. Since each individual transaction is small, the overhead of parallelizing each query may exceed the potential benefits. Interquery parallelism would be a better choice, allowing many transactions to run in parallel.

b. Partition skew can be a performance issue in this type of system, especially with the use of shared-nothing parallelism. A load imbalance among the processors of the distributed system can significantly reduce the speedup gained by parallel execution. For example, if all transactions happen to involve only the data in a single partition, the processors not associated with that partition will not be used at all.

c. Since *account_type_master* is rarely updated, it can be replicated in its entirety across all nodes. If the *account* relation is updated frequently and accesses are well distributed, it should be partitioned across nodes.

22.15 What is the motivation for work-stealing with virtual nodes in a shared-memory setting? Why might work-stealing not be as efficient in a shared-nothing setting?

Answer:

Work stealing helps address the issue of skew. Multiple tasks (each virtual nodes corresponds to a task) can be initially assigned to each real node. If some real nodes finish processing early, they can steal tasks (virtual nodes) that are waiting to be processed by real nodes that are taking more time to finish their assigned tasks. Since all the required data are in shared memory, it is quite cheap to access the data from any of the real nodes.

However, in a shared-nothing setting, the tasks would have already been distributed across the nodes, and moving them again over the network will have an overhead. If the skew is large enough it makes sense to steal work, but for smaller amounts of skew the overhead may result in very little benefit.

22.16 The attribute on which a relation is partitioned can have a significant impact on the cost of a query.

a. Given a workload of SQL queries on a single relation, what attributes would be candidates for partitioning?

b. How would you choose between the alternative partitioning techniques, based on the workload?

c. Is it possible to partition a relation on more than one attribute? Explain your answer.

Answer:

a. The candidate attributes would be

   i. Attributes on which one or more queries have a selection condition. the corresponding selection condition can then be evaluated at a single processor instead of being evaluated at all processors.

   ii. Attributes involved in join conditions. If such an attribute is used for partitioning, it is possible to perform the join without repartitioning the relation. This effect is particularly beneficial for very large relations, for which repartitioning can be very expensive.

   iii. Attributes involved in group-by clauses; similar to joins, it is possible to perform aggregation without repartitioning the corresponding relation.

b. A cost-based approach works best in choosing between alternatives. In this approach, candidate partitioning choices are generated, and for each candidate the cost of executing all the queries/updates in a workload is estimated. The choice leading to the least cost is picked.

   One issue is that the number of candidate choices is generally very large. Algorithms and heuristics designed to limit the number of candidates for which costs need to be estimated are widely used in practice.

   Another issue is that the workload may have a very large number of queries/updates. Techniques to reduce this number include the following: (a) combining repeated occurrences of a query that only differ in constants, replacing them with one parameterized query along with a count of number of occurrences and (b) dropping queries which are very cheap, or not likely to be affected by the partitioning choice.

c. It is possible to partition a relation on more than one attribute in two ways. One is to involve multiple attributes in a single composite partitioning key. The other way is to keep more than one copy of the same relation, partitioned in different ways. The latter approach is increases update costs, but it can speed up some queries significantly.

22.17 Consider system that is processing a stream of tuples for a relation r with attributes (A, B, C, *timestamp*) Suppose the goal of a parallel stream processing

system is to compute the number of tuples for each A value in each 5 minute window (based on the timestamp of the tuple). What would be the topic and the topic partitions? Explain why.

Answer:

There would be a topic corresponding to relation r. The topic would be partitioned on $r.A$, to allow parallel processing across different A values.

CHAPTER 23

Parallel and Distributed Transaction Processing

Exercises

23.14 What characteristics of an application make it easy to scale the application by using a key-value store, and what characteristics rule out deployment on key-value stores?

Answer:

Any application that is easy to partition, and does not need strong guarantees of consistency across partitions, is ideally suited to the running on a key-value store.

Any database application that needs transactional consistency for updates to multiple records, in particular records that cannot be meaninfully grouped together, would be hard to implement successfully in the cloud. For examples include bank records, academic records of students, and many other types of organizational records have complex schemas with relationships that cannot be easily partitioned, yet need to be atomically updated.

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

Answer:

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

23.16 In the majority protocol, what should the reader do if it finds different values from different copies, to (a) decide what is the correct value, and (b) to bring

the copies back to consistency? If the reader does not bother to bring the copies back to consistency, would it affect correctness of the protocol?

Answer:
Make sure it has a majority of copies, and use the copy that has the highest version number. To bring the copies back to consistency, it should write the latest value (along with the version number) to all the copies that have older versions. No locks are needed when performing this step. If this step were not performed, correctness would not be affected.

23.17 If we apply a distributed version of the multiple-granularity protocol of Chapter 18 to a distributed database, the site responsible for the root of the DAG may become a bottleneck. Suppose we modify that protocol as follows:

- Only intention-mode locks are allowed on the root.
- All transactions are given the strongest intention-mode lock (IX) on the root automatically.

Show that these modifications alleviate this problem without allowing any nonserializable schedules.

Answer:
Serializability is assured since we have not changed the rules for the multiple granularity protocol. Since transactions are automatically granted an IX locks on the root node, and are not given other kinds of locks on it, there is no need to send any lock requests to the root. Thus the bottleneck is relieved.

23.18 Discuss the advantages and disadvantages of the two methods that we presented in Section 23.3.4 for generating globally unique timestamps.

Answer:
The centralized approach has the problem of a possible bottleneck at the central site and the problem of electing a new central site if it goes down. The distributed approach has the problem that many messages must be exchanged to keep the system fair, or one site can get ahead of all other sites and dominate the database.

23.19 Spanner provides read-only transactions a snapshot view of data, using multi-version two-phase locking.

a. In the centralized multi-version 2PL scheme, read-only transactions never wait. But in Spanner, reads may have to wait. Explain why.

b. Using an older timestamp for the snapshot can reduce waits, but has some drawbacks. Explain why, and what the drawbacks are.

Answer:

a. Unlike centralized commit, which can be instantaneous, commit in Spanner may have to wait for prepared transactions to commit; these transactions have already been assigned a timestamp, but we don't know whether they will commit or not. Till the decision is known, a snapshot read will have to wain.

b. By using an older timestamp, waits can be reduced or avoided, since there will be fewer or no transactions in prepared state with a timestamp less than that of the snapshot. The tradeoff is that the snapshot may not be current, so the readonly transaction may see stale data.

23.20 Merkle trees can be made short and fat (like B^+-trees) or thin and tall (like binary search trees). Which option would be better if you are comparing data across two sites that are geographically separated, and why?

Answer:
Short and fat trees are better, since comparison requires multiple communication round trips, with at least one per level if some items differ. A short and fat tree reduces the number of round trips, which is important since round trip delay can be quite large if sites are geographically separated.

23.21 Why is the notion of term important when an election is used to choose a coordinator? What are the analogies between elections with terms and elections used in a democracy?

Answer:
Since a coordinator might die or get disconnected, a new coordinator must be elected. Without a notion of a term, a site would have to change it vote, which could allow multiple nodes to win a single election. With the notion of a term, and assuming the sites follow the protocol and cannot change their vote in a term, at most one site can get the majority vote in a term.

In a democracy, elections are held periodically; voters cannot vote twice in the same election, but can vote again in a subsequent election, and are allowed to change the decision at that point.

23.22 For correct execution of a replicated state machine, the actions must be deterministic. What could happen if an action is non-deterministic?

Answer:
If an action is non-deterministic, for example it tosses a coin, or depends on the time when it is executed, different state machines may enter different states, even though they got exactly the same input. Queries would then give different answers corresponding to the same state, depending on which machine is queried, which is not acceptable.

CHAPTER 24

Advanced Indexing Techniques

Exercises

24.10 The stepped merge variant of the LSM tree allows multiple trees per level. What are the tradeoffs in having more trees per level?

Answer:
Having more trees in a level reduces the write cost, since each level is then larger, leading to a lower number of levels in total. However, reads are penalized, since a read may have to read each tree in each level. While Bloom filters can reduce the read cost for point queries on a single value, there is still a significant overhead. Range queries cannot benefit from Bloom filters and become correspondingly more expensive with more trees in a level.

24.11 Suppose you want to use the idea of a quadtree for data in three dimensions. How would the resultant data structure (called an *octtree*) divide up space?

Answer:
The key idea is to divide each dimension in two. Just as with two dimensions we get 4 regions, with three dimensions, we get 8 regions. The resultant data structure is called a octtree since it has 8 children for each internal node.

24.12 Explain the distinction between closed and open hashing. Discuss the relative merits of each technique in database applications.

Answer:
Open hashing may place keys with the same hash function value in different buckets. Closed hashing always places such keys together in the same bucket. Thus in this case, different buckets can be of different sizes, though the implementation may be by linking together fixed size buckets using overflow chains. Deletion is difficult with open hashing as *all* the buckets may have to inspected before we can ascertain that a key value has been deleted, whereas in closed hashing only that bucket whose address is obtained by hashing the key value need be inspected. Deletions are more common in databases and hence closed hashing is more appropriate for them. For a small, static set of data lookups

may be more efficient using open hashing. The symbol table of a compiler would be a good example.

24.13 What are the causes of bucket overflow in a hash file organization? What can be done to reduce the occurrence of bucket overflows?

Answer:
The causes of bucket overflow are:

a. Our estimate of the number of records that the relation will have was too low, and hence the number of buckets allotted was not sufficient.

b. Skew in the distribution of records to buckets. This may happen either because there are many records with the same search key value, or because the the hash function chosen did not have the desirable properties of uniformity and randomness.

To reduce the occurrence of overflows, we can:

a. Choose the hash function more carefully, and make better estimates of the relation size.

b. If the estimated size of the relation is n_r and number of records per block is f_r, allocate $(n_r/f_r) * (1+d)$ buckets instead of (n_r/f_r) buckets. Here d is a fudge factor, typically around 0.2. Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that some of the skew is handled and the probability of overflow is reduced.

24.14 Why is a hash structure not the best choice for a search key on which range queries are likely?

Answer:
A range query cannot be answered efficiently using a hash index, we will have to read all the buckets. This is because key values in the range do not occupy consecutive locations in the buckets, they are distributed uniformly and randomly throughout all the buckets.

24.15 Our description of static hashing assumes that a large contiguous stretch of disk blocks can be allocated to a static hash table. Suppose you can allocate only C contiguous blocks. Suggest how to implement the hash table, if it can be much larger than C blocks. Access to a block should still be efficient.

Answer:
A separate list/table as shown below can be created.

Starting address of first set of C blocks

C

Starting address of next set of C blocks

2*C* and so on

Desired block address = Starting adress (from the table depending on the block number) + blocksize * (blocknumber % C)

For each set of *C* blocks, a single entry is added to the table. In this case, locating a block requires 2 steps: First we use the block number to find the actual block address, and then we can access the desired block.

CHAPTER 25

Advanced Application Development

Exercises

25.10 Database tunning:

a. What are the three broad levels at which a database system can be tuned to improve performance?

b. Give two examples of how tuning can be done for each of the levels.

Answer:

a. We refer to performance tuning of a database system as the modification of some system components in order to improve transaction response times or overall transaction throughput. Database systems can be tuned at various levels to enhance performance, viz.

i. Schema and transaction design

ii. Buffer manager and transaction manager

iii. Access and storage structures

iv. Hardware - disks, CPU, busses, etc.

b. We describe some examples for performance tuning of some of the major components of the database system.

i. **Tuning the schema**
In this chapter we have seen two examples of schema tuning, viz. vertical partition of a relation (or conversely - join of two relations), and denormalization (or conversely - normalization). These examples reflect the general scenario, and ideas therein can be applied to tune other schemas.

ii. **Tuning the transactions**

One approach used to speed up query execution is to improve the its plan. Suppose that we need the natural join of two relations - say *account* and *depositor* from our sample bank database. A *sort–merge-join* (Section 15.5.4) on the attribute *account-number* may be quicker than a simple nested-loop join on the relations.

Other ways of tuning transactions are breaking up long update transactions and combining related sets of queries into a single query. Generic examples for these approaches are given in this chapter.

For client-server systems, wherein the query has to be transmitted from client to server, the query transmission time itself may form a large fraction of the total query cost. Using *stored procedures* can significantly reduce the queries' response time.

iii. **Tuning the buffer manager**
The buffer manager can be made to increase or decrease the number of pages in the buffer according to changing page-fault rates. However, it must be noted that a larger number of pages may mean higher costs for latch management and maintenance of other data structures like free-lists and page map tables.

iv. **Tuning the transaction manager**
The transaction schedule affects system performance. A query that computes statistics for customers at each branch of the bank will need to scan the relations *account* and *depositor*. During these scans, no updates to any customer's balance will be allowed. Thus, the response time for the update transactions is high. Large queries are best executed when there are few updates, such as at night.

Checkpointing also incurs some cost. If recovery time is not critical, it is preferable to examine a long log (during recovery) rather than spend a lot of (checkpointing) time during normal operation. Hence it may be worthwhile to tune the checkpointing interval according to the expected rate of crashes and the required recovery time.

v. **Tuning the access and storage structures**
A query's response time can be improved by creating an appropriate index on the relation. For example, consider a query in which a depositor enquires about her balance in a particular account. This query would result in the scan of the relation *account* if it has no index on *account-number*. Similar indexing considerations also apply to computing joins, i.e., an index on *account-number* in the *account* relation saves the time needed to scan *account* when a natural join of *account* is taken with *depositor*.

In contrast, performance of update transactions may suffer due to indexing. Let us assume that frequent updates to the balance are required. Also suppose that there is an index on *balance* (presumably for range queries) in *account*. Now, for each update to the value of the balance, the index too will have to be updated. In addition, concurrent updates to the index structure will require additional locking overheads. Note that the response time for each update would not be more if there were no index on *balance*.

The type of index chosen also affects performance. For a range query, an order-preserving index (like B-trees) is better than a hashed index.

Clustering of data affects the response time for some queries. For example, assume that the tuples of the *account* relation are clustered on *branch-name*. Then the average execution time for a query that finds the total balance amount deposited at a particular branch can be improved. Even more benefit accrues from having a clustered index on *branch-name*.

If the database system has more than one disk, *declustering* of data will enable parallel access. Suppose that we have five disks, and that in a hypothetical situation, each customer has five accounts, and each account has a lot of historical information that needs to be accessed. Storing one account per customer per disk will enable parallel access to all accounts of a particular customer. Thus, the speed of a scan on *depositor* will increase about fivefold.

vi. **Tuning the hardware**

The hardware for the database system typically consists of disks, the processor, and the interconnecting architecture (busses, etc.). Each of these components may be a bottleneck, and by increasing the number of disks or their block sizes, or using a faster processor, or improving the bus architecture, one may obtain an improvement in system performance.

25.11 When carrying out performance tuning, should you try to tune your hardware (by adding disks or memory) first, or should you try to tune your transactions (by adding indices or materialized views) first. Explain your answer.

Answer:

The three levels of tuning - hardware, database system parameters, schema and transactions - interact with one another, so we must consider them together when tuning a system. For example, tuning at the transaction level may result in the hardware bottleneck changing from the disk system to the CPU, or vice versa.

25.12 Suppose that your application has transactions that each access and update a single tuple in a very large relation stored in a B^+-tree file organization. Assume that all internal nodes of the B^+-tree are in memory, but only a very small fraction of the leaf pages can fit in memory. Explain how to calculate the minimum number of disks required to support a workload of 1000 transactions per second. Also calculate the required number of disks, using values for disk parameters given in Section 12.3.

Answer:

Given that all internal nodes of the B^+-tree are in memory, only a very small fraction of the leaf pages can fit in memory. We can deduce that each one I/O transaction that access and update a single tuple requires just 1 I/O operation. The disk with the parameters given in the chapter would support a little under 100 random-access I/O operations of 4 kilbytes each per second. So, the number of disks needed to support a workload of 1000 transactions is 1000/100 = 10 disks.

The disk parameters given in Section 12.3 are almost the same as the values in this chapter. So, the number of disks required will be around 10 in this case also.

25.13 What is the motivation for splitting a long transaction into a series of small ones? What problems could arise as a result, and how can these problems be averted?

Answer:

Long update transactions cause a lot of log information to be written and hence extend the checkpointing interval and also the recovery time after a crash. A transaction that performs many updates may even cause the system log to overflow before the transaction commits.

To avoid these problems with a long update transaction, it may be advisable to break it up into smaller transactions. This can be seen as a *group* transaction being split into many small *mini-batch* transactions. The same effect is obtained by executing both the group transaction and the mini-batch transactions, which are scheduled in the order that their operations appear in the group transaction.

However, executing the mini-batch transactions in place of the group transaction has some costs, such as extra effort when recovering from system failures. Also, even if the group transaction satisfies the *isolation* requirement, the mini-batch may not. Thus the transaction manager can release the locks held by the mini-batch only when the last transaction in the mini-batch completes execution.

25.14 Suppose the price of memory falls by half, and the speed of disk access (number of accesses per second) doubles, while all other factors remain the same. What would be the effect of this change on the 5-minute and 1-minute rule?

Answer:
There will be no effect of these changes on the 5 minute or the 1 minute rule. The value of n, i.e., the frequency of page access at the break-even point, is proportional to the product of memory price and speed of disk access, other factors remaining constant. So when memory price falls by half and access speed doubles, n remains the same.

25.15 List at least four features of the TPC benchmarks that help make them realistic and dependable measures.

Answer:
Some features that make the TPC benchmarks realistic and dependable are -

a. Ensuring full support for ACID properties of transactions,

b. Calculating the throughput by observing the *end-to-end* performance,

c. Making sizes of relations proportional to the expected rate of transaction arrival, and

d. Measuring the dollar cost per unit of throughput.

25.16 Why was the TPC-D benchmark replaced by the TPC-H and TPC-R benchmarks?

Answer:
Various TPC-D queries can be significantly speeded up by using materialized views and other redundant information, but the overheads of using them should be properly accounted for. Hence TPC-R and TPC-H were introduced as refinements of TPC-D, both of which use same the schema and workload. TPC-R models periodic reporting queries, and the database running it is permitted to use materialized views. TPC-H, on the other hand, models ad hoc querying and prohibits materialized views and other redundant information.

25.17 Explain what application characteristics would help you decide which of TPC-C, TPC-H, or TPC-R best models the application.

Answer:
Depending on the application characteristics, different benchmarks are used to model it.
The TPC-C benchmark is widely used for transaction processing. It is appropriate for applications which concentrate on the main activities in an order-entry environment, such as entering and delivering orders, recording payments, checking status of orders, and monitoring levels of stock.
The TPC-H (H represents *adhoc*) benchmark models the applications which prohibit materialized views and other redundant information and permits indices only on primary and foreign keys. This benchmark models ad hoc querying where the queries are not known beforehand.

The TPC-R (R represents for *reporting*) models the applications which has queries, inserts, updates, and deletes. The application is permitted to use materialized views and other redundant information.

25.18 Given that the LDAP functionality can be implemented on top of a database system, what is the need for the LDAP standard?

Answer:

The reasons are:

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

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

CHAPTER 26

Blockchain Databases

Exercises

26.12 In what order are blockchain transactions serialized?

Answer:
The order is determined by the ordering of blocks on the blockchain and the order of transactions within a block. A valid block ensures that each transaction that takes output from another transaction as input is not a double-spend transaction.

26.13 Since blockchains are immutable, how is a transaction abort implemented so as not to violate immutability?

Answer:
Aborted transactions are not placed on the blockchain and thus their changes were never made visible. Unlike in a traditional database system with an update-in-place policy, there is nothing to undo.

26.14 Since pointers in a blockchain include a cryptographic hash of the previous block, why is there the additional need for replication of the blockchain to ensure immutability?

Answer:
Replication is the defense against an attacker replacing the entire blockchain.

26.15 Suppose a user forgets or loses her or his private key? How is the user affected?

Answer:
Loss of a private key is a disaster. Unless the user has given the private key to an exchange, that user will have permanently lost all currency stored on the blockchain.

26.16 How is the difficulty of proof-of-work mining adjusted as more nodes join the network, thus increasing the total computational power of the network? Describe the process in detail.

Answer:
As more nodes join (or upgrade, for that matter), the computational power of the network increases and so nonces are discovered more quickly. This increases the rate at which blocks are mined. The blockchain system code is written such that when this occurs the target for the hash is adjusted to a smaller value, lowering the probability of guessing a successful nonce. That, in turn, lowers the mining rate. Continuous adjustment keeps the mining rate within a bounded range.

26.17 Why is Byzantine consensus a poor consensus mechanism in a public blockchain?

Answer:
Byzantine consensus uses a majority vote. Therefore an attacker can add small, cheap nodes (such as old computers or old mobile phones) that need to serve no useful function beyond voting for the attacker's agenda. Thus is it cheap to overwhelm a public blockchain. Permissioned blockchains control admission of nodes, which prevents such attacks.

26.18 Explain how off-chain transaction processing can enhance throughput. What are the trade-offs for this benefit?

Answer:
Off-chain processing avoids the overhead of distributed consensus and validation. The off-chain network can run at rates similar to that of a traditional database system. However, this comes without the protections and public visibility that a blockchain provides, since aggregated transactions are posted to the blockchain only periodically or at the termination of an agreement.

26.19 Choose an enterprise of personal interest to you and explain how blockchain technology could be employed usefully in that business.

Answer:
No answer provided. Student answers should be judged for technical accuracy and business relevance.