CHAPTER 1

Introduction

Practice Exercises

1.1 This chapter has described several major advantages of a database system. What are two disadvantages?
Answer: Two disadvantages associated with database systems are listed below.

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

Answer:

- **Data redundancy and inconsistency**. This would be relevant to metadata to some extent, although not to the actual video data, which is not updated. There are very few relationships here, and none of them can lead to redundancy.
- **Difficulty in accessing data**. If video data is only accessed through a few predefined interfaces, as is done in video sharing sites today,

this will not be a problem. However, if an organization needs to find video data based on specific search conditions (beyond simple keyword queries) if meta data were stored in files it would be hard to find relevant data without writing application programs. Using a database would be important for the task of finding data.

- **Data isolation**. Since data is not usually updated, but instead newly created, data isolation is not a major issue. Even the task of keeping track of who has viewed what videos is (conceptually) append only, again making isolation not a major issue. However, if authorization is added, there may be some issues of concurrent updates to authorization information.
- **Integrity problems**. It seems unlikely there are significant integrity constraints in this application, except for primary keys. if the data is distributed, there may be issues in enforcing primary key constraints. Integrity problems are probably not a major issue.
- **Atomicity problems**. When a video is uploaded, metadata about the video and the video should be added atomically, otherwise there would be an inconsistency in the data. An underlying recovery mechanism would be required to ensure atomicity in the event of failures.
- **Concurrent-access anomalies**. Since data is not updated, concurrent access anomalies would be unlikely to occur.
- **Security problems**. These would be a issue if the system supported authorization.

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

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

CHAPTER 1

Introduction

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

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

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

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

Exercises

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

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

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

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

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

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

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

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

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

a. interaction with the file manager.

b. integrity enforcement.

c. security enforcement.

d. backup and recovery.

e. concurrency control.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 2

Introduction to the Relational Model

Practice Exercises

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

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

- Inserting a tuple:

  (10111, Ostrom, Economics, 110,000)

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

- Deleting the tuple:

  (Biology, Watson, 90000)

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

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

Figure 2.1 Relational database for Practice Exercise 2.1.

2.3 Consider the *time_slot* relation. Given that a particular time slot can meet more than once in a week, explain why *day* and *start_time* are part of the primary key of this relation, while *end_time* is not.
Answer: The attributes *day* and *start_time* are part of the primary key since a particular class will most likely meet on several different days, and may even meet more than once in a day. However, *end_time* is not part of the primary key since a particular class that starts at a particular time on a particular day cannot end at more than one time.

2.4 In the instance of *instructor* shown in Figure **??**, no two instructors have the same name. From this, can we conclude that *name* can be used as a superkey (or primary key) of *instructor*?
Answer: No. For this possible instance of the instructor table the names are unique, but in general this may not be always the case (unless the university has a rule that two instructors cannot have the same name, which is a rather unlikey scenario).

2.5 What is the result of first performing the cross product of *student* and *advisor*, and then performing a selection operation on the result with the predicate *s_id* = ID? (Using the symbolic notation of relational algebra, this query can be written as $\sigma_{s\_id=ID}(student \times advisor)$.)
Answer: The result attributes include all attribute values of student followed by all attributes of advisor. The tuples in the result are as follows. For each student who has an advisor, the result has a row containing that students attributes, followed by an *s_id* attribute identical to the students ID attribute, followed by the *i_id* attribute containing the ID of the students advisor.
Students who do not have an advisor will not appear in the result. A student who has more than one advisor will appear a corresponding number of times in the result.

2.6 Consider the following expressions, which use the result of a relational algebra operation as the input to another operation. For each expression, explain in words what the expression does.

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

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

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

Answer:

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

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

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

c. Provide a list of consisting of

ID, $name$, $course\_id$

of all students who took any course in the university.

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

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

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

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

Answer:

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

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

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

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

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

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

Answer:

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

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

CHAPTER 2

Introduction to the Relational Model

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

Exercises

2.9 Consider the bank database of Figure 2.15.

a. What are the appropriate primary keys?

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

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

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

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

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

Answer:

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

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

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

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

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

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

is a relation based on that schema.

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

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

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

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

Answer:

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

2.15 Discuss the relative merits of procedural and nonprocedural languages.

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

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

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

CHAPTER 3

Introduction to SQL

Exercises

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

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

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

c. Find the highest salary of any instructor.

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

e. Find the enrollment of each section that was offered in Autumn 2009.

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

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

Answer:

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

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

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

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

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

c. Find the highest salary of any instructor.

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

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

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

e. Find the enrollment of each section that was offered in Autumn 2009.
One way of writing the query is as follows.

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

Note that if a section does not have any students taking it, it would not appear in the result. One way of ensuring such a section appears with a count of 0 is to replace **natural join** by the **natural left outer join** operation, covered later in Chapter 4. Another way is to use a subquery in the **select** clause, as follows.

```
select  course_id, sec_id,
        (select count(ID)
        from   takes
        where  takes.year = section.year
               and takes.semester = section.semester
               and takes.course_id = section.course_id
               and takes.section_id = section.section_id)
        from   section
where   semester = 'Autumn'
and     year = 2009
```

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

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

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

As an alternative to using a nested subquery in the **from** clause, it is possible to use a **with** clause, as illustrated in the answer to the next part of this question.
A subtle issue in the above query is that if no section had any enrollment, the answer would be empty, not 0. We can use the alternative using a subquery, from the previous part of this question, to ensure the count is 0 in this case.

g. Find the sections that had the maximum enrollment in Autumn 2009. The following answer uses a **with** clause to create a temporary view, simplifying the query.

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

It is also possible to write the query without the **with** clause, but the subquery to find enrollment would get repeated twice in the query.

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

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

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

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

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

Answer:

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

```
select sum(credits * points)
from   (takes natural join course) natural join grade_points
whereID = '12345'
```

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

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

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

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

```
select  sum(credits * points)/sum(credits) as GPA
from    (takes natural join course) natural join grade_points
where   ID = '12345'
```

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

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

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

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

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

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

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

3.3

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

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

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

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

Answer:

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

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

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

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

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

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

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

a. Find the total number of people who owned cars that were involved in accidents in 1989.

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

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

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

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

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

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

Figure ??. Insurance database.

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

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

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

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

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

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

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

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

b. Find the number of students with each grade.

Answer:

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

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

b. Find the number of students with each grade.

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

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

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

Answer:

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

3.8 Consider the SQL query

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

Figure 3.1 Banking database for Exercises 3.8 and 3.15.

select $p.a1$
from p, $r1$, $r2$
where $p.a1 = r1.a1$ **or** $p.a1 = r2.a1$

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

f. Find the company that has the most employees.

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

Answer:

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

Figure 3.20. Employee database.

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

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

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

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

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

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

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

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

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

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

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

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

If people may work for several companies and we wish to consider the *total* earnings of each person, the problem is more complex. It can be solved by using a nested subquery, but we illustrate below how to solve it using the **with** clause.

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

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

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

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

Below is a solution using standard SQL.

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

f. Find the company that has the most employees.

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

CHAPTER 3

Introduction to SQL

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

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

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

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

Exercises

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

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

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

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

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

Answer:

a. SQL query:

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

b. SQL query:

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

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

c. SQL query:

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

d. SQL query:

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

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

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

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

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

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

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

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

Answer:

a. SQL query:

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

b. SQL query:

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

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

c. SQL query:

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

d. SQL query:

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

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

e. SQL query:

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

delete from section
where course_id = 'CS-001'

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

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

f. SQL query:

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

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

Answer:

a. SQL query:

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

Figure 3.18 Insurance database for Exercises 3.4 and 3.14.

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

b. SQL query:

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

c. SQL query:

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

d. SQL query:

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

e. SQL query:

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

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

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

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

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

a. SQL query:

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

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

b. SQL query:

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

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

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

Figure 3.19 Banking database for Exercises 3.8 and 3.15.

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

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

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

Answer:

a. SQL query:

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

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

b. SQL query:

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

c. SQL query:

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

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

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

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

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

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

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

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

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

e. Find the company that has the smallest payroll.

Answer:

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

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

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

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

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

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

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

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

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

e. Find the company that has the smallest payroll.

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.21 Library database for Exercise 3.21.

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

3.22 Rewrite the **where** clause

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

without using the **unique** construct.

Answer:

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

3.23 Consider the query:

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

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

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

3.24 Consider the query:

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

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

Answer:

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

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

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

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

CHAPTER 4

Intermediate SQL

Exercises

4.1 Write the following queries in SQL:

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

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

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "—".

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

Answer:

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

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

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

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

```
select ID, name,
    (select count(*) as 'Number of sections'
    from teaches T where T.id = I.id)
from instructor I
```

c. Display the list of all course sections offered in Spring 2010, along with the names of the instructors teaching the section. If a section has more than one instructor, it should appear as many times in the result as it has instructors. If it does not have any instructor, it should still appear in the result with the instructor name set to "–".

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

The query may also be written using the **coalesce** operator, by replacing **decode**(..) by **coalesce**(*name*, '–'). A more complex version of the query can be written using union of join result with another query that uses a subquery to find courses that do not match; refer to exercise 4.2.

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

b. Is the above pattern, with C null and D not null possible in the result of the first expression? Explain why or why not.

Answer:

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

b. It is not possible for D to be not null while C is null in the result of the first expression, since in the subexpression s natural left outer join t, it is not possible for C to be null while D is not null. In the overall expression C can be null if and only if some r tuple does

not have a matching B value in s. However in this case D will also be null.

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

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

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

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

Hint: use the queries from Exercise **??**.

Answer:

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

instructor = {(12345,'Guass', 'Physics', 10000)}
teaches = {(12345, 'EE321', 1, 'Spring', 2009)}
course = {('EE321', 'Magnetism', 'Elec. Eng.', 6)}

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

c. Consider the query

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

In the above query, we would lose some sections if *teaches*.ID is allowed to be **NULL** and such tuples exist. If, just because

teaches.ID is a foreign key to *instructor*, we did not create such a tuple, the error in the above query would not be detected.

4.5 Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise **??**; recall that we used a relation *grade_points*(*grade*, *points*) to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.
Answer: We should not add credits for courses with a null grade; further to to correctly handle the case where a student has not completed any course, we should make sure we don't divide by zero, and should instead return a null value.
We break the query into a subquery that finds sum of credits and sum of credit-grade-points, taking null grades into account The outer query divides the above to get the average, taking care of divide by 0.

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

The view defined above takes care of **NULL** grades by considering the creditpoints to be 0, and not adding the corresponding credits in *credit_sum*.
The query above ensures that if the student has not taken any course with non-NULL credits, and has *credit_sum* = 0 gets a gpa of **NULL**. This avoid the division by 0, which would otherwise have resulted.
An alternative way of writing the above query would be to use *student* **natural left outer join** *gpa*, in order to consider students who have not taken any course.

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

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

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

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

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

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

Answer:

CHAPTER 4

Intermediate SQL

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

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

Exercises

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

Answer:

a.

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

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

Figure 4.11 Employee database for Figure 4.7 and 4.12.

b.

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

4.13 Under what circumstances would the query

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

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

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

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

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

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

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

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

Answer:

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

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

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

Figure 4.12 Employee database for Exercise 4.16.

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

Answer:

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

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

a. Propose a syntax for expressing such constraints.

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

Answer:

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

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

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

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

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

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

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

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

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

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

CHAPTER 5

Advanced SQL

Practice Exercises

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

Answer: FILL

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

Answer:

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

5.10 Given a relation $S(student, subject, marks)$, write a query to find the top n students by total marks, by using ranking.

Answer: We assume that multiple students do not have the same marks since otherwise the question is not deterministic; the query below deterministically returns all students with the same marks as the n student, so it may return more than n students.

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

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

Answer:

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

CHAPTER 5

Advanced SQL

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

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

Exercises

5.12 Consider the following relations for a company database:

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

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

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

Figure 5.26 Java code for Exercise 5.12.

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

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

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

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

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

Answer:

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

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

c. The method is shown below.

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

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

Answer:

a. Same as for JDBC.

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

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

5.15 Consider an employee database with two relations

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

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

a. Using SQL functions as appropriate.

b. Without using SQL functions.

Answer:

a.

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

b.

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

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

Answer:

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

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

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

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

prereq_depth(*course_id*, *prereq_id*, *depth*)

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

Answer:

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

select *
from prereq_depth
```

5.19 Consider the relational schema

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

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

Answer:

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

    select *
        from total_part
```

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

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

SQL function:

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

returns table(number integer)

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

  insert into newpart
    select subpart_id, count
```

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

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

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

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

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

JDBC function:

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

b. **avg**

c. Standard deviation

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

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

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

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

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

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

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

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

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

which we get by expanding the formula

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

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

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

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

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

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

Answer:

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

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

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

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

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

Answer:

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

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

Answer:

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

CHAPTER 6

Formal Relational Query Languages

Practice Exercises

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

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

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

c. Find the highest salary of any instructor.

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

e. Find the enrollment of each section that was offered in Autumn 2009.

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

If people may not work for any company:

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

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

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

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

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

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

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

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

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

Given the above definition:

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

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

Answer:

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

b. The required expression is as follows:
$r \leftarrow \Pi_{ID, course\_id}(takes)$

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

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

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

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

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

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

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

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

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

6.5 Let the following relation schemas be given:

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

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

a. $\Pi_A(r)$

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

c. $r \times s$

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

Answer:

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

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

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

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

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

a. $\Pi_A(r_1)$

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

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

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

Answer:

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

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

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

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

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

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

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

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

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

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

Answer:

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

b. $r \bowtie s$

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

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

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

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

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

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

Answer:

a.

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

b.

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

c.

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

d.

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

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

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

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

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

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

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

can be written as $R1 \bowtie_P R2$.

An SQL join expression of the form

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

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

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

CHAPTER 6

Formal Relational Query Languages

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

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

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

Exercises

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

a. Using an aggregate function.

b. Without using any aggregate functions.

Answer:

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

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

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

a. Find the company with the most employees.

b. Find the company with the smallest payroll.

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

Answer:

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

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

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

6.14 Consider the following relational schema for a library:

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

Write the following queries in relational algebra.

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If people may not work for any company:

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

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

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

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

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

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

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

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

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

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

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

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

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

Answer:

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

b. $r \bowtie s$

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

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

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

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

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

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

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

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

a. $r \ltimes s$

b. $r \rtimes s$

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

Answer:

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

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

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

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

CHAPTER 7

Database Design and the E-R Model

Practice Exercises

7.1 **Answer:** The E-R diagram is shown in Figure 7.1. Payments are modeled as weak entities since they are related to a specific policy.
Note that the participation of accident in the relationship *participated* is not total, since it is possible that there is an accident report where the participating car is unknown.

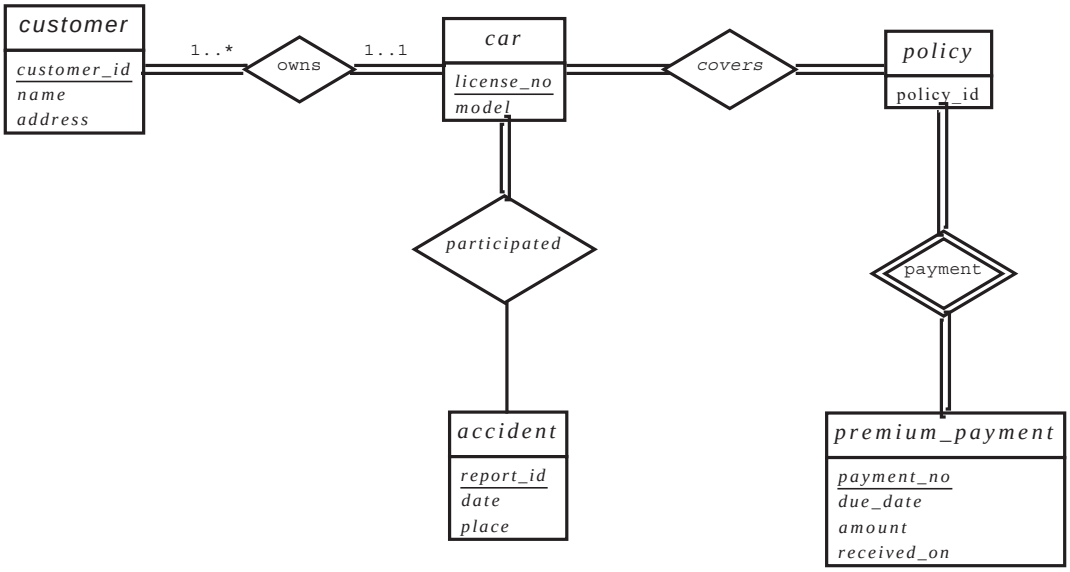


Figure 7.1 E-R diagram for a car insurance company.

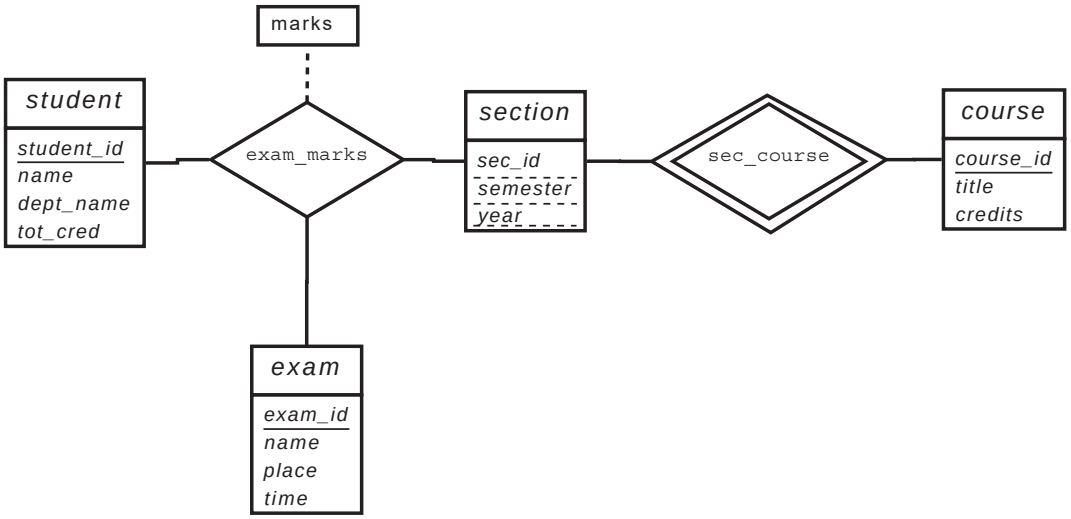


Figure 7.2 E-R diagram for marks database.

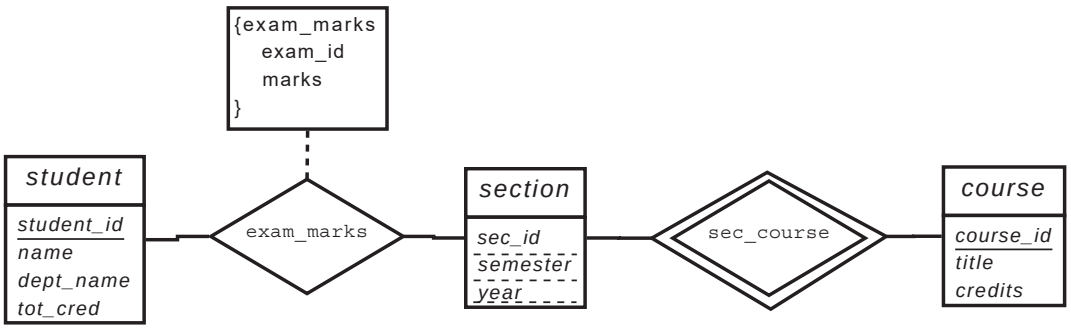


Figure 7.3 Another E-R diagram for marks database.

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

a. The E-R diagram is shown in Figure 7.2. Note that an alterantive is to model examinations as weak entities related to a section, rather than as a strong entity. The marks relationship would then be a binary relationship between *student* and *exam*, without directly involving *section*.

b. The E-R diagram is shown in Figure 7.3. Note that here we have not modeled the name, place and time of the exam as part of the relationship attributes. Doing so would result in duplication of the information, once per student, and we would not be able to record this information without an associated student. If we wish to represent this information, we would need to retain a separate entity corresponding to each exam.

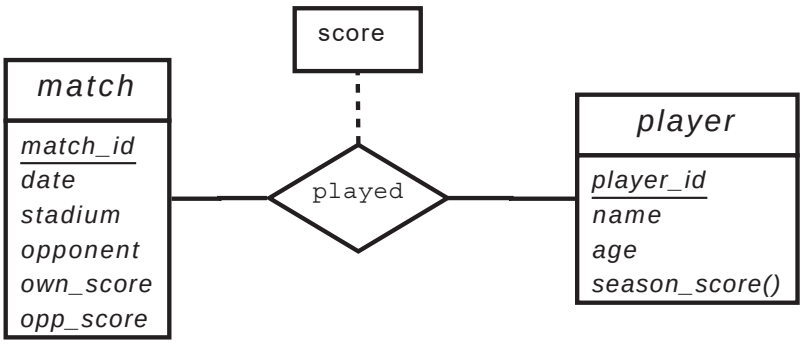


Figure 7.4 E-R diagram for favourite team statistics.

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

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

7.5 **Answer:**

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

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

7.6 **Answer:**

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

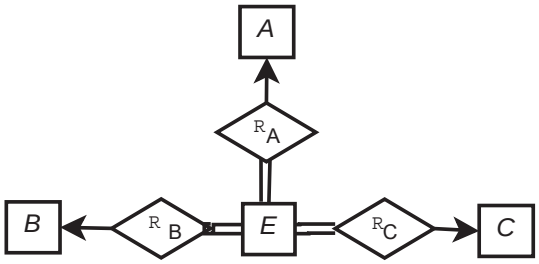


Figure 7.5 E-R diagram for Exercise 7.6b.

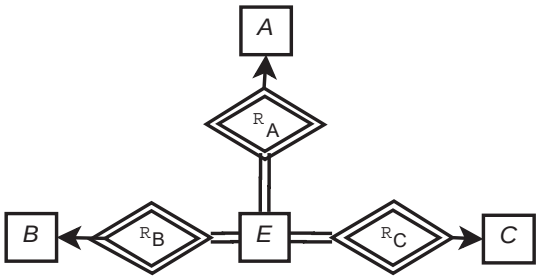


Figure 7.6 E-R diagram for Exercise 7.6d.

b. See Figure 7.5. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B and C.

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

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

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

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

7.9 **Answer:**

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

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

7.11 **Answer:**

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

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

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

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

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

7.13 **Answer:**

a. The E-R diagram is shown in Figure 7.7. The primary key attributes *student_id* and *instructor_id* are assumed to be immutable, that is they are not allowed to change with time. All other attributes are assumed to potentially change with time.
Note that the diagram uses multivalued composite attributes such as *valid_times* or *name*, with sub attributes such as *start_time* or *value*.

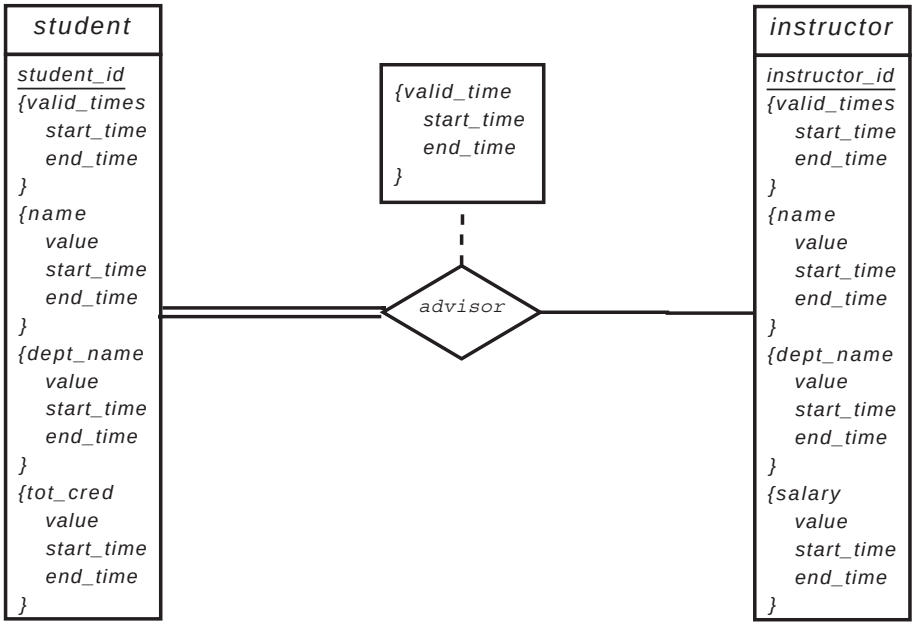


Figure 7.7 E-R diagram for Exercise 7.13

The *value* attribute is a subattribute of several attributes such as *name*, *tot_cred* and *salary*, and refers to the name, total credits or salary during a particular interval of time.

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

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

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

CHAPTER 7

Database Design and the E-R Model

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

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

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

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

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

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

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

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

Exercises

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

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

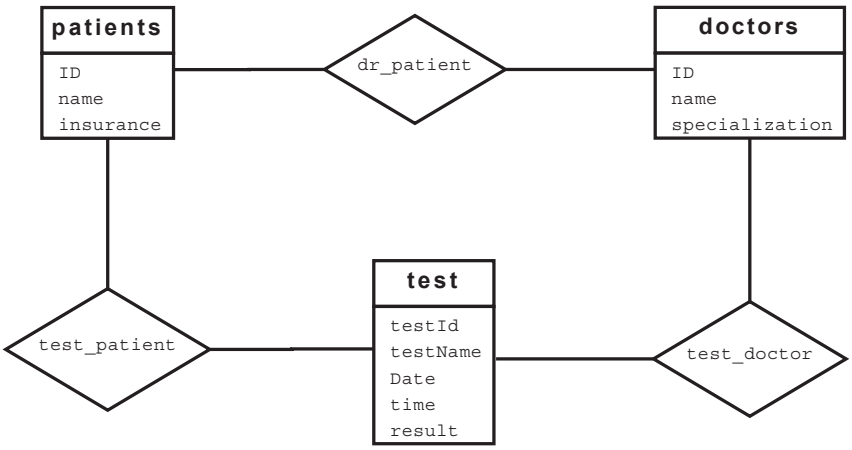


Figure 7.1 E-R diagram for a hospital.

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

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

Answer:

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

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

Answer:

a. Car insurance tables:

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

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

b. Student Exam tables:
   i. Ternary Relationship:

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

   ii. Binary relationship:

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

c. Player Match tables:

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

7.17 Extend the E-R diagram of Practice Exercise 7.3 to track the same information for all teams in a league.
Answer: See Figure 7.2. Note that we assume a player can play in only one team; if a player may switch teams, we would have to track for each match which team the player was in, which we could do by turning the relationship *played* into a ternary relationship.

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

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

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

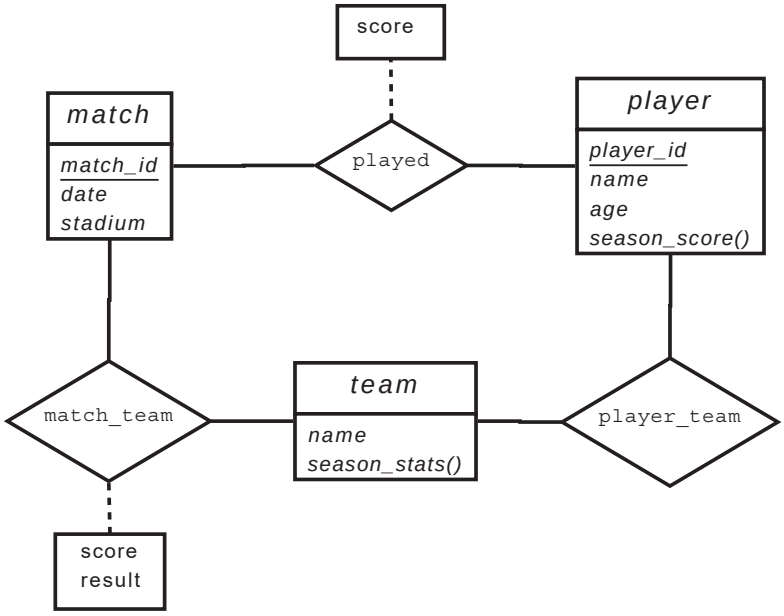


Figure 7.2 E-R diagram for all teams statistics.

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

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

a. List the entity sets and their primary keys.

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

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

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

a. Entity sets:

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

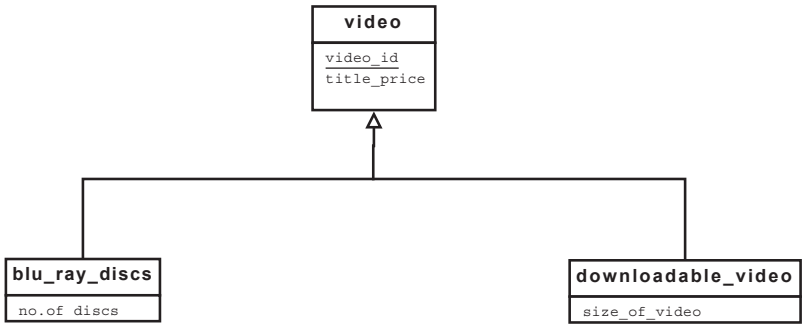


Figure 7.3 ER Diagram for Exercise 7.20 (b)

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

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

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

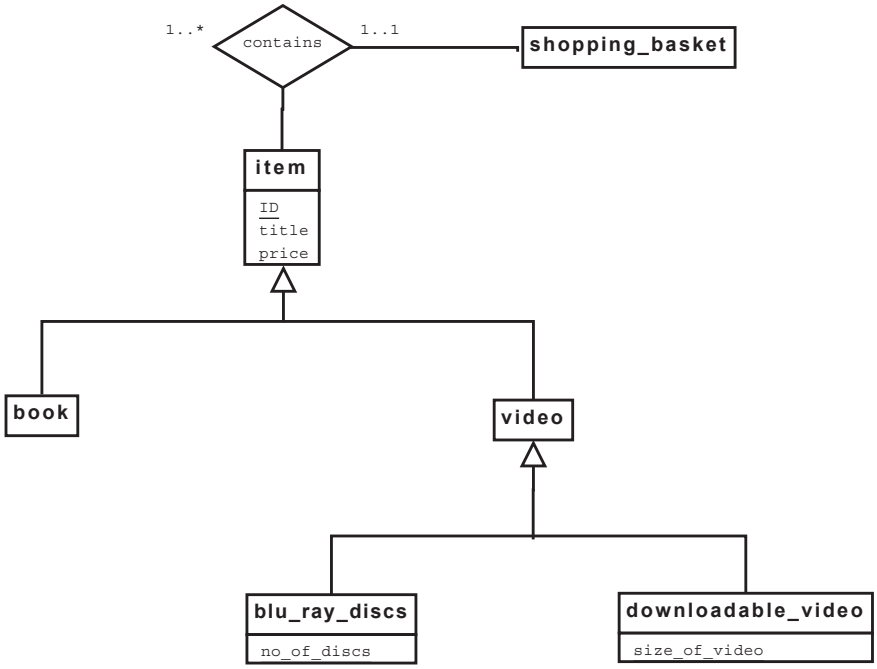


Figure 7.4 ER Diagram for Exercise 7.20 (c)

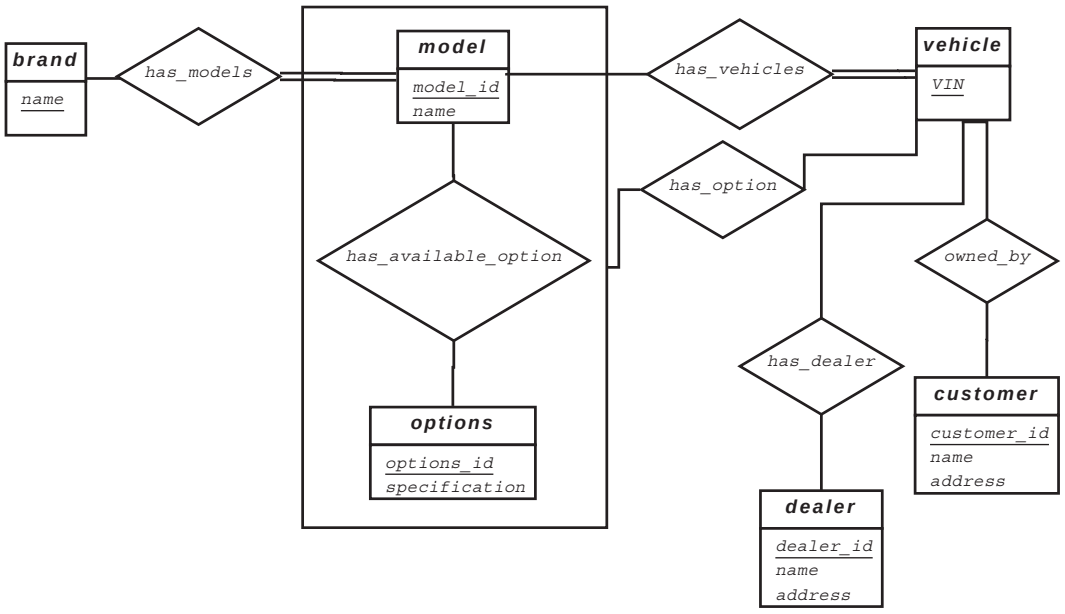


Figure 7.5 ER Diagram for Exercise 7.21

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

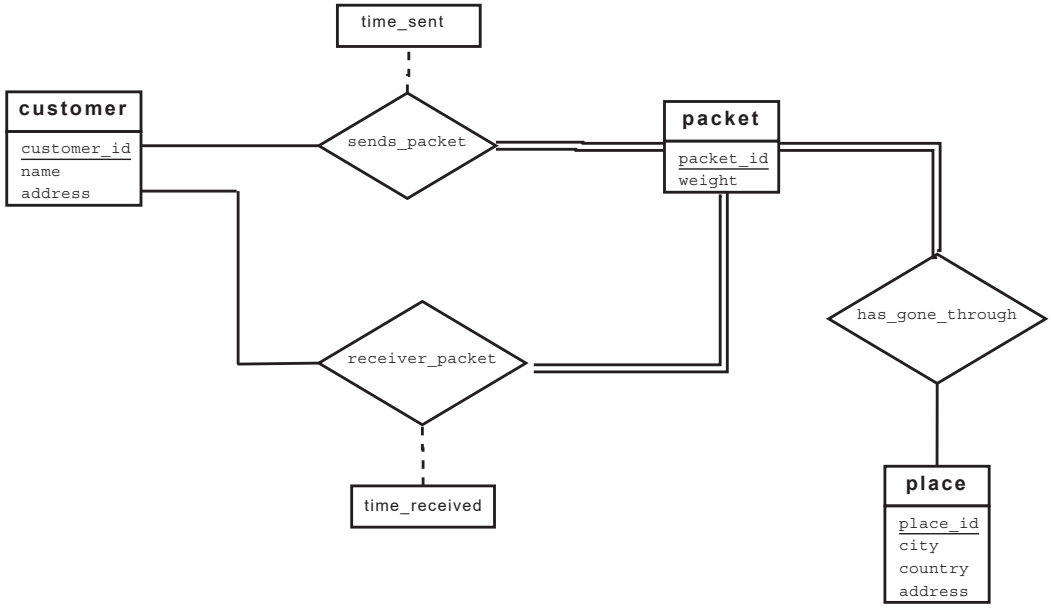
Your design should include an E-R diagram, a set of relational schemas, and a list of constraints, including primary-key and foreign-key constraints.

Answer:

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

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

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



As an alternative:

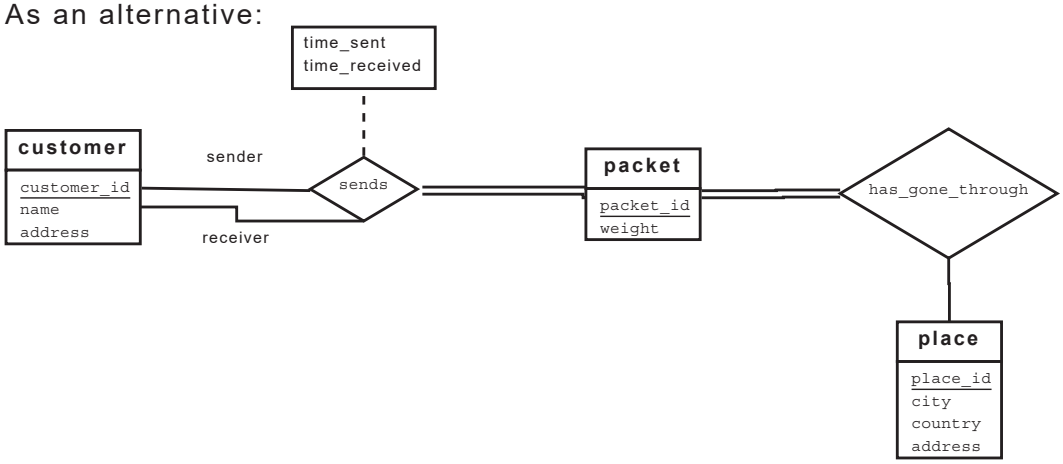


Figure 7.6 ER Diagram Alternatives for Exercise 7.22

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

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

Answer:

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

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

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

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

Answer:

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

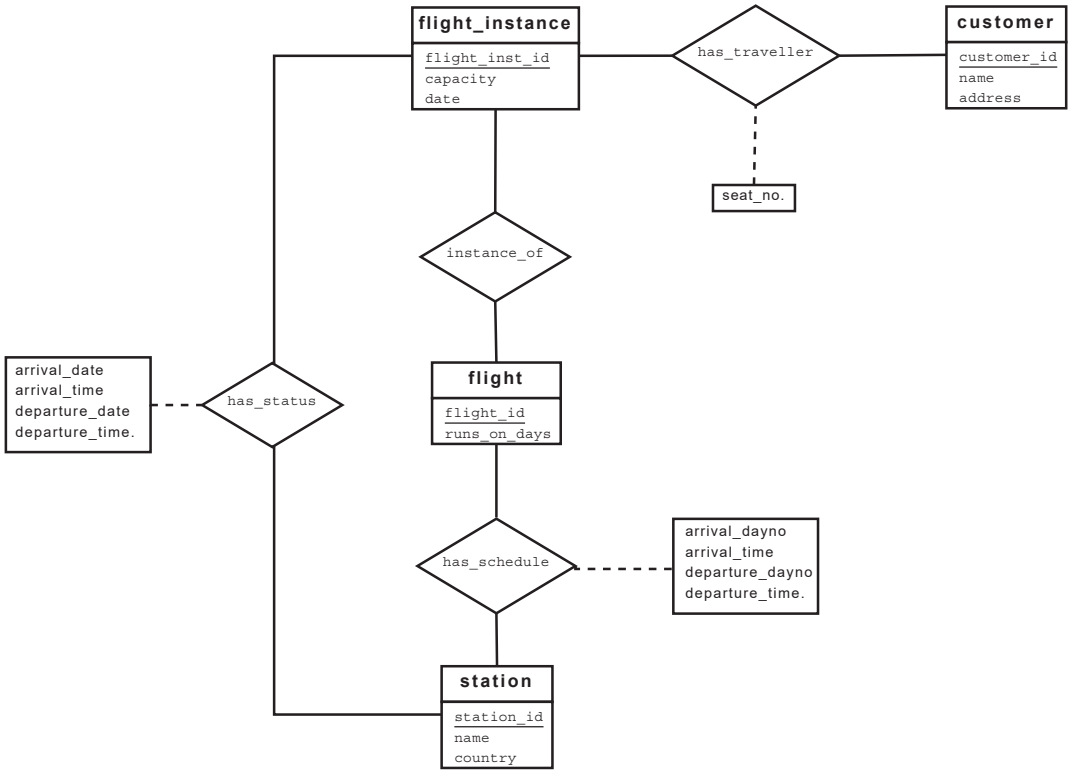


Figure 7.7 ER Diagram for Exercise 7.23

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

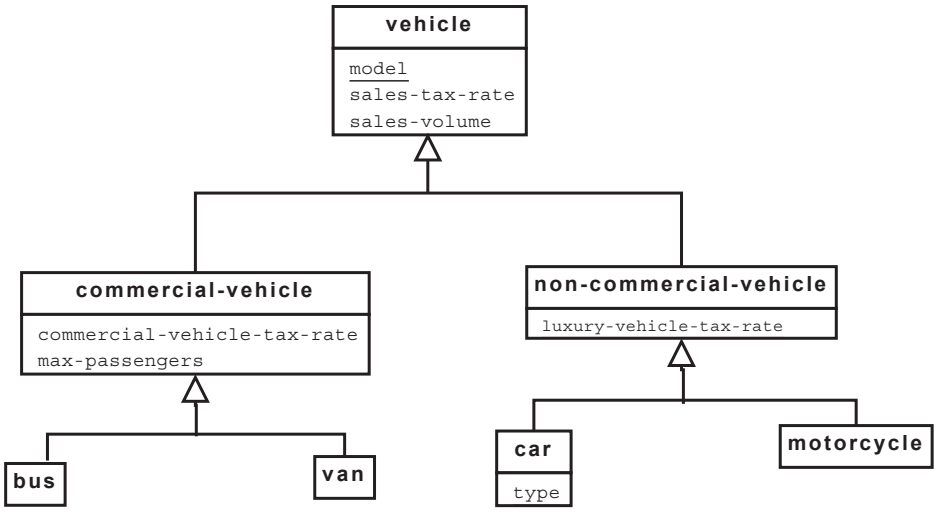


Figure 7.8 E-R diagram of motor-vehicle sales company.

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

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

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

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

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

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

7.28 Explain the distinction between disjoint and overlapping constraints.

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

7.29 Explain the distinction between total and partial constraints.

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

CHAPTER 8

Relational Database Design

Exercises

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

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

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

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

Answer: A decomposition $\{R_1, R_2\}$ is a lossless-join decomposition if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$. Let $R_1 = (A, B, C)$, $R_2 = (A, D, E)$, and $R_1 \cap R_2 = A$. Since A is a candidate key (see Practice Exercise 8.6), Therefore $R_1 \cap R_2 \rightarrow R_1$.

8.2 List all functional dependencies satisfied by the relation of Figure 8.17.
Answer: The nontrivial functional dependencies are: $A \rightarrow B$ and $C \rightarrow B$, and a dependency they logically imply: $AC \rightarrow B$. There are 19 trivial functional dependencies of the form $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$. C does not functionally determine A because the first and third tuples have the same C but different A values. The same tuples also show B does not functionally determine A. Likewise, A does not functionally determine C because the first two tuples have the same A value and different C values. The same tuples also show B does not functionally determine C.

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

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

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

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

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

Answer: To prove that:

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

Following the hint, we derive:

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

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

Answer: Proof using Armstrong's axioms of the Pseudotransitivity Rule:
if $\alpha \rightarrow \beta$ and $\gamma\,\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$.

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

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

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

List the candidate keys for R.

Answer: Note: It is not reasonable to expect students to enumerate all of F^+. Some shorthand representation of the result should be acceptable as long as the nontrivial members of F^+ are found.

Starting with $A \rightarrow BC$, we can conclude: $A \rightarrow B$ and $A \rightarrow C$.

Since $A \rightarrow B$ and $B \rightarrow D$, $A \rightarrow D$	(decomposition, transitive)
Since $A \rightarrow CD$ and $CD \rightarrow E$, $A \rightarrow E$	(union, decomposition, transitive)
Since $A \rightarrow A$, we have	(reflexive)
$A \rightarrow ABCDE$ from the above steps	(union)
Since $E \rightarrow A$, $E \rightarrow ABCDE$	(transitive)
Since $CD \rightarrow E$, $CD \rightarrow ABCDE$	(transitive)
Since $B \rightarrow D$ and $BC \rightarrow CD$, $BC \rightarrow ABCDE$	(augmentative, transitive)
Also, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow D$, etc.	

Therefore, any functional dependency with A, E, BC, or CD on the left hand side of the arrow is in F^+, no matter which other attributes appear in the FD. Allow * to represent any set of attributes in R, then F^+ is $BD \rightarrow B$, $BD \rightarrow D$, $C \rightarrow C$, $D \rightarrow D$, $BD \rightarrow BD$, $B \rightarrow D$, $B \rightarrow B$, $B \rightarrow BD$, and all FDs of the form $A* \rightarrow \alpha$, $BC* \rightarrow \alpha$, $CD* \rightarrow \alpha$, $E* \rightarrow \alpha$ where α is any subset of $\{A, B, C, D, E\}$. The candidate keys are A, BC, CD, and E.

8.7 Using the functional dependencies of Practice Exercise 8.6, compute the canonical
cover F_c.

Answer: The given set of FDs F is:-

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

The left side of each FD in F is unique. Also none of the attributes in the left side or right side of any of the FDs is extraneous. Therefore the canonical cover F_c is equal to F.

8.8 Consider the algorithm in Figure 8.18 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 8.8 (Section 8.4.2) and that it computes α^+ correctly.
Answer: The algorithm is correct because:

- If A is added to $result$ then there is a proof that $\alpha \rightarrow A$. To see this, observe that $\alpha \rightarrow \alpha$ trivially so α is correctly part of $result$. If $A \notin \alpha$ is added to $result$ there must be some FD $\beta \rightarrow \gamma$ such that $A \in \gamma$ and β is already a subset of $result$. (Otherwise $fdcount$ would be nonzero and the **if** condition would be false.) A full proof can be given by induction on the depth of recursion for an execution of **addin**, but such a proof can be expected only from students with a good mathematical background.
- If $A \in \alpha^+$, then A is eventually added to $result$. We prove this by induction on the length of the proof of $\alpha \rightarrow A$ using Armstrong's axioms. First observe that if procedure **addin** is called with some argument β, all the attributes in β will be added to $result$. Also if a particular FD's *fdcount* becomes 0, all the attributes in its tail will definitely be added to $result$. The base case of the proof, $A \in \alpha \Rightarrow A \in \alpha^+$, is obviously true because the first call to **addin** has the argument α. The inductive hypotheses is that if $\alpha \rightarrow A$ can be proved in n steps or less then $A \in result$. If there is a proof in $n+1$ steps that $\alpha \rightarrow A$, then the last step was an application of either reflexivity, augmentation or transitivity on a fact $\alpha \rightarrow \beta$ proved in n or fewer steps. If reflexivity or augmentation was used in the $(n+1)^{st}$ step, A must have been in $result$ by the end of the n^{th} step itself. Otherwise, by the inductive hypothesis $\beta \subseteq result$. Therefore the dependency used in proving $\beta \rightarrow \gamma$, $A \in \gamma$ will have $fdcount$ set to 0 by the end of the n^{th} step. Hence A will be added to $result$.

To see that this algorithm is more efficient than the one presented in the chapter note that we scan each FD once in the main program. The resulting array $appears$ has size proportional to the size of the given FDs. The recursive calls to **addin** result in processing linear in the size of $appears$. Hence the algorithm has time complexity which is linear in the size of the given FDs. On the other hand, the algorithm given in the text has quadratic time complexity, as it may perform the loop as many times as the number of FDs, in each loop scanning all of them once.

8.9 Given the database schema $R(a, b, c)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $b \rightarrow c$ holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present. (Although part of the SQL standard, such assertions are not supported by any database implementation currently.)
Answer:

a. The query is given below. Its result is non-empty if and only if $b \rightarrow c$ does not hold on r.

```
select b
from r
group by b
having count(distinct c) > 1
```

b.

```
create assertion b_to_c check
    (not exists
        (select b
         from r
         group by b
         having count(distinct c) > 1
        )
    )
```

8.10 Our discussion of lossless-join decomposition implicitly assumed that attributes on the left-hand side of a functional dependency cannot take on null values. What could go wrong on decomposition, if this property is violated?

Answer: The natural join operator is defined in terms of the cartesian product and the selection operator. The selection operator, gives *unknown* for any query on a null value. Thus, the natural join excludes all tuples with null values on the common attributes from the final result. Thus, the decomposition would be lossy (in a manner different from the usual case of lossy decomposition), if null values occur in the left-hand side of the functional dependency used to decompose the relation. (Null values in attributes that occur only in the right-hand side of the functional dependency do not cause any problems.)

8.11 In the BCNF decomposition algorithm, suppose you use a functional dependency $\alpha \rightarrow \beta$ to decompose a relation schema $r(\alpha, \beta, \gamma)$ into $r_1(\alpha, \beta)$ and $r_2(\alpha, \gamma)$.

a. What primary and foreign-key constraint do you expect to hold on the decomposed relations?

b. Give an example of an inconsistency that can arise due to an erroneous update, if the foreign-key constraint were not enforced on the decomposed relations above.

c. When a relation is decomposed into 3NF using the algorithm in Section 8.5.2, what primary and foreign key dependencies would you expect will hold on the decomposed schema?

Answer:

a. α should be a primary key for r_1, and α should be the foreign key from r_2, referencing r_1.

b. If the foreign key constraint is not enforced, then a deletion of a tuple from r_1 would not have a corresponding deletion from the referencing tuples in r_2. Instead of deleting a tuple from r, this would amount to simply setting the value of α to null in some tuples.

c. For every schema $r_i(\alpha\beta)$ added to the schema because of a rule $\alpha \rightarrow \beta$, α should be made the primary key. Also, a candidate key γ for the original relation is located in some newly created relation r_k, and is a primary key for that relation.
Foreign key constraints are created as follows: for each relation r_i created above, if the primary key attributes of r_i also occur in any other relation r_j, then a foreign key constraint is created from those attributes in r_j, referencing (the primary key of) r_i.

8.12 Let $R_1, R_2, \ldots, R_n$ be a decomposition of schema U. Let $u(U)$ be a relation, and let $r_i = \Pi_{R_i}(u)$. Show that

$$u \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

Answer: Consider some tuple t in u.
Note that $r_i = \Pi_{R_i}(u)$ implies that $t[R_i] \in r_i$, $1 \leq i \leq n$. Thus,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] \in r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

By the definition of natural join,

$$t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n] = \Pi_\alpha (\sigma_\beta (t[R_1] \times t[R_2] \times \ldots \times t[R_n]))$$

where the condition β is satisfied if values of attributes with the same name in a tuple are equal and where $\alpha = U$. The cartesian product of single tuples generates one tuple. The selection process is satisfied because all attributes with the same name must have the same value since they are projections from the same tuple. Finally, the projection clause removes duplicate attribute names.
By the definition of decomposition, $U = R_1 \cup R_2 \cup \ldots \cup R_n$, which means that all attributes of t are in $t[R_1] \bowtie t[R_2] \bowtie \ldots \bowtie t[R_n]$. That is, t is equal to the result of this join.
Since t is any arbitrary tuple in u,

$$u \subseteq r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$$

8.13 Show that the decomposition in Practice Exercise 8.1 is not a dependency-preserving decomposition.
Answer: The dependency $B \rightarrow D$ is not preserved. F_1, the restriction of F to (A, B, C) is $A \rightarrow ABC$, $A \rightarrow AB$, $A \rightarrow AC$, $A \rightarrow BC$,

$A \rightarrow B, A \rightarrow C, A \rightarrow A, B \rightarrow B, C \rightarrow C, AB \rightarrow AC, AB \rightarrow ABC, AB \rightarrow BC, AB \rightarrow AB, AB \rightarrow A, AB \rightarrow B, AB \rightarrow C, AC$ (same as AB), BC (same as AB), ABC (same as AB). F_2, the restriction of F to (C, D, E) is $A \rightarrow ADE, A \rightarrow AD, A \rightarrow AE, A \rightarrow DE, A \rightarrow A, A \rightarrow D, A \rightarrow E, D \rightarrow D, E$ (same as A), AD, AE, DE, ADE (same as A). $(F_1 \cup F_2)^+$ is easily seen not to contain $B \rightarrow D$ since the only FD in $F_1 \cup F_2$ with B as the left side is $B \rightarrow B$, a trivial FD. We shall see in Practice Exercise 8.15 that $B \rightarrow D$ is indeed in F^+. Thus $B \rightarrow D$ is not preserved. Note that $CD \rightarrow ABCDE$ is also not preserved.

A simpler argument is as follows: F_1 contains no dependencies with D on the right side of the arrow. F_2 contains no dependencies with B on the left side of the arrow. Therefore for $B \rightarrow D$ to be preserved there must be an FD $B \rightarrow \alpha$ in F_1^+ and $\alpha \rightarrow D$ in F_2^+ (so $B \rightarrow D$ would follow by transitivity). Since the intersection of the two schemes is A, $\alpha = A$. Observe that $B \rightarrow A$ is not in F_1^+ since $B^+ = BD$.

8.14 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless-join decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

Answer: Let F be a set of functional dependencies that hold on a schema R. Let $\sigma = \{R_1, R_2, \ldots, R_n\}$ be a dependency-preserving 3NF decomposition of R. Let X be a candidate key for R.

Consider a legal instance r of R. Let $j = \Pi_X(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$. We want to prove that $r = j$.

We claim that if t_1 and t_2 are two tuples in j such that $t_1[X] = t_2[X]$, then $t_1 = t_2$. To prove this claim, we use the following inductive argument – Let $F' = F_1 \cup F_2 \cup \ldots \cup F_n$, where each F_i is the restriction of F to the schema R_i in σ. Consider the use of the algorithm given in Figure 8.8 to compute the closure of X under F'. We use induction on the number of times that the for loop in this algorithm is executed.

- *Basis* : In the first step of the algorithm, $result$ is assigned to X, and hence given that $t_1[X] = t_2[X]$, we know that $t_1[result] = t_2[result]$ is true.
- *Induction Step* : Let $t_1[result] = t_2[result]$ be true at the end of the k th execution of the for loop.
  Suppose the functional dependency considered in the $k+1$ th execution of the for loop is $\beta \rightarrow \gamma$, and that $\beta \subseteq result$. $\beta \subseteq result$ implies that $t_1[\beta] = t_2[\beta]$ is true. The facts that $\beta \rightarrow \gamma$ holds for some attribute set R_i in σ, and that $t_1[R_i]$ and $t_2[R_i]$ are in $\Pi_{R_i}(r)$ imply that $t_1[\gamma] = t_2[\gamma]$ is also true. Since γ is now added to $result$ by the algorithm, we know that $t_1[result] = t_2[result]$ is true at the end of the $k+1$ th execution of the for loop.

Since σ is dependency-preserving and X is a key for R, all attributes in R are in $result$ when the algorithm terminates. Thus, $t_1[R] = t_2[R]$ is true, that is, $t_1 = t_2$ – as claimed earlier.
Our claim implies that the size of $\Pi_X(j)$ is equal to the size of j. Note also that $\Pi_X(j) = \Pi_X(r) = r$ (since X is a key for R). Thus we have proved that the size of j equals that of r. Using the result of Practice Exercise 8.12, we know that $r \subseteq j$. Hence we conclude that $r = j$.
Note that since X is trivially in 3NF, $\sigma \cup \{X\}$ is a dependency-preserving lossless-join decomposition into 3NF.

8.15 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless-join decompositions of R' into BCNF.
Answer: Given the relation $R' = (A, B, C, D)$ the set of functional dependencies $F' = A \rightarrow B, C \rightarrow D, B \rightarrow C$ allows three distinct BCNF decompositions.

$$R_1 = \{(A, B), (C, D), (B, C)\}$$

is in BCNF as is

$$R_2 = \{(A, B), (C, D), (A, C)\}$$

$$R_2 = \{(A, B), (C, D), (A, C)\}$$

$$R_3 = \{(B, C), (A, D), (A, B)\}$$

8.16 Let a *prime* attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let A be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that A is *transitively dependent* on α. We can restate our definition of 3NF as follows: A relation schema R is in 3NF with respect to a set F of functional dependencies if there are no nonprime attributes A in R for which A is transitively dependent on a key for R.
Show that this new definition is equivalent to the original one.
Answer: Suppose R is in 3NF according to the textbook definition. We show that it is in 3NF according to the definition in the exercise. Let A be a nonprime attribute in R that is transitively dependent on a key α for R. Then there exists $\beta \subseteq R$ such that $\beta \rightarrow A$, $\alpha \rightarrow \beta$, $A \notin \alpha$, $A \notin \beta$, and $\beta \rightarrow \alpha$ does not hold. But then $\beta \rightarrow A$ violates the textbook definition of 3NF since

- $A \notin \beta$ implies $\beta \rightarrow A$ is nontrivial
- Since $\beta \rightarrow \alpha$ does not hold, β is not a superkey
- A is not any candidate key, since A is nonprime

Now we show that if R is in 3NF according to the exercise definition, it is in 3NF according to the textbook definition. Suppose R is not in 3NF according the the textbook definition. Then there is an FD $\alpha \rightarrow \beta$ that fails all three conditions. Thus

- $\alpha \rightarrow \beta$ is nontrivial.
- α is not a superkey for R.
- Some A in $\beta - \alpha$ is not in any candidate key.

This implies that A is nonprime and $\alpha \rightarrow A$. Let γ be a candidate key for R. Then $\gamma \rightarrow \alpha$, $\alpha \rightarrow \gamma$ does not hold (since α is not a superkey), $A \notin \alpha$, and $A \notin \gamma$ (since A is nonprime). Thus A is transitively dependent on γ, violating the exercise definition.

8.17 A functional dependency $\alpha \rightarrow \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$. We say that β is *partially dependent* on α. A relation schema R is in **second normal form** (2NF) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

Answer: Referring to the definitions in Practice Exercise 8.16, a relation schema R is said to be in 3NF if there is no non-prime attribute A in R for which A is transitively dependent on a key for R.

We can also rewrite the definition of 2NF given here as :

"A relation schema R is in 2NF if no non-prime attribute A is partially dependent on any candidate key for R."

To prove that every 3NF schema is in 2NF, it suffices to show that if a non-prime attribute A is partially dependent on a candidate key α, then A is also transitively dependent on the key α.

Let A be a non-prime attribute in R. Let α be a candidate key for R. Suppose A is partially dependent on α.

- From the definition of a partial dependency, we know that for some proper subset β of α, $\beta \rightarrow A$.
- Since $\beta \subset \alpha$, $\alpha \rightarrow \beta$. Also, $\beta \rightarrow \alpha$ does not hold, since α is a candidate key.
- Finally, since A is non-prime, it cannot be in either β or α.

Thus we conclude that $\alpha \rightarrow A$ is a transitive dependency. Hence we have proved that every 3NF schema is also in 2NF.

8.18 Give an example of a relation schema R and a set of dependencies such that R is in BCNF but is not in 4NF.

Answer:

$R(A, B, C)$
$A \twoheadrightarrow B$

```
result := ∅;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in α+ */
for i := 1 to |F| do
   begin
      let β → γ denote the ith FD;
      fdcount [i] := |β|;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
      appears [A] := NIL;
      for i := 1 to |F| do
         begin
            let β → γ denote the ith FD;
            if A ∈ β then add i to appears [A];
         end
   end
addin (α);
return (result);

procedure addin (α);
for each attribute A in α do
   begin
      if A ∉ result then
         begin
            result := result ∪ {A};
            for each element i of appears[A] do
               begin
                  fdcount [i] := fdcount [i] − 1;
                  if fdcount [i] := 0 then
                     begin
                        let β → γ denote the ith FD;
                        addin (γ);
                     end
               end
         end
   end
```

Figure 8.18. An algorithm to compute α^+.

CHAPTER 8

Relational Database Design

This chapter presents the principles of relational database design. Undergraduates frequently find this chapter difficult. It is acceptable to cover only Sections 8.1 and 8.3 for classes that find the material particularly difficult. However, a careful study of data dependencies and normalization is a good way to introduce students to the formal aspects of relational database theory.

There are many ways of stating the definitions of the normal forms. We have chosen a style which we think is the easiest to present and which most clearly conveys the intuition of the normal forms.

Exercises

8.19 Give a lossless-join decomposition into BCNF of schema R of Exercise 8.1.

Answer: From Exercise 8.6, we know that $B \rightarrow D$ is nontrivial and the left hand side is not a superkey. By the algorithm of Figure 8.11 we derive the relations $\{(A, B, C, E), (B, D)\}$. This is in BCNF.

8.20 Give a lossless-join, dependency-preserving decomposition into 3NF of schema R of Practice Exercise 8.1.

Answer: First we note that the dependencies given in Practice Exercise 8.1 form a canonical cover. Generating the schema from the algorithm of Figure 8.12 we get

$$R' = \{(A, B, C), (C, D, E), (B, D), (E, A)\}.$$

Schema (A, B, C) contains a candidate key. Therefore R' is a third normal form dependency-preserving lossless-join decomposition.

Note that the original schema $R = (A, B, C, D, E)$ is already in 3NF. Thus, it was not necessary to apply the algorithm as we have done above. The single original schema is trivially a lossless join, dependency-preserving decomposition.

8.21 Normalize the following schema, with given constraints, to 4NF.

$$
\begin{aligned}
&\textit{books}(\textit{accessionno}, \textit{isbn}, \textit{title}, \textit{author}, \textit{publisher})\\
&\textit{users}(\textit{userid}, \textit{name}, \textit{deptid}, \textit{deptname})\\
&\textit{accessionno} \rightarrow \textit{isbn}\\
&\textit{isbn} \rightarrow \textit{title}\\
&\textit{isbn} \rightarrow \textit{publisher}\\
&\textit{isbn} \twoheadrightarrow \textit{author}\\
&\textit{userid} \rightarrow \textit{name}\\
&\textit{userid} \rightarrow \textit{deptid}\\
&\textit{deptid} \rightarrow \textit{deptname}
\end{aligned}
$$

Answer: In *books*, we see that

$$\textit{isbn} \twoheadrightarrow \textit{title}, \textit{publisher}, \textit{author}$$

and yet, *isbn* is not a super key. Thus, we break *books* into

$$
\begin{aligned}
&\textit{books\_accnno}(\textit{accessionno}, \textit{isbn})\\
&\textit{books\_details}(\textit{isbn}, \textit{title}, \textit{publisher}, \textit{author})
\end{aligned}
$$

After this, we still have

$$\textit{isbn} \twoheadrightarrow \textit{author}$$

but neither is *isbn* a primary key of *book_details*, nor are the attributes of *book_details* equal to $\{\textit{isbn}\} \cup \{\textit{author}\}$. Therefore we decompose *book_details* again into

$$
\begin{aligned}
&\textit{books\_details1}(\textit{isbn}, \textit{title}, \textit{publisher})\\
&\textit{books\_authors}(\textit{isbn}, \textit{author})
\end{aligned}
$$

Similarly, in *users*,

$$\textit{deptid} \rightarrow \textit{deptname}$$

and yet, *deptid* is not a super key. Hence, we break *users* to

$$
\begin{aligned}
&\textit{users}(\textit{userid}, \textit{name}, \textit{deptid})\\
&\textit{departments}(\textit{deptid}, \textit{deptname})
\end{aligned}
$$

We verify that there are no further functional or multivalued dependencies that cause violation of 4NF, so the final set of relations are:

books_accnno(*accessionno*, *isbn*)
books_details1(*isbn*, *title*, *publisher*)
books_authors(*isbn*, *author*) *users*(*userid*, *name*, *deptid*)
departments(*deptid*, *deptname*)

8.22 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

Answer:

- Repetition of information is a condition in a relational database where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult, and can lead to inconsistent data if updates are done to one copy of the value, but not to another.
- Inability to represent information is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values otherwise a tuple without the unrelated information cannot be inserted into the relation.
- Inability to represent information can also occur because of loss of information which results from the decomposition of one relation into two relations, which cannot be combined to recreate the original relation. Such a lossy decomposition may happen implicitly, even without explicitly carrying out decomposition, if the initial relational schema itself corresponds to the decomposition.

8.23 Why are certain functional dependencies called *trivial* functional dependencies?

Answer: Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations.

8.24 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

Answer: The definition of functional dependency is: $\alpha \rightarrow \beta$ holds on R if in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Reflexivity rule: if α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.
Assume $\exists\, t_1$ and t_2 such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	since $\beta \subseteq \alpha$
$\alpha \rightarrow \beta$	definition of FD

Augmentation rule: if $\alpha \rightarrow \beta$, and γ is a set of attributes, then $\gamma\,\alpha \rightarrow \gamma\,\beta$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\gamma\,\alpha] = t_2[\gamma\,\alpha]$

$t_1[\gamma] = t_2[\gamma]$	$\gamma \subseteq \gamma\,\alpha$
$t_1[\alpha] = t_2[\alpha]$	$\alpha \subseteq \gamma\,\alpha$
$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma\,\beta] = t_2[\gamma\,\beta]$	$\gamma\,\beta = \gamma \cup \beta$
$\gamma\,\alpha \rightarrow \gamma\,\beta$	definition of FD

Transitivity rule: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.
Assume $\exists\, t_1,\ t_2$ such that $t_1[\alpha] = t_2[\alpha]$

$t_1[\beta] = t_2[\beta]$	definition of $\alpha \rightarrow \beta$
$t_1[\gamma] = t_2[\gamma]$	definition of $\beta \rightarrow \gamma$
$\alpha \rightarrow \gamma$	definition of FD

8.25 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation r that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

Answer: Consider the following rule: if $A \rightarrow B$ and $C \rightarrow B$, then $A \rightarrow C$. That is, $\alpha = A$, $\beta = B$, $\gamma = C$. The following relation r is a counterexample to the rule.

r:

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2

Note: $A \rightarrow B$ and $C \rightarrow B$, (since no 2 tuples have the same C value, $C \rightarrow B$ is true trivially). However, it is not the case that $A \rightarrow C$ since the same A value is in two tuples, but the C value in those tuples disagree.

8.26 Use Armstrong's axioms to prove the soundness of the decomposition rule.

Answer: The decomposition rule, and its derivation from Armstrong's axioms are given below:
if $\alpha \rightarrow \beta\gamma$, then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

$\alpha \rightarrow \beta\gamma$	given
$\beta\gamma \rightarrow \beta$	reflexivity rule
$\alpha \rightarrow \beta$	transitivity rule
$\beta\gamma \rightarrow \gamma$	reflexive rule
$\alpha \rightarrow \gamma$	transitive rule

8.27 Using the functional dependencies of Practice Exercise 8.6, compute B^+.

Answer: Computing B^+ by the algorithm in Figure 8.8 we start with $result = \{B\}$. Considering FDs of the form $\beta \rightarrow \gamma$ in F, we find that the only dependencies satisfying $\beta \subseteq result$ are $B \rightarrow B$ and $B \rightarrow D$. Therefore $result = \{B, D\}$. No more dependencies in F apply now. Therefore $B^+ = \{B, D\}$

8.28 Show that the following decomposition of the schema R of Practice Exercise 8.1 is not a lossless-join decomposition:

(A, B, C)
(C, D, E).

Hint: Give an example of a relation r on schema R such that

$$\Pi_{A,B,C}(r) \bowtie \Pi_{C,D,E}(r) \neq r$$

Answer: Following the hint, use the following example of r:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

With $R_1 = (A, B, C)$, $R_2 = (C, D, E)$:

a. $\Pi_{R_1}(r)$ would be:

A	B	C
a_1	b_1	c_1
a_2	b_2	c_1

b. $\Pi_{R_2}(r)$ would be:

C	D	E
c_1	d_1	e_1
c_1	d_2	e_2

c. $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$ would be:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_1	d_2	e_2
a_2	b_2	c_1	d_1	e_1
a_2	b_2	c_1	d_2	e_2

Clearly, $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \neq r$. Therefore, this is a lossy join.

8.29 Consider the following set F of functional dependencies on the relation schema $r(A, B, C, D, E, F)$:

$$A \rightarrow BCD$$
$$BC \rightarrow DE$$
$$B \rightarrow D$$
$$D \rightarrow A$$

a. Compute B^+.

b. Prove (using Armstrong's axioms) that AF is a superkey.

c. Compute a canonical cover for the above set of functional dependencies F; give each step of your derivation with an explanation.

d. Give a 3NF decomposition of r based on the canonical cover.

e. Give a BCNF decomposition of r using the original set of functional dependencies.

f. Can you get the same BCNF decomposition of r as above, using the canonical cover?

Answer:

a. $B \rightarrow BD$ (third dependency)
$BD \rightarrow ABD$ (fourth dependency)
$ABD \rightarrow ABCD$ (first dependency)
$ABCD \rightarrow ABCDE$ *(second dependency)*

Thus, $B^+ = ABCDE$

b. Prove (using Armstrong's axioms) that AF is a superkey.

$A \rightarrow BCD$ (Given)
$A \rightarrow ABCD$ (Augmentation with A)
$BC \rightarrow DE$ (Given)
$ABCD \rightarrow ABCDE$ (Augmentation with ABCD)
$A \rightarrow ABCDE$ (Transitivity)
$AF \rightarrow ABCDEF$ (Augmentation with F)

c. We see that D is extraneous in dep. 1 and 2, because of dep. 3. Removing these two, we get the new set of rules

$$A \rightarrow BC$$
$$BC \rightarrow E$$
$$B \rightarrow D$$
$$D \rightarrow A$$

Now notice that B^+ is $ABCDE$, and in particular, the FD $B \rightarrow E$ can be determined from this set. Thus, the attribute C is extraneous in

the third dependency. Removing this attribute, and combining with the third FD, we get the final canonical cover as :

$$A \rightarrow BC$$
$$B \rightarrow DE$$
$$D \rightarrow A$$

Here, no attribute is extraneous in any FD.

d. We see that there is no FD in the canonical cover such that the set of attributes is a subset of any other FD in the canonical cover. Thus, each each FD gives rise to its own relation, giving

$$r_1(A, B, C)$$
$$r_2(B, D, E)$$
$$r_3(D, A)$$

Now the attribute F is not dependent on any attribute. Thus, it must be a part of every superkey. Also, none of the relations in the above schema have F, and hence, none of them have a superkey. Thus, we need to add a new relation with a superkey.

$$r_4(A, F)$$

e. We start with

$$r(A, B, C, D, E, F)$$

We see that the relation is not in BCNF because of the first FD. Hence, we decompose it accordingly to get

$$r_1(A, B, C, D)\ r_2(A, E, F)$$

Now we notice that $A \rightarrow E$ is an FD in F^+, and causes r_2 to violate BCNF. Once again, decomposing r_2 gives

$$r_1(A, B, C, D)\ r_2(A, F)\ r_3(A, E)$$

This schema is now in BCNF.

f. Can you get the same BCNF decomposition of r as above, using the canonical cover?
If we use the functional dependencies in the preceding canonical cover directly, we cannot get the above decomposition. However, we can infer the original dependencies from the canonical cover, and if we use those for BCNF decomposition, we would be able to get the same decomposition.

8.30 List the three design goals for relational databases, and explain why each is desirable.

Answer: The three design goals are lossless-join decompositions, dependency preserving decompositions, and minimization of repetition of

information. They are desirable so we can maintain an accurate database, check correctness of updates quickly, and use the smallest amount of space possible.

8.31 In designing a relational database, why might we choose a non-BCNF design?

Answer: BCNF is not always dependency preserving. Therefore, we may want to choose another normal form (specifically, 3NF) in order to make checking dependencies easier during updates. This would avoid joins to check dependencies and increase system performance.

8.32 Given the three goals of relational-database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Practice Exercise 8.17 for the definition of 2NF.)

Answer: The three design goals of relational databases are to avoid

- Repetition of information
- Inability to represent information
- Loss of information.

2NF does not prohibit as much repetition of information since the schema (A, B, C) with dependencies $A \rightarrow B$ and $B \rightarrow C$ is allowed under 2NF, although the same (B, C) pair could be associated with many A values, needlessly duplicating C values. To avoid this we must go to 3NF. Repetition of information is allowed in 3NF in some but not all of the cases where it is allowed in 2NF. Thus, in general, 3NF reduces repetition of information. Since we can always achieve a lossless join 3NF decomposition, there is no loss of information needed in going from 2NF to 3NF.

Note that the decomposition $\{(A, B), (B, C)\}$ is a dependency-preserving and lossless-loin 3NF decomposition of the schema (A, B, C). However, in case we choose this decomposition, retrieving information about the relationship between A, B and C requires a join of two relations, which is avoided in the corresponding 2NF decomposition.

Thus, the decision of which normal form to choose depends upon how the cost of dependency checking compares with the cost of the joins. Usually, the 3NF would be preferred. Dependency checks need to be made with *every* insert or update to the instances of a 2NF schema, whereas, only some queries will require the join of instances of a 3NF schema.

8.33 Given a relational schema $r(A, B, C, D)$, does $A \twoheadrightarrow BC$ logically imply $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$? If yes prove it, else give a counter example.

Answer: $A \twoheadrightarrow BC$ holds on the following table:

r :

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_1	b_1	c_1	d_2
a_1	b_2	c_2	d_1

If $A \twoheadrightarrow B$, then we know that there exists t_1 and t_3 such that $t_1[B] = t_3[B]$. Thus, we must choose one of the following for t_1 and t_3:

- $t_1 = r_1$ and $t_3 = r_3$, or $t_1 = r_3$ and $t_3 = r_1$:
  Choosing either $t_2 = r_2$ or $t_2 = r_4$, $t_3[C] \neq t_2[C]$.
- $t_1 = r_2$ and $t_3 = r_4$, or $t_1 = r_4$ and $t_3 = r_2$:
  Choosing either $t_2 = r_1$ or $t_2 = r_3$, $t_3[C] \neq t_2[C]$.

Therefore, the condition $t_3[C] = t_2[C]$ can not be satisfied, so the conjecture is false.

8.34 Explain why 4NF is a normal form more desirable than BCNF.

Answer: 4NF is more desirable than BCNF because it reduces the repetition of information. If we consider a BCNF schema not in 4NF (see Practice Exercise 7.16), we observe that decomposition into 4NF does not lose information provided that a lossless join decomposition is used, yet redundancy is reduced.

CHAPTER 9

Application Design and Development

Practice Exercises

9.1 What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs?

Answer: The CGI interface starts a new process to service each request, which has a significant operating system overhead. On the other hand, servelets are run as threads of an existing process, avoiding this overhead. Further, the process running threads could be the Web server process itself, avoiding interprocess communication which can be expensive. Thus, for small to moderate sized tasks, the overhead of Java is less than the overheads saved by avoiding process creating and communication.
For tasks involving a lot of CPU activity, this may not be the case, and using CGI with a C or C++ program may give better performance.

9.2 List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.

Answer: Most computers have limits on the number of simultaneous connections they can accept. With connectionless protocols, connections are broken as soon as the request is satisfied, and therefore other clients can open connections. Thus more clients can be served at the same time. A request can be routed to any one of a number of different servers to balance load, and if a server crashes another can take over without the client noticing any problem.

The drawback of connectionless protocols is that a connection has to be reestablished every time a request is sent. Also, session information has to be sent each time in form of cookies or hidden fields. This makes them slower than the protocols which maintain connections in case state information is required.

9.3 Consider a carelessly written Web application for an online-shopping site, which stores the price of each item as a hidden form variable in the Web page sent to the customer; when the customer submits the form, the information from the hidden form variable is used to compute the bill for the customer. What is the loophole in this scheme? (There was a real instance where the loophole was exploited by some customers of an online-shopping site, before the problem was detected and fixed.)

Answer: A hacker can edit the HTML source code of the Web page, and replace the value of the hidden variable price with whatever value they want, and use the modified Web page to place an order. The Web application would then use the user-modified value as the price of the product.

9.4 Consider another carelessly written Web application, which uses a servlet that checks if there was an active session, but does not check if the user is authorized to access that page, instead depending on the fact that a link to the page is shown only to authorized users. What is the risk with this scheme? (There was a real instance where applicants to a college admissions site could, after logging into the Web site, exploit this loophole and view information they were not authorized to see; the unauthorized access was however detected, and those who accessed the information were punished by being denied admission.)

Answer: Although the link to the page is shown only to authorized users, an unauthorized user may somehow come to know of the existence of the link (for example, from an unauthorized user, or via Web proxy logs). The user may then login to the system, and access the unauthorized page by entering its URL in the browser. If the check for user authorization was inadvertantly left out from that page, the user will be able to see the result of the page.

The HTTP referer attribute can be used to block a naive attempt to exploit such loopholes, by ensuring the referer value is from a valid page of the Web site. However, the referer attribute is set by the browser, and can be spoofed, so a malicious user can easily work around the referer check.

9.5 List three ways in which caching can be used to speed up Web server performance.

Answer: Caching can be used to improve performance by exploiting the commonalities between transactions.

a. If the application code for servicing each request needs to open a connection to the database, which is time consuming, then a pool of open connections may be created before hand, and each request uses one from those.

b. The results of a query generated by a request can be cached. If same request comes agian, or generates the same query, then the cached result can be used instead of connecting to database again.

c. The final webpage generated in response to a request can be cached. If the same request comes again, then the cached page can be outputed.

9.6 The netstat command (available on Linux and on Windows) shows the active network connections on a computer. Explain how this command can be used to find out if a particular Web page is not closing connections that it opened, or if connection pooling is used, not returning connections to the connection pool. You should account for the fact that with connection pooling, the connection may not get closed immediately.

Answer: The tester should run netstat to find all connections open to the machine/socket used by the database. (If the application server is separate from the database server, the command may be executed at either of the machines). Then, the Web page being tested should be accessed repeatedly (this can be automated by using tools such as JMeter to generate page accesses). The number of connections to the database would go from 0 to some value (depending on the number of connections retained in the pool), but after some time the number of connections should stop increasing. If the number keeps increasing, the code underlying the Web page is clearly not closing connections or returning the connection to the pool.

9.7 Testing for SQL-injection vulnerability:

a. Suggest an approach for testing an application to find if it is vulnerable to SQL injection attacks on text input.

b. Can SQL injection occur with other forms of input? If so, how would you test for vulnerability?

Answer:

a. One approach is to enter a string containing a single quote in each of the input text boxes of each of the forms provided by the application, to see if the application correctly saves the value. If it does not save the value correctly, and/or gives an error message, it is vulnerable to SQL injection.

b. Yes, SQL injection can even occur with selection inputs such as drop-down menus, by modifying the value sent back to the server when the input value is chosen, for example by editing the page directly, or in the browser's DOM tree. A test tool should be able to modify the values sent to the application, inserting a single quote into the value; the test tool should also spoof (modify) the HTTP refer attribute to be a valid value, to bypass any attempt by the application to check the refer attribute.

9.8 A database relation may have the values of certain attributes encrypted for security. Why do database systems not support indexing on encrypted attributes? Using your answer to this question, explain why database systems do not allow encryption of primary-key attributes.

Answer: It is not possible in general to index on an encrypted value, unless all occurrences of the value encrypt to the same value (and even in this case, only equality predicates would be supported). However, mapping all occurrences of a value to the same encrypted value is risky, since statistical analysis can be used to reveal common values, even without decryption; techniques based on adding random "salt" bits are used to prevent such analysis, but they make indexing impossible. One possible workaround is to store the index unencrypted, but then the index can be used to leak values. Another option is to keep the index encrypted, but then the database system should know the decryption key, to decrypt required parts of the index on the fly. Since this required modifying large parts of the database system code, databases typically do not support this option.
The primary-key constraint has to be checked by the database when tuples are inserted, and if the values are encrypted as above, the database system will not be able to detect primary key violations. Therefore database systems that support encryption of specified attributes do not allow primary-key attributes, or for that matter foreign-key attributes, to be encrypted.

9.9 Exercise 9.8 addresses the problem of encryption of certain attributes. However, some database systems support encryption of entire databases. Explain how the problems raised in Exercise 9.8 are avoided when the entire database is encrypted.
Answer: When the entire database is encrypted, it is easy for the database to perform decryption as data is fetched from disk into memory, so in-memory storage is unencrypted. With this option, everything in the database, including indices, is encrypted when on disk, but un-encrypted in memory. As a result, only the data access layer of the database system code needs to be modified to perform encryption, leaving other layers untouched. Thus, indices can be used unchanged, and primary-key and foreign-key constraints enforced without any change to the corresponding layers of the database system code.

9.10 Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?
Answer: The key problem with digital certificates (when used offline, without contacting the certificate issuer) is that there is no way to withdraw them.

For instance (this actually happened, but names of the parties have been changed) person C claims to be an employee of company X and get a new public key certified by the certifying authority A. Suppose the authority A incorrectly believed that C was acting on behalf of company X, it gives C a certificate $cert$. Now, C can communicate with person Y, who checks the certificate $cert$ presenetd by C, and believes the public key contained in $cert$ really belongs to X. Now C would communicate with Y using the public key, and Y trusts the communication is from company X.

Person Y may now reveal confidential information to C, or accept purchase order from C, or execute programs certified by C, based on the public key, thinking he is actually communicating with company X. In each case there is potential for harm to Y.

Even if A detects the impersonation, as long as Y does not check with A (the protocol does not require this check), there is no way for Y to find out that the certificate is forged.

If X was a certification authority itself, further levels of fake certificates can be created. But certificates that are not part of this chain would not be affected.

9.11 Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Answer: A scheme for storing passwords would be to encrypt each password (after adding randomly generated "salt" bits to prevent dictionary attacks), and then use a hash index on the user-id to store/access the encrypted password. The password being used in a login attempt is then encrypted (if randomly generated "salt" bits were used initially these bits should be stored with the user-id, and used when encrypting the user-supplied password). The encrypted value is then compared with the stored encrypted value of the correct password. An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist. Thus, "one-way" encryption functions, such as secure hashing functions, which do not support decryption can be used for this task. The secure hashing algorithm SHA-1 is widely used for such one-way encryption.

CHAPTER 9

Application Design and 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

```
<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>
```

Servlet Code

```
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();
    }
}
```

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.

Answer: **HTML form**

```
<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>
```

Schema

```
CREATE TABLE SUBMISSION_COUNT (
    value integer unique,
    scount integer not null);
```

Servlet Code

```
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();
      }
}
```

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**

```
<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>
```

Schema

```
CREATE TABLE USERAUTH(
    userid integer primary key,
    username varchar(100) unique,
    password varchar(20)
);
```

Servlet Code

```
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();
    }
}
```

9.15 What is an SQL injection attack? Explain how it works, and what precautions must be taken to prevent SQL injection attacks.

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.

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 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:

```
// 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();
    }
}
```

9.17 Explain the terms CRUD and REST.

Answer:

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 use Ajax to communicate with a server after you type in each character (sometimes starting after the 3rd or 4th character), and respond 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 organizations 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 would execute the script, which can carry out actions such as sending private cookie information back to the malicious user, or execute 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 multi-factor authentication? How does it help safeguard against stolen passwords?

Answer: In multi-factor authentication (with two-factor authentication as a special case), where 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 9.7.5, 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.section_id= takes.section_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 it reaches 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 above 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 certificate) to encrypt 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) is 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

Storage and File Structure

Practice Exercises

10.1 **Answer:** This arrangement has the problem that P_i and B_{4i-3} are on the same disk. So if that disk fails, reconstruction of B_{4i-3} is not possible, since data and parity are both lost.

10.2 **Answer:**

a. It is stored as an array containing physical page numbers, indexed by logical page numbers. This representation gives an overhead equal to the size of the page address for each page.

b. It takes 32 bits for every page or every 4096 bytes of storage. Hence, it takes 64 megabytes for the 64 gigabyte of flash storage.

c. If the mapping is such that, every p consecutive logical page numbers are mapped to p consecutive physical pages, we can store the mapping of the first page for every p pages. This reduces the in memory structure by a factor of p. Further, if p is an exponent of 2, we can avoid some of the least significant digits of the addresses stored.

10.3 **Answer:**

a. To ensure atomicity, a block write operation is carried out as follows:

   i. Write the information onto the first physical block.

   ii. When the first write completes successfully, write the same information onto the second physical block.

   iii. The output is declared completed only after the second write completes successfully.

   During recovery, each pair of physical blocks is examined. If both are identical and there is no detectable partial-write, then no further actions are necessary. If one block has been partially rewritten, then we replace its contents with the contents of the other block. If there

has been no partial-write, but they differ in content, then we replace the contents of the first block with the contents of the second, or vice versa. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates both copies) or results in no change.
The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

b. The idea is similar here. For any block write, the information block is written first followed by the corresponding parity block. At the time of recovery, each set consisting of the n^{th} block of each of the disks is considered. If none of the blocks in the set have been partially-written, and the parity block contents are consistent with the contents of the information blocks, then no further action need be taken. If any block has been partially-written, it's contents are reconstructed using the other blocks. If no block has been partially-written, but the parity block contents do not agree with the information block contents, the parity block's contents are reconstructed.

10.4 Answer:

a. Although moving record 6 to the space for 5, and moving record 7 to the space for 6, is the most straightforward approach, it requires moving the most records, and involves the most accesses.

b. Moving record 7 to the space for 5 moves fewer records, but destroys any ordering in the file.

c. Marking the space for 5 as deleted preserves ordering and moves no records, but requires additional overhead to keep track of all of the free space in the file. This method may lead to too many "holes" in the file, which if not compacted from time to time, will affect performance because of reduced availability of contiguous free records.

10.5 Answer: (We use "↑ i" to denote a pointer to record "i".)
The original file of Figure 10.7.

header				↑ 1
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				↑ 4
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

a. The file after **insert** (24556, Turnamian, Finance, 98000).

header				↑ 4
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

b. The file after **delete** record 2.

header				↑ 2
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2				↑ 4
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

The free record chain could have alternatively been from the header to 4, from 4 to 2, and finally from 2 to 6.

c. The file after **insert** (34556, Thompson, Music, 67000).

header				↑ 4
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	24556	Turnamian	Finance	98000
record 2	34556	Thompson	Music	67000
record 3	22222	Einstein	Physics	95000
record 4				↑ 6
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

10.6 Answer:

The relation *section* with three tuples is as follows.

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-347	1	Fall	2009	Taylor	3128	C

The relation *takes* with five students for each section is as follows.

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2009	A
00128	CS-347	1	Fall	2009	A-
12345	CS-347	1	Fall	2009	A
12345	CS-101	1	Fall	2009	C
17968	BIO-301	1	Summer	2010	null
23856	CS-347	1	Fall	2009	A
45678	CS-101	1	Fall	2009	F
54321	CS-101	1	Fall	2009	A-
54321	CS-347	1	Fall	2009	A
59762	BIO-301	1	Summer	2010	null
76543	CS-101	1	Fall	2009	A
76543	CS-347	1	Fall	2009	A
78546	BIO-301	1	Summer	2010	null
89729	BIO-301	1	Summer	2010	null
98988	BIO-301	1	Summer	2010	null

The multitable clustering for the above two instances can be taken as:

BIO-301	1	Summer	2010	Painter	514	A
17968	BIO-301	1	Summer	2010	null	
59762	BIO-301	1	Summer	2010	null	
78546	BIO-301	1	Summer	2010	null	
89729	BIO-301	1	Summer	2010	null	
98988	BIO-301	1	Summer	2010	null	
CS-101	1	Fall	2009	Packard	101	H
00128	CS-101	1	Fall	2009	A	
12345	CS-101	1	Fall	2009	C	
45678	CS-101	1	Fall	2009	F	
54321	CS-101	1	Fall	2009	A-	
76543	CS-101	1	Fall	2009	A	
CS-347	1	Fall	2009	Taylor	3128	C
00128	CS-347	1	Fall	2009	A-	
12345	CS-347	1	Fall	2009	A	
23856	CS-347	1	Fall	2009	A	
54321	CS-347	1	Fall	2009	A	
76543	CS-347	1	Fall	2009	A	

10.7 **Answer:**

a. Everytime a record is inserted/deleted, check if the usage of the block has changed levels. In that case, update the corresponding

bits. Note that we don't need to access the bitmaps at all unless the usage crosses a boundary, so in most of the cases there is no overhead.

b. When free space for a large record or a set of records is sought, then multiple free list entries may have to be scanned before finding a proper sized one, so overheads are much higher. With bitmaps, one page of bitmap can store free info for many pages, so I/O spent for finding free space is minimal. Similarly, when a whole block or a large part of it is deleted, bitmap technique is more convenient for updating free space information.

10.8 **Answer:** Hash table is the common option for large database buffers. The hash function helps in locating the appropriate bucket, on which linear search is performed.

10.9 **Answer:**

a. MRU is preferable to LRU where $R_1 \bowtie R_2$ is computed by using a nested-loop processing strategy where each tuple in R_2 must be compared to each block in R_1. After the first tuple of R_2 is processed, the next needed block is the first one in R_1. However, since it is the least recently used, the LRU buffer management strategy would replace that block if a new block was needed by the system.

b. LRU is preferable to MRU where $R_1 \bowtie R_2$ is computed by sorting the relations by join values and then comparing the values by proceeding through the relations. Due to duplicate join values, it may be necessary to "back-up" in one of the relations. This "backing-up" could cross a block boundary into the most recently used block, which would have been replaced by a system using MRU buffer management, if a new block was needed.

Under MRU, some unused blocks may remain in memory forever. In practice, MRU can be used only in special situations like that of the nested-loop strategy discussed in Exercise 10.9a.

CHAPTER 10

Storage and File Structure

This chapter presents basic file structure concepts. The chapter really consists of two parts—the first dealing with relational databases, and the second with object-oriented databases. The second part can be omitted without loss of continuity for later chapters.

Many computer science undergraduates have covered some of the material in this chapter in a prior course on data structures or on file structures. Even if students' backgrounds are primarily in data structures, this chapter is still important since it addresses data structure issues as they pertain to disk storage. Buffer management issues, covered in Section 10.8.1 should be familiar to students who have taken an operating systems course. However, there are database-specific aspects of buffer management that make this section worthwhile even for students with an operating system background.

Exercises

10.10 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.

10.11 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 amongst the sectors. But that is better than the loss of data in case of no remapping!

10.12 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 on-going 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.

10.13 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 has 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.

10.14 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.

10.15 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.

10.16 If possible, determine the buffer-management strategy used by the operating system running on your local computer system and what mechanisms it provides to control replacement of pages. Discuss how the control on replacement that it provides would be useful for the implementation of database systems.

Answer: The typical OS uses variants of LRU, which are cheaper to implement than LRU, for buffer replacement. LRU and its variants are often a bad strategy for databases. As explained in Section 10.8.2 of the text, MRU is the best strategy for nested loop join. In general no single strategy handles all scenarios well, and the database system should be able to manage its own buffer cache for which the replacement policy takes into account all the performance related issues.

Many operating systems provide mechanisms to lock pages in memory, which can be used to ensure buffer pages stay in memory. However, operating systems today generally do not allow any other control on page replacement.

10.17 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.

10.18 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 smallest 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.

10.19 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.

10.20 If you have data that should not be lost on disk failure, and the data are 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.

10.21 In earlier generation disks the number of sectors per track was the same across all tracks. Current generation disks have more sectors per track on outer tracks, and fewer sectors per track on inner tracks (since they are shorter in length). What is the effect of such a change on each of the three main indicators of disk speed?

Answer: The main performance effect of storing more sectors on the outer tracks and fewer sectors on the inner tracks is that the disk's **data-transfer** rate will be greater on the outer tracks than the inner tracks. This is because the disk spins at a constant rate, so more sectors pass underneath the drive head in a given amount of time when the arm is positioned on an outer track than when on an inner track.

In fact, some high-performance systems are optimized by storing data only in outer tracks, so that disk arm movement is minimized while maximizing data-transfer rates.

10.22 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 11

Indexing and Hashing

Practice Exercises

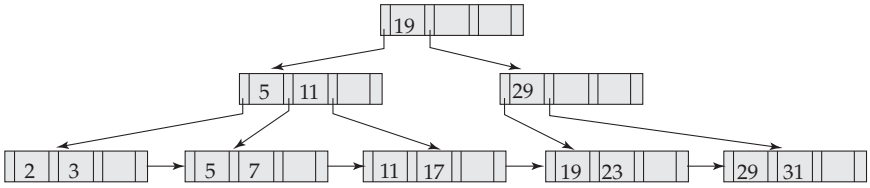
11.1 **Answer:** Reasons for not keeping indices on every attribute include:

- Every index requires additional CPU time and disk I/O overhead during inserts and deletions.
- Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes).
- Each extra index requires additional storage space.
- For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist.

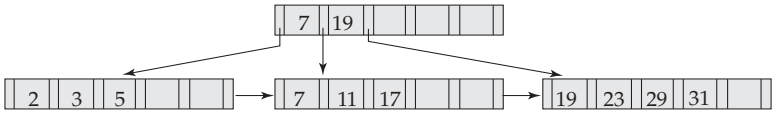
11.2 **Answer:** In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient.

11.3 **Answer:** The following were generated by inserting values into the B^+-tree in ascending order. A node (other than the root) was never allowed to have fewer than $\lceil n/2 \rceil$ values/pointers.

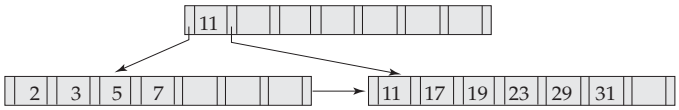
a.



b.



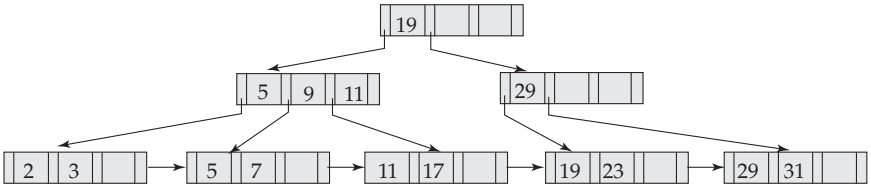
c.



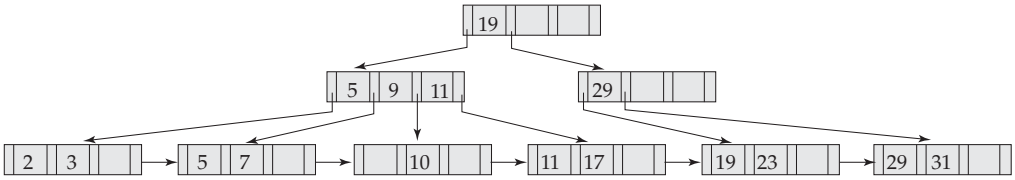
11.4 Answer:

- With structure 11.3.a:

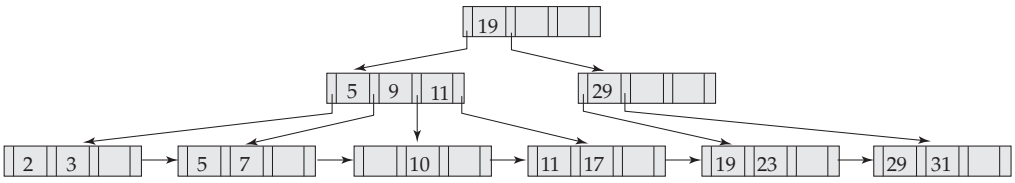
  Insert 9:



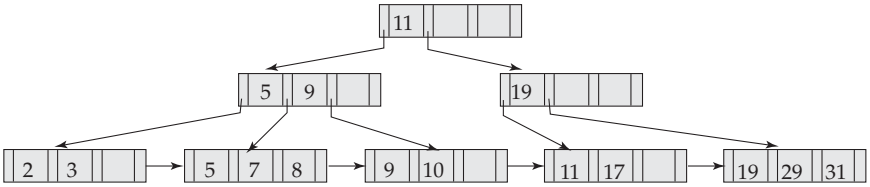
  Insert 10:



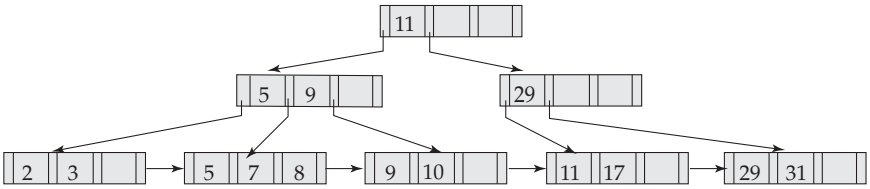
  Insert 8:



Delete 23:

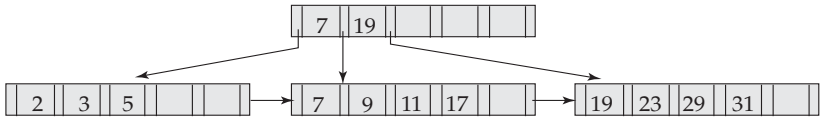


Delete 19:

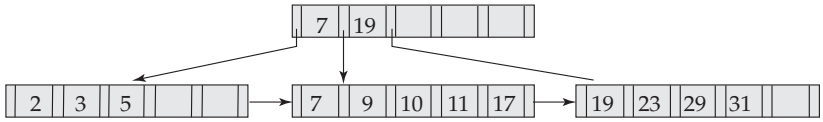


- With structure 11.3.b:

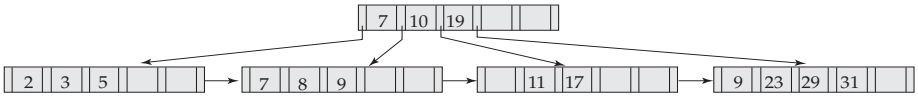
  Insert 9:



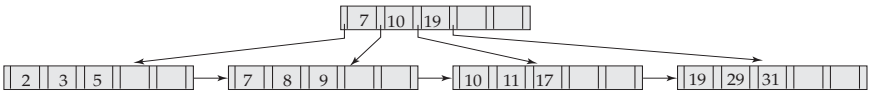
  Insert 10:



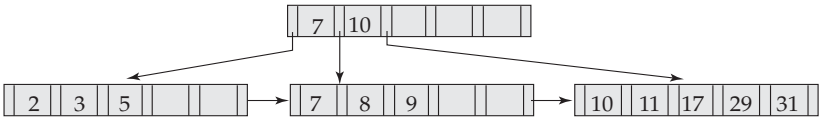
  Insert 8:



Delete 23:

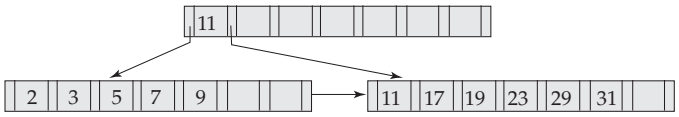


Delete 19:

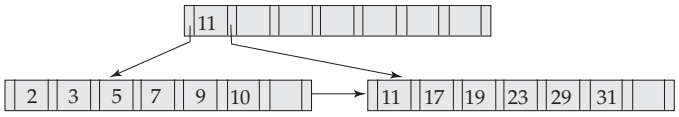


- With structure 11.3.c:

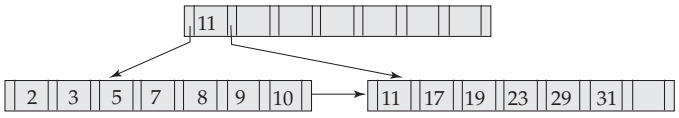
  Insert 9:



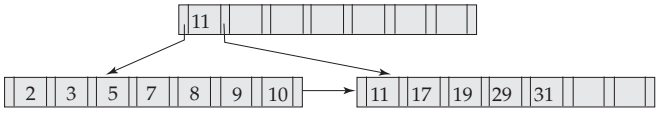
  Insert 10:



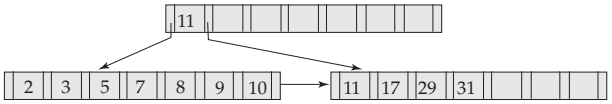
  Insert 8:



  Delete 23:

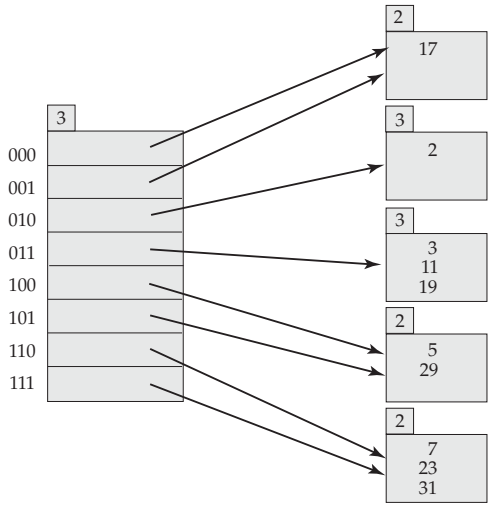


  Delete 19:



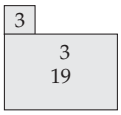
11.5 **Answer:** If there are K search-key values and $m - 1$ siblings are involved in the redistribution, the expected height of the tree is: $log_{\lfloor (m-1)n/m \rfloor}(K)$

11.6 **Answer:** Extendable hash structure



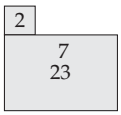
11.7 **Answer:**

a. Delete 11: From the answer to Exercise 11.6, change the third bucket to:

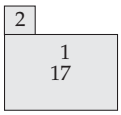


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

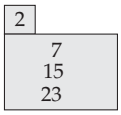
b. Delete 31: From the answer to 11.6, change the last bucket to:



c. Insert 1: From the answer to 11.6, change the first bucket to:



d. Insert 15: From the answer to 11.6, change the last bucket to:



11.8 **Answer:** The pseudocode is shown in Figure 11.1.

11.9 **Answer:** Let i denote the number of bits of the hash value used in the hash table. Let **bsize** denote the maximum capacity of each bucket. The pseudocode is shown in Figure 11.2.
Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket j differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

When merge is successful, further merging may be possible, which is handled by a recursive call to *coalesce* at the end of the function.

11.10 **Answer:** If the hash table is currently using i bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly i.
Consider a bucket j with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to i, then reset the count to 1. If the bucket is being split and i_j is one less that i, then increase the count by 1. It the bucket if being coalesced, and i_j is equal to i then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.
However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e. a count for each value of

```
function findIterator(value V) {
/* Returns an iterator for the search on the value V */
    Iterator iter();
    Set iter.value = V;
    Set C = root node
    while (C is not a leaf node) begin
        Let i = samllest number such that V <= C.K_i
        if there is no such number i then begin
            Let P_m = last non-null pointer in the node
            Set C = C.P_m;
        end
        else Set C = C.P_i;
    end
    /* C is a leaf node */
    Let i be the least value such that K_i = V
    if there is such a value i then begin
        Set iter.index = i;
        Set iter.page = C;
        Set iter.active = TRUE;
    end
    else if (V is the greater than the largest value in the leaf) then begin
        if (C.P_n.K_1 = V) then begin
            Set iter.page = C.P_n;
            Set iter.index = 1;
            Set iter.active = TRUE;
        end
        else Set iter.active = FALSE;
    end
    else Set iter.active = FALSE;
    return (iter)
}
Class Iterator {
    variables:
        value V /* The value on which the index is searched */
        boolean active /* Stores the current state of the iterator (TRUE or FALSE)*/
        int index /* Index of the next matching entry (if active is TRUE) */
        PageID page /* Page Number of the next matching entry (if active is TRUE)*/

    function next() {
        if (active) then begin
            Set retPage = page;
            Set retIndex = index;
            if (index + 1 = page.size) then begin
                page = page.P_n
                index = 0
            end
            else index = index + 1;
            if (page.K_index ≠ V)
                then active=FALSE;
            return(retPage, retIndex)
        end
        else return null;
    }
}
```

Figure 11.1 Pseudocode for findIterator and the Iterator class

```
delete(value K_l)
begin
    j = first i high-order bits of h(K_l);
    delete value K_l from bucket j;
    coalesce(bucket j);
end

coalesce(bucket j)
begin
    i_j = bits used in bucket j;
    k = any bucket with first (i_j − 1) bits same as that
        of bucket j while the bit i_j is reversed;
    i_k = bits used in bucket k;
    if(i_j ≠ i_k)
        return; /* buckets cannot be merged */
    if(entries in j + entries in k > bsize)
        return; /* buckets cannot be merged */
    move entries of bucket k into bucket j;

    decrease the value of i_j by 1;
    make all the bucket-address-table entries,
    which pointed to bucket k, point to j;

    coalesce(bucket j);
end
```

Figure 11.2 Pseudocode for deletion

common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when sufficient number of entries at the end of count array become 0.

11.11 **Answer:** We reproduce the instructor relation below.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

a. Bitmap for *salary*, with S_1, S_2, S_3 and S_4 representing the given intervals in the same order

S_1	0	0	1	0	0	0	0	0	0	0	0	0
S_2	0	0	0	0	0	0	0	0	0	0	0	0
S_3	1	0	0	0	1	0	0	1	0	0	0	0
S_4	0	1	0	1	0	1	1	0	1	1	1	1

b. The question is a bit trivial if there is no bitmap on the *dept_name* attribute. The bitmap for the *dept_name* attribute is:

Comp. Sci	1	0	0	0	0	0	1	0	0	0	1	0
Finance	0	1	0	0	0	0	0	0	1	0	0	0
Music	0	0	1	0	0	0	0	0	0	0	0	0
Physics	0	0	0	1	0	1	0	0	0	0	0	0
History	0	0	0	0	1	0	0	1	0	0	0	0
Biology	0	0	0	0	0	0	0	0	0	1	0	0
Elec. Eng.	0	0	0	0	0	0	0	0	0	0	0	1

To find all instructors in the Finance department with salary of 80000 or more, we first find the intersection of the Finance department bitmap and S_4 bitmap of *salary* and then scan on these records for salary of 80000 or more.
Intersection of Finance department bitmap and S_4 bitmap of *salary*.

S_4	0	1	0	1	0	1	1	0	1	1	1	1
Finance	0	1	0	0	0	0	0	0	1	0	0	0
$S_4 \cap$ Finance	0	1	0	0	0	0	0	0	1	0	0	0

Scan on these records with salary 80000 or more gives Wu and Singh as the instructors who satisfy the given query.

11.12 **Answer:** If the index entries are inserted in ascending order, the new entries get directed to the last leaf node. When this leaf node gets filled, it is split into two. Of the two nodes generated by the split, the left node is left untouched and the insertions takes place on the right node. This makes the occupancy of the leaf nodes to about 50 percent, except the last leaf.

If keys that are inserted are sorted in descending order, the above situation would still occur, but symmetrically, with the right node of a split never getting touched again, and occupancy would again be 50 percent for all nodes other than the first leaf.

11.13 **Answer:**

a. The cost to locate the page number of the required leaf page for an insertion is negligible since the non-leaf nodes are in memory. On the leaf level it takes one random disk access to read and one random disk access to update it along with the cost to write one page. Insertions which lead to splitting of leaf nodes require an additional page write. Hence to build a B^+-tree with n_r entries it takes a maximum of $2 * n_r$ random disk accesses and $n_r + 2 * (n_r/f)$ page writes. The second part of the cost comes from the fact that in the worst case each leaf is half filled, so the number of splits that occur is twice n_r/f.

   The above formula ignores the cost of writing non-leaf nodes, since we assume they are in memory, but in reality they would also be written eventually. This cost is closely approximated by $2*(n_r/f)/f$, which is the number of internal nodes just above the leaf; we can add further terms to account for higher levels of nodes, but these are much smaller than the number of leaves and can be ignored.

b. Substituting the values in the above formula and neglecting the cost for page writes, it takes about $10,000,000 * 20$ milliseconds, or 56 hours, since each insertion costs 20 milliseconds.

c.

```
function insert_in_leaf(value K, pointer P)
    if(tree is empty) create an empty leaf node L, which is also the root
    else Find the last leaf node in the leaf nodes chain L
    if (L has less than n − 1 key values)
        then insert (K,P) at the first available location in L
    else begin
        Create leaf node L1
        Set L.P_n = L1;
        Set K1 = last value from page L
        insert_in_parent(1, L, K1, L1)
        insert (K,P) at the first location in L1
    end


function insert_in_parent(level l, pointer P, value K, pointer P1)
    if (level l is empty) then begin
        Create an empty non-leaf node N, which is also the root
        insert(P, K, P1) at the starting of the node N
        return
    else begin
        Find the right most node N at level l
        if (N has less than n pointers)
            then insert(K, P1) at the first available location in N
        else begin
            Create a new non-leaf page N1
            insert (P1) at the starting of the node N
            insert_in_parent(l + 1, pointer N, value K, pointer N1)
        end
    end
```

The insert_in_leaf function is called for each of the value, pointer pairs in ascending order. Similar function can also be build for descending order. The search for the last leaf or non-leaf node at any level can be avoided by storing the current last page details in an array.

The last node in each level might be less than half filled. To make this index structure meet the requirements of a B^+-tree, we can redistribute the keys of the last two pages at each level. Since the last but one node is always full, redistribution makes sure that both of then are at least half filled.

11.14 **Answer:** In a B^+-tree index or file organization, leaf nodes that are adjacent to each other in the tree may be located at different places on disk. When a file organization is newly created on a set of records, it is possible to allocate blocks that are mostly contiguous on disk to leafs nodes that are contiguous in the tree. As insertions and deletions occur

on the tree, sequentiality is increasingly lost, and sequential access has to wait for disk seeks increasingly often.

a. One way to solve this problem is to rebuild the index to restore sequentiality.

b. i. In the worst case each n-block unit and each node of the B^+-tree is half filled. This gives the worst case occupancy as 25 percent.

ii. No. While splitting the n-block unit the first $n/2$ leaf pages are placed in one n-block unit, and the remaining in the second n-block unit. That is, every n-block split maintains the order. Hence, the nodes in the n-block units are consecutive.

iii. In the regular B^+-tree construction, the leaf pages might not be sequential and hence in the worst case, it takes one seek per leaf page. Using the block at a time method, for each n-node block, we will have at least $n/2$ leaf nodes in it. Each n-node block can be read using one seek. Hence the worst case seeks comes down by a factor of $n/2$.

iv. Allowing redistribution among the nodes of the same block, does not require additional seeks, where as, in regular B^+-tree we require as many seeks as the number of leaf pages involved in the redistribution. This makes redistribution for leaf blocks efficient with this scheme. Also the worst case occupancy comes back to nearly 50 percent. (Splitting of leaf nodes is preferred when the participating leaf nodes are nearly full. Hence nearly 50 percent instead of exact 50 percent)

CHAPTER 11

Indexing and Hashing

This chapter covers indexing techniques ranging from the most basic one to highly specialized ones. Due to the extensive use of indices in database systems, this chapter constitutes an important part of a database course.

A class that has already had a course on data-structures would likely be familiar with hashing and perhaps even B^+-trees. However, this chapter is necessary reading even for those students since data structures courses typically cover indexing in main memory. Although the concepts carry over to database access methods, the details (e.g., block-sized nodes), will be new to such students.

The sections on B-trees (Sections 11.4.5) and bitmap indexing (Section 11.9) may be omitted if desired.

Exercises

11.15 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.

11.16 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.

11.17 For each B^+-tree of Practice Exercise 11.3, 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 11.3a:**

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 12.3b:

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 12.3c:

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.

11.18 The solution presented in Section 11.3.4 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 more 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.

11.19 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.

11.20 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.

11.21 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.

11.22 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. 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?
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 operation. 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. 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.
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, 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, 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. Under what conditions on n_1 and n_2 would the index be an efficient way of finding records satisfying $10 < A < 50 \wedge 5 < B < 10$.
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

dont satisfy the second condition are also output with an additional cost of h each (worst case).

11.23 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 arises 2 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 bu 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 which is sufficeint to distinguish between the key values in the subtrees that it seperates. The full name can be stored in the leaf nodes, this way we dont lose any information and also maximize the average fanout of internal nodes.

11.24 Suppose a relation is stored in a B^+-tree file organization. Suppose secondary indices stored record identifiers that are pointers to records on disk.

a. What would be the effect on the secondary indices if a node split happens 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 accesing data using a secondary index.

11.25 Show how to compute existence bitmaps 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*.

11.26 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.

11.27 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
$2C$
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 12

Query Processing

Practice Exercises

12.1 Assume (for simplicity in this exercise) that only one tuple fits in a block and memory holds at most 3 blocks. Show the runs created on each pass of the sort-merge algorithm, when applied to sort the following tuples on the first attribute: (kangaroo, 17), (wallaby, 21), (emu, 1), (wombat, 13), (platypus, 3), (lion, 8), (warthog, 4), (zebra, 11), (meerkat, 6), (hyena, 9), (hornbill, 2), (baboon, 12).

Answer: We will refer to the tuples (kangaroo, 17) through (baboon, 12) using tuple numbers t_1 through t_{12}. We refer to the j^{th} run used by the i^{th} pass, as r_{ij}. The initial sorted runs have three blocks each. They are:

$$
\begin{aligned}
r_{11} &= \{t_3, t_1, t_2\} \\
r_{12} &= \{t_6, t_5, t_4\} \\
r_{13} &= \{t_9, t_7, t_8\} \\
r_{14} &= \{t_{12}, t_{11}, t_{10}\}
\end{aligned}
$$

Each pass merges three runs. Therefore the runs after the end of the first pass are:

$$
\begin{aligned}
r_{21} &= \{t_3, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\} \\
r_{22} &= \{t_{12}, t_{11}, t_{10}\}
\end{aligned}
$$

At the end of the second pass, the tuples are completely sorted into one run:

$$
r_{31} = \{t_{12}, t_3, t_{11}, t_{10}, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}
$$

12.2 Consider the bank database of Figure 12.13, where the primary keys are underlined, and the following SQL query:

select *T.branch_name*
from *branch T*, *branch S*
where *T.assets* > *S.assets* **and** *S.branch_city* = "Brooklyn"

Write an efficient relational-algebra expression that is equivalent to this query. Justify your choice.

Answer:

Query:

$$\Pi_{T.branch\_name}((\Pi_{branch\_name,\, assets}(\rho_T(branch))) \bowtie_{T.assets\, >\, S.assets} (\Pi_{assets}\ (\sigma_{(branch\_city\, =\, \text{'Brooklyn'})}(\rho_S(branch)))))$$

This expression performs the theta join on the smallest amount of data possible. It does this by restricting the right hand side operand of the join to only those branches in Brooklyn, and also eliminating the unneeded attributes from both the operands.

12.3 Let relations $r_1(A, B, C)$ and $r_2(C, D, E)$ have the following properties: r_1 has 20,000 tuples, r_2 has 45,000 tuples, 25 tuples of r_1 fit on one block, and 30 tuples of r_2 fit on one block. Estimate the number of block transfers and seeks required, using each of the following join strategies for $r_1 \bowtie r_2$:

a. Nested-loop join.

b. Block nested-loop join.

c. Merge join.

d. Hash join.

Answer:

r_1 needs 800 blocks, and r_2 needs 1500 blocks. Let us assume M pages of memory. If $M > 800$, the join can easily be done in 1500 + 800 disk accesses, using even plain nested-loop join. So we consider only the case where $M \leq 800$ pages.

a. Nested-loop join:
Using r_1 as the outer relation we need $20000 * 1500 + 800 = 30,000,800$ disk accesses, if r_2 is the outer relation we need $45000 * 800 + 1500 = 36,001,500$ disk accesses.

b. Block nested-loop join:
If r_1 is the outer relation, we need $\lceil \frac{800}{M-1} \rceil * 1500 + 800$ disk accesses, if r_2 is the outer relation we need $\lceil \frac{1500}{M-1} \rceil * 800 + 1500$ disk accesses.

c. Merge-join:
Assuming that r_1 and r_2 are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil log_{M-1}(1500/M) \rceil +$

2) + 800(2$\lceil log_{M-1}(800/M)\rceil$ + 2) disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

d. Hash-join:
We assume no overflow occurs. Since r_1 is smaller, we use it as the build relation and r_2 as the probe relation. If $M > 800/M$, i.e. no need for recursive partitioning, then the cost is $3(1500+800) = 6900$ disk accesses, else the cost is $2(1500+800)\lceil log_{M-1}(800) - 1\rceil + 1500 + 800$ disk accesses.

12.4 The indexed nested-loop join algorithm described in Section 12.5.3 can be inefficient if the index is a secondary index, and there are multiple tuples with the same value for the join attributes. Why is it inefficient? Describe a way, using sorting, to reduce the cost of retrieving tuples of the inner relation. Under what conditions would this algorithm be more efficient than hybrid merge join?

Answer:
If there are multiple tuples in the inner relation with the same value for the join attributes, we may have to access that many blocks of the inner relation for each tuple of the outer relation. That is why it is inefficient. To reduce this cost we can perform a join of the outer relation tuples with just the secondary index leaf entries, postponing the inner relation tuple retrieval. The result file obtained is then sorted on the inner relation addresses, allowing an efficient physical order scan to complete the join.
Hybrid merge–join requires the outer relation to be sorted. The above algorithm does not have this requirement, but for each tuple in the outer relation it needs to perform an index lookup on the inner relation. If the outer relation is much larger than the inner relation, this index lookup cost will be less than the sorting cost, thus this algorithm will be more efficient.

12.5 Let r and s be relations with no indices, and assume that the relations are not sorted. Assuming infinite memory, what is the lowest-cost way (in terms of I/O operations) to compute $r \bowtie s$? What is the amount of memory required for this algorithm?

Answer:
We can store the entire smaller relation in memory, read the larger relation block by block and perform nested loop join using the larger one as the outer relation. The number of I/O operations is equal to $b_r + b_s$, and memory requirement is $min(b_r, b_s) + 2$ pages.

12.6 Consider the bank database of Figure 12.13, where the primary keys are underlined. Suppose that a B^+-tree index on *branch_city* is available on relation *branch*, and that no other index is available. List different ways to handle the following selections that involve negation:

a. $\sigma_{\neg(branch\_city < \text{“Brooklyn”})}(branch)$

b. $\sigma_{\neg(branch\_city = \text{“Brooklyn”})}(branch)$

c. $\sigma_{\neg(branch\_city < \text{“Brooklyn”} \vee assets < 5000)}(branch)$

Answer:

a. Use the index to locate the first tuple whose *branch_city* field has value “Brooklyn”. From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch_city* field is anything other than “Brooklyn”.

c. This query is equivalent to the query

$$\sigma_{(branch\_city \geq \text{'Brooklyn'} \wedge assets < 5000)}(branch)$$

Using the *branch-city* index, we can retrieve all tuples with *branch-city* value greater than or equal to “Brooklyn” by following the pointer chains from the first “Brooklyn” tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

12.7 Write pseudocode for an iterator that implements indexed nested-loop join, where the outer relation is pipelined. Your pseudocode must define the standard iterator functions *open*(), *next*(), and *close*(). Show what state information the iterator must maintain between calls.

Answer: Let *outer* be the iterator which returns successive tuples from the pipelined outer relation. Let *inner* be the iterator which returns successive tuples of the inner relation having a given value at the join attributes. The *inner* iterator returns these tuples by performing an index lookup. The functions **IndexedNLJoin::open**, **IndexedNLJoin::close** and **IndexedNLJoin::next** to implement the indexed nested-loop join iterator are given below. The two iterators *outer* and *inner*, the value of the last read outer relation tuple t_r and a flag $done_r$ indicating whether the end of the outer relation scan has been reached are the state information which need to be remembered by **IndexedNLJoin** between calls.

```
IndexedNLJoin::open()
begin
    outer.open();
    inner.open();
    done_r := false;
    if(outer.next() ≠ false)
        move tuple from outer's output buffer to t_r;
    else
        done_r := true;
end
```

```
IndexedNLJoin::close()
begin
    outer.close();
    inner.close();
end
```

```
boolean IndexedNLJoin::next()
begin
    while(¬done_r)
    begin
        if(inner.next(t_r[JoinAttrs]) ≠ false)
        begin
            move tuple from inner's output buffer to t_s;
            compute t_r ⋈ t_s and place it in output buffer;
            return true;
        end
        else
            if(outer.next() ≠ false)
            begin
                move tuple from outer's output buffer to t_r;
                rewind inner to first tuple of s;
            end
            else
                done_r := true;
    end
    return false;
end
```

12.8 Design sort-based and hash-based algorithms for computing the relational division operation (see Practise Exercises of Chapter 6 for a definition of the division operation).

Answer: Suppose $r(T \cup S)$ and $s(S)$ be two relations and $r \div s$ has to be computed.

For sorting based algorithm, sort relation s on S. Sort relation r on (T, S). Now, start scanning r and look at the T attribute values of the first tuple. Scan r till tuples have same value of T. Also scan s simultaneously and check whether every tuple of s also occurs as the S attribute of r, in a fashion similar to merge join. If this is the case, output that value of T and proceed with the next value of T. Relation s may have to be scanned multiple times but r will only be scanned once. Total disk accesses, after

sorting both the relations, will be $|r| + N * |s|$, where N is the number of distinct values of T in r.
We assume that for any value of T, all tuples in r with that T value fit in memory, and consider the general case at the end. Partition the relation r on attributes in T such that each partition fits in memory (always possible because of our assumption). Consider partitions one at a time. Build a hash table on the tuples, at the same time collecting all distinct T values in a separate hash table. For each value of T, Now, for each value V_T of T, each value s of S, probe the hash table on (V_T, s). If any of the values is absent, discard the value V_T, else output the value V_T.
In the case that not all r tuples with one value for T fit in memory, partition r and s on the S attributes such that the condition is satisfied, run the algorithm on each corresponding pair of partitions r_i and s_i. Output the intersection of the T values generated in each partition.

12.9 What is the effect on the cost of merging runs if the number of buffer blocks per run is increased, while keeping overall memory available for buffering runs fixed?
Answer: Seek overhead is reduced, but the the number of runs that can be merged in a pass decreases potentially leading to more passes. A value of b_b that minimizes overall cost should be chosen.

CHAPTER 12

Query Processing

This chapter describes the process by which queries are executed efficiently by a database system. The chapter starts off with measures of cost, then proceeds to algorithms for evaluation of relational algebra operators and expressions. This chapter applies concepts from Chapter 2, 6, 10, and 11.

Query processing algorithms can be covered without tedious and distracting details of size estimation. Although size estimation is covered later, in Chapter 13, the presentation there has been simplified by omitting some details. Instructors can choose to cover query processing but omit query optimization, without loss of continuity with later chapters.

Exercises

12.10 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 seek time of 1 microsecond, and a transfer rate of 40 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* $\times$ *Disk seek cost*) + (*Number of block transfers* $\times$ *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 12.4, 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:

- **Case 1**: $b_b = 1$
  The number of disk seeks is: 5002×10^3. Therefore the cost of sorting the relation is: $(5002 \times 10^3) \times (1 \times 10^{-6}) + (25 \times 10^6) \times (10^{-4}) = 5.002 + 2500$ 2506 seconds.
- **Case 2**: $b_b = 100$
  The number of disk seeks is: 52×10^3. Therefore the cost of sorting the relation is: $(52 \times 10^3) \times (1 \times 10^{-6}) + (25 \times 10^6) \times (10^{-4}) = 0.052 + 2500$, which is approx $= 2500$ seconds.

12.11 Consider the following extended relational-algebra operators. Describe how to implement each operation using sorting, and using hashing.

a. **Semijoin** ($\ltimes_\theta$): $r \ltimes_\theta s$ is defined as $\Pi_R(r \bowtie_\theta s)$, where R is the set of attributes in the schema of r; that it selects those tuples r_i in r for which there is a tuple s_j in s such that r_i and s_j satisfy predicate θ.

b. **Anti-semijoin** ($\bar{\ltimes}_\theta$): $r \bar{\ltimes}_\theta s$ is defined as $r - \Pi_R(r \bowtie_\theta s)$; that it it selects those tuples r_i in r for which there is no tuple s_j in s such that r_i and s_j satisfy predicate θ.

Answer: As in the case of join algorithms, semijoin and anti-semijoin can be done efficiently if the join conditions are equijoin conditions. We describe below how to efficiently handle the case of equijoin conditions using sorting and hashing. With arbitrary join conditions, sorting and hashing cannot be used; (block) nested loops join needs to be used instead.

a. **Semijoin:**

- **Semijoin using Sorting:** Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge al-

gorithm and add tuples of r to the result whenever the join attributes of the current tuples of r and s match.

- **Semijoin using Hashing:** Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a value, add the current tuple of r to the result.
  Note that if r and s are large, they can be partitioned on the join attributes first, and the above procedure applied on each partition. If r is small but s is large, a hash index can be built on r, and probed using s; and if an s tuple matches an r tuple, the r tuple can be output and deleted from the hash index.

b. **Anti-semijoin:**

- **Anti-Semijoin using Sorting**: Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge algorithm and add tuples of r to the result if no tuple of s satisfies the join predicate for the corresponding tuple of r.
- **Anti-Semijoin using Hashing**: Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a null value, add the current tuple of r to the result.
  As for semijoin, partitioning can be used if r and s are large. An index on r can be used instead of an index on s, but then when an s tuple matches an r tuple, the r tuple is deleted from the index. After processing all s tuples, all remaining r tuples in the index are output as the result of the anti-semijoin operation.

12.12 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.

12.13 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.

12.14 Estimate the number of block transfers and seeks required by your solution to Exercise 12.13 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Practice Exercise 12.3.

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 by 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 $\lceil n_o/n_2 \rceil$.

12.15 The hash-join algorithm as described in Section 12.5.5 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 $t_r \; ⟕ \; t_s$. To get the natural right outer join $t_r \; ⟖ \; t_s$, 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 Figures A.9 and A.10. 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.

12.16 Pipelining is used to avoid writing intermediate results to disk. 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 output of the sort of r can be pipelined to the merge join without being written back to disk.

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. Using pipelining, output from the sorting operation on r is written to a buffer B. When B is full, the merge-join processes tuples from B, joining them with tuples from s until B is empty. At this point, the sorting operation is resumed and B is refilled. This process continues until the merge-join is complete.

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.

12.17 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 follows:

```
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
```

12.18 Suppose you have to compute ${}_{A}\mathcal{G}_{sum(C)}(r)$ as well as ${}_{A,B}\mathcal{G}_{sum(C)}(r)$. 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.

CHAPTER 13

Query Optimization

Practice Exercises

13.1 Show that the following equivalences hold. Explain how you can apply them to improve the efficiency of certain queries:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$, where θ uses only attributes from E_1.

Answer:

a. $E_1 \bowtie_\theta (E_2 - E_3) = (E_1 \bowtie_\theta E_2 - E_1 \bowtie_\theta E_3)$.
Let us rename $(E_1 \bowtie_\theta (E_2 - E_3))$ as R_1, $(E_1 \bowtie_\theta E_2)$ as R_2 and $(E_1 \bowtie_\theta E_3)$ as R_3. It is clear that if a tuple t belongs to R_1, it will also belong to R_2. If a tuple t belongs to R_3, $t[E_3$'s attributes] will belong to E_3, hence t cannot belong to R_1. From these two we can say that

$$\forall t,\ t \in R_1 \ \Rightarrow\ t \in (R_2 - R_3)$$

It is clear that if a tuple t belongs to $R_2 - R_3$, then $t[R_2$'s attributes] $\in E_2$ and $t[R_2$'s attributes] $\notin E_3$. Therefore:

$$\forall t,\ t \in (R_2 - R_3) \ \Rightarrow\ t \in R_1$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side join will produce many tuples which will finally be removed from the result. The left hand side expression can be evaluated more efficiently.

b. $\sigma_\theta({}_A\mathcal{G}_F(E)) = {}_A\mathcal{G}_F(\sigma_\theta(E))$, where θ uses only attributes from A.
θ uses only attributes from A. Therefore if any tuple t in the output of ${}_A\mathcal{G}_F(E)$ is filtered out by the selection of the left hand side, all the tuples in E whose value in A is equal to $t[A]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E)) \ \Rightarrow\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E))$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin {}_A\mathcal{G}_F(\sigma_\theta(E)) \ \Rightarrow\ t \notin \sigma_\theta({}_A\mathcal{G}_F(E))$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids performing the aggregation on groups which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

c. $\sigma_\theta(E_1 \supset\!\bowtie E_2) = \sigma_\theta(E_1) \supset\!\bowtie E_2$ where θ uses only attributes from E_1.
θ uses only attributes from E_1. Therefore if any tuple t in the output of $(E_1 \supset\!\bowtie E_2)$ is filtered out by the selection of the left hand side, all the tuples in E_1 whose value is equal to $t[E_1]$ are filtered out by the selection of the right hand side. Therefore:

$$\forall t,\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2) \ \Rightarrow\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2$$

Using similar reasoning, we can also conclude that

$$\forall t,\ t \notin \sigma_\theta(E_1) \supset\!\bowtie E_2 \ \Rightarrow\ t \notin \sigma_\theta(E_1 \supset\!\bowtie E_2)$$

The above two equations imply the given equivalence.
This equivalence is helpful because evaluation of the right hand side avoids producing many output tuples which are anyway going to be removed from the result. Thus the right hand side expression can be evaluated more efficiently than the left hand side expression.

13.2 For each of the following pairs of expressions, give instances of relations that show the expressions are not equivalent.

a. $\Pi_A(R - S)$ and $\Pi_A(R) - \Pi_A(S)$.

b. $\sigma_{B<4}({}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(R))$ and ${}_A\mathcal{G}_{max(B)\ \textbf{as}\ B}(\sigma_{B<4}(R))$.

c. In the preceding expressions, if both occurrences of max were replaced by min would the expressions be equivalent?

d. $(R \supset\!\bowtie S) \supset\!\bowtie T$ and $R \supset\!\bowtie (S \supset\!\bowtie T)$
In other words, the natural left outer join is not associative. (Hint: Assume that the schemas of the three relations are $R(a, b1)$, $S(a, b2)$, and $T(a, b3)$, respectively.)

e. $\sigma_\theta(E_1 \supset\!\bowtie E_2)$ and $E_1 \supset\!\bowtie \sigma_\theta(E_2)$, where θ uses only attributes from E_2.

Answer:

a. $R = \{(1, 2)\},\ S = \{(1, 3)\}$
The result of the left hand side expression is $\{(1)\}$, whereas the result of the right hand side expression is empty.

b. $R = \{(1, 2), (1, 5)\}$
The left hand side expression has an empty result, whereas the right hand side one has the result $\{(1, 2)\}$.

c. Yes, on replacing the max by the min, the expressions will become equivalent. Any tuple that the selection in the rhs eliminates would not pass the selection on the lhs if it were the minimum value, and would be eliminated anyway if it were not the minimum value.

d. $R = \{(1, 2)\}$, $S = \{(2, 3)\}$, $T = \{(1, 4)\}$. The left hand expression gives $\{(1, 2, null, 4)\}$ whereas the the right hand expression gives $\{(1, 2, 3, null)\}$.

e. Let R be of the schema (A, B) and S of (A, C). Let $R = \{(1, 2)\}$, $S = \{(2, 3)\}$ and let θ be the expression $C = 1$. The left side expression's result is empty, whereas the right side expression results in $\{(1, 2, null)\}$.

13.3 SQL allows relations with duplicates (Chapter 3).

a. Define versions of the basic relational-algebra operations σ, Π, $\times$, $\bowtie$, $-$, $\cup$, and $\cap$ that work on relations with duplicates, in a way consistent with SQL.

b. Check which of the equivalence rules 1 through 7.b hold for the multiset version of the relational-algebra defined in part a.

Answer:

a. We define the multiset versions of the relational-algebra operators here. Given multiset relations r_1 and r_2,

i. σ
Let there be c_1 copies of tuple t_1 in r_1. If t_1 satisfies the selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$, otherwise there are none.

ii. Π
For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

iii. $\times$
If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, then there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

iv. $\bowtie$
The output will be the same as a cross product followed by a selection.

v. $-$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 - c_2$ copies of t in $r_1 - r_2$, provided that $c_1 - c_2$ is positive.

vi. $\cup$

If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $c_1 + c_2$ copies of t in $r_1 \cup r_2$.

vii. $\cap$
If there are c_1 copies of tuple t in r_1 and c_2 copies of t in r_2, then there will be $min(c_1, c_2)$ copies of t in $r_1 \cap r_2$.

b. All the equivalence rules 1 through 7.b of section 13.2.1 hold for the multiset version of the relational-algebra defined in the first part. There exist equivalence rules which hold for the ordinary relational-algebra, but do not hold for the multiset version. For example consider the rule :-

$$A \cap B \quad = \quad A \cup B \; - \; (A - B) \; - \; (B - A)$$

This is clearly valid in plain relational-algebra. Consider a multiset instance in which a tuple t occurs 4 times in A and 3 times in B. t will occur 3 times in the output of the left hand side expression, but 6 times in the output of the right hand side expression. The reason for this rule to not hold in the multiset version is the asymmetry in the semantics of multiset union and intersection.

13.4 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$, with primary keys A, C, and E, respectively. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750 tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$, and give an efficient strategy for computing the join.

Answer:

- The relation resulting from the join of r_1, r_2, and r_3 will be the same no matter which way we join them, due to the associative and commutative properties of joins. So we will consider the size based on the strategy of $((r_1 \bowtie r_2) \bowtie r_3)$. Joining r_1 with r_2 will yield a relation of at most 1000 tuples, since C is a key for r_2. Likewise, joining that result with r_3 will yield a relation of at most 1000 tuples because E is a key for r_3. Therefore the final relation will have at most 1000 tuples.
- An efficient strategy for computing this join would be to create an index on attribute C for relation r_2 and on E for r_3. Then for each tuple in r_1, we do the following:
  a. Use the index for r_2 to look up at most one tuple which matches the C value of r_1.
  b. Use the created index on E to look up in r_3 at most one tuple which matches the unique value for E in r_2.

13.5 Consider the relations $r_1(A, B, C)$, $r_2(C, D, E)$, and $r_3(E, F)$ of Practice Exercise 13.4. Assume that there are no primary keys, except the entire schema. Let $V(C, r_1)$ be 900, $V(C, r_2)$ be 1100, $V(E, r_2)$ be 50, and $V(E, r_3)$ be 100. Assume that r_1 has 1000 tuples, r_2 has 1500 tuples, and r_3 has 750

tuples. Estimate the size of $r_1 \bowtie r_2 \bowtie r_3$ and give an efficient strategy for computing the join.

Answer: The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r_1, $1500/V(C, r_2)$ = 15/11 tuples (on the average) of r_2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r_3 to yield a result of approximately 10,227 tuples ($15000/11 \times 750/100 = 10227$). A good strategy should join r_1 and r_2 first, since the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

13.6 Suppose that a B^+-tree index on *building* is available on relation *department*, and that no other index is available. What would be the best way to handle the following selections that involve negation?

a. $\sigma_{\neg(building < \text{“Watson”})}(department)$

b. $\sigma_{\neg(building = \text{“Watson”})}(department)$

c. $\sigma_{\neg(building < \text{“Watson”} \vee budget < 50000)}(department)$

Answer:

a. Use the index to locate the first tuple whose *building* field has value “Watson”. From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *building* field is anything other than “Watson”.

c. This query is equivalent to the query:

$$\sigma_{building \geq \text{'Watson'} \wedge budget < 5000)}(department).$$

Using the *building* index, we can retrieve all tuples with *building* value greater than or equal to “Watson” by following the pointer chains from the first “Watson” tuple. We also apply the additional criteria of $budget < 5000$ on every tuple.

13.7 Consider the query:

```
select *
from r, s
where upper(r.A) = upper(s.A);
```

where “upper” is a function that returns its input argument with all lowercase letters replaced by the corresponding uppercase letters.

a. Find out what plan is generated for this query on the database system you use.

b. Some database systems would use a (block) nested-loop join for this query, which can be very inefficient. Briefly explain how hash-join or merge-join can be used for this query.

Answer:

a. First create relations r and s, and add some tuples to the two relations, before finding the plan chosen; or use existing relations in place of r and s. Compare the chosen plan with the plan chosen for a query directly equating $r.A = s.B$. Check the estimated statistics too. Some databases may give the same plan, but with vastly different statistics.
(On PostgreSQL, we found that the optimizer used the merge join plan described in the answer to the next part of this question.)

b. To use hash join, hashing should be done after applying the upper() function to $r.A$ and $s.A$. Similarly, for merge join, the relations should be sorted on the result of applying the upper() function on $r.A$ and $s.A$. The hash or merge join algorithms can then be used unchanged.

13.8 Give conditions under which the following expressions are equivalent

$$_{A,B}\mathcal{G}_{agg(C)}(E_1 \bowtie E_2) \quad \text{and} \quad (_{A}\mathcal{G}_{agg(C)}(E_1)) \bowtie E_2$$

where *agg* denotes any aggregation operation. How can the above conditions be relaxed if *agg* is one of **min** or **max**?
Answer: The above expressions are equivalent provided E_2 contains only attributes A and B, with A as the primary key (so there are no duplicates). It is OK if E_2 does not contain some A values that exist in the result of E_1, since such values will get filtered out in either expression. However, if there are duplicate values in $E_2.A$, the aggregate results in the two cases would be different.
If the aggregate function is min or max, duplicate A values do not have any effect. However, there should be no duplicates on (A, B); the first expression removes such duplicates, while the second does not.

13.9 Consider the issue of interesting orders in optimization. Suppose you are given a query that computes the natural join of a set of relations S. Given a subset $S1$ of S, what are the interesting orders of $S1$?
Answer: The interesting orders are all orders on subsets of attributes that can potentially participate in join conditions in further joins. Thus, let T be the set of all attributes of $S1$ that also occur in any relation in $S - S1$. Then every ordering of every subset of T is an interesting order.

13.10 Show that, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. *Hint:* A **complete binary tree** is one where every internal node has exactly two children. Use the fact that the number of different complete

binary trees with n leaf nodes is:

$$\frac{1}{n}\binom{2(n-1)}{(n-1)}$$

If you wish, you can derive the formula for the number of complete binary trees with n nodes from the formula for the number of binary trees with n nodes. The number of binary trees with n nodes is:

$$\frac{1}{n+1}\binom{2n}{n}$$

This number is known as the **Catalan number**, and its derivation can be found in any standard textbook on data structures or algorithms.

Answer: Each join order is a complete binary tree (every non-leaf node has exactly two children) with the relations as the leaves. The number of different complete binary trees with n leaf nodes is $\frac{1}{n}\binom{2(n-1)}{(n-1)}$. This is because there is a bijection between the number of complete binary trees with n leaves and number of binary trees with $n-1$ nodes. Any complete binary tree with n leaves has $n-1$ internal nodes. Removing all the leaf nodes, we get a binary tree with $n-1$ nodes. Conversely, given any binary tree with $n-1$ nodes, it can be converted to a complete binary tree by adding n leaves in a unique way. The number of binary trees with $n-1$ nodes is given by $\frac{1}{n}\binom{2(n-1)}{(n-1)}$, known as the Catalan number. Multiplying this by $n!$ for the number of permutations of the n leaves, we get the desired result.

13.11 Show that the lowest-cost join order can be computed in time $O(3^n)$. Assume that you can store and look up information about a set of relations (such as the optimal join order for the set, and the cost of that join order) in constant time. (If you find this exercise difficult, at least show the looser time bound of $O(2^{2n})$.)

Answer: Consider the dynamic programming algorithm given in Section 13.4. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x=2$. Thus the total cost is less than 3^n.

13.12 Show that, if only left-deep join trees are considered, as in the System R optimizer, the time taken to find the most efficient join order is around $n2^n$. Assume that there is only one interesting sort order.

Answer: The derivation of time taken is similar to the general case, except that instead of considering $2^{k+1}-2$ subsets of size less than or equal to

k for A, we only need to consider $k+1$ subsets of size exactly equal to k. That is because the right hand operand of the topmost join has to be a single relation. Therefore the total cost for finding the best join order for all subsets of size $k+1$ is $\binom{n}{k+1}(k+1)$, which is equal to $n\binom{n-1}{k}$. Summing over all k from 1 to $n-1$ using the binomial expansion of $(1+x)^{n-1}$ with $x=1$, gives a total cost of less than $n2^{n-1}$.

13.13 Consider the bank database of Figure 13.9, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Write a nested query on the relation *account* to find, for each branch with name starting with B, all accounts with the maximum balance at the branch.

b. Rewrite the preceding query, without using a nested subquery; in other words, decorrelate the query.

c. Give a procedure (similar to that described in Section 13.4.4) for decorrelating such queries.

Answer:

a. The nested query is as follows:

```
select  S.acount_number
from    account S
where   S.branch_name like 'B%' and
        S.balance =
        (select max(T.balance)
        from account T
        where T.branch_name = S.branch_name)
```

b. The decorrelated query is as follows:

```
create table t1 as
        select branch_name, max(balance)
        from account
        group by branch_name
select  account_number
from    account, t1
where   account.branch_name like 'B%' and
        account.branch_name = t1.branch_name and
        account.balance = t1.balance
```

c. In general, consider the queries of the form:

select $\cdots$
from L_1
where P_1 **and**
A_1 **op**
(**select f**(A_2)
from L_2
where P_2)

where, f is some aggregate function on attributes A_2, and *op* is some boolean binary operator. It can be rewritten as

create table t_1 **as**
select f(A_2),V
from L_2
where P_2^1
group by V
select $\cdots$
from L_1, t_1
where P_1 **and** P_2^2 **and**
A_1 **op** $t_1.A_2$

where P_2^1 contains predicates in P_2 without selections involving correlation variables, and P_2^2 introduces the selections involving the correlation variables. V contains all the attributes that are used in the selections involving correlation variables in the nested query.

13.14 The set version of the semijoin operator $\ltimes$ is defined as follows:

$$r \ltimes_\theta s = \Pi_R(r \bowtie_\theta s)$$

where R is the set of attributes in the schema of r. The multiset version of the semijoin operation returns the same set of tuples, but each tuple has exactly as many copies as it had in r.
Consider the nested query we saw in Section 13.4.4 which finds the names of all instructors who taught a course in 2007. Write the query in relational algebra using the multiset semjoin operation, ensuring that the number of duplicates of each name is the same as in the SQL query. (The semijoin operation is widely used for decorrelation of nested queries.)
Answer: The query can be written as follows:
$instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{year=2007}(teaches))$

CHAPTER 13

Query Optimization

This chapter describes how queries are optimized. It starts off with statistics used for query optimization, and outlines how to use these statistics to estimate selectivities and query result sizes used for cost estimation. Equivalence rules are covered next, followed by a description of a query optimization algorithm modeled after the classic System R optimization algorithm, and coverage of nested subquery optimization. The chapter ends with a description of materialized views, their role in optimization and a description of incremental view-maintenance algorithms.

It should be emphasized that the estimates of query sizes and selectivities are approximate, even if the assumptions made, such as uniformity, hold. Further, the cost estimates for various algorithms presented in Chapter 12 assume only a minimal amount of memory, and are thus worst case estimates with respect to buffer space availability. As a result, cost estimates are never very accurate. However, practical experience has shown that such estimates tend to be reasonably accurate, and plans optimal with respect to estimated cost are rarely much worse than a truly optimal plan.

We do *not* expect students to memorize the size-estimates, and we stress only the process of arriving at the estimates, not the exact values. Precision in terms of estimated cost is not a worthwhile goal, so estimates off by a few I/O operations can be considered acceptable.

The theory in this chapter is ideally backed up by lab assignments where students study the query execution plans generated by one or more database systems. Most database products have an "explain plan" feature that lets the user find the evaluation plan used on a query. It is worthwhile asking students to explore the plans generated for different queries, with and without indices. Assignment may be designed in which students measure the performance speedup provided by indices. A more challenging assignment is to design tests to see how clever the query optimizer is, and to guess from these experiments which of the optimization techniques covered in the chapter are used in the system.

Exercises

13.15 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.

13.16 Show how to derive the following equivalences by a sequence of transformations using the equivalence rules in Section 13.2.1.

a. $\sigma_{\theta_1 \wedge \theta_2 \wedge \theta_3}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(E)))$

b. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$, 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 $\sigma_{\theta_1}(E_1 \bowtie_{\theta_3} (\sigma_{\theta_2}(E_2)))$.

13.17 Consider the two expressions $\sigma_\theta(E_1 \supset\!\bowtie E_2)$ 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 $\sigma_{deptname<'Z'}(dept \rightarrowtail\!\bowtie emp)$.

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* = 'Einstein"', the two expressions would be equivalent, that is they would always have the same result, since any tuple that is in in *dept* $\rightarrowtail\!\bowtie$ *emp* but not in *dept* $\bowtie$ *emp* would not satisfy the condition since its *name* attribute would be null.

13.18 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 13.2.1 complete? Hint: Consider the equivalence $\sigma_{3=5}(r) = \{\}$.

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 13.2.1 is not complete. The expressions $\sigma_{3=5}(r)$ and $\{\}$ are equivalent, but this cannot be shown by using just these rules.

13.19 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.

13.20 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.

13.21 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 the above query using the multiset version of the semijoin operation, defined in Exercise 13.14.

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.

13.22 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 $v = r \supset\!\bowtie s$, when a set of tuples i_r is inserted in r, we add the tuples $i_r \supset\!\bowtie s$ to the view. When i_r is deleted from r, we delete $i_r \supset\!\bowtie s$ 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 $r \supset\!\bowtie s$) 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.

13.23 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% of the tuples in r are deleted. Then recompution is better that incremental view maintenance. This is because in incremental view maintenance, the 70% of the deleted tuples have to be joined with s while in recompution, just the remaining 30% 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.

13.24 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 attribue 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.

13.25 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.

13.26 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 14

Transactions

Practice Exercises

14.1 **Answer:** Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

14.2 **Answer:** There are several steps in the creation of a file. A storage area is assigned to the file in the file system, a unique i-number is given to the file and an i-node entry is inserted into the i-list. Deletion of file involves exactly opposite steps.
For the file system user in UNIX, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor though, many of the internal file system actions need to have transaction semantics. All the steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

14.3 **Answer:** Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured. In contrast, most users of file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

14.4 **Answer:** If a transaction is very long or when it fetches data from a slow disk, it takes a long time to complete. In absence of concurrency, other transactions will have to wait for longer period of time. Average responce time will increase. Also when the transaction is reading data from disk, CPU is idle. So resources are not properly utilized. Hence concurrent execution becomes important in this case. However, when the transactions are short or the data is available in memory, these problems do not occur.

14.5 **Answer:** Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practice are based on conflict serializability—they actually permit only a subset of

conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

14.6 **Answer:** There is a serializable schedule corresponding to the precedence graph below, since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, T_1, T_2, T_3, T_4, T_5.

14.7 **Answer:** A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.

14.8 **Answer:**

a. A schedule showing the Lost Update Anomaly:

T_1	T_2
read(A)	
	read(A)
	write(A)
write(A)	

In the above schedule, the value written by the transaction T_2 is lost because of the write of the transaction T_1.

b. Lost Update Anomaly in Read Committed Isolation Level

T_1	T_2
lock-S(A)	
read(A)	
unlock(A)	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
write(A)	
unlock(A)	
commit	

The locking in the above schedule ensures the Read Committed isolation level. The value written by transaction T_2 is lost due to T_1's write.

c. Lost Update Anomaly is not possible in Repeatable Read isolation level. In repeatable read isolation level, a transaction T_1 reading a

data item X, holds a shared lock on X till the end. This makes it impossible for a newer transaction T_2 to write the value of X (which requires X-lock) until T_1 finishes. This forces the serialization order T_1, T_2 and thus the value written by T_2 is not lost.

14.9 **Answer:**

Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200 respectively.

Suppose that transaction T_1 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances. Suppose that concurrent transaction T_2 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances.

Since each of the transactions checks the integrity constraints on its own snapshot, if they run concurrently each will believe that the sum of the balances after the withdrawal is \$100 and therefore its withdrawal does not violate the integrity constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both of them can commit. This is a non-serializable execution which results into a serious problem of withdrawal of more money.

14.10 **Answer:**

Consider a web-based airline reservation system. There could be many concurrent requests to see the list of available flights and available seats in each flight and to book tickets. Suppose, there are two users A and B concurrently accessing this web application, and only one seat is left on a flight.

Suppose that both user A and user B execute transactions to book a seat on the flight, and suppose that each transaction checks the total number of seats booked on the flight, and inserts a new booking record if there are enough seats left. Let T_3 and T_4 be their respective booking transactions, which run concurrently. Now T_3 and T_4 will see from their snapshots that one ticket is available and insert new booking records. Since the two transactions do not update any common data item (tuple), snapshot isolation allows both transactions to commit. This results in an extra booking, beyond the number of seats available on the flight.

However, this situation is usually not very serious since cancellations often resolve the conflict; even if the conflict is present at the time the flight is to leave, the airline can arrange a different flight for one of the passengers on the flight, giving incentives to accept the change. Using snapshot isolation improves the overall performance in this case since the booking transactions read the data from their snapshots only and do not block other concurrent transactions.

14.11 **Answer:**

The given situation will not cause any problem for the definition of conflict serializability since the ordering of operations on each data item is necessary for conflict serializability, whereas the ordering of operations on different data items is not important.

T_1	T_2
read(A)	
	read(B)
write(B)	

For the above schedule to be conflict serializable, the only ordering requirement is **read**(B) -> **write**(B). **read**(A) and **read**(B) can be in any order. Therefore, as long as the operations on a data item can be totally ordered, the definition of conflict serializability should hold on the given multiprocessor system.

CHAPTER 14

Transactions

This chapter provides an overview of transaction processing. It first motivates the problems of atomicity, consistency, isolation and durability, and introduces the notion of ACID transactions. It then presents some naive schemes, and their drawbacks, thereby motivating the techniques described in Chapters 15 and 16. The rest of the chapter describes the notion of schedules and the concept of serializability.

We strongly recommend covering this chapter in a first course on databases, since it introduces concepts that every database student should be aware of. Details on how to implement the transaction properties are covered in Chapters 15 and 16.

In the initial presentation to the ACID requirements, the isolation requirement on concurrent transactions does not insist on serializability. Following Haerder and Reuter [1983], isolation just requires that the events within a transaction must be hidden from other transactions running concurrently, in order to allow rollback. However, later in the chapter, and in most of the book (except in Chapter 26), we use the stronger condition of serializability as a requirement on concurrent transactions.

Exercises

14.12 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.

14.13 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* → *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 which it made to the database are undone, and the transaction 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.

14.14 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. Both of these are described in the chapter.

14.15 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 $A = B = 0$ 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 non-serializable schedule.

T_1	T_2
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)	

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).

14.16 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)
unlock(A)	
	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 the unlock instruction is added to show how 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.

14.17 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. 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.

14.18 Why do database systems support concurrent execution of transactions, in spite of the extra programming 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 two good reasons for allowing concurrency:

- Improved throughput and resource utilization. A transaction my involve I/O activity, 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 among them. It reduces the unpredictable delays and the average response time.

14.19 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 can not 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.

14.20 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 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.

14.21 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 non-serializable as a result.

b. Give an example of a schedule where one transaction uses the pred_read operation on relation r and another concurrent transactions 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 non-serializable.

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 15

Concurrency Control

Practice Exercises

15.1 **Answer:** Suppose two-phase locking does not ensure serializability. Then there exists a set of transactions T_0, T_1 ... T_{n-1} which obey 2PL and which produce a nonserializable schedule. A non-serializable schedule implies a cycle in the precedence graph, and we shall show that 2PL cannot produce such cycles. Without loss of generality, assume the following cycle exists in the precedence graph: $T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_{n-1} \rightarrow T_0$. Let α_i be the time at which T_i obtains its last lock (i.e. T_i's lock point). Then for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$. Then for the cycle we have

$$\alpha_0 < \alpha_1 < \alpha_2 < ... < \alpha_{n-1} < \alpha_0$$

Since $\alpha_0 < \alpha_0$ is a contradiction, no such cycle can exist. Hence 2PL cannot produce non-serializable schedules. Because of the property that for all transactions such that $T_i \rightarrow T_j$, $\alpha_i < \alpha_j$, the lock point ordering of the transactions is also a topological sort ordering of the precedence graph. Thus transactions can be serialized according to their lock points.

15.2 **Answer:**

a. Lock and unlock instructions:

T_{34}:
lock-S(A)
read(A)
lock-X(B)
read(B)
if $A = 0$
then $B := B + 1$
write(B)
unlock(A)
unlock(B)

T_{35}:
lock-S(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

b. Execution of these transactions can result in deadlock. For example, consider the following partial schedule:

T_{31}	T_{32}
lock-S (A)	
	lock-S (B)
	read(B)
read(A)	
lock-X (B)	
	lock-X (A)

The transactions are now deadlocked.

15.3 **Answer:** Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.

15.4 **Answer:**
Consider two nodes A and B, where A is a parent of B. Let dummy vertex D be added between A and B. Consider a case where transaction T_2 has a lock on B, and T_1, which has a lock on A wishes to lock B, and T_3 wishes to lock A. With the original tree, T_1 cannot release the lock on A until it gets the lock on B. With the modified tree, T_1 can get a lock on D, and release the lock on A, which allows T_3 to proceed while T_1 waits for T_2. Thus, the protocol allows locks on vertices to be released earlier to other transactions, instead of holding them when waiting for a lock on a child. A generalization of idea based on edge locks is described in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

15.5 **Answer:** Consider the tree-structured database graph given below.

Schedule possible under tree protocol but not under 2PL:

T_1	T_2
lock *(A)*	
lock *(B)*	
unlock *(A)*	
	lock *(A)*
lock *(C)*	
unlock *(B)*	
	lock *(B)*
	unlock *(A)*
	unlock *(B)*
unlock *(C)*	

Schedule possible under 2PL but not under tree protocol:

T_1	T_2
lock *(A)*	
	lock *(B)*
lock *(C)*	
	unlock *(B)*
unlock *(A)*	
unlock *(C)*	

15.6 **Answer:**

The proof is in Kedem and Silberschatz, "Locking Protocols: From Exclusive to Shared Locks," JACM Vol. 30, 4, 1983. (The proof of serializability and deadlock freedom of the original tree protocol may be found in Silberschatz and Kedem, "Consistency in Hierarchical Database Systems", JACM Vol. 27, 1, Jan 1980.)

It is worth noting that if the protocol were modified to allow update transactions to lock any data item first, non-serializable executions can occur. Intuitively, a long running readonly transaction T_0 may precede an update transaction T_1 on node a, but continues to holds a lock on child node b. After T_1 updates a and commits, readonly transaction T_2 reads a, and thus T_1 precedes T_2. However, T_2 now overtakes T_1, share locking b and its child c (not possible with only exclusive locks), and reads c and

commits. Subsequently, transaction T_3 updates c and commits, after which T_0 share locks and reads c. Thus, there is a cycle

$$T_0 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_0$$

In effect, two readonly transactions see two different orderings of the same two update transactions. By requiring update transactions to always start at the root, the protocol ensures that the above situation cannot occur. In the above example T_3 would be forced to start from the root, and thus T_0 cannot be serialized after T_3.
The formal proof is by induction on the number of read-only transactions (and, we must admit, not very intuitive). We present it after introducing some notation first:
Q^X : the set of update transactions.
Q^S :the set of Read-only transactions.
$L(T_i)$: the set of data items locked by transaction T_i
Let Q the new protocol. Suppose T_i and T_k are update transactions, and T_j a read only transaction; then if T_j overlaps with T_i and T_k in the set of data items it accesses, since T_i and T_k both start from the root, T_i and T_k must also overlap on the set of data items accessed. Formally,
$\forall T_i \in Q^X \forall T_j \in Q^S \forall T_k \in Q^X$
$[((L(T_i) \bigcap L(T_j)) \neq \phi \wedge (L(T_j) \bigcap L(T_k)) \neq \phi) \Rightarrow (L(T_i) \bigcap L(T_k)) \neq \phi]$
Consider an arbitrary schedule of transactions in protocol Q. Let k be the number of Read-only transactions which participate in this schedule. We show by induction that there exist no minimal cycles in $(T, \rightarrow)$, where $T_i \rightarrow T_j$ denotes that T_i precedes T_j due to a conflict on some data item. Let the cycle without loss of generality be,
$T_0 \rightarrow T_1 ... \rightarrow T_{m-1} \rightarrow T_0$
Here, we know that $m \geq 2k$, as there must be an Update transaction on each side of a Read-only transaction, otherwise the cycle will not be formed.

- $k = 0$: In this case each T_i is in Q^X, so Q becomes the original tree protocol which ensures the serializability and deadlock freedom.
- $k = 1$: Let T_i be the unique transaction in Q^S. We can replace T_i by T_i^X. The resulting schedule follows the original tree protocol which ensures serializability and deadlock freedom.
- $k > 1$: Let $T_i \wedge Q^S$, then $\{T_{i-1}, T_{i+1}\} \in Q^X$. As, $m \geq 2k \geq 4, i-1 \neq i+1$ (modulo m). By the assumption of the given protocol's definition,
  $L(T_{i-1}) \bigcap L(T_{i+1}) \neq \phi$
  Thus, $T_{i-1} \rightarrow T_{i+1}$ or $T_{i+1} \rightarrow T_{i-1}$; in either case, the original cycle was not minimal (in the first case, T_i can be deleted from the cycle, in the second case there is a cycle involving T_{i-i}, T_i and T_{i+1}.

By contradiction, there can be no such cycle, and thus the new protocol ensures serializability.

To show deadlock freedom, consider the waits for graph. If there is any edge from an update transaction T_j to another update transaction T_i, it is easy to see that T_i locks the root before T_j. Similarly if there is an edge from an update txn T_j to any readonly transaction T_k, which in turn can only wait for an update transaction T_i, it can be seen that T_i must have locked the root before T_j. In other words, an older update transaction cannot wait (directly or indirectly) on a younger update transaction, and thus, there can be no cycle in the waits for graph.

15.7 **Answer:**
The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979. The proof is quite non-trivial, and you may skip the details if you wish.
Consider,

- $G(V, A)$: the directed acyclic graph of the data items.
- T_0, T_1, T_2,.....,T_m are the participating transactions.
- $E(T_i)$ is the first vertex locked by transaction T_i.
- $\eta : V \rightarrow \{1,2,3,....\}$ such that for each $u, w \in V$ if $< u, w > \in A$ then $\eta(u) < \eta(w)$
- $F(T_i, T_j)$ is that $v \in L(T_i) \bigcap L(T_j)$ for which $\eta(v)$ is minimal
- For each $e \in V$, define a relation $\omega_e \subseteq T \times T$ such that $T_i \omega_e T_j$ for $e \in L(T_i) \bigcap L(T_j)$ iff T_i successfully locked e and T_j either never (successfully) locked e or locked e, only after T_i unlocked it.
- Define also, $T_i \omega T_j = \exists e[T_i \omega_e T_j]$

a. **Lemma 1** We first show that, for the given protocol,
If $L(T_i) \bigcap L(T_j) \neq \phi$ then $F(T_i, T_j) \in \{E(T_i), E(T_j)\}$
Assume by contradiction that,
$F(T_i, T_j) \notin \{E(T_i), E(T_j)\}$
But then, by the locking protocol (as both T_i and T_j) had to lock more than **half** of the predecessors of $F(T_i, T_j)$), it follows that some predecessor of $F(T_i, T_j)$ is in $L(T_i) \bigcap L(T_j)$ contradicting the definition of $F(T_i, T_j)$.

b. **Lemma 2** Now we show that,
$T_i \omega_u T_j$ for all $u \in L(T_i) \bigcap L(T_j)$ iff $T_i \omega_{F(T_i,T_j)} T_j$.

If $u = F(T_i, T_j)$ then $T_i \omega_{F(T_i,T_j)} T_j$. is trivially true. If $u \neq F(T_i, T_j)$, then $u \notin \{E(T_i), E(T_j)\}$. It thus follows that some predecessor of w of u was successfully locked by both T_i and T_j and this u was

locked by T_i and T_j when they issued the instructions to lock u. Thus, $T_i \omega_w T_j \Leftrightarrow T_i \omega_u T_j$ (and $T_j \omega_w T_i \Leftrightarrow T_j \omega_u T_i$).
Now by induction: $T_i \omega_{F(T_i,T_j)} T_j \Leftrightarrow T_i \omega_w T_j$ and the result follows.

Now, we prove that the given protocol ensures serializability and deadlock freedom by induction on the length of minimal cycle.

a. $m = 2$: The protocol ensures no minimal cycles as shown in the above Lemma 2

b. $m > 2$: Assume by contradiction that
$T_0 \omega T_1 \omega T_2 \ldots\ldots \omega T_{m-1} \omega T_0$ is a minimal cycle of length m. We will consider two cases:

i. $F(T_i, T_{i+1})$'s are not all distinct. It follows that,

$$L(T_i) \bigcap L(T_j) \bigcap L(T_k) \neq \phi \text{ for } 0 \leq i \leq j \leq k \leq m-1$$

A. Assume $< i, j, k > = < i, i+1, i+2 >$. Then it easily follows that $T_i \omega T_{i+2}$ and (*) is not a minimal cycle.

B. Assume $< i, j, k > \neq < i, i+1, i+2 >$. If say $|j - i| > 1$ then as either $T_i \omega T_j$ or $T_j \omega T_i$ and (*) is not a minimal cycle. If $|j - i| = 1$ and $|k - j| > 1$ then the proof is analogous..

ii. All $F(T_i, T_{i+1})$'s are distinct. Then for some i

$$\eta(F(T_i, T_{i+1})) < \eta(F(T_{i+1}, T_{i+2})) \qquad (**)$$

$$\text{and } \eta(F(T_{i+1}, T_{i+2})) > \eta(F(T_{i+2}, T_{i+3})) \qquad (***)$$

By (**), $E(T_{i+1}) \neq F(T_{i+1}, T_{i+2})$ and by (***), $E(T_{i+2}) \neq F(T_{i+1}, T_{i+2})$.

Thus, $F(T_{i+1}, T_{i+2}) \notin \{E(T_{i+1}), E(T_{i+2})\}$, a contradiction to Lemma 1.

We have thus shown that the given protocol ensures serializability and deadlock freedom.

15.8 Answer:
The proof is Silberschatz and Kedem, "A Family of Locking Protocols for Database Systems that Are Modeled by Directed Graphs", IEEE Trans. on Software Engg. Vol. SE-8, No. 6, Nov 1982.
The proof is rather complex; we omit details, which may be found in the above paper.

15.9 **Answer:** The access protection mechanism can be used to implement page level locking. Consider reads first. A process is allowed to read a page only after it read-locks the page. This is implemented by using `mprotect` to initially turn off read permissions to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler associated with protection violation then requests a read lock on the page, and after the lock is acquired, it uses `mprotect` to allow read access to the page by the process, and finally allows the process to continue. Write access is handled similarly.

15.10 **Answer:**

a. Serializability can be shown by observing that if two transactions have an I mode lock on the same item, the increment operations can be swapped, just like read operations. However, any pair of conflicting operations must be serialized in the order of the lock points of the corresponding transactions, as shown in Exercise 15.1.

b. The **increment** lock mode being compatible with itself allows multiple incrementing transactions to take the lock simultaneously thereby improving the concurrency of the protocol. In the absence of this mode, an **exclusive** mode will have to be taken on a data item by each transaction that wants to increment the value of this data item. An exclusive lock being incompatible with itself adds to the lock waiting time and obstructs the overall progress of the concurrent schedule.
In general, increasing the **true** entries in the compatibility matrix increases the concurrency and improves the throughput.

The proof is in Korth, "Locking Primitives in a Database System," JACM Vol. 30, 1983.

15.11 **Answer:** It would make no difference. The write protocol is such that the most recent transaction to write an item is also the one with the largest timestamp to have done so.

15.12 **Answer:** If a transaction needs to access a large a set of items, multiple granularity locking requires fewer locks, whereas if only one item needs to be accessed, the single lock granularity system allows this with just one lock. Because all the desired data items are locked and unlocked together in the multiple granularity scheme, the locking overhead is low, but concurrency is also reduced.

15.13 **Answer:** In the concurrency control scheme of Section 15.5 choosing **Start**(T_i) as the timestamp of T_i gives a subset of the schedules allowed by choosing **Validation**(T_i) as the timestamp. Using **Start**(T_i) means that whoever started first must finish first. Clearly transactions could enter the validation phase in the same order in which they began executing,

but this is overly restrictive. Since choosing **Validation**(T_i) causes fewer nonconflicting transactions to restart, it gives the better response times.

15.14 **Answer:**

- Two-phase locking: Use for simple applications where a single granularity is acceptable. If there are large read-only transactions, multiversion protocols would do better. Also, if deadlocks must be avoided at all costs, the tree protocol would be preferable.
- Two-phase locking with multiple granularity locking: Use for an application mix where some applications access individual records and others access whole relations or substantial parts thereof. The drawbacks of 2PL mentioned above also apply to this one.
- The tree protocol: Use if all applications tend to access data items in an order consistent with a particular partial order. This protocol is free of deadlocks, but transactions will often have to lock unwanted nodes in order to access the desired nodes.
- Timestamp ordering: Use if the application demands a concurrent execution that is equivalent to a particular serial ordering (say, the order of arrival), rather than *any* serial ordering. But conflicts are handled by roll-back of transactions rather than waiting, and schedules are not recoverable. To make them recoverable, additional overheads and increased response time have to be tolerated. Not suitable if there are long read-only transactions, since they will starve. Deadlocks are absent.
- Validation: If the probability that two concurrently executing transactions conflict is low, this protocol can be used advantageously to get better concurrency and good response times with low overheads. Not suitable under high contention, when a lot of wasted work will be done.
- Multiversion timestamp ordering: Use if timestamp ordering is appropriate but it is desirable for read requests to never wait. Shares the other disadvantages of the timestamp ordering protocol.
- Multiversion two-phase locking: This protocol allows read-only transactions to always commit without ever waiting. Update transactions follow 2PL, thus allowing recoverable schedules with conflicts solved by waiting rather than roll-back. But the problem of deadlocks comes back, though read-only transactions cannot get involved in them. Keeping multiple versions adds space and time overheads though, therefore plain 2PL may be preferable in low conflict situations.

15.15 **Answer:** A transaction waits on *a.* disk I/O and *b.* lock acquisition. Transactions generally wait on disk reads and not on disk writes as disk

writes are handled by the buffering mechanism in asynchronous fashion and transactions update only the in-memory copy of the disk blocks.

The technique proposed essentially separates the waiting times into two phases. The first phase – where transaction is executed without acquiring any locks and without performing any writes to the database – accounts for almost all the waiting time on disk I/O as it reads all the data blocks it needs from disk if they are not already in memory. The second phase—the transaction re-execution with strict two-phase locking—accounts for all the waiting time on acquiring locks. The second phase may, though rarely, involve a small waiting time on disk I/O if a disk block that the transaction needs is flushed to memory (by buffer manager) before the second phase starts.

The technique may increase concurrency as transactions spend almost no time on disk I/O with locks held and hence locks are held for shorter time. In the first phase the transaction reads all the data items required —and not already in memory—from disk. The locks are acquired in the second phase and the transaction does almost no disk I/O in this phase. Thus the transaction avoids spending time in disk I/O with locks held.

The technique may even increase disk throughput as the disk I/O is not stalled for want of a lock. Consider the following scenario with strict two-phase locking protocol: A transaction is waiting for a lock, the disk is idle and there are some item to be read from disk. In such a situation disk bandwidth is wasted. But in the proposed technique, the transaction will read all the required item from the disk without acquiring any lock and the disk bandwidth may be properly utilized.

Note that the proposed technique is most useful if the computation involved in the transactions is less and most of the time is spent in disk I/O and waiting on locks, as is usually the case in disk-resident databases. If the transaction is computation intensive, there may be wasted work. An optimization is to save the updates of transactions in a temporary buffer, and instead of reexecuting the transaction, to compare the data values of items when they are locked with the values used earlier. If the two values are the same for all items, then the buffered updates of the transaction are executed, instead of reexecuting the entire transaction.

15.16 **Answer:** Consider two transactions T_1 and T_2 shown below.

T_1	T_2
write(p)	
	read(p)
	read(q)
write(q)	

Let TS(T_1) < TS(T_2) and let the timestamp test at each operation except write(q) be successful. When transaction T_1 does the timestamp test for write(q) it finds that TS(T_1) < R-timestamp(q), since TS(T_1) < TS(T_2) and R-timestamp(q) = TS(T_2). Hence the writeoperation fails and transaction T_1 rolls back. The cascading results in transaction T_2 also being rolled back as it uses the value for item p that is written by transaction T_1.
If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

15.17 **Answer:** In the text, we considered two approaches to dealing with the phantom phenomenon by means of locking. The coarser granularity approach obviously works for timestamps as well. The B^+-tree index based approach can be adapted to timestamping by treating index buckets as data items with timestamps associated with them, and requiring that all read accesses use an index. We now show that this simple method works. Suppose a transaction T_i wants to access all tuples with a particular range of search-key values, using a B^+-tree index on that search-key. T_i will need to read all the buckets in that index which have key values in that range. It can be seen that any delete or insert of a tuple with a key-value in the same range will need to write one of the index buckets read by T_i. Thus the logical conflict is converted to a conflict on an index bucket, and the phantom phenomenon is avoided.

15.18 **Answer:** Note: The tree-protocol of Section 15.1.5 which is referred to in this question, is different from the multigranularity protocol of Section 15.3 and the B^+-tree concurrency protocol of Section 15.10.

One strategy for early lock releasing is given here. Going down the tree from the root, if the currently visited node's child is not full, release locks held on all nodes except the current node, request an X-lock on the child node, after getting it release the lock on the current node, and then descend to the child. On the other hand, if the child is full, retain all locks held, request an X-lock on the child, and descend to it after getting the lock. On reaching the leaf node, start the insertion procedure. This strategy results in holding locks only on the full index tree nodes from the leaf upwards, until and including the first non-full node.

An optimization to the above strategy is possible. Even if the current node's child is full, we can still release the locks on all nodes but the current one. But after getting the X-lock on the child node, we split it right away. Releasing the lock on the current node and retaining just the lock on the appropriate split child, we descend into it making it the current node. With this optimization, at any time at most two locks are held, of a parent and a child node.

15.19 **Answer:**

a. validation test for first-committer-wins scheme: Let Start(T_i), Commit(T_i) and be the timestamps associated with a transaction T_i and the update set for T_i be update_set(T_i). Then for all transactions T_k with

Commit(T_k) < Commit(T_i), one of the following two conditions must hold:

- If Commit(T_k) < Start(T_k), T_k completes its execution before T_i started, the serializability is maintained.
- If Start(T_i) < Commit(T_k) < Commit(T_i) and update_set(T_i) and update_set(T_k) do not intersect

b. Validation test for first-committer-wins scheme with W-timestamps for data items: If a transaction T_i writes a data item Q, then the W-timestamp(Q) is set to Commit(T_i). For the validation test of a transaction T_i to pass, following condition must hold:

- For each data item Q written by T_i, W-timestamp(Q) < Start(T_i)

c. First-updater-wins scheme:

i. For a data item Q written by T_i, the W-timestamp is assigned the timestamp when the write occurred in T_i

ii. Since the validation is done after acquiring the exclusive locks and the exclusive locks are held till the end of the transaction, the data item cannot be modified inbetween the lock acquisition and commit time. So, the result of validation test for a transaction would be the same at the commit time as that at the update time.

iii. Because of the exclusive locking, at the most one transaction can acquire the lock on a data item at a time and do the validation testing. Thus, two or more transactions can not do validation testing for the same data item simultaneously.

CHAPTER 15

Concurrency Control

This chapter describes how to control concurrent execution in a database, in order to ensure the isolation properties of transactions. A variety of protocols are described for this purpose. If time is short, some of the protocols may be omitted. We recommend covering, at the least, two-phase locking (Sections 15.1.1), through 15.1.3, deadlock detection and recovery (Section 15.2, omitting Section 15.2.1), and the phantom phenomenon (Section 15.8.3). The most widely used techniques would thereby be covered.

It is worthwhile pointing out how the graph-based locking protocols generalize simple protocols, such as ordered acquisition of locks, which students may have studied in an operating system course. Although the timestamp protocols by themselves are not widely used, multiversion two-phase locking (Section 15.6.2) is of increasing importance since it allows long read-only transactions to run concurrently with updates.

The phantom phenomenon is often misunderstood by students as showing that two-phase locking is incorrect. It is worth stressing that transactions that scan a relation must read some data to find out what tuples are in the relation; as long as this data is itself locked in a two-phase manner, the phantom phenomenon will not arise.

Exercises

15.20 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.

15.21 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

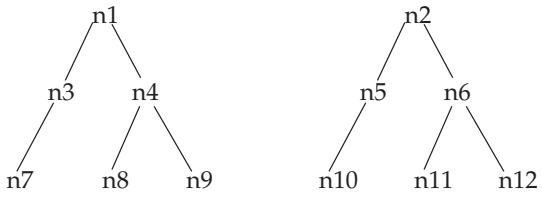
Answer: It is relatively simple to implement, imposes low rollback overhead because of cascadeless schedules, and usually allows an acceptable level of concurrency.

15.22 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 2 trees:



We have 2 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.

15.23 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.

15.24 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.

15.25 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.

15.26 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 intend-shared declaration has no meaning.

15.27 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.

15.28 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.

15.29 Show that there are schedules that are possible under the two-phase locking protocol, but are 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.

15.30 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.

15.31 As discussed in Exercise 15.19, 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 what is 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 non-serializable 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.

15.32 Outline the key similarities and differences between the timestamp based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 15.19, and the optimistic-concurrency-control-without-read-validation scheme, described in Section 15.9.3.

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.

15.33 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.

15.34 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.

15.35 Give example schedules to show that with key-value locking, if any of lookup, insert, or delete do 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 ion 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 phantom phenomenon.

15.36 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 transacations 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 till its released. But inorder 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.

15.37 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 2 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, lets assume A is a higher numbered item then B.

a. T_i executes write(A) before T_j executes read(A). So, there's a edge T_i->T_j.

b. T_j executes write(B) before T_i executes read(A). So, there's a edge T_j->T_i.

There's a cycle in the graph which means the given schedule is not conflict serializable.

CHAPTER 16

Recovery System

Practice Exercises

16.1 Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.

Answer: Within a single transaction in undo-list, suppose a data item is updated more than once, say from 1 to 2, and then from 2 to 3. If the undo log records are processed in forward order, the final value of the data item would be incorrectly set to 2, whereas by processing them in reverse order, the value is set to 1. The same logic also holds for data items updated by more than one transaction on undo-list.

Using the same example as above, but assuming the transaction committed, it is easy to see that if redo processing processes the records in forward order, the final value is set correctly to 3, but if done in reverse order, the final value would be set incorrectly to 2.

16.2 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a media (disk) failure?

Answer: Checkpointing is done with log-based recovery schemes to reduce the time required for recovery after a crash. If there is no checkpointing, then the entire log must be searched after a crash, and all transactions undone/redone from the log. If checkpointing had been performed, then most of the log-records prior to the checkpoint can be ignored at the time of recovery.

Another reason to perform checkpoints is to clear log-records from stable storage as it gets full.

Since checkpoints cause some loss in performance while they are being taken, their frequency should be reduced if fast recovery is not critical. If we need fast recovery checkpointing frequency should be increased. If

the amount of stable storage available is less, frequent checkpointing is unavoidable.
Checkpoints have no effect on recovery from a disk crash; archival dumps are the equivalent of checkpoints for recovery from disk crashes.

16.3 Some database systems allow the administrator to choose between two forms of logging: *normal logging*, used to recover from system crashes, and *archival logging*, used to recover from media (disk) failure. When can a log record be deleted, in each of these cases, using the recovery algorithm of Section 16.4?
Answer: **Normal logging**: The following log records cannot be deleted, since they may be required for recovery:

a. Any log record corresponding to a transaction which was active during the most recent checkpoint (i.e. which is part of the <checkpoint L> entry)

b. Any log record corresponding to transactions started after the recent checkpoint.

All other log records can be deleted. After each checkpoint, more records become candidates for deletion as per the above rule.
Deleting a log record while retaining an earlier log record would result in gaps in the log, and would require more complex log processing. Therefore in practise, systems find a point in the log such that all earlier log records can be deleted, and delete that part of the log. Often, the log is broken up into multiple files, and a file is deleted when all log records in the file can be deleted.

Archival logging: Archival logging retains log records that may be needed for recovery from media failure (such as disk crashes). Archival dumps are the equivalent of checkpoints for recovery from media failure. The rules for deletion above can be used for archival logs, but based on the last archival dump instead of the last checkpoint. The frequency of archival dumps would be lesser than checkpointing, since a lot of data has to be written. Thus more log records would need to be retained with archival logging.

16.4 Describe how to modify the recovery algorithm of Section 16.4 to implement savepoints, and to perform rollback to a savepoint. (Savepoints are described in Section 16.8.3.)
Answer: A savepoint can be performed as follows:

a. Output onto stable storage all log records for that transaction which are currently in main memory.

b. Output onto stable storage a log record of the form <savepoint T_i>, where T_I is the transaction identifier.

To roll back a currently executing transaction partially till a particular savepoint, execute undo processing for that transaction, till the savepoint is reached. Redo log records are generated as usual during the undo phase above.
It is possible to perform repeated undo to a single savepoint by writing a fresh savepoint record after rolling back to that savepoint. The above algorithm can be extended to support multiple savepoints for a single transaction, by giving each savepoint a name. However, once undo has rolled back past a savepoint, it is no longer possible to undo upto that savepoint.

16.5 Suppose the deferred modification technique is used in a database.

a. Is the old-value part of an update log record required any more? Why or why not?

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo-phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions, and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique, if transactions perform a large number of updates?

Answer:

a. The old-value part of an update log record is not required. If the transaction has committed, then the old value is no longer necessary as there would be no need to undo the transaction. And if the transaction was active when the system crashed, the old values are still safe in the stable storage as they haven't been modified yet.

b. During the redo phase, the undo list need not be maintained any more, since the stable storage does not reflect updates due to any uncommitted transaction.

c. A data item read will first issue a read request on the local memory of the transaction. If it is found there, it is returned. Otherwise, the item is loaded from the database buffer into the local memory of the transaction and then returned.

d. If a single transaction performs a large number of updates, there is a possibility of the transaction running out of memory to store the local copies of the data items.

16.6 The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a B^+-tree.

a. Suggest how to share as many nodes as possible between the new copy and the shadow-copy of the B^+-tree, assuming that updates are made only to leaf entries, with no insertions and deletions.

b. Even with the above optimization, logging is much cheaper than a shadow-copy scheme, for transactions that perform small updates. Explain why.

Answer:

a. To begin with, we start with the copy of just the root node pointing to the shadow-copy. As modifications are made, the leaf entry where the modification is made and all the nodes in the path from that leaf node till the root, are copied and updated. All other nodes are shared.

b. For transactions that perform small updates, the shadow-paging scheme, would copy multiple pages for a single update, even with the above optimization. Logging, on the other hand just requires small records to be created for every update; the log records are physically together in one page or a few pages, and thus only a few log page I/O operations are required to commit a transaction. Furthermore, the log pages written out across subsequent transaction commits are likely to be adjacent physically on disk, minimizng disk arm movement.

16.7 Suppose we (incorrectly) modify the recovery algorithm of Section 16.4 to not log actions taken during transaction rollback. When recovering from a system crash, transactions that were rolled back earlier would then be included in undo-list, and rolled back again. Give an example to show how actions taken during the undo phase of recovery could result in an incorrect database state. (Hint: Consider a data item updated by an aborted transaction, and then updated by a transaction that commits.)
Answer: Consider the following log records generated with the (incorrectly) modified recovery algorithm:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_2 start>
4. <T_2, A, 1000, 2000>
5. <T_2 commit>

A rollback actually happened between steps 2 and 3, but there are no log records reflecting the same. Now, this log data is processed by the recovery algorithm. At the end of the redo phase, T_1 would get added to the undo-list, and the value of A would be 2000. During the undo phase,

since T_1 is present in the undo-list, the recovery algorithm does an undo of statement 2 and A takes the value 1000. The update made by T_2, though commited, is lost.
The correct sequence of logs is as follows:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_1, A, 1000>
4. <T_1 abort>
5. <T_2 start>
6. <T_2, A, 1000, 2000>
7. <T_2 commit>

This would make sure that T_1 would not get added to the undo-list after the redo phase.

16.8 Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.
Answer: If a transaction allocates a page to a relation, even if the transaction is rolled back, the page allocation should not be undone because other transactions may have stored records in the same page. Such operations that should not be undone are called nested top actions in ARIES. They can be modeled as operations whose undo action does nothing. In ARIES such operations are implemented by creating a dummy CLR whose UndoNextLSN is set such that the transaction rollback skips the log records generated by the operation.

16.9 Suppose a transaction deletes a record, and the free space generated thus is allocated to a record inserted by another transaction, even before the first transaction commits.

a. What problem can occur if the first transaction needs to be rolled back?

b. Would this problem be an issue if page-level locking is used instead of tuple-level locking?

c. Suggest how to solve this problem while supporting tuple-level locking, by logging post-commit actions in special log records, and executing them after commit. Make sure your scheme ensures that such actions are performed exactly once.

Answer:

a. If the first transaction needs to be rolled back, the tuple deleted by that transaction will have to be restored. If undo is performed in the usual physical manner using the old values of data items, the space allocated to the new tuple would get overwritten by the transaction

undo, damaging the new tuples, and associated data structures on the disk block. This means that a logical undo operation has to be performed i.e., an insert has to be performed to undo the delete, which complicates recovery.
On related note, if the second transaction inserts with the same key, integrity constraints might be violated on rollback.

b. If page level locking is used, the free space generated by the first transaction is not allocated to another transaction till the first one commits. So this problem will not be an issue if page level locking is used.

c. The problem can be solved by deferring freeing of space till after the transaction commits. To ensure that space will be freed even if there is a system crash immediately after commit, the commit log record can be modified to contain information about freeing of space (and other similar operations) which must be performed after commit. The execution of these operations can be performed as a transaction and log records generated, following by a post-commit log record which indicates that post commit processing has been completed for the transaction.
During recovery, if a commit log record is found with post-commit actions, but no post-commit log record is found, the effects of any partial execution of post-commit operations are rolled back during recovery, and the post commit operations are reexecuted at the end of recovery. If the post-commit log record is found, the post-commit actions are not reexecuted. Thus, the actions are guaranteed to be executed exactly once.
The problem of clashes on primary key values can be solved by holding key-level locks so that no other transaction can use the key till the first transaction completes.

16.10 Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)
Answer: Interactive transactions are more difficult to recover from than batch transactions because some actions may be irrevocable. For example, an output (write) statement may have fired a missile, or caused a bank machine to give money to a customer. The best way to deal with this is to try to do all output statements at the end of the transaction. That way if the transaction aborts in the middle, no harm will be have been done.
Output operations should ideally be done atomically; for example, ATM machines often count out notes, and deliver all the notes together instead of delivering notes one-at-a-time. If output operations cannot be done atomically, a physical log of output operations, such as a disk log of events, or even a video log of what happened in the physical world can be

maintained, to allow perform recovery to be performed manually later, for example by crediting cash back to a customers account.

16.11 Sometimes a transaction has to be undone after it has committed because it was erroneously executed, for example because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.

Suggest a modification to the recovery algorithm of Section 16.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be re-executed logically, if the updates are available in the form of SQL but cannot be re-executed using their log records. Why?

Answer:

a. Consider the a bank account A with balance \$100. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. The log records corresponding to the updates of A by transactions T_1 and T_2 would be $< T_1, A, 100, 110 >$ and $< T_2, A, 110, 120 >$ resp.
Say, we wish to undo transaction T_1. The normal transaction undo mechanism will replaces the value in question— A in this example— by the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism the resulting balance would be \$100 and we would, in effect, undo both transactions, whereas we intend to undo only transaction T_1.

b. Let the erroneous transaction be T_e.

- Identify the latest archival dump, say D, before the log record $< T_e, START >$. Restore the database using the dump.
- Redo all log records starting from the dump D till the log record $< T_e, COMMIT >$. Some transaction—apart from transaction T_e —would be active at the commit time of transaction T_e. Let S_1 be the set of such transactions.
- Rollback T_e and the transactions in the set S_1. This completes point-in-time recovery.
In case logical redo is possible, later transactions can be rexecuted logically assuming log records containing logical redo

information were written for every transaction. To perform logical redo of later transactions, scan the log further starting from the log record $< T_e, COMMIT >$ till the end of the log. Note the transactions that were started after the commit point of T_e. Let the set of such transactions be S_2. Re-execute the transactions in set S_1 and S_2 logically.

c. Consider again an example from the first item. Let us assume that both transactions are undone and the balance is reverted back to the original value \$100.

Now we wish to redo transaction T_2. If we redo the log record $< T_2, A, 110, 120 >$ corresponding to transaction T_2 the balance would become \$120 and we would, in effect, redo both transactions, whereas we intend to redo only transaction T_2.

CHAPTER 16

Recovery System

This chapter covers failure models and a variety of failure recovery techniques. Recovery in a real-life database systems supporting concurrent transactions is rather complicated. To help the student understand concepts better, the chapter presents recovery models in increasing degree of complexity.

The coverage of recovery in the 6th edition has changed significantly from the previous editions. In particular, the number of different algorithms presented has been reduced from earlier, avoiding the confusion between algorithms that perform undo first, then redo, versus algorithms that perform redo first, then undo; we have standardized on the latter in this edition.

Coverage of recovery algorithms now follows the sequence below:

- In Section 16.3 we describe a number of key concepts in recovery, before presenting the basic recovery algorithm in Section 16.4; the algorithm in this section supports concurrency with strict two-phase locking.
- Sections 16.5 and 16.6 discuss how to extend the recovery algorithm to deal with issues in buffer management and failure of non-volatile storage.
- Section 16.7 discusses how to handle early lock release by using logical undo operations; the algorithm described here was referred to as the "Advanced Recovery Algorithm" in the 5th edition. The coverage here has been extended, with more examples.
- Section 16.8 presents the ARIES recovery algorithm, variants of which are widely used in practice. ARIES includes some support for early lock release, along with several optimizations that speed up recovery greatly.

Exercises

16.12 Explain the difference between the three storage types—volatile, non-volatile, 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.

Non-volatile 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 non-volatile storage is typically several times slower. Stable storage is slower than non-volatile storage because of the cost of data replication.

16.13 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.

16.14 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.

16.15 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 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.

16.16 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.

16.17 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 jusing 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 necessary 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.

16.18 Consider the log in Figure 16.5. Suppose there is a crash just before the $< T_0$ abort$>$ log record 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 16.7 for the case of logical operations) can help avoid carrying out redundant operations, which create such redundant log records.

16.19 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 16.4, 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 till the beginning of this transaction. This will increase the recovery time in the case of the recovery algorithm of Section 16.4. However, in the case of ARIES the effect is not that bad as 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 till 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.

16.20 Consider the log in Figure 16.6. Suppose there is a crash during recovery, just before the operation abort log record is written for operation O_1. Explain what would happen when the system recovers again.

Answer: **Errata note:** The question above in the book has the text ".. just before after the operation abort"; the word "after" should be deleted from the text.

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, which were their original values before T_0 or T_1 started.

16.21 Compare log-based recovery with the shadow-copy scheme in terms of their overheads, for the case when data is 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 avoids log writes, at the expense of increased IO 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 system would unnecessarily write old value, 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.

16.22 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 the 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 the 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.)

16.23 Explain the difference between a system crash and a "disaster."

Answer: In a system crash, the CPU goes down, and 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.

16.24 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 as 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.

16.25 The Oracle database system uses undo log records to provide a snapshot view of the database, under snapshot-isolation. The snapshot view seen by transaction T_i reflects updates of all transactions that had committed when T_i started, and the updates of T_i; updates of all other transactions are not visible to T_i.

Describe a scheme for buffer handling whereby transactions are given a snapshot view of pages in the buffer. Include details of how to use the log to generate the snapshot view. You can assume that operations as well as their undo actions affect only one page.

Answer: First, determine if a transaction is currently modifying the buffer. If not, then return the current contents of the buffer. Otherwise, examine the records in the undo log pertaining to this buffer. Make a copy of the buffer, then for each relevant operation in the undo log, apply the operation to the buffer copy starting with the most recent operation and working backwards until the point at which the modifying transaction began. Finally, return the buffer copy as the snapshot buffer.

CHAPTER 17

Database System Architectures

Practice Exercises

17.1 Instead of storing shared structures in shared memory, an alternative architecture would be to store them in the local memory of a special process, and access the shared data by interprocess communication with the process. What would be the drawback of such an architecture?

Answer: The drawbacks would be that two interprocess messages would be required to acquire locks, one for the request and one to confirm grant. Interprocess communication is much more expensive than memory access, so the cost of locking would increase. The process storing the shared structures could also become a bottleneck.

The benefit of this alternative is that the lock table is protected better from erroneous updates since only one process can access it.

17.2 In typical client–server systems the server machine is much more powerful than the clients; that is, its processor is faster, it may have multiple processors, and it has more memory and disk capacity. Consider instead a scenario where client and server machines have exactly the same power. Would it make sense to build a client–server system in such a scenario? Why? Which scenario would be better suited to a data-server architecture?

Answer: With powerful clients, it still makes sense to have a client-server system, rather than a fully centralized system. If the data-server architecture is used, the powerful clients can off-load all the long and compute intensive transaction processing work from the server, freeing it to perform only the work of satisfying read-write requests. even if the transaction-server model is used, the clients still take care of the user-interface work, which is typically very compute-intensive.

A fully distributed system might seem attractive in the presence of powerful clients, but client-server systems still have the advantage of simpler concurrency control and recovery schemes to be implemented

on the server alone, instead of having these actions distributed in all the machines.

17.3 Consider a database system based on a client–server architecture, with the server acting as a data server.

a. What is the effect of the speed of the interconnection between the client and the server on the choice between tuple and page shipping?

b. If page shipping is used, the cache of data at the client can be organized either as a tuple cache or a page cache. The page cache stores data in units of a page, while the tuple cache stores data in units of tuples. Assume tuples are smaller than pages. Describe one benefit of a tuple cache over a page cache.

Answer:

a. We assume that tuples are smaller than a page and fit in a page. If the interconnection link is slow it is better to choose tuple shipping, as in page shipping a lot of time will be wasted in shipping tuples that might never be needed. With a fast interconnection though, the communication overheads and latencies, not the actual volume of data to be shipped, becomes the bottle neck. In this scenario page shipping would be preferable.

b. Two benefits of an having a tuple-cache rather than a page-cache, even if page shipping is used, are:

   i. When a client runs out of cache space, it can replace objects without replacing entire pages. The reduced caching granularity might result in better cache-hit ratios.

   ii. It is possible for the server to ask clients to return some of the locks which they hold, but don't need (lock de-escalation). Thus there is scope for greater concurrency. If page caching is used, this is not possible.

17.4 Suppose a transaction is written in C with embedded SQL, and about 80 percent of the time is spent in the SQL code, with the remaining 20 percent spent in C code. How much speedup can one hope to attain if parallelism is used only for the SQL code? Explain.
Answer: Since the part which cannot be parallelized takes 20% of the total running time, the best speedup we can hope for has to be less than 5.

17.5 Some database operations such as joins can see a significant difference in speed when data (for example, one of the relations involved in a join) fits in memory as compared to the situation where the data does not fit in memory. Show how this fact can explain the phenomenon of **superlinear speedup**, where an application sees a speedup greater than the amount of resources allocated to it.
Answer: FILL

17.6 Parallel systems often have a network structure where sets of n processors connect to a single Ethernet switch, and the Ethernet switches themselves connect to another Ethernet switch. Does this architecture correspond to a bus, mesh or hypercube architecture? If not, how would you describe this interconnection architecture?

Answer: FILL

CHAPTER 17

Database-System Architectures

The chapter is suitable for an introductory course. We recommend covering it, at least as self-study material, since students are quite likely to use the non-centralized (particularly client-server) database architectures when they enter the real world. The material in this chapter could potentially be supplemented by the two-phase commit protocol (2PC), (Section 19.4.1 from Chapter 19) to give students an overview of the most important details of non-centralized database architectures.

Exercises

17.7 Why is it relatively easy to port a database from a single processor machine to a multiprocessor machine if individual queries need not be parallelized?

Answer: Porting is relatively easy to a shared memory multiprocessor machine. Databases designed for single-processor machines already provide multitasking, allowing multiple processes to run on the same processor in a time-shared manner, giving a view to the user of multiple processes running in parallel. Thus, coarse-granularity parallel machines logically appear to be identical to single-processor machines, making the porting relatively easy.

The only difference is that a multiprocessor machine has multiple processor caches, and depending on the specific machine architecture, it is possible that a memory write executed on one processor may not be visible to a read on another processor for some time. To ensure that writes are visible at other processors, the database system needs to execute a special instruction (often called a *fence* instruction) which flushes a data item from all processor caches, ensuring the next read sees the latest value. Typically this instruction is executed just before releasing a latch or lock.

Porting a database to a shared disk or shared nothing multiprocessor architecture is clearly a little harder.

17.8 Transaction-server architectures are popular for client–server relational databases, where transactions are short. On the other hand, data-server

architectures are popular for client–server object-oriented database systems, where transactions are expected to be relatively long. Give two reasons why data servers may be popular for object-oriented databases but not for relational databases.

Answer: Data servers are good if data transfer is small with respect to computation, which is often the case in applications of OODBs such as computer aided design. In contrast, in typical relational database applications such as transaction processing, a transaction performs little computation but may touch several pages, which will result in a lot of data transfer with little benefit in a data server architecture. Another reason is that structures such as indices are heavily used in relational databases, and will become spots of contention in a data server architecture, requiring frequent data transfer. There are no such points of frequent contention in typical OODB applications which involve complex data that is rarely updated concurrently by multiple processes.

17.9 What is lock de-escalation, and under what conditions is it required? Why is it not required if the unit of data shipping is an item?

Answer: In a client-server system with page shipping, when a client requests an item, the server typically grants a lock not on the requested item, but on the *page* having the item, thus implicitly granting locks on all the items in the page. The other items in the page are said to be *prefetched*. If some other client subsequently requests one of the prefetched items, the server may ask the owner of the page lock to transfer back the lock on this item. If the page lock owner doesn't need this item, it de-escalates the page lock that it holds, to item locks on all the items that it is actually accessing, and then returns the locks on the unwanted items. The server can then grant the latter lock request.

If the unit of data shipping is an item, there are no coarser granularity locks; even if prefetching is used, it is typically implemented by granting individual locks on each of the prefetched items. Thus when the server asks for a return of a lock, there is no question of de-escalation, the requested lock is just returned if the client has no use for it.

17.10 Suppose you were in charge of the database operations of a company whose main job is to process transactions. Suppose the company is growing rapidly each year, and has outgrown its current computer system. When you are choosing a new parallel computer, what measure is most relevant—speedup, batch scaleup, or transaction scaleup? Why?

Answer: With increasing scale of operations, we expect that the number of transactions submitted per unit time increases. On the other hand, we wouldn't expect most of the individual transactions to grow longer, nor would we require that a given transaction should execute more quickly now than it did before. Hence transaction scale-up is the most relevant measure in this scenario.

17.11 Database systems are typically implemented as a set of processes (or threads) sharing a shared memory area.

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 necessary to control access to the shared data structures. Since many database transactions only involve reading from a data structure, **reader-writer** locks should be used, allowing multiple processes to concurrently read from a data structure by using the lock in **shared** mode. 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 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 are 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 non-serial 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.

17.12 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 thus allowing direct access to shared memory is a security risk.

17.13 What are the factors that can work against linear scaleup 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 which affects scale-up. In a shared-nothing system, interprocess communication overheads will be the main impeding factor. Since there is no shared memory, acquiring locks, and other activities requiring message passing between processes will take more time with increased parallelism.

17.14 Memory systems can be divided into multiple modules, each of which can be serving a separate request at a given time. What impact would 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 would become 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.

17.15 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.

CHAPTER 18

Parallel Databases

Practice Exercises

18.1 In a range selection on a range-partitioned attribute, it is possible that only one disk may need to be accessed. Describe the benefits and drawbacks of this property.

Answer: If there are few tuples in the queried range, then each query can be processed quickly on a single disk. This allows parallel execution of queries with reduced overhead of initiating queries on multiple disks.

On the other hand, if there are many tuples in the queried range, each query takes a long time to execute as there is no parallelism within its execution. Also, some of the disks can become hot-spots, further increasing response time.

Hybrid range partitioning, in which small ranges (a few blocks each) are partitioned in a round-robin fashion, provides the benefits of range partitioning without its drawbacks.

18.2 What form of parallelism (interquery, interoperation, or intraoperation) is likely to be the most important for each of the following tasks?

a. Increasing the throughput of a system with many small queries

b. Increasing the throughput of a system with a few large queries, when the number of disks and processors is large

Answer:

a. When there are many small queries, inter-query parallelism gives good throughput. Parallelizing each of these small queries would increase the initiation overhead, without any significant reduction in response time.

b. With a few large queries, intra-query parallelism is essential to get fast response times. Given that there are large number of processors and disks, only intra-operation parallelism can take advantage of the parallel hardware – for queries typically have

few operations, but each one needs to process a large number of tuples.

18.3 With pipelined parallelism, it is often a good idea to perform several operations in a pipeline on a single processor, even when many processors are available.

a. Explain why.

b. Would the arguments you advanced in part *a* hold if the machine has a shared-memory architecture? Explain why or why not.

c. Would the arguments in part *a* hold with independent parallelism? (That is, are there cases where, even if the operations are not pipelined and there are many processors available, it is still a good idea to perform several operations on the same processor?)

Answer:

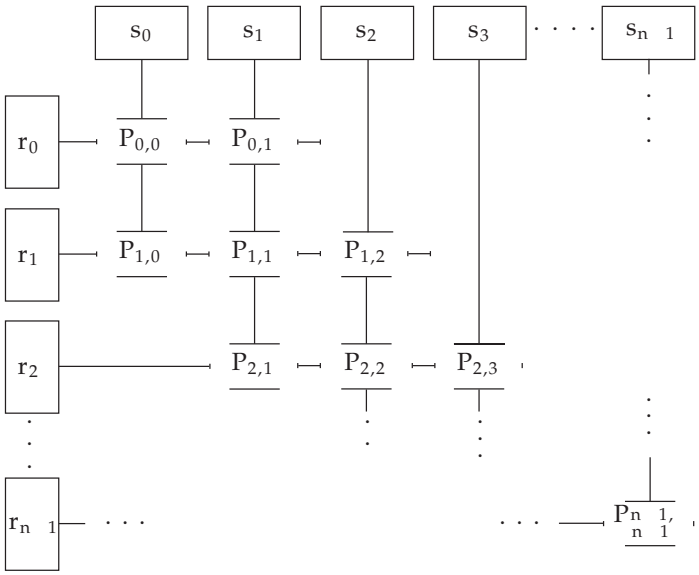
a. The speed-up obtained by parallelizing the operations would be offset by the data transfer overhead, as each tuple produced by an operator would have to be transferred to its consumer, which is running on a different processor.

b. In a shared-memory architecture, transferring the tuples is very efficient. So the above argument does not hold to any significant degree.

c. Even if two operations are independent, it may be that they both supply their outputs to a common third operator. In that case, running all three on the same processor may be better than transferring tuples across processors.

18.4 Consider join processing using symmetric fragment and replicate with range partitioning. How can you optimize the evaluation if the join condition is of the form $| r.A - s.B | \leq k$, where k is a small constant? Here, $| x |$ denotes the absolute value of x. A join with such a join condition is called a **band join**.

Answer: Relation r is partitioned into n partitions, $r_0, r_1, \ldots, r_{n-1}$, and s is also partitioned into n partitions, $s_0, s_1, \ldots, s_{n-1}$. The partitions are replicated and assigned to processors as shown below.



Each fragment is replicated on 3 processors only, unlike in the general case where it is replicated on n processors. The number of processors required is now approximately $3n$, instead of n^2 in the general case. Therefore given the same number of processors, we can partition the relations into more fragments with this optimization, thus making each local join faster.

18.5 Recall that histograms are used for constructing load-balanced range partitions.

a. Suppose you have a histogram where values are between 1 and 100, and are partitioned into 10 ranges, 1–10, 11–20, . . . , 91–100, with frequencies 15, 5, 20, 10, 10, 5, 5, 20, 5, and 5, respectively. Give a load-balanced range partitioning function to divide the values into 5 partitions.

b. Write an algorithm for computing a balanced range partition with p partitions, given a histogram of frequency distributions containing n ranges.

Answer:

a. A partitioning vector which gives 5 partitions with 20 tuples in each partition is: [21, 31, 51, 76]. The 5 partitions obtained are $1-20, 21-30, 31-50, 51-75$ and $76-100$. The assumption made in arriving at this partitioning vector is that within a histogram range, each value is equally likely.

b. Let the histogram ranges be called $h_1, h_2, \ldots, h_h$, and the partitions $p_1, p_2, \ldots, p_p$. Let the frequencies of the histogram ranges be

$n_1, n_2, \ldots, n_h$. Each partition should contain N/p tuples, where $N = \Sigma_{i=1}^{h} n_i$.
To construct the load balanced partitioning vector, we need to determine the value of the k_1^{th} tuple, the value of the k_2^{th} tuple and so on, where $k_1 = N/p, k_2 = 2N/p$ etc, until k_{p-1}. The partitioning vector will then be $[k_1, k_2, \ldots, k_{p-1}]$. The value of the k_i^{th} tuple is determined as follows. First determine the histogram range h_j in which it falls. Assuming all values in a range are equally likely, the k_i^{th} value will be

$$s_j + (e_j - s_j) * \frac{k_{ij}}{n_j}$$

where

s_j	:	first value in h_j
e_j	:	last value in h_j
k_{ij}	:	$k_i - \Sigma_{l=1}^{j-1} n_l$

18.6 Large-scale parallel database systems store an extra copy of each data item on disks attached to a different processor, to avoid loss of data if one of the processors fails.

a. Instead of keeping the extra copy of data items from a processor at a single backup processor, it is a good idea to partition the copies of the data items of a processor across multiple processors. Explain why.

b. Explain how virtual-processor partitioning can be used to efficiently implement the partitioning of the copies as described above.

c. What are the benefits and drawbacks of using RAID storage instead of storing an extra copy of each data item?

Answer: FILL

18.7 Suppose we wish to index a large relation that is partitioned. Can the idea of partitioning (including virtual processor partitioning) be applied to indices? Explain your answer, considering the following two cases (assuming for simplicity that partitioning as well as indexing are on single attributes):

a. Where the index is on the partitioning attribute of the relation.

b. Where the index is on an attribute other than the partitioning attribute of the relation.

Answer: FILL

18.8 Suppose a well-balanced range-partitioning vector had been chosen for a relation, but the relation is subsequently updated, making the partitioning unbalanced. Even if virtual-processor partitioning is used,

a particular virtual processor may end up with a very large number of tuples after the update, and repartitioning would then be required.

a. Suppose a virtual processor has a significant excess of tuples (say, twice the average). Explain how repartitioning can be done by splitting the partition, thereby increasing the number of virtual processors.

b. If, instead of round-robin allocation of virtual processors, virtual partitions can be allocated to processors in an arbitrary fashion, with a mapping table tracking the allocation. If a particular node has excess load (compared to the others), explain how load can be balanced.

c. Assuming there are no updates, does query processing have to be stopped while repartitioning, or reallocation of virtual processors, is carried out? Explain your answer.

Answer: FILL

CHAPTER 18

Parallel Databases

This chapter is suitable for an advanced course, but can also be used for independent study projects by students of a first course. The chapter covers several aspects of the design of parallel database systems — partitioning of data, parallelization of individual relational operations, and parallelization of relational expressions. The chapter also briefly covers some systems issues, such as cache coherency and failure resiliency.

The most important applications of parallel databases today are for warehousing and analyzing large amounts of data. Therefore partitioning of data and parallel query processing are covered in significant detail. Query optimization is also of importance, for the same reason. However, parallel query optimization is still not a fully solved problem; exhaustive search, as is used for sequential query optimization, is too expensive in a parallel system, forcing the use of heuristics.

The description of parallel query processing algorithms is based on the shared-nothing model. Students may be asked to study how the algorithms can be improved if shared-memory machines are used instead.

Exercises

18.9 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 speed-up and fast response time.

Hash partitioning
For point queries on the partitioning attributes, this gives the fastest response, as each disk can process a differnt 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.

18.10 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 equal number of tuples per range will give a good partitioning vector with very low skew.

18.11 Give an example of a join that is not a simple equi-join for which partitioned parallelism can be used. What attributes should be used for partitioning?

Answer: We give two examples of such joins.

a. $r \bowtie_{(r.A=s.B) \wedge (r.A<s.C)} s$
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,

b. $r \bowtie_{(r.A \geq (\lfloor s.B/20 \rfloor) * 20) \wedge (r.A < ((\lfloor s.B/20 \rfloor)+1) * 20)} s$
This is a query in which an r tuple and an s tuple join with each other if they fall into the same range of values. Hence partitioned parallelism applies naturally to this scenario, even though the join is not an equi-join.

For both the queries, r should be partitioned on attribute A and s on attribute B. For the second query, the partitioning of s should actually be done on $(\lfloor s.B/20 \rfloor) * 20$.

18.12 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 the 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 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 12.

f. The left outer join can be computed using an extension of the Fragment-and-Replicate scheme to compute non equi-joins. Consider $r \mathbin{⟕} s$. 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 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.

18.13 Describe the benefits and drawbacks of pipelined parallelism.

Answer:

- **Benefits:** No need to write intermediate relations to disk only to read them back immediately.
- **Drawbacks:**
  a. Cannot take advantage of high degrees of parallelism, as typical queries do not have large number of operations.
  b. Not possible to pipeline operators which need to look at all the input before producing any output.
  c. Since each operation executes on a single processor, the most expensive ones take a long time to finish. Thus speed-up will be low despite the use of parallelism.

18.14 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 amongst the processors of the distributed system can significant 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 entirety across all nodes. If the *account* relation is updated frequently and accesses are well-distributed, it should be partitioned across nodes.

18.15 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 has 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 by one parametrized query along with a count of number of occurrences and (b) dropping queries which are very cheap anyway, 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 can speed up some queries significantly.

CHAPTER 19

Distributed Databases

Practice Exercises

19.1 How might a distributed database designed for a local-area network differ from one designed for a wide-area network?

Answer: Data transfer on a local-area network (LAN) is much faster than on a wide-area network (WAN). Thus replication and fragmentation will not increase throughput and speed-up on a LAN, as much as in a WAN. But even in a LAN, replication has its uses in increasing reliability and availability.

19.2 To build a highly available distributed system, you must know what kinds of failures can occur.

a. List possible types of failure in a distributed system.

b. Which items in your list from part a are also applicable to a centralized system?

Answer:

a. The types of failure that can occur in a distributed system include

   i. Site failure.

   ii. Disk failure.

   iii. Communication failure, leading to disconnection of one or more sites from the network.

b. The first two failure types can also occur on centralized systems.

19.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Practice Exercise 19.2a, explain how 2PC ensures transaction atomicity despite the failure.

Answer: A proof that 2PC guarantees atomic commits/aborts inspite of site and link failures, follows. The main idea is that after all sites reply with a <**ready** *T*> message, only the co-ordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a

site can happen only after it ascertains the co-ordinator's decision, either directly from the co-ordinator, or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

a. A site can abort a transaction T (by writing an <**abort** T> log record) only under the following circumstances:

   i. It has not yet written a <**ready** T> log-record. In this case, the co-ordinator could not have got, and will not get a <**ready** T> or <**commit** T> message from this site. Therefore only an abort decision can be made by the co-ordinator.

   ii. It has written the <**ready** T> log record, but on inquiry it found out that some other site has an <**abort** T> log record. In this case it is correct for it to abort, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually aborting.

   iii. It is itself the co-ordinator. In this case also no site could have committed, or will commit in the future, because commit decisions can be made only by the co-ordinator.

b. A site can commit a transaction T (by writing an <**commit** T> log record) only under the following circumstances:

   i. It has written the <**ready** T> log record, and on inquiry it found out that some other site has a <**commit** T> log record. In this case it is correct for it to commit, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually committing.

   ii. It is itself the co-ordinator. In this case no other participating site can abort/ would have aborted, because abort decisions are made only by the co-ordinator.

19.4 Consider a distributed system with two sites, A and B. Can site A distinguish among the following?

- B goes down.
- The link between A and B goes down.
- B is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Answer:

Site A cannot distinguish between the three cases until communication has resumed with site B. The action which it performs while B is inaccessible must be correct irrespective of which of these situations has actually

occurred, and must be such that B can re-integrate consistently into the distributed system once communication is restored.

19.5 The persistent messaging scheme described in this chapter depends on timestamps combined with discarding of received messages if they are too old. Suggest an alternative scheme based on sequence numbers instead of timestamps.

Answer: We can have a scheme based on sequence numbers similar to the scheme based on timestamps. We tag each message with a sequence number that is unique for the (sending site, receiving site) pair. The number is increased by 1 for each new message sent from the sending site to the receiving site.

The receiving site stores and acknowledges a received message only if it has received all lower numbered messages also; the message is stored in the *received-messages* relation.

The sending site retransmits a message until it has received an ack from the receiving site containing the sequence number of the transmitted message, or a higher sequence number. Once the acknowledgment is received, it can delete the message from its send queue.

The receiving site discards all messages it receives that have a lower sequence number than the latest stored message from the sending site. The receiving site discards from *received-messages* all but the (number of the) most recent message from each sending site (message can be discarded only after being processed locally).

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of messages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

19.6 Give an example where the read one, write all available approach leads to an erroneous state.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Let one of the sites, say s, be down when T_1 is executed and transaction t_2 reads the balance from site s. One can see that the balance at the primary site would be \$110 at the end.

19.7 Explain the difference between data replication in a distributed system and the maintenance of a remote backup site.

Answer: In remote backup systems all transactions are performed at the primary site and the data is replicated at the remote backup site. The

remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.
The distributed systems offer greater availability by having multiple copies of the data at different sites whereas the remote backup systems offer lesser availability at lower cost and execution overhead.
In a distributed system, transaction code runs at all the sites whereas in a remote backup system it runs only at the primary site. The distributed system transactions follow two-phase commit to have the data in consistent state whereas a remote backup system does not follow two-phase commit and avoids related overhead.

19.8 Give an example where lazy replication can lead to an inconsistent database state even when updates get an exclusive lock on the primary (master) copy.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Suppose the copy of the balance at one of the sites, say s, is not consistent – due to lazy replication strategy – with the primary copy after transaction T_1 is executed and let transaction T_2 read this copy of the balance. One can see that the balance at the primary site would be \$110 at the end.

19.9 Consider the following deadlock-detection algorithm. When transaction T_i, at site S_1, requests a resource from T_j, at site S_3, a request message with timestamp n is sent. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_1. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_3 only if T_j has received the request message and cannot immediately grant the requested resource. A request from T_i to T_j in the same site is handled in the usual manner; no timestamps are associated with the edge (T_i, T_j). A central coordinator invokes the detection algorithm by sending an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. Note that such a graph contains all the local information that the site has about the state of the real graph. The wait-for graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.

When the controller has received a reply from each site, it constructs a graph as follows:

- The graph contains a vertex for every transaction in the system.
- The graph has an edge (T_i, T_j) if and only if:
  - There is an edge (T_i, T_j) in one of the wait-for graphs.

- An edge (T_i, T_j, n) (for some n) appears in more than one wait-for graph.

Show that, if there is a cycle in the constructed graph, then the system is in a deadlock state, and that, if there is no cycle in the constructed graph, then the system was not in a deadlock state when the execution of the algorithm began.

Answer: Let us say a cycle $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_m \rightarrow T_i$ exists in the graph built by the controller. The edges in the graph will either be local edges of the from (T_k, T_l) or distributed edges of the form (T_k, T_l, n). Each local edge (T_k, T_l) definitely implies that T_k is waiting for T_l. Since a distributed edge (T_k, T_l, n) is inserted into the graph only if T_k's request has reached T_l and T_l cannot immediately release the lock, T_k is indeed waiting for T_l. Therefore every edge in the cycle indeed represents a transaction waiting for another. For a detailed proof that this imlies a deadlock refer to Stuart et al. [1984].

We now prove the converse implication. As soon as it is discovered that T_k is waiting for T_l:

a. a local edge (T_k, T_l) is added if both are on the same site.

b. The edge (T_k, T_l, n) is added in both the sites, if T_k and T_l are on different sites.

Therefore, if the algorithm were able to collect all the local wait-for graphs at the same instant, it would definitely discover a cycle in the constructed graph, in case there is a circular wait at that instant. If there is a circular wait at the instant when the algorithm began execution, none of the edges participating in that cycle can disappear until the algorithm finishes. Therefore, even though the algorithm cannot collect all the local graphs at the same instant, any cycle which existed just before it started will anyway be detected.

19.10 Consider a relation that is fragmented horizontally by *plant_number*:

employee (*name, address, salary, plant_number*)

Assume that each fragment has two replicas: one stored at the New York site and one stored locally at the plant site. Describe a good processing strategy for the following queries entered at the San Jose site.

a. Find all employees at the Boca plant.

b. Find the average salary of all employees.

c. Find the highest-paid employee at each of the following sites: Toronto, Edmonton, Vancouver, Montreal.

d. Find the lowest-paid employee in the company.

Answer:

a. i. Send the query $\Pi_{name}(employee)$ to the Boca plant.
   ii. Have the Boca location send back the answer.

b. i. Compute average at New York.
   ii. Send answer to San Jose.

c. i. Send the query to find the highest salaried employee to Toronto, Edmonton, Vancouver, and Montreal.
   ii. Compute the queries at those sites.
   iii. Return answers to San Jose.

d. i. Send the query to find the lowest salaried employee to New York.
   ii. Compute the query at New York.
   iii. Send answer to San Jose.

19.11 Compute $r \ltimes s$ for the relations of Figure 19.9.

Answer: The result is as follows.

$r \ltimes s =$

A	B	C
1	2	3
5	3	2

19.12 Give an example of an application ideally suited for the cloud and another that would be hard to implement successfully in the cloud. Explain your answer.

Answer: Any application that is easy to partition, and does not need strong guarantees of consistency across partitions, is ideally suited to the cloud. For example, Web-based document storage systems (like Google docs), and Web based email systems (like Hotmail, Yahoo! mail or GMail), are ideally suited to the cloud. The cloud is also ideally suited to certain kinds of data analysis tasks where the data is already on the cloud; for example, the Google Map-Reduce framework, and Yahoo! Hadoop are widely used for data analysis of Web logs such as logs of URLs clicked by users.

Any database application that needs transactional consistency would be hard to implement successfully in the cloud; examples include bank records, academic records of students, and many other types of organizational records.

19.13 Given that the LDAP functionality can be implemented on top of a database system, what is the need for the LDAP standard?

Answer: The reasons are:

a. Directory access protocols are simplified protocols that cater to a limited type of access to data.

b. Directory systems provide a simple mechanism to name objects in a hierarchical fashion which can be used in a distributed directory system to specify what information is stored in each of the directory servers. The directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

19.14 Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.

b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

Answer:

a. We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the co-ordinator, and requiring that the lock be requested on the data item which resides on the currently elected co-ordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

19.15 Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.

b. Show how you could use a ticket scheme to ensure global serializability.

Answer:

a. The same system as in the answer to Exercise 19.14 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.

CHAPTER 19

Distributed Databases

Distributed databases in general, and *heterogeneous* distributed databases in particular, are of increasing practical importance, as organizations attempt to integrate databases across physical and organizational boundaries. Such interconnection of databases to create a distributed or multidatabase is in fact proving crucial to competitiveness for many companies. This chapter reconsiders the issues addressed earlier in the text, such as query processing, recovery and concurrency control, from the standpoint of distributed databases.

This is a long chapter, and is appropriate only for an advanced course. Single topics may be chosen for inclusion in an introductory course. Good choices include distributed data storage, heterogeneity and two-phase commit.

Exercises

19.16 Discuss the relative advantages of centralized and distributed databases.

Answer:

- A distributed database allows a user convenient and transparent access to data which is not stored at the site, while allowing each site control over its own local data. A distributed database can be made more reliable than a centralized system because if one site fails, the database can continue functioning, but if the centralized system fails, the database can no longer continue with its normal operation. Also, a distributed database allows parallel execution of queries and possibly splitting one query into many parts to increase throughput.
- A centralized system is easier to design and implement. A centralized system is cheaper to operate because messages do not have to be sent.

19.17 Explain how the following differ: fragmentation transparency, replication transparency, and location transparency.

Answer:

a. With fragmentation transparency, the user of the system is unaware of any fragmentation the system has implemented. A user may for-

mulate queries against global relations and the system will perform the necessary transformation to generate correct output.

b. With replication transparency, the user is unaware of any replicated data. The system must prevent inconsistent operations on the data. This requires more complex concurrency control algorithms.

c. Location transparency means the user is unaware of where data are stored. The system must route data requests to the appropriate sites.

19.18 When is it useful to have replication or fragmentation of data? Explain your answer.

Answer: Replication is useful when there are many read-only transactions at different sites wanting access to the same data. They can all execute quickly in parallel, accessing local data. But updates become difficult with replication. Fragmentation is useful if transactions on different sites tend to access different parts of the database.

19.19 Explain the notions of transparency and autonomy. Why are these notions desirable from a human-factors standpoint?

Answer: Autonomy is the amount of control a single site has over the local database. It is important because users at that site want quick and correct access to local data items. This is especially true when one considers that local data will be most frequently accessed in a database. Transparency hides the distributed nature of the database. This is important because users should not be required to know about location, replication, fragmentation or other implementation aspects of the database.

19.20 If we apply a distributed version of the multiple-granularity protocol of Chapter 15 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 all possible intention-mode locks 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 all intention 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.

19.21 Study and summarize the facilities that the database system you are using provides for dealing with inconsistent states that can be reached with lazy propagation of updates.

Answer:

PostgreSQL does not have built-in support for replication. However, there are several external projects that add replication support to the database engine.

- **Slony-I** adds basic master-slave replication functionality to PostgreSQL with updates to the replicated databases performed lazily. Slony-I uses the trigger mechanism of PostgreSQL to implement replication. A consistent view of the replicated data is provided by preserving transactions across the replicated sites. Thus, the replicated sites will always have a consistent, although potentially older, version of the data.
  Slony-I does not support multi-master replication, and therefore does not have to deal with inconsistent states due to updates at multiple sites.
- **Postgres-R** extends PostgreSQL to add synchronous replication, thus avoiding the consistency issues of performing lazy updates. However, Postgres-R is experimental software, and not ready for production use.

Several databases support lazy multi-master replication, which involves the risk of inconsistent data. The first problem lies in detecting the conflict, and the second problem lies in resolving the conflict. Key values and timestamps are the primary means of detecting conflicts:

- Uniqueness conflicts are detected when there are duplicates on a primary key.
- Update conflicts are detected when two sites update the same item independently. Timestamps or version numbers sent with updates are used to detect such conflicts, by recording the timestamp/version number prior to and after the update.
- Delete conflicts occur when a transaction at one site deletes a tuple, which another site concurrently updates (before the delete propagated to that site).

Oracle provides several options for resolving conflicts:

- Latest time stamp: Most recent update wins
- Overwrite: Overwrites current value with new value, without checking for conflict.
- Discard: Ignores the value
- Additive: Difference of the two values is added to the current value (current value = current value + (new value - old value)). Alternatives include retaining the minimum or maximum value.

As an alternative to using the predefined conflict resolution policies in Oracle, administrators can create stored procedures to resolve each type of conflict on a relation.
In Microsoft SQL Server 2008, multi-master replication is called peer-to-peer replication. SQL Server supports conflict detection between all combinations of insert, delete and update, based on the primary key, and a node identifier plus a version number stored with each tuple. In case conflicts are detected, the default is to stop replication, but the default can be overridden using stored procedures to resolve conflicts.
In addition, SQL Server also supports a form of replication called Merge Replication, which is based on a single-publisher/multiple-subscriber model, where each subscriber can update data and send updates back to the publisher. The resolution model simply retains one of the conflicting updates, with the publisher having higher priority, and a priority scheme between subscribers used to choose the winning subscriber in case two subscribers updated the same data.

19.22 Discuss the advantages and disadvantages of the two methods that we presented in Section 19.5.2 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 exchanges to keep the system fair, or one site can get ahead of all other sites and dominate the database.

19.23 Consider the relations:

employee (*name, address, salary, plant_number*)
machine (*machine_number, type, plant_number*)

Assume that the *employee* relation is fragmented horizontally by *plant_number*, and that each fragment is stored locally at its corresponding plant site. Assume that the *machine* relation is stored in its entirety at the Armonk site. Describe a good strategy for processing each of the following queries.

a. Find all employees at the plant that contains machine number 1130.

b. Find all employees at plants that contain machines whose type is "milling machine."

c. Find all machines at the Almaden plant.

d. Find employee $\bowtie$ machine.

Answer:

a. i. Perform $\Pi_{plant\_number}\ (\sigma_{machine\_number=1130}\ (machine))$ at Armonk.

ii. Send the query Π_{name} (*employee*) to all site(s) which are in the result of the previous query.

iii. Those sites compute the answers.

iv. Union the answers at the destination site.

b. This strategy is the same as in part *a* of this exercise, except the first step should be to perform

$$\Pi_{plant\_number}\ (\sigma_{type=\text{“milling machine"}}\ (machine))$$ at Armonk.

c. i. Perform $\sigma_{plant\_number\ =\ x}\ (machine)$ at Armonk, where x is the plant_number for Almaden.

ii. Send the answers to the destination site.

d. Strategy 1:

i. Group *machine* at Armonk by plant number.

ii. Send the groups to the sites with the corresponding plant_number.

iii. Perform a local join between the local data and the received data.

iv. Union the results at the destination site.

Strategy 2:

Send the *machine* relation at Armonk, and all the fragments of the *employee* relation to the destination site. Then perform the join at the destination site.

There is parallelism in the join computation according to the first strategy but not in the second. Nevertheless, in a WAN the amount of data to be shipped is the main cost factor. We expect that each plant will have more than one machine, hence the result of the local join at each site will be a cross-product of the employee tuples and machines at that plant. This cross-product's size is greater than the size of the *employee* fragment at that site. As a result the second strategy will result in less data shipping, and will be more efficient.

19.24 For each of the strategies of Exercise 19.23, state how your choice of a strategy depends on:

a. The site at which the query was entered.

b. The site at which the result is desired.

Answer:

a. Assuming that the cost of shipping the query itself is minimal, the site at which the query was submitted does not affect our strategy for query evaluation.

b. For the first query, we find out the plant numbers where the machine number 1130 is present, at Armonk. Then the employee tuples at all those plants are shipped to the destination site. We can see that this strategy is more or less independent of the destination site. The same can be said of the second query. For the third query, the selection is performed at Armonk and results shipped to the destination site. This strategy is obviously independent of the destination site.
For the fourth query, we have two strategies. The first one performs local joins at all the plant sites and their results are unioned at the destination site. In the second strategy, the *machine* relation at Armonk as well as all the fragments of the *employee* relation are first shipped to the destination, where the join operation is performed. There is no obvious way to optimize these two strategies based on the destination site. In the answer to Exercise 19.23 we saw the reason why the second strategy is expected to result in less data shipping than the first. That reason is independent of destination site, and hence we can in general prefer strategy two to strategy one, regardless of the destination site.

19.25 Is the expression $r_i \ltimes r_j$ necessarily equal to $r_j \ltimes r_i$? Under what conditions does $r_i \ltimes r_j = r_j \ltimes r_i$ hold?
Answer: In general, $r_i \ltimes r_j \neq r_j \ltimes r_i$. This can be easily seen from Exercise 19.11, in which $r \ltimes s \neq s \ltimes r$. $r \ltimes s$ was given in 19.11, while

$s \ltimes r =$

C	D	E
3	4	5
3	6	8
2	3	2

By definition, $r_i \ltimes r_j = \Pi_{R_i}(r_i \bowtie r_j)$ and $r_j \ltimes r_i = \Pi_{R_j}(r_i \bowtie r_j)$, where R_i and R_j are the schemas of r_i and r_j respectively. For $\Pi_{R_i}(r_i \bowtie r_j)$ to be always equal to $\Pi_{R_j}(r_i \bowtie r_j)$, the schemas R_i and R_j must be the same.

19.26 If a cloud data-storage service is used to store two relations r and s and we need to join r and s, why might it be useful to maintain the join as a materialized view? In your answer, be sure to distinguish among various meanings of "useful": overall throughput, efficient use of space, and response time to user queries.
Answer: Performing a join on a cloud data-storage system can be very expensive, if either of the relations to be joined is partitioned on attributes

other than the join attributes, since a very large amount of data would need to be transferred to perform the join. However, if $r \bowtie s$ is maintained as a materialized view, it can be updated at a relatively low cost each time each time either r or s is updated, instead of incurring a very large cost when the query is executed. Thus, queries are benefitted at some cost to updates.

Considering the various notions of usefulness, with the materialized view, overall throughput will be much better if the join query is executed reasonably often relative to updates, but may be worse if the join is rarely used, but updates are frequent.

The materialized view will certainly require extra space, but given that disk capacities are very high relative to IO (seek) operations and transfer rates, the extra space is likely to not be an major overhead since a large number of disks are needed anyway to handle the IO load and data transfer load.

The materialized view will obviously be very useful to evaluate join queries, reducing time greatly by avoiding a large amount of data transfer across machines.

19.27 Why do cloud-computing services support traditional database systems best by using a virtual machine instead of running directly on the service provider's actual machine?

Answer: By using a virtual machine, if a physical machine fails, virtual machines running on that physical machine can be restarted quickly on one or more other physical machines, improving availability. (Assuming of course that data remains accessible, either by storing multiple copies of data, or by storing data in an highly available external storage system.)

19.28 Describe how LDAP can be used to provide multiple hierarchical views of data, without replicating the base-level data.

Answer: This can be done using referrals. For example an organization may maintain its information about departments either by geography (i.e. all departments in a site of the the organization) or by structure (i.e. information about a department from all sites). These two hierarchies can be maintained by defining two different schemas with department information at a site as the base information. The entries in the two hierarchies will refer to the base information entry using referrals.

CHAPTER 20

Data Analysis and Mining

Practice Exercises

20.1 Describe benefits and drawbacks of a source-driven architecture for gathering of data at a data warehouse, as compared to a destination-driven architecture.

Answer: In a destination-driven architecture for gathering data, data transfers from the data sources to the data-warehouse are based on demand from the warehouse, whereas in a source-driven architecture, the transfers are initiated by each source.

The benefits of a source-driven architecture are

- Data can be propagated to the destination as soon as it becomes available. For a destination-driven architecture to collect data as soon as it is available, the warehouse would have to probe the sources frequently, leading to a high overhead.
- The source does not have to keep historical information. As soon as data is updated, the source can send an update message to the destination and forget the history of the updates. In contrast, in a destination-driven architecture, each source has to maintain a history of data which have not yet been collected by the data warehouse. Thus storage requirements at the source are lower for a source-driven architecture.

On the other hand, a destination-driven architecture has the following advantages.

- In a source-driven architecture, the source has to be active and must handle error conditions such as not being able to contact the warehouse for some time. It is easier to implement passive sources, and a single active warehouse. In a destination-driven architecure, each source is required to provide only a basic functionality of executing queries.
- The warehouse has more control on when to carry out data gathering activities, and when to process user queries; it is not a good

idea to perform both simultaneously, since they may conflict on locks.

20.2 Why is column-oriented storage potentially advantageous in a database system that supports a data warehouse?
Answer: No Answer

20.3 Suppose that there are two classification rules, one that says that people with salaries between $10,000 and $20,000 have a credit rating of *good*, and another that says that people with salaries between $20,000 and $30,000 have a credit rating of *good*. Under what conditions can the rules be replaced, without any loss of information, by a single rule that says people with salaries between $10,000 and $30,000 have a credit rating of *good*?
Answer: Consider the following pair of rules and their confidence levels :

No.	Rule	Conf.
1.	$\forall$ *persons* P, $10000 < P.salary \leq 20000 \Rightarrow P.credit = good$	60%
2.	$\forall$ *persons* P, $20000 < P.salary \leq 30000 \Rightarrow P.credit = good$	90%

The new rule has to be assigned a confidence-level which is between the confidence-levels for rules 1 and 2. Replacing the original rules by the new rule will result in a loss of confidence-level information for classifying persons, since we cannot distinguish the confidence levels of perople earning between 10000 and 20000 from those of people earning between 20000 and 30000. Therefore we can combine the two rules without loss of information only if their confidences are the same.

20.4 Consider the schema depicted in Figure 20.2. Give an SQL query to summarize sales numbers and price by store and date, along with the hierarchies on store and date.
Answer: query:

```
select store-id, city, state, country,
       date, month, quarter, year,
       sum(number), sum(price)
from sales, store, date
where sales.store-id = store.store-id and
       sales.date = date.date
groupby rollup(country, state, city, store-id),
       rollup(year, quarter, month, date)
```

20.5 Consider a classification problem where the classifier predicts whether a person has a particular disease. Suppose that 95% of the people tested do not suffer from the disease. (That is, *pos* corresponds to 5% and *neg* to 95% of the test cases.) Consider the following classifiers:

- Classifier C_1 which always predicts negative (a rather useless classifier of course).
- Classifier C_2 which predicts positive in 80% of the cases where the person actually has the disease, but also predicts positive in 5% of the cases where the person does not have the disease.
- Classifier C_3 which predicts positive in 95% of the cases where the person actually has the disease, but also predicts positive in 20% of the cases where the person does not have the disease.

Given the above classifiers, answer the following questions.

a. For each of the above classifiers, compute the accuracy, precision, recall and specificity.

b. If you intend to use the results of classification to perform further screening for the disease, how would you choose between the classifiers.

c. On the other hand, if you intend to use the result of classification to start medication, where the medication could have harmful effects if given to someone who does not have the disease, how would you choose between the classifiers?

Answer: No Answer

CHAPTER 20

Data Warehousing and Mining

This chapter covers data warehousing and data mining. Considering the growing importance of all the topics covered in this chapter, some of the sections of the chapter can be assigned as supplementary reading material, even in an introductory course. The material in the chapter is also suitable for laying the groundwork for an advanced course, or for professionals to keep in touch with recent developments.

Exercises

20.6 Draw a diagram that shows how the *classroom* relation of our university example as shown in Appendix A would be stored under a column-oriented storage structure.

Answer: The relation would be stored in three files, one per attribute, as shown below. We assume that the row number can be inferred implicitly from position, by using fixed size space for each attribute. Otherwise, the row number would also have to be stored explicitly.

building
Packard
Painter
Taylor
Watson
Watson

room_number
101
514
3128
100
120

capacity
500
10
70
30
50

20.7 Explain why the nested-loops join algorithm (see Section 12.5.1) would work poorly on database stored in a column-oriented manner. Describe an alternative algorithm that would work better and explain why your solution is better.

Answer: If the nested-loops join algorithm is used as is, it would require tuples for each of the relations to be assembled before they are joined. Assembling tuples can be expensive in a column store, since each attribute may come from a separate area of the disk; the overhead of assembly would be particularly wasteful if many tuples do not satisfy the join condition and would be discarded. In such a situation it would be better to first find which tuples match by accessing only the join columns of the relations. Sort-merge join, hash join, or indexed nested loops join can be used for this task. After the join is performed, only tuples that get output by the join need to be assembled; assembly can be done by sorting the join result on the record identifier of one of the relations and accessing the corresponding attributes, then resorting on record identifiers of the other relation to access its attributes.

20.8 Construct a decision-tree classifier with binary splits at each node, using tuples in relation $r(A, B, C)$ shown below as training data; attribute C denotes the class. Show the final tree, and with each node show the best split for each attribute along with its information gain value.

(1, 2, a), (2, 1, a), (2, 5, b), (3, 3, b), (3, 6, b),
(4, 5, b), (5, 5, c), (6, 3, b), (6, 7, c)

Answer: Figure 20.1 shows one possible decision tree for the data. Using the Gini purity metric, the purity of the initial data set is

$$1 - \sum_{i-1}^{k} p_i^2 = 1 - ((\frac{2}{9})^2 + (\frac{5}{9})^2 + (\frac{2}{9})^2) = 0.595259$$

The first branch splits on $B <= 2$, giving a purity score of $1 - 1^2 = 0$ for those attributes with $B <= 2$ (all are classified as a), and a purity score of

$$1 - ((\frac{2}{7})^2 + (\frac{5}{7})^2) = 0.40816$$

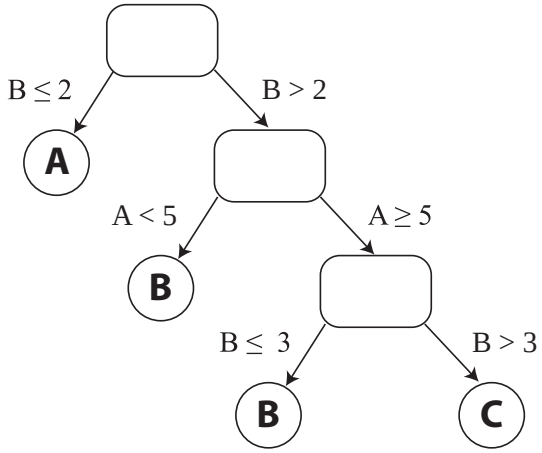


Figure 20.1 Decision tree for data on relation $r(A, B, C)$

for the remaining items. The weighted purity of the entire set is

$$\frac{2}{9} * 0 + \frac{7}{9} * 0.40816 = 0.31746$$

The information gain from this split is $0.595259 - 0.31746 = 0.27513$. Next, we split on $A < 5$. The 4 data items with $A < 5$ all have class b, and thus have purity 0. The remaining 3 items have purity

$$1 - ((\frac{1}{3})^2 + (\frac{2}{3})^2) = 0.44444$$

The weighted purity of these sets is

$$\frac{4}{7} * 0 + \frac{3}{7} * 0.44444 = 0.19048$$

The information gain from the second split is $0.40816 - 0.19048 = 0.21769$. Finally, we split on $B <= 3$. One data item satisfies this predicate and has class b. The other two items both have class c. The purity of these two sets is 0. The information gain from the final split is $0.21769 - 0 = 0.21769$.

20.9 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.
$\forall\, transactions\ T,\ \textbf{true} \Rightarrow buys(T, jeans)$	50%	50%
$\forall\, transactions\ T,\ \textbf{true} \Rightarrow buys(T, t\text{-}shirts)$	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%

20.10 Consider the problem of finding large itemsets.

a. Describe how to find the support for a given collection of itemsets by using a single scan of the data. Assume that the itemsets and associated information, such as counts, will fit in memory.

b. Suppose an itemset has support less than j. Show that no superset of this itemset can have support greater than or equal to j.

Answer:

a. Let $\{S_1, S_2, \ldots, S_n\}$ be the collection of item-sets for which we want to find the support. Associate a counter $count(S_i)$ with each item-set S_i.
Initialize each counter to zero. Now examine the transactions one-by-one. Let $S(T)$ be the item-set for a transaction T. For each item-set S_i that is a subset of $S(T)$, increment the corresponding counter $count(S_i)$.
When all the transactions have been scanned, the values of $count(S_i)$ for each i will give the support for item-set S_i.

b. Let A be an item-set. Consider any item-set B which is a superset of A. Let τ_A and τ_B be the sets of transactions that purchase all items in A and all items in B, respectively. For example, suppose A is $\{a, b, c\}$, and B is $\{a, b, c, d\}$.
A transaction that purchases all items from B must also have purchased all items from A (since $A \subseteq B$). Thus, every transaction in τ_B is also in τ_A. This implies that the number of transactions in τ_B is at most the number of transactions in τ_A. In other words, the support for B is at most the support for A.
Thus, if any item-set has support less than j, all supersets of this item-set have support less than j.

20.11 Create a small example of a set of transactions showing that although many transactions contain two items, that is, the itemset containing the two items has a high support, purchase of one of the items may have a negative correlation with purchase of the other.

Answer: The following set of transactions involve fruit purchases:

Transaction_ID	Item
T-1	orange
T-1	banana
T-1	apple
T-2	orange
T-2	banana
T-3	orange
T-3	apple
T-4	orange
T-4	banana
T-4	grapes
T-5	banana
T-5	apple
T-6	banana
T-6	grapes

Consider the association rule

$$orange \Rightarrow banana$$

This rule is satisfied in 3 out of the 6 transactions, so the support value is 50 percent. However, the correlation between purchasing oranges and purchasing bananas in this data set is -0.32.

20.12 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.

20.13 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 21

Information Retrieval

Practice Exercises

21.1 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the Practice Exercises in this chapter to the query "SQL relation".

Answer: We do not consider the questions containing neither of the keywords as their relevance to the keywords is zero. The number of words in a question include stop words. We use the equations given in Section 21.2 to compute relevance; the log term in the equation is assumed to be to the base 2.

Q#	#wo--rds	# "SQL"	#"rela--tion"	"SQL" term freq.	"relation" term freq.	"SQL" relv.	"relation" relv.	Tota relv.
1	84	1	1	0.0170	0.0170	0.0002	0.0002	0.0004
4	22	0	1	0.0000	0.0641	0.0000	0.0029	0.0029
5	46	1	1	0.0310	0.0310	0.0006	0.0006	0.0013
6	22	1	0	0.0641	0.0000	0.0029	0.0000	0.0029
7	33	1	1	0.0430	0.0430	0.0013	0.0013	0.0026
8	32	1	3	0.0443	0.1292	0.0013	0.0040	0.0054
9	77	0	1	0.0000	0.0186	0.0000	0.0002	0.0002
14	30	1	0	0.0473	0.0000	0.0015	0.0000	0.0015
15	26	1	1	0.0544	0.0544	0.0020	0.0020	0.0041

21.2 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

Answer: Let S be a set of n keywords. An algorithm to find all documents that contain at least k of these keywords is given below :

This algorithm calculates a reference count for each document identifier. A reference count of i for a document identifier d means that at least i of the keywords in S occur in the document identified by d. The algorithm maintains a list of records, each having two fields – a document identifier, and the reference count for this identifier. This list is maintained sorted on the document identifier field.

```
initialize the list L to the empty list;
for (each keyword c in S) do
begin
    D := the list of documents identifiers corresponding to c;
    for (each document identifier d in D) do
        if (a record R with document identifier as d is on list L) then
                R.reference_count := R.reference_count + 1;
        else begin
                make a new record R;
                R.document_id := d;
                R.reference_count := 1;
                add R to L;
        end;
end;
for (each record R in L) do
    if (R.reference_count >= k) then
        output R;
```

Note that execution of the second *for* statement causes the list D to "merge" with the list L. Since the lists L and D are sorted, the time taken for this merge is proportional to the sum of the lengths of the two lists. Thus the algorithm runs in time (at most) proportional to n times the sum total of the number of document identifiers corresponding to each keyword in S.

21.3 Suggest how to implement the iterative technique for computing PageRank given that the T matrix (even in adjacency list representation) does not fit in memory.
Answer: No answer

21.4 Suggest how a document containing a word (such as "leopard") can be indexed such that it is efficiently retrieved by queries using a more general concept (such as "carnivore" or "mammal"). You can assume that the concept hierarchy is not very deep, so each concept has only a few generalizations (a concept can, however, have a large number of specializations). You can also assume that you are provided with a function that returns the concept for each word in a document. Also suggest how a query using a specialized concept can retrieve documents using a more general concept.
Answer: Add doc to index lists for more general concepts also.

21.5 Suppose inverted lists are maintained in blocks, with each block noting the largest popularity rank and TF-IDF scores of documents in the remaining blocks in the list. Suggest how merging of inverted lists can stop early if the user wants only the top K answers.

Answer: For all documents whose scores are not complete use upper bounds to compute best possible score. If K th largest completed score is greater than the largest upper bound among incomplete scores output the K top answers.No answer

CHAPTER 21

Information Retrieval

This chapter covers advanced querying techniques for databases and information retrieval. Advanced querying techniques include decision support systems, online analytical processing, including SQL:1999 support for OLAP, and data mining.

Although information retrieval has been considered as a separate field from databases in the research community, there are strong connections. Distributed information retrieval is growing in importance with the explosion of documents on the world wide web and the resultant importance of web search techniques.

Considering the growing importance of all the topics covered in this chapter, some of the sections of the chapter can be assigned as supplementary reading material, even in an introductory course. These could include OLAP, some parts of data mining, and some parts of information retrieval. The material in the chapter is also suitable for laying the groundwork for an advanced course, or for professionals to keep in touch with recent developments.

Exercises

21.6 Using a simple definition of term frequency as the number of occurrences of the term in a document, give the TF-IDF scores of each term in the set of documents consisting of this and the next exercise.

Answer:

Term frequency TF(d, t) = log(1 + n(d, t)/n(d))
where n(d, t) denotes the number of occurences of term t in the document d and n(d) denotes the number of terms in the document.

using - log(1 + 1/75)
a - log(1 + 5/75)
simple - log(1 + 1/75)
definition - log(1 + 1/75)
of - log(1 + 6/75)
term - log(1 + 3/75)
frequency - log(1 + 1/75)

as - log(1 + 1/75)
the - log(1 + 1/75)
number - log(1 + 7/75)
occurences - log(1 + 1/75)
in - log(1 + 2/75)
document - log(1 + 1/75)
give - log(1 + 1/75)
TFIDF - log(1 + 1/75)
scores - log(1 + 1/75)
each - log(1 + 3/75)
set - log(1 + 1/75)
documents - log(1 + 3/75)
consisting - log(1 + 1/75)
this - log(1 + 1/75)
and - log(1 + 2/75)
next - log(1 + 1/75)
exercise - log(1 + 1/75)
create - log(1 + 1/75)
small - log(1 + 2/75)
example - log(1 + 1/75)
4 - log(1 + 1/75)
with - log(1 + 1/75)
PageRank - log(1 + 3/75)
inverted - log(1 + 1/75)
lists - log(1 + 1/75)
sorted - log(1 + 1/75)
by - log(1 + 1/75)
you - log(1 + 1/75)
do - log(1 + 1/75)
not - log(1 + 1/75)
need - log(1 + 1/75)
to - log(1 + 1/75)
compute - log(1 + 1/75)
just - log(1 + 1/75)
assume - log(1 + 1/75)
some - log(1 + 1/75)
values - log(1 + 1/75)
page - log(1 + 1/75)

21.7 Create a small example of four small documents, each with a PageRank, and create inverted lists for the documents sorted by the PageRank. You do not need to compute PageRank, just assume some values for each page.

Answer: Given 4 documents - A, B, C, D where the PageRanks are decreasing in that order, which means A has the highest PageRank and D has the lowest PageRank. We have, pages that are pointed to from more

web pages have higher PageRank. Similarly, pages pointed to by Web pages with a high PageRank will also have a higher PageRank.
One way of creating an inverted list is:

a. B, C, D all point to A. $A <- B$, $A <- C$, $A <- D$.

b. A points to B. $B <- A$.

c. B points to C. $C <- B$.

d. C points to D, $D <- C$.

21.8 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 occurences 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 of term frequency may not make sense. The relevance of a term may not depend on the number of occurences 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 not also be very relavant in this case. For example, the primary key value and some other key may be having the inverse document frequency of 1, but we cant assume their weights to be equal.
The similarity of the two tuples can be measured by the *cosinesimilarity* metric. But one major difference is 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 doesnt increase the similarity factor.

21.9 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 tobe visited, and thus will have a higer PageRank. Similarly, pages pointed 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 to each page increases, which only increases the PageRank.

21.10 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 adverstisements that are similar to A. One appraoch 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 metric $'cosinesimilarity'$ can also be used to determine which advertisements to supply for a page, given the page contents.

21.11 One way to create a keyword-specific version of PageRank is to modify the random jump such that a jump is only possible to pages containing the keyword. Thus pages that do not contain the keyword but are close (in terms of links) to pages that contain the keyword also get a nonzero rank for that keyword.

a. Give equations defining such a keyword-specific version of PageRank.

b. Give a formula for computing the relevance of a page to a query containing multiple keywords.

Answer:

a. Give equations defining such a keyword-specific version of PageRank.

$$P[j] = \delta/N_i + (1-\delta) * \sum_{i=1}^{N} (T[i, j] * P]i])$$

where δ is a constant between 0 and 1, N is the number of pages, N_i is the number of pages containing the keyword; δ represents the probability of a step in the walk being a jump to a page containing the keyword.

b. Give a formula for computing the relevance of a page to a query containing multiple keywords.
The relevance of a document of a document to a query comtaining multiple keywords is estimated by combining by adding the relevance measures of the page to each keyword. Some weights also might be considered.

$$r(d, Q) = \sum_{t=1}^{n} TF(d, t) * IDF(t)$$

where Q is the set of keywords of size n, TF is the $termfrequency$ and IDf is the $inversedocumentfrequency$. This measure can be further

refined if the user is permitted to specify weights $w(t)$ for terms in the query, in which case the user specified weights are also taken into account by multiplying $TF(d, t)$ by $w(t)$ in the above formula.

21.12 The idea of popularity ranking using hyperlinks can be extended to relational and XML data, using foreign key and IDREF edges in place of hyperlinks. Suggest how such a ranking scheme may be of value in the following applications:

a. A bibliographic database that has links from articles to authors of the articles and links from each article to every article that it references.

b. A sales database that has links from each sales record to the items that were sold.

Also suggest why prestige ranking can give less than meaningful results in a movie database that records which actor has acted in which movies.

Answer:

a. A bibliographic database, which has links from articles to authors of the articles and links from each article to every article that it references.
This helps us in ranking articles according to their popularity. If an article is referenced by many articles, then its more popular, so each article has a rank associated with it. and we could also find the authors for those articles.

b. A sales database which has links from each sales record to the items that were sold.
This helps us in determining wich items are the most popular. If the item is referenced by many sales records, then its more popular. So, ranking the database helps in determining the popularity of the items that were sold.

A movie which has many actors associated with it is deemed to be more popular when prestige ranking is taken into account. Same goes with the actor as well. But that may not be true in real life, the popularity of a movie or that of an actor cannot be determined by ranking the movie lists which map the actors to the movies.

21.13 What is the difference between a false positive and a false drop? If it is essential that no relevant information be missed by an information retrieval query, is it acceptable to have either false positives or false drops? Why?

Answer: False drop - A few relevant documents may not be retrieved. False positive - A few irrelevant documents may be retrieved. It is acceptable to have false positives but not any false drops when it is essential that no relevant information is to missed because by permitting false positive; the system can later filter the results away later by looking at the keywords than they actually contain, but by permitting false drops, some

relvant information is missed out. By allowing false positives and not allowing false drops, no relevant information is missed out.

CHAPTER 22

Object-Based Databases

Practice Exercises

22.1 A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity.
- Sports cars: horsepower, renter age requirement.
- Vans: number of passengers.
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive).

Construct an SQL schema definition for this database. Use inheritance where appropriate.

Answer: For this problem, we use table inheritance. We assume that **MyDate**, **Color** and **DriveTrainType** are pre-defined types.

create type *Vehicle*
(*vehicle_id* **integer**,
license_number **char(15)**,
manufacturer **char(30)**,
model **char(30)**,
purchase_date **MyDate**,
color **Color**)

create table *vehicle* **of type** *Vehicle*

create table *truck*
(*cargo_capacity* **integer**)
under *vehicle*

create table *sportsCar*

```
    (horsepower integer
     renter_age_requirement integer)
    under vehicle

create table van
    (num_passengers integer)
    under vehicle

create table offRoadVehicle
    (ground_clearance real
     driveTrain DriveTrainType)
    under vehicle
```

22.2 Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

Emp = (ename, ChildrenSet **multiset***(Children), SkillSet* **multiset***(Skills))*
Children = (name, birthday)
Skills = (type, ExamSet **setof***(Exams))*
Exams = (year, city)

a. Define the above schema in SQL, with appropriate types for each attribute.

b. Using the above schema, write the following queries in SQL.

   i. Find the names of all employees who have a child born on or after January 1, 2000.

   ii. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

   iii. List all skill types in the relation *Emp*.

Answer:

a. No Answer.

b. Queries in SQL.

   i. Program:

```
select ename
from emp as e, e.ChildrenSet as c
where 'March' in
        (select birthday.month
         from c
        )
```

   ii. Program:

```
select e.ename
from emp as e, e.SkillSet as s, s.ExamSet as x
where s.type = 'typing' and x.city = 'Dayton'
```

iii. Program:

```
select distinct s.type
from emp as e, e.SkillSet as s
```

22.3 Consider the E-R diagram in Figure 22.5, which contains composite, multivalued, and derived attributes.

a. Give an SQL schema definition corresponding to the E-R diagram.

b. Give constructors for each of the structured types defined above.

Answer:

a. The corresponding SQL:1999 schema definition is given below. Note that the derived attribute *age* has been translated into a method.

```
create type Name
    (first_name varchar(15),
     middle_initial char,
     last_name varchar(15))
create type Street
    (street_name varchar(15),
     street_number varchar(4),
     apartment_number varchar(7))
create type Address
    (street Street,
     city varchar(15),
     state varchar(15),
     zip_code char(6))
create table customer
    (name Name,
     customer_id varchar(10),
     address Adress,
     phones  char(7)  array[10],
     dob date)
method integer age()
```

b.
```
create function Name (f varchar(15), m char, l varchar(15))
returns Name
begin
    set first_name = f;
    set middle_initial = m;
    set last_name = l;
end
create function Street (sname varchar(15), sno varchar(4), ano varchar(7))
```

```
returns Street
begin
    set street_name = sname;
    set street_number = sno;
    set apartment_number =ano;
end
create function Address (s Street, c varchar(15), sta varchar(15), zip varchar(6))
returns Address
begin
    set street = s;
    set city = c;
    set state =sta;
    set zip_code =zip;
end
```

22.4 Consider the relational schema shown in Figure 22.6.

a. Give a schema definition in SQLcorresponding to the relational schema, but using references to express foreign-key relationships.

b. Write each of the queries given in Exercise 6.13 on the above schema, using SQL.

Answer:

a. The schema definition is given below. Note that backward references can be addedbut they are not so important as in OODBS because queries can be written in SQL and joins can take care of integrity constraints.

```
create type Employee
    (person_name varchar(30),
     street varchar(15),
     city varchar(15))
create type Company
    (company_name varchar(15),
    (city varchar(15))
create table employee of Employee
create table company of Company
create type Works
    (person ref(Employee) scope employee,
    comp ref(Company) scope company,
    salary int)
create table works of Works
create type Manages
    (person ref(Employee) scope employee,
    (manager ref(Employee) scope employee)
create table manages of Manages
```

b. i. **select** *comp*− >*name*

from *works*
group by *comp*
having count(*person*) $\geq$ **all(select count(**$person$**)**
from *works*
group by *comp*)

ii. **select** $comp->name$
from *works*
group by *comp*
having sum(*salary*) $\leq$ **all(select sum(**$salary$**)**
from *works*
group by *comp*)

iii. **select** $comp->name$
from *works*
group by *comp*
having avg(*salary*) $>$ **(select avg(**$salary$**)**
from *works*
where $comp->company\_name$="First Bank Corporation")

22.5 Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language–based OODB, object relational; do not specify a commercial product) you would recommend. Justify your recommendation.

a. A computer-aided design system for a manufacturer of airplanes.

b. A system to track contributions made to candidates for public office.

c. An information system to support the making of movies.

Answer:

a. A computer-aided design system for a manufacturer of airplanes:
An OODB system would be suitable for this. That is because CAD requires complex data types, and being computation oriented, CAD tools are typically used in a programming language environment needing to access the database.

b. A system to track contributions made to candidates for public office:
A relational system would be apt for this, as data types are expected to be simple, and a powerful querying mechanism is essential.

c. An information system to support the making of movies:
Here there will be extensive use of multimedia and other complex data types. But queries are probably simple, and thus an object relational system is suitable.

22.6 How does the concept of an object in the object-oriented model differ from the concept of an entity in the entity-relationship model?

Answer: An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it, and these messages are implemented by the methods.

CHAPTER 22

Object-Based Databases

This chapter describes extensions to relational database systems to provide complex data types and object-oriented features. Such extended systems are called object-relational systems. Since the chapter was introduced in the 3^{th} edition most commercial database systems have added some support for object-relational features, and these features have been standardized as part of SQL:1999.

It would be instructive to assign students exercises aimed at finding applications where the object-relational model, in particular complex objects, would be better suited than the traditional relational model.

Exercises

22.7 Redesign the database of Practice Exercise 22.2 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.

Employee-details = (*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:-

$$\begin{aligned} ename, cname &\rightarrow bday, bmonth, byear \\ ename &\twoheadrightarrow cname, bday, bmonth, byear \\ ename, stype &\twoheadrightarrow xyear, xcity \end{aligned}$$

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:-

Employee = (*ename*)
Child = (*ename, cname, bday, bmonth, byear*)
Skill = (*ename, stype, xyear, xcity*)

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.

22.8 Consider the schema from Practice Exercise 22.2.

a. Give SQL DDL statements to create a relation *EmpA* which has the same information as *Emp*, but where multiset-valued attributes *ChildrenSet*, *SkillsSet* and *ExamsSet* are replaced by array-valued attributes *ChildrenArray*, *SkillsArray* and *ExamsArray*.

b. Write a query to convert data from the schema of *Emp* to that of *EmpA*, with the array of children sorted by birthday, the array of skills by the skill type and the array of exams by the year.

c. Write an SQL statement to update the *Emp* relation by adding a child Jeb, with a birthdate of February 5, 2001, to the employee named George.

d. Write an SQL statement to perform the same update as above but on the *EmpA* relation. Make sure that the array of children remains sorted by year.

Answer:

a.

```
create type Exams (year int, city varchar(30))
create type SkillsA (type varchar(30),
    ExamsArray Exams array [20])
create type Children (name varchar(30), birthday date)
create table EmpA (ename varchar(30),
    ChildrenArray Children array [10],
    SkillArray SkillsA array [25])
```

b.

```
select ename,
    array(select name, birthday
          from unnest(E.ChildrenSet) as CS
          order by CS.birthday) as ChildrenArray,
    array(select type,
         array(select year, city
```

```
                    from unnest(SS.ExamSet)
                    order by SS.year) as ExamsArray
               from unnest(E.SkillSet) as SS)
          as SkillsArray
from Emp as E
```

c.

```
update Emp
set ChildrenSet = ChildrenSet union
    multiset[(''Jeb'', ''2/5/2001'')]
where ename = ''George''
```

d. We make use of the infix array concatenation operator, "||", which takes two arrays as arguments and returns the concatenation of the two arrays.

```
update EmpA
set ChildrenArray = array(
    select name, birthday
    from unnest(ChildrenArray ||
        array[(''Jeb'', ''2/5/2001'')])
    order by birthday)
where ename = ''George''
```

22.9 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 22.4. 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.

22.10 Explain the distinction between a type x and a reference type **ref**(x). Under what circumstances would you choose to use a reference type?

Answer: If the type of an attribute is x, then in each tuple of the table, corresponding to that attribute, there is an actual object of type x . If its type is **ref**(x), then in each tuple, corresponding to that attribute, there is a *reference* to some object of type x. We choose a reference type for an attribute, if that attribute's intended purpose is to refer to an independent object.

22.11 Consider the E-R diagram in Figure 22.7, 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.
```
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*

b.

select *
from *person*
where *ID* **not in**
(**select** *ID* **from** *secretary*)

c. Give an SQL query to print the names of people who are neither employees nor students.

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.

22.12 Suppose a JDO database had an object *A*, which references object *B*, which in turn references object *C*. Assume all objects are on disk initially. Suppose a program first dereferences *A*, then dereferences *B* by following the reference from *A*, and then finally dereferences *C*. Show the objects that are represented in memory after each dereference, along with their state (hollow or filled, and values in their reference fields).

Answer: See figures 22.1 through 22.3. Gray boxes indicate persistent objects and white boxes indicate hollow objects.

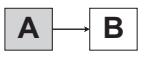


Figure 22.1 State of the program after A is referenced.

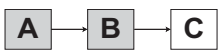


Figure 22.2 State of the program after B is referenced.

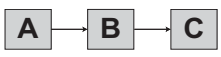


Figure 22.3 State of the program after C is referenced.

CHAPTER 23

XML

Practice Exercises

23.1 Give an alternative representation of university information containing the same data as in Figure 23.1, but using attributes instead of subelements. Also give the DTD or XML Schema for this representation.
Answer:

a. The XML representation of data using attributes is shown in Figure 23.100.

b. The DTD for the bank is shown in Figure 23.101.

```
<university>
    <department dept_name="Comp. Sci." building="Taylor"
            budget="100000">
    </department>
    <department dept_name="Biology" building="Watson"
            budget="90000">
    </department>
    <course course_id="CS-101" title="Intro. to Computer Science"
            dept_name="Comp. Sci." credits="4">
    </course>
    <course course_id="BIO-301" title="Genetics"
            dept_name="Biology." credits="4">
    </course>
    <instructor IID="10101" name="Srinivasan"
            dept_name="Comp. Sci." salary="65000">
    </instructor>
    <instructor IID="83821" name="Brandt"
            dept_name="Comp. Sci" salary="92000">
    </instructor>
    <instructor IID="76766" name="Crick"
            dept_name="Biology" salary="72000">
    </instructor>
    <teaches IID="10101" course_id="CS-101">
    </teaches>
    <teaches IID="83821" course_id="CS-101">
    </teaches>
    <teaches IID="76766" course_id="BIO-301">
    </teaches>
</university>
```

Figure 23.100 XML representation.

23.2 Give the DTD or XML Schema for an XML representation of the following nested-relational schema:

Emp = (ename, ChildrenSet **setof***(Children), SkillsSet* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, ExamsSet **setof***(Exams))*
Exams = (year, city)

Answer: Query:

```
<!DOCTYPE db [
        <!ELEMENT emp (ename, children*, skills*)>
        <!ELEMENT children (name, birthday)>
        <!ELEMENT birthday (day, month, year)>
        <!ELEMENT skills (type, exams+)>
        <!ELEMENT exams (year, city)>
        <!ELEMENT ename( #PCDATA )>
        <!ELEMENT name( #PCDATA )>
        <!ELEMENT day( #PCDATA )>
        <!ELEMENT month( #PCDATA )>
        <!ELEMENT year( #PCDATA )>
        <!ELEMENT type( #PCDATA )>
        <!ELEMENT city( #PCDATA )>
] >
```

```
<!DOCTYPE university [
        <!ELEMENT department >
        <!ATTLIST department
                dept_name ID #REQUIRED
                building CDATA #REQUIRED
                budget CDATA #REQUIRED >
        <!ELEMENT instructor >
        <!ATTLIST instructor
                IID ID #REQUIRED
                name CDATA #REQUIRED
                dept_name IDREF #REQUIRED >
                salary CDATA #REQUIRED >
        <!ELEMENT course >
        <!ATTLIST course
                course_id ID #REQUIRED
                title CDATA #REQUIRED
                dept_name IDREF #REQUIRED >
                credits CDATA #REQUIRED >
        <!ELEMENT teaches >
        <!ATTLIST teaches
                IID IDREF #REQUIRED >
                course_id IDREF #REQUIRED
] >
```

Figure 23.101 The DTD for the university.

23.3 Write a query in XPath on the schema of Practice Exercise 23.2 to list all skill types in *Emp*.
Answer: Code:

```
/db/emp/skills/type
```

23.4 Write a query in XQuery on the XML representation in Figure 23.11 to find the total salary of all instructors in each department.
Answer: Query:

```
for $b in distinct (/university/department/dept_name)
return
<dept-total>
    <dept_name> $b/text() </dept_name>
    let $s := sum (/university/instructor[dept_name=$b]/salary)
    return <total-salary> $s </total-salary>
</dept-total>
```

23.5 Write a query in XQuery on the XML representation in Figure 23.1 to compute the left outer join of department elements with course elements. (Hint: Use universal quantification.)
Answer: Query:

```
<lojoin>
for $d in /university/department,
    $c in /university/course
where $c/dept_name = $d/dept_name
return <dept-course> $d $c </dept-course>
|
for $d in /university/department,
where every $c in /university/course satisfies
(not ($c/dept_name =$d/dept_name))
return <dept-course > $c </dept-course >
</lojoin>
```

23.6 Write queries in XQuery to output course elements with associated instructor elements nested within the course elements, given the university information representation using ID and IDREFS in Figure 23.11.

Answer: The answer in XQuery is

```
<university-2>
    for $c in /university/course
    return
        <course>
            <course_id> $c/* </course_id>
            for $a in $c/id(@instructors)
            return $a
        </course>
</university-2>
```

23.7 Give a relational schema to represent bibliographical information specified according to the DTD fragment in Figure 23.16. The relational schema must keep track of the order of author elements. You can assume that only books and articles appear as top-level elements in XML documents.

Answer: Relation schema:

book (bid, title, year, publisher, place)
article (artid, title, journal, year, number, volume, pages)
book_author (bid, first_name,last_name, order)
article_author (artid, first_name,last_name, order)

23.8 Show the tree representation of the XML data in Figure 23.1, and the representation of the tree using *nodes* and *child* relations described in Section 23.6.2.

Answer: The answer is shown in Figure 23.102.

nodes(1,element,university,–)
nodes(2,element,department,–)
nodes(3,element,department,–)
nodes(4,element,course,–)
nodes(5,element,course,–)
nodes(6,element,instructor,–)
nodes(7,element,instructor,–)
nodes(8,element,instructor,–)
nodes(9,element,teaches,–)
nodes(10,element,teaches,–)
nodes(11,element,teaches,–)
child(2,1) child(3,1) child(4,1)
child(5,1) child(6,1)
child(7,1) child(8,1) child(9,1)

Continued in Figure 23.103

Figure 23.102 Relational Representation of XML Data as Trees.

23.9 Consider the following recursive DTD:

```
<!DOCTYPE parts [
    <!ELEMENT part (name, subpartinfo*)>
    <!ELEMENT subpartinfo (part, quantity)>
    <!ELEMENT name ( #PCDATA )>
    <!ELEMENT quantity ( #PCDATA )>
] >
```

a. Give a small example of data corresponding to this DTD.

b. Show how to map this DTD to a relational schema. You can assume that part names are unique; that is, wherever a part appears, its subpart structure will be the same.

c. Create a schema in XML Schema corresponding to this DTD.

Answer:

a. The answer is shown in Figure 23.104.

b. Show how to map this DTD to a relational schema.

```
part(partid,name)
subpartinfo(partid, subpartid, qty)
```

Attributes partid and subpartid of subpartinfo are foreign keys to part.

c. The XML Schema for the DTD is as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="parts" type="partsType" />
<xs:complexType name="partType">
    <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="subpartinfo" type="subpartinfoType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
<xs:complexType name="subpartinfoType"/>
    <xs:sequence>
        <xs:element name="part" type="partType"/>
        <xs:element name="quantity" type="xs:string"/>
    </xs:sequence>
</xs:schema>
```

child(10,1) child(11,1)
nodes(12,element,dept_name,Comp. Sci.)
nodes(13,element,building,Taylor)
nodes(14,element,budget,100000)
child(12,2) child(13,2) child(14,2)
nodes(15,element,dept_name,Biology)
nodes(16,element,building,Watson)
nodes(17,element,budget,90000)
child(15,3) child(16,3) child(17,3)
nodes(18,element,course_id,CS-101)
nodes(19,element,title,Intro. to Computer Science)
nodes(20,element,dept_name,Comp. Sci.)
nodes(21,element,credits,4)
child(18,4) child(19,4) child(20,4)child(21,4)
nodes(22,element,course_id,BIO-301)
nodes(23,element,title,Genetics)
nodes(24,element,dept_name,Biology)
nodes(25,element,credits,4)
child(22,5) child(23,5) child(24,5)child(25,5)
nodes(26,element,IID,10101)
nodes(27,element,name,Srinivasan)
nodes(28,element,dept_name,Comp. Sci.)
nodes(29,element,salary,65000)
child(26,6) child(27,6) child(28,6)child(29,6)
nodes(30,element,IID,83821)
nodes(31,element,name,Brandt)
nodes(32,element,dept_name,Comp. Sci.)
nodes(33,element,salary,92000)
child(30,7 child(31,7) child(32,7)child(33,7)
nodes(34,element,IID,76766)
nodes(35,element,dept_name,Biology)
nodes(36,element,salary,72000)
child(34,8) child(35,8) child(36,8)
nodes(37,element,IID,10101)
nodes(38,element,course_id,CS-101)
child(37,9) child(38,9)
nodes(39,element,IID,83821)
nodes(40,element,course_id,CS-101)
child(39,10) child(40,10)
nodes(41,element,IID,76766)
nodes(42,element,course_id,BIO-301)
child(41,11) child(42,11)

Figure 23.103 Continuation of Figure 23.102.

```
<parts>
    <part>
        <name> bicycle </name>
        <subpartinfo>
            <part>
                <name> wheel </name>
                <subpartinfo>
                    <part>
                        <name> rim </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> spokes </name>
                    </part>
                    <qty> 40 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> tire </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> brake </name>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> gear </name>
            </part>
            <qty> 3 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> frame </name>
            </part>
            <qty> 1 </qty>
        </subpartinfo>
    </part>
</parts>
```

Figure 23.104 Example Parts Data in XML.

CHAPTER 23

XML

XML is today widely used in the exchange of data between applications, and for storing data, whether simple or complex, in flat files. All the recent standard formats for storing documents, spreadsheets, presentations, and so on are based on XML, cementing the success of XML. In the context of databases, XML has been quite successful, although not with anything like the success of XML outside of databases; relational databases continue to be used for most mission critical data. However, support for storing XML in databases has improved significantly in the last 5 years.

Our view of XML is decidedly database centric. In this view, XML is a data model that provides a number of features beyond that provided by the relational model, in particular the ability to package related information into a single unit, by using nested structures. Specific application domains for data representation and interchange need their own standards that define the data schema.

Given the extensive nature of XML and related standards, this chapter only attempts to provide an introduction, and does not attempt to provide a complete description. For a course that intends to explore XML in detail, supplementary material may be required. These could include online information on XML and books on XML.

Exercises

23.10 Show, by giving a DTD, how to represent the non-1NF *books* relation from Section 22.2, using XML.

Answer:

```
<!DOCTYPE bib [
        <!ELEMENT book (title, author+, publisher, keyword+)>
        <!ELEMENT publisher (pub-name, pub-branch) >
        <!ELEMENT title ( #PCDATA )>
        <!ELEMENT author ( #PCDATA )>
        <!ELEMENT keyword ( #PCDATA )>
        <!ELEMENT pub-name( #PCDATA )>
        <!ELEMENT pub-branch( #PCDATA )>
] >
```

23.11 Write the following queries in XQuery, assuming the schema from Practice Exercise 23.2.

a. Find the names of all employees who have a child who has a birthday in March.

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

c. List all skill types in *Emp*.

Answer:

a. Find the names of all employees who have a child who has a birthday in March.

```
for $e in /db/emp,
    $m in distinct-values($e/children/birthday/month)
where $m = 'March'
return $e/ename
```

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

```
for $e in /db/emp
    $s in $e/skills[type='typing']
    $exam in $s/exams
where $exam/city= 'Dayton'
return $e/ename
```

c. List all skill types in *Emp*.

```
for $t in distinct-values (/db/emp/skills/type)
return $t
```

23.12 Consider the XML data shown in Figure 23.3. Suppose we wish to find purchase orders that ordered two or more copies of the part with identifier 123. Consider the following attempt to solve this problem:

```
for $p in purchaseorder
where $p/part/id = 123 and $p/part/quantity >= 2
return $p
```

Explain why the query may return some purchase orders that order less than two copies of part 123. Give a correct version of the above query.

Answer: Reason:
The expression x =y evaluates to true if any of the values returned by the first expression is equal to any one of the values returned by the second expression. In this case, if any of the values in the part/quantity >= 2 then it evaluates to true, so the query returns parts which have id = 123 irrespective of the quantity.

Query:

```
for $b in purchaseorder
where some $p in $b/part satisfies
      $p/id = 123 AND $p/quantity >= 2
return {$b}
```

23.13 Give a query in XQuery to flip the nesting of data from Exercise 23.10. That is, at the outermost level of nesting the output must have elements corresponding to authors, and each such element must have nested within it items corresponding to all the books written by the author.

Answer:

```
<bib>
    for $x in distinct-values(/books/author)
        return
            <author> <name> { $x } </name>
            { for $y in /books[author = $x/author],
                return <book>
                    <title> { $y/title } </title>
                    <publisher> { $y/publisher } </publisher>
                    <keyword> { $y/keyword } </keyword>
                </book>
            } </author>
</bib>
```

23.14 Give the DTD for an XML representation of the information in Figure 7.29. Create a separate element type to represent each relationship, but use ID and IDREF to implement primary and foreign keys.

Answer:

```
<!DOCTYPE bookstore [
    <!ELEMENT basket (contains+, basket-of)>
    <!ATTLIST basket
        basketid ID #REQUIRED >
    <!ELEMENT customer (name, address, phone)>
    <!ATTLIST customer
        email ID #REQUIRED >
    <!ELEMENT book (year, title, price, written-by, published-by)>
    <!ATTLIST book
        ISBN ID #REQUIRED >
    <!ELEMENT warehouse (address, phone, stocks)>
    <!ATTLIST warehouse
        code ID #REQUIRED >
    <!ELEMENT author (name, address, URL)>
    <!ATTLIST author
        authid ID #REQUIRED >
    <!ELEMENT publisher (address, phone, URL)>
    <!ATTLIST publisher
        name ID #REQUIRED >
    <!ELEMENT basket-of >
    <!ATTLIST basket-of
        owner IDREF #REQUIRED >
    <!ELEMENT contains >
    <!ATTLIST contains
        book IDREF #REQUIRED
        number CDATA #REQUIRED >
    <!ELEMENT stocks >
    <!ATTLIST stocks
        book IDREF #REQUIRED
        number CDATA #REQUIRED >
    <!ELEMENT written-by >
    <!ATTLIST written-by
        authors IDREFS #REQUIRED >
    <!ELEMENT published-by >
    <!ATTLIST published-by
        publisher IDREF #REQUIRED >
    <!ELEMENT name (#PCDATA )>
    <!ELEMENT address (#PCDATA )>
    <!ELEMENT phone (#PCDATA )>
```

```
    <!ELEMENT year (#PCDATA )>
    <!ELEMENT title (#PCDATA )>
    <!ELEMENT price (#PCDATA )>
    <!ELEMENT number (#PCDATA )>
    <!ELEMENT URL (#PCDATA )>
] >
```

23.15 Give an XML Schema representation of the DTD from Exercise 23.14.
Answer:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="basket" type="BasketType">
    <xs:element name="customer">
        <xs:complexType>
        <xs:attribute name="email" use="required">
        <xs:sequence>
            <xs:element name="name" type="xs:string">
            <xs:element name="address" type="xs:string">
            <xs:element name="phone" type="xs:decimal">
        </xs:sequence>
            </xs:complexType>
    </xs:element>
    <xs:key name="customerKey">
        <xs:selector xpath="/bookstore/customer">
        <xs:field xpath="email">
    </xs:key>
    <xs:element name="book">
        <xs:complexType>
            <xs:attribute name="ISBN" use="required">
            <xs:sequence>
            <xs:element name="year" type="xs:decimal">
            <xs:element name="title" type="xs:string">
            <xs:element name="price" type="xs:decimal">
            <xs:element name="written-by" type="xs:string">
            <xs:element name="published-by" type="xs:string">
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:key name="bookKey">
        <xs:selector xpath="/bookstore/book">
        <xs:field xpath="ISBN">
    </xs:key>
    <xs:element name="warehouse">
        <xs:complexType>
```

```
                <xs:attribute name="code" use="required">
                <xs:sequence>
                <xs:element name="address" type="xs:string">
                <xs:element name="phone" type="xs:decimal">
                <xs:element name="stocks" type="xs:decimal">
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:key name="warehouseKey">
        <xs:selector xpath="/bookstore/warehouse">
        <xs:field xpath="code">
</xs:key>
<xs:element name="author">
        <xs:complexType>
                <xs:attribute name="authid" use="required">
                <xs:sequence>
                <xs:element name="name" type="xs:string">
                <xs:element name="address" type="xs:string">
                <xs:element name="URL" type="xs:string">
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:key name="authorKey">
        <xs:selector xpath="/bookstore/author">
        <xs:field xpath="authid">
</xs:key>
<xs:element name="publisher">
        <xs:complexType>
                <xs:attribute name="name" use="required">
                <xs:sequence>
                <xs:element name="address" type="xs:string">
                <xs:element name="phone" type="xs:decimal">
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:key name="publisherKey">
        <xs:selector xpath="/bookstore/publisher">
        <xs:field xpath="name">
</xs:key>
<xs:element name="basket-of">
        <xs:attribute name="owner" use="required">
</xs:element>
<xs:keyref name="basketCustomerFKey" refer="customerKey">
        <xs:selector xpath="/bookstore/customer">
        <xs:field xpath="owner">
```

```
        </xs:key>
        <xs:element name="contains">
            <xs:attribute name="book" use="required">
            <xs:attribute name="number" use="required" type="xs:decimal"'>
        </xs:element>
        <xs:keyref name="bookBasketFKey" refer="bookKey">
            <xs:selector xpath="/bookstore/book">
            <xs:field xpath="book/">
        </xs:key>
        <xs:element name="stocks">
            <xs:attribute name="book" use="required">
            <xs:attribute name="number" use="required" type="xs:decimal"'>
        </xs:element>
        <xs:keyref name="bookwarehouseFKey" refer="bookKey">
            <xs:selector xpath="/bookstore/book">
            <xs:field xpath="book/">
        </xs:key>
        <xs:element name="written-by">
            <xs:attribute name="authors" use="required">
        </xs:element>
        <xs:keyref name="authorbookFKey" refer="authorKey">
            <xs:selector xpath="/bookstore/author">
            <xs:field xpath="authors">
        </xs:key>
        <xs:element name="published-by">
            <xs:attribute name="publisher" use="required">
        </xs:element>
        <xs:keyref name="bookpublisherFKey" refer="publisherKey">
            <xs:selector xpath="/bookstore/publisher">
            <xs:field xpath="publisher">
        </xs:key>
            <xs:complexType name="BasketType">
                <xs:attribute name="basketid" use="required">
                <xs:sequence>
                <xs:element name="contains" minOccurs="1"
                        maxOccurs="unbounded">
                <xs:element name="basket-of">
                </xs:sequence>
            </xs:complexType>
        <xs:key name="basketKey">
            <xs:selector xpath="/bookstore/basket">
            <xs:field xpath="basketid">
        </xs:key>
</xs:schema>
```

23.16 Write queries in XQuery on the bibliography DTD fragment in Figure 23.16, to do the following:

a. Find all authors who have authored a book and an article in the same year.

b. Display books and articles sorted by year.

c. Display books with more than one author.

d. Find all books that contain the word "database" in their title and the word "Hank" in an author's name (whether first or last).

Answer:

a. Find all authors who have authored a book and an article in the same year.

```
for $a in distinct-values (/bib/book/author),
      $y in /bib/book[author=$a]/year,
      $art in /bib/article[author=$a and year=$y]
return $a
```

b. Display books and articles sorted by year.

```
for $a in ((/bib/book) | (/bib/article))
order by $a/year
return $a
```

c. Display books with more than one author.

```
for $b in ((/bib/book[author/count()>1])
return $b
```

d. Find all books that contain the word "database" in their title and the word "Hank" in an author's name (whether first or last).

```
for $a in /bibliography/book/
where $b in $a satisfies
      (contains($b/title,"database") AND
      (contains($b/author(@last_name),"Hank")
            OR contains($b/author(@fiest_name),"Hank"))
return $a
```

23.17 Give a relational mapping of the XML purchase order schema illustrated in Figure 23.3, using the approach described in Section 23.6.2.3. Suggest how to remove redundancy in the relational schema, if item identifiers functionally determine the description and purchase and supplier names functionally determine the purchase and supplier address, respectively.

Answer: Removing redundancy:
Item ids functionally determine the desription The table 'item' can be broken down into 2 tables

item(item_id, quantity, price, itemlist_id)
and
item_info(item_id, description)

Then the item description doesnt get repeated whenever the item appears on the itemlist.
Purchaser_names functionally determine the address The table purchaser can be subdivided as shown below

purchaser(name, purcahseorder_id)
and
purchaser_info(name, address)

Supplier_names functionally determine the address The table supplier can be subdivided as shown below

supplier(name, purchaseorder_id)
and
supplier_info(name, address)

The address does not get repeated every time the supplier or purchaser is involved in a purchase.

```
create table purchase_order
        (purchaseorder_id char(20),
        purchaser_name char(20),
        supplier_name char(20),
        itemlist_id char(20),
        total_cost numeric(16,2),
        payment_terms char(20),
        shipping_mode char(20),
        primary key (identifier),
        foreign key (purchaser_name) references purchaser,
        foreign key (supplier_name) references supplier,
        foreign key (itemlist_id) references itemlist)
```

```
create table purchaser
        (purchaser_name char(20),
        address char(20),
        purchaseorder_id char(20),
        primary key (name),
        foreign key (purchaseorder_id) references purchase_order)


create table supplier
        (supplier_name char(20),
        address char(20),
        purchaseorder_id char(20),
        primary key (name),
        foreign key (purchaseorder_id) references purchase_order)


create table itemlist
        (itemlist_id char(20),
        item_id char(20),
        purchaseorder_id char(20),
        primary key (itemlist_id),
        foreign key (item_id) references item,
        foreign key (purchaseorder_id) references purchase_order)

create table item
        (item_id char(20),
        description char(20),
        quantity numeric(16),
        price numeric(16,2),
        itemlist_id char(20),
        primary key (item_id),
        foreign key (itemlist_id) references itemlist)
```

23.18 Write queries in SQL/XML to convert university data from the relational schema we have used in earlier chapters to the *university-1* and *university-2* XML schemas.

Answer: **university-1:**

```
select xmlelement(name "department",
        xmlelement(name "dept_name", d.dept_name),
        xmlelement(name "building" d.building),
        xmlelement(name "budget" d.budget),
        xmlelement(name "courses",
        (select xmlagg(xmlelement(name "course",
                xmlelement(name "course_id" c.course_id),
                xmlelement(name "title" c.title),
                xmlelement(name "credits" c.credits)))
                from course c
                where c.dept_name = d.dept_name
                order by c.course_id)))
from department d
select xmlelement(name "instructor",
        xmlelement(name "IID", i.IID),
        xmlelement(name "name", i.name),
        xmlelement(name "dept_name", i.dept_name),
        xmlelement(name "salary", i.salary),
        (select xmlagg(xmlelement(name "course_id", t.course_id))
                from teaches t
                where teaches.IID = i.IID
                order by t.course_id))
from instructor i
```

university-2:

```
select xmlelement(name “instructor”,
        xmlelement(name “IID”, i.IID),
        xmlelement(name “name”, i.name),
        xmlelement(name “dept_name”, i.dept_name),
        xmlelement(name “salary”, i.salary),
        xmlelement(name “teaches”,
        (select xmlagg(xmlelement(name “course”,
                xmlelement(name “course_id” c.course_id),
                xmlelement(name “title” c.title),
                xmlelement(name “credits” c.credits)
                from course c, teaches t
                where c.course_id = t.course_id and t.IID = i.IID
                order by c.course_id)))))
from instructor i
select xmlelement(name “department”,
        xmlelement(name “dept_name”, d.dept_name),
        xmlelement(name “building” d.building),
        xmlelement(name “budget” d.budget),
        (select xmlagg(xmlelement(name “course_id”, c.course_id))
                from course c
                where c.dept_name = d.dept_name)
from department d
```

23.19 As in Exercise 23.18, write queries to convert university data to the *university-1* and *university-2* XML schemas, but this time by writing XQuery queries on the default SQL/XML database to XML mapping.

Answer: **university-1**:

```
<university-1> {
    for $x in /university/department/row
    return
        <department>
            <dept_name> {$x/dept_name} </dept_name>
            <building> {$x/building} </building>
            <budget> {$x/budget} </budget>
            for $c in /university/course/row[dept_name=$x/dept_name]
            return <course>
                <course_id> {$c/course_id} </course_id>
                <title> {$c/title} </title>
                <credits> {$c/credits} </credits>
            </course>
        </department>
    for $i in /university/instuctor/row
    return
        <instructor>
            <IID> {$i/IID} </IID>
            <name> {$i/name} </name>
            <dept_name> {$i/dept_name} </dept_name>
            <salary> {$i/salary} </salary>
                for $t in /university/teaches/row[IID=$i/IID]
                return {
                    <course_id> {$t/course_id} </course_id>
                }
        </instructor>
} </university-1>
```

university-2:

```
<university-2> {
    for $x in /university/instructor/row/
    return
        <instructor>
            <IID> {$i/IID} </IID>
            <name> {$i/name} </name>
            <dept_name> {$i/dept_name} </dept_name>
            <salary> {$i/salary} </salary>
            <teaches> {
                for $t in /university/teaches/row[IID=$i/IID]
                    $c in /university/course/row[$t.course_id=$c/course_id]
                return <course>
                    <course_id> {$c/course_id} </course_id>
                    <title> {$c/title} </title>
                    <dept_name> {$c/dept_name} </dept_name>
                    <credits> {$c/credits} </credits>
                </course>
            } </teaches>
        </instructor>
    for $x in /university/department/row/
    return
        <department>
            <dept_name> {$x/dept_name} </dept_name>
            <building> {$x/building} </building>
            <budget> {$x/budget} </budget>
                for $c in /university/course/row[dept_name=$x/dept_name]
                return
                    <course_id> {$c/course_id} </course_id>
        </department>
} </university-2>
```

23.20 One way to shred an XML document is to use XQuery to convert the schema to an SQL/XML mapping of the corresponding relational schema, and then use the SQL/XML mapping in the backward direction to populate the relation.

As an illustration, give an XQuery query to convert data from the *university-1* XML schema to the SQL/XML schema shown in Figure 23.15.

Answer:

```
<department> {
    for $x in /university-1/department
    return
        <row>
            <dept_name> { $x/dept_name } </dept_name>
            <building> { $x/building } </building>
            <budget> { $x/budget } </budget>
        </row>
} </department>

<course> {
    for $x in /university-1/department/course
    return
        <row> {$x} </row>
} </course>

<instructor> {
    for $x in /university-1/instructor
    return
        <row>
            <IID> { $x/IID } </IID>
            <name> { $x/name } </name>
            <dept_name> { $x/dept_name } </dept_name>
            <salary> { $x/salary } </salary>
        </row>
} <instructor>

<teaches> {
    for $i in /university-1/instructor, $cid in $i/course_id
    return
        <row>
            <IID> { $i/IID } </IID>
            <course_id> { $cid } </course_id>
        </row>
} <teaches>
```

23.21 Consider the example XML schema from Section 23.3.2, and write XQuery queries to carry out the following tasks:

a. Check if the key constraint shown in Section 23.3.2 holds.

b. Check if the keyref constraint shown in Section 23.3.2 holds.

Answer:

a. Check if the key constraint shown in Section 23.3.2 holds.

```
let $x = /university/department
let $y = distinct-values(/university/department/dept_name)
return {
      if fn:count($y) == fn:count($x)
            then 1
            else 0
}
```

b. Check if the keyref constraint shown in Section 23.3.2 holds.

```
return
      (every $c in /university/course satisfies
            (some $d in /university/department satisfies
                  $c/dept_name = $d/dept_name))
```

23.22 Consider Practice Exercise 23.7, and suppose that authors could also appear as top-level elements. What change would have to be done to the relational schema?

Answer: Author id can be added as an attrribute to the author top element. It can be used to refer to the author in the book and article topelements and also keep track of the order.

CHAPTER 24

Advanced Application Development

Practice Exercises

24.1 Many applications need to generate sequence numbers for each transaction.

a. If a sequence counter is locked in two-phase manner, it can become a concurrency bottleneck. Explain why this may be the case.

b. Many database systems support built-in sequence counters that are not locked in two-phase manner; when a transaction requests a sequence number, the counter is locked, incremented and unlocked.

i. Explain how such counters can improve concurrency.

ii. Explain why there may be gaps in the sequence numbers belonging to the final set of committed transactions.

Answer: If two-phase locking is used on the counter, the counter must remain locked in exclusive mode until the transaction is done acquiring locks. During that time, the counter is unavailable and no concurrent transactions can be started regardless of whether they would have data conflicts with the transaction holding the counter.
If locking is done outside the scope of a two-phase locking protocol, the counter is locked only for the brief time it takes to increment the counter. SInce there is no other operation besides increment performed on the counter, this creates no problem except when a transaction aborts. During an abort, the counter *cannot* be restored to its old value but instead should remain as it stands. This means that some counter values are unused since those values were assigned to aborted transactions. To see why we cannot restore the old counter value, assume transaction T_a has counter value 100 and then T_b is given the next value, 101. If T_a aborts and the counter were restored to 100, the value 100 would be given to some other transaction T_c and then 101 would be given to

a second transaction T_d. Now we have two non-aborted transactions with the same counter value.

24.2 Suppose you are given a relation $r(a, b, c)$.

a. Give an example of a situation under which the performance of equality selection queries on attribute a can be greatly affected by how r is clustered.

b. Suppose you also had range selection queries on attribute b. Can you cluster r in such a way that the equality selection queries on $r.a$ and the range selection queries on $r.b$ can both be answered efficiently? Explain your answer.

c. If clustering as above is not possible, suggest how both types of queries can be executed efficiently by choosing appropriate indices.

Answer:

a. If r is not clustered on a, tuples of r with a particular a-value could be scattered throughout the area in which r is stored. This would make it necessary to scan all of r. Clustering on a combined with an index, would allow tuples of r with a particular a-value to be retrieved directly.

b. This is possible only if there is a special relationship between a and b such as "if tuple t_1 has an a-value less than the a-value of tuple t_2, then t_1 must have a b-value less than that of t_2". Aside from special cases, such a clustering is not possible since the sort order of the tuples on a is different from the sort order on b.

c. The physical order of the tuples cannot always be suited to both queries. If r is clustered on a and we have a B^+-tree index on b, the first query can be executed very efficiently. For the second, we can use the usual B^+-tree range-query algorithm to obtain pointers to all the tuples in the result relation, then sort those those pointers so that any particular disk block is accessed only once.

24.3 Suppose that a database application does not appear to have a single bottleneck; that is, CPU and disk utilization are both high, and all database queues are roughly balanced. Does that mean the application cannot be tuned further? Explain your answer.
Answer: it may still be tunable. Example using a materialized view may reduce both CPU and disk utilization

24.4 Suppose a system runs three types of transactions. Transactions of type A run at the rate of 50 per second, transactions of type B run at 100 per second, and transactions of type C run at 200 per second. Suppose the mix of transactions has 25 percent of type A, 25 percent of type B, and 50 percent of type C.

a. What is the average transaction throughput of the system, assuming there is no interference between the transactions?

b. What factors may result in interference between the transactions of different types, leading to the calculated throughput being incorrect?

Answer:

a. Let there be 100 transactions in the system. The given mix of transaction types would have 25 transactions each of type A and B, and 50 transactions of type C. Thus the time taken to execute transactions only of type A is 0.5 seconds and that for transactions only of type B or only of type C is 0.25 seconds. Given that the transactions do not interfere, the total time taken to execute the 100 transactions is $0.5 + 0.25 + 0.25 = 1$ second. i.e, the average overall transaction throughput is 100 *transactions per second*.

b. One of the most important causes of transaction interference is lock contention. In the previous example, assume that transactions of type A and B are update transactions, and that those of type C are queries. Due to the speed mismatch between the processor and the disk, it is possible that a transaction of type A is holding a lock on a "hot" item of data and waiting for a disk write to complete, while another transaction (possibly of type B or C) is waiting for the lock to be released by A. In this scenario some CPU cycles are wasted. Hence, the observed throughput would be lower than the calculated throughput.
Conversely, if transactions of type A and type B are disk bound, and those of type C are CPU bound, and there is no lock contention, observed throughput may even be better than calculated.
Lock contention can also lead to deadlocks, in which case some transaction(s) will have to be aborted. Transaction aborts and restarts (which may also be used by an optimistic concurrency control scheme) contribute to the observed throughput being lower than the calculated throughput.
Factors such as the limits on the sizes of data-structures and the variance in the time taken by book-keeping functions of the transaction manager may also cause a difference in the values of the observed and calculated throughput.

24.5 List some benefits and drawbacks of an anticipatory standard compared to a reactionary standard.
Answer: In the absence of an anticipatory standard it may be difficult to reconcile between the differences among products developed by various organizations. Thus it may be hard to formulate a reactionary standard without sacrificing any of the product development effort. This problem has been faced while standardizing pointer syntax and access mechanisms for the ODMG standard.

On the other hand, a reactionary standard is usually formed after extensive product usage, and hence has an advantage over an anticipatory standard - that of built-in pragmatic experience. In practice, it has been found that some anticipatory standards tend to be over-ambitious. SQL-3 is an example of a standard that is complex and has a very large number of features. Some of these features may not be implemented for a long time on any system, and some, no doubt, will be found to be inappropriate.

CHAPTER 24

Advanced Application Development

Exercises

24.6 Find out all performance information your favorite database system provides. Look for at least the following: what queries are currently executing or executed recently, what resources each of them consumed (CPU and I/O), what fraction of page requests resulted in buffer misses (for each query, if available), and what locks have a high degree of contention. You may also be able to get information about CPU and I/O utilization from the operating system.

Answer:

- Postgresql: The *EXPLAIN* command lets us see what query plan the system creates for any query. The numbers that are currently quoted by *EXPLAIN* are:
  a. Estimated start-up cost
  b. Estimated total cost
  c. Estimated number of rows output by this plan node
  d. Estimated average width of rows
- SQL: There is a Microsoft tool called *SQL Profiler*. The data is logged, and then the performance can be monitored. The following performance counters can be logged.
  a. Memory
  b. Physical Disk
  c. Process
  d. Processor
  e. SQLServer:Access Methods
  f. SQLServer:Buffer Manager
  g. SQLServer:Cache Manager

h. SQLServer:Databases
i. SQLServer:General Statistics
j. SQLServer:Latches
k. SQLServer:Locks
l. SQLServer:Memory Manager
m. SQLServer:SQL Statistics
n. SQLServer:SQL Settable

24.7 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 12.5.4) 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 is 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 scanning *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 partic-

ular 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 where 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 five-fold.

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 by improving the bus architecture, one may obtain an improvement in system performance.

24.8 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 3 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.

24.9 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 10.2.

Answer: Given that all internal nodes of the B^+-tree are in memory, and only a very small fraction of the leaf pages can fit in memory. We can deduce that each 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, number of disks needed to support a workload of 1000 transactions is 1000/100 = 10 disks.

The disk parameters given in Section 10.2 are almost the same as the values in the current chapter. So, the number of disks required will be around 10 in this case also.

24.10 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.

24.11 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.

24.12 List at least 4 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.

24.13 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. Hence TPC-R and TPC-H

were introduced as refinements of TPC-D, both of which use same schema and workload. TPC-R models periodic reporting queries, and the database running it is permited to use materialized views. TPC-H, on the other hand, models ad hoc querying, and prohibits materialized views and other redundant information.

24.14 Explain what application characteristics would help you decide which of TPCC, TPC-H, or TPC-R best models the application.

Answer: Depending on the application characterictics, different benchmarks are used to model it.

The TPCC 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 TPCH (H represents *adhoc*) benchmark models the applications which prohibit materialized views and other redundant information, and permits indices only on primary and foriegn keys. This benchmark models ad-hoc querying where the queries are not known beforehand.

The TPCR (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.

CHAPTER 25

Advanced Data Types and New Applications

Practice Exercises

25.1 What are the two types of time, and how are they different? Why does it make sense to have both types of time associated with a tuple?

Answer: A temporal database models the changing states of some aspects of the real world. The time intervals related to the data stored in a temporal database may be of two types - *valid time* and *transaction time*. The valid time for a fact is the set of intervals during which the fact is true in the real world. The transaction time for a data object is the set of time intervals during which this object is part of the physical database. Only the transaction time is system dependent and is generated by the database system.

Suppose we consider our sample bank database to be bitemporal. Only the concept of valid time allows the system to answer queries such as - "What was Smith's balance two days ago?". On the other hand, queries such as - "What did we record as Smith's balance two days ago?" can be answered based on the transaction time. The difference between the two times is important. For example, suppose, three days ago the teller made a mistake in entering Smith's balance and corrected the error only yesterday. This error means that there is a difference between the results of the two queries (if both of them are executed today).

25.2 Suppose you have a relation containing the x, y coordinates and names of restaurants. Suppose also that the only queries that will be asked are of the following form: The query specifies a point, and asks if there is a restaurant exactly at that point. Which type of index would be preferable, R-tree or B-tree? Why?

Answer: The given query is not a range query, since it requires only searching for a point. This query can be efficiently answered by a B-tree index on the pair of attributes (x, y).

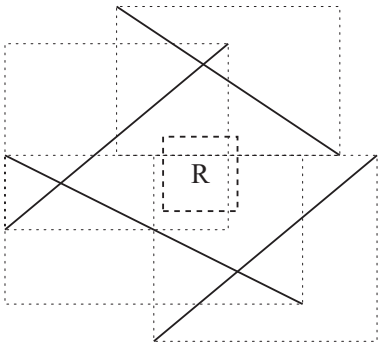
25.3 Suppose you have a spatial database that supports region queries (with circular regions) but not nearest-neighbor queries. Describe an algorithm to find the nearest neighbor by making use of multiple region queries.

Answer: Suppose that we want to search for the nearest neighbor of a point P in a database of points in the plane. The idea is to issue multiple region queries centered at P. Each region query covers a larger area of points than the previous query. The procedure stops when the result of a region query is non-empty. The distance from P to each point within this region is calculated and the set of points at the smallest distance is reported.

25.4 Suppose you want to store line segments in an R-tree. If a line segment is not parallel to the axes, the bounding box for it can be large, containing a large empty area.

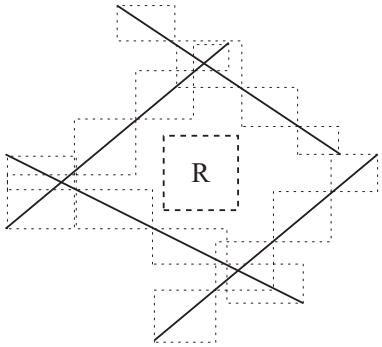
- Describe the effect on performance of having large bounding boxes on queries that ask for line segments intersecting a given region.
- Briefly describe a technique to improve performance for such queries and give an example of its benefit. Hint: You can divide segments into smaller pieces.

Answer: Large bounding boxes tend to overlap even where the region of overlap does not contain any information. The following figure:



shows a region R within which we have to locate a segment. Note that even though none of the four segments lies in R, due to the large bounding boxes, we have to check each of the four bounding boxes to confirm this.

A significant improvement is observed in the follwoing figure:



where each segment is split into multiple pieces, each with its own bounding box. In the second case, the box R is not part of the boxes indexed by the R-tree. In general, dividing a segment into smaller pieces causes the bounding boxes to be smaller and less wasteful of area.

25.5 Give a recursive procedure to efficiently compute the spatial join of two relations with R-tree indices. (Hint: Use bounding boxes to check if leaf entries under a pair of internal nodes may intersect.)

Answer: Following is a recursive procedure for computing spatial join of two R-trees.

```
SpJoin(node n1, node n2)
begin
    if(the bounding boxes of n1 and n2 do not intersect)
        return;
    if(both n1 and n2 are leaves)
        output all pairs of entries (e1, e2) such that
            e1 ∈ n1 and e2 ∈ n2, and e1 and e2 overlap;
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    for each ns1 in NS1 and ns2 in NS2;
        SpJoin(ns1, ns2);
end
```

25.6 Describe how the ideas behind the RAID organization (Section 10.3) can be used in a broadcast-data environment, where there may occasionally be noise that prevents reception of part of the data being transmitted.

Answer: The concepts of RAID can be used to improve reliability of the broadcast of data over wireless systems. Each block of data that is to be broadcast is split into *units* of equal size. A checksum value is calculated for each unit and appended to the unit. Now, parity data for these units is calculated. A checksum for the parity data is appended to it to form a parity unit. Both the data units and the parity unit are then broadcast one after the other as a single transmission.

On reception of the broadcast, the receiver uses the checksums to verify whether each unit is received without error. If one unit is found to be in error, it can be reconstructed from the other units.

The size of a unit must be chosen carefully. Small units not only require more checksums to be computed, but the chance that a burst of noise corrupts more than one unit is also higher. The problem with using large units is that the probability of noise affecting a unit increases; thus there is a tradeoff to be made.

25.7 Define a model of repeatedly broadcast data in which the broadcast medium is modeled as a virtual disk. Describe how access time and data-transfer rate for this virtual disk differ from the corresponding values for a typical hard disk.

Answer: We can distinguish two models of broadcast data. In the case of a pure broadcast medium, where the receiver cannot communicate with the broadcaster, the broadcaster transmits data with periodic cycles of retransmission of the entire data, so that new receivers can catch up with all the broadcast information. Thus, the data is broadcast in a continuous cycle. This period of the cycle can be considered akin to the worst case rotational latency in a disk drive. There is no concept of seek time here. The value for the cycle latency depends on the application, but is likely to be at least of the order of seconds, which is much higher than the latency in a disk drive.

In an alternative model, the receiver can send requests back to the broadcaster. In this model, we can also add an equivalent of disk access latency, between the receiver sending a request, and the broadcaster receiving the request and responding to it. The latency is a function of the volume of requests and the bandwidth of the broadcast medium. Further, queries may get satisfied without even sending a request, since the broadcaster happened to send the data either in a cycle or based on some other receivers request. Regardless, latency is likely to be at least of the order of seconds, again much higher than the corresponding values for a hard disk.

A typical hard disk can transfer data at the rate of 1 to 5 megabytes per second. In contrast, the bandwidth of a broadcast channel is typically only a few kilobytes per second. Total latency is likely to be of the order of seconds to hundreds or even thousands of seconds, compared to a few milliseconds for a hard disk.

25.8 Consider a database of documents in which all documents are kept in a central database. Copies of some documents are kept on mobile

computers. Suppose that mobile computer A updates a copy of document 1 while it is disconnected, and, at the same time, mobile computer B updates a copy of document 2 while it is disconnected. Show how the version-vector scheme can ensure proper updating of the central database and mobile computers when a mobile computer reconnects.

Answer: Let C be the computer onto which the central database is loaded. Each mobile computer (host) i stores, with its copy of each document d, a version-vector – that is a set of version numbers $V_{d,i,j}$, with one entry for each other host j that stores a copy of the document d, which it could possibly update.

Host A updates document 1 while it is disconnected from C. Thus, according to the version vector scheme, the version number $V_{1,A,A}$ is incremented by one.

Now, suppose host A re-connects to C. This pair exchanges version-vectors and finds that the version number $V_{1,A,A}$ is greater than $V_{1,C,A}$ by 1, (assuming that the copy of document 1 stored host A was updated most recently only by host A). Following the version-vector scheme, the version of document 1 at C is updated and the change is reflected by an increment in the version number $V_{1,C,A}$. Note that these are the only changes made by either host.

Similarly, when host B connects to host C, they exchange version-vectors, and host B finds that $V_{1,B,A}$ is one less than $V_{1,C,A}$. Thus, the version number $V_{1,B,A}$ is incremented by one, and the copy of document 1 at host B is updated.

Thus, we see that the version-vector scheme ensures proper updating of the central database for the case just considered. This argument can be very easily generalized for the case where multiple off-line updates are made to copies of document 1 at host A as well as host B and host C. The argument for off-line updates to document 2 is similar.

CHAPTER 25

Advanced Data Types and New Applications

This chapter covers advanced data types and new applications, including temporal datababases, spatial and geographic databases, multimedia databases, and mobility and personal databases. In particular, the data types mentioned above have grown in importance in recent years, and commercial database systems are increasingly providing support for such data types through extensions to the database system variously called cartridges or extenders.

This chapter is suited as a means to lay the groundwork for an advanced course. Some of the material, such as temporal and spatial data types, may be suitable for self-study in a first course.

Exercises

25.9 Will functional dependencies be preserved if a relation is converted to a temporal relation by adding a time attribute? How is the problem handled in a temporal database?

Answer: Functional dependencies may be violated when a relation is augmented to include a time attribute. For example, suppose we add a time attribute to the relation *account* in our sample bank database. The dependency *account-number*→*balance* may be violated since a customer's balance would keep changing with time.

To remedy this problem temporal database systems have a slightly different notion of functional dependency, called *temporal functional dependency*. For example, the temporal functional dependency:

$$account\text{-}number \xrightarrow{\tau} balance$$

over *Account-schema* means that for each instance *account* of *Account-schema*, all snapshots of *account* satisfy the functional dependency

account-number→*balance*;

i.e at any time instance, each account will have a unique bank balance corresponding to it.

25.10 Consider two-dimensional vector data where the data items do not overlap. Is it possible to convert such vector data to raster data? If so, what are the drawbacks of storing raster data obtained by such conversion, instead of the original vector data?

Answer: To convert non-overlapping vector data to raster data, we set the values for exactly those pixels that lie on any one of the data items (regions); the other pixels have a default value.
The disadvantages to this approach are: loss of precision in location information (since raster data loses resolution), a much higher storage requirement, and loss of abstract information (like the shape of a region).

25.11 Study the support for spatial data offered by the database system that you use, and implement the following:

a. A schema to represent the geographic location of restaurants along with features such as the cuisine served at the restaurant and the level of expensiveness.

b. A query to find moderately priced restaurants that serve Indian food and are within 5 miles of your house (assume any location for your house).

c. A query to find for each restaurant the distance from the nearest restaurant serving the same cuisine and with the same level of expensiveness.

Answer: PostgreSQL includes support for R-tree indices over spatial data, as well as a number of built-in geometric data types (points, boxes, circles, lines, and paths) to represent spatial data, and functions to manipulate this data.

a.

```
create table restaurants (
    name varchar(30),
    location point,
    cusine varchar(30),
    price int)
```

b. Assume your house is at coordinates (21.5, 14.2), and that a price value of 2 means "moderately priced".
<-> is the PostgreSQL operator representing "distance between".

```
select name
```

```
from restaurants
where ((point '(21.5, 14.2)') <-> location) < 5.0
    and cusine = 'Indian'
    and price <= 2
```

c.

```
select r1.name, min(r1.location <-> r2.location)
from restaurants as r1, restaurants as r2
where r1.cusine = r2.cusine
    and r1.price = r2.price
group by r1.name
```

25.12 What problems can occur in a continuous-media system if data are delivered either too slowly or too fast?

Answer: Continuous media systems typically handle a large amount of data, which have to be delivered at a steady rate. Suppose the system provides the picture frames for a television set. The delivery rate of data from the system should be matched with the frame display rate of the TV set. If the delivery rate is too low, the display would periodically freeze or blank out, since there will be no new data to be displayed for some time. On the other hand, if the delivery rate is too high, the data buffer at the destination TV set will overflow causing loss of data; the lost data will never get displayed.

25.13 List three main features of mobile computing over wireless networks that are distinct from traditional distributed systems.

Answer: Some of the main distinguishing features are as follows.

- In distributed systems, disconnection of a host from the network is considered to be a *failure*, whereas allowing such disconnection is a *feature* of mobile systems.
- Distributed systems are usually centrally administered, whereas in mobile computing, each personal computer that participates in the system is administered by the user (owner) of the machine and there is little central administration, if any.
- In conventional distributed systems, each machine has a fixed location and network address(es). This is not true for mobile computers, and in fact, is antithetical to the very purpose of mobile computing.
- Queries made on a mobile computing system may involve the location and velocity of a host computer.
- Each computer in a distributed system is allowed to be arbitrarily large and may consume a lot of (almost) uninterrupted electrical power. Mobile systems typically have small computers that run on low wattage, short-lived batteries.

25.14 List three factors that need to be considered in query optimization for mobile computing that are not considered in traditional query optimizers.

Answer: The most important factor influencing the cost of query processing in traditional database systems is that of disk I/O. However, in mobile computing, minimizing the amount of energy required to execute a query is an important task of a query optimizer. To reduce the consumption of energy (battery power), the query optimizer on a mobile computer must minimize the size and number of queries to be transmitted to remote computers as well as the time for which the disk is spinning.

In traditional database systems, the cost model typically does not include connection time and the amount of data transferred. However, mobile computer users are usually charged according to these parameters. Thus, these parameters should also be minimized by a mobile computer's query optimizer.

25.15 Give an example to show that the version-vector scheme does not ensure serializability. (Hint: Use the example from Practice Exercise 25.8, with the assumption that documents 1 and 2 are available on both mobile computers A and B, and take into account the possibility that a document may be read without being updated.)

Answer: Consider the example given in the previous exercise. Suppose that both host A and host B are not connected to each other. Further, assume that identical copies of document 1 and document 2 are stored at host A and host B.

Let $\{X = 5\}$ be the initial contents of document 1, and $\{X = 10\}$ be the initial contents of document 2. Without loss of generality, let us assume that all version-vectors are initially zero.

Suppose host A updates the number its copy of document 1 with that in its copy of document 2. Thus, the contents of both the documents (at host A) are now $\{X = 10\}$. The version number $V_{1,A,A}$ is incremented to 1.

While host B is disconnected from host A, it updates the number in its copy of document 2 with that in its copy of document 1. Thus, the contents of both the documents (at host B) are now $\{X = 5\}$. The version number $V_{2,B,B}$ is incremented to 1.

Later, when host A and host B connect, they exchange version-vectors. The version-vector scheme updates the copy of document 1 at host B to $\{X = 10\}$, and the copy of document 2 at host A to $\{X = 5\}$. Thus, both copies of each document are identical, viz. document 1 contains $\{X = 10\}$ and document 2 contains $\{X = 5\}$.

However, note that a serial schedule for the two updates (one at host A and another at host B) would result in both documents having the *same* contents. Hence this example shows that the version-vector scheme does not ensure serializability.

CHAPTER 26

Advanced Transaction Processing

Practice Exercises

26.1 Like database systems, workflow systems also require concurrency and recovery management. List three reasons why we cannot simply apply a relational database system using 2PL, physical undo logging, and 2PC.

Answer:

a. The tasks in a workflow have dependencies based on their status. For example the starting of a task may be conditional on the outcome (such as commit or abort) of some other task. All the tasks cannot execute independently and concurrently, using 2PC just for atomic commit.

b. Once a task gets over, it will have to expose its updates, so that other tasks running on the same processing entity don't have to wait for long. 2PL is too strict a form of concurrency control, and is not appropriate for workflows.

c. Workflows have their own consistency requirements; that is, failure-atomicity. An execution of a workflow must finish in an *acceptable termination state*. Because of this, and because of early exposure of uncommitted updates, the recovery procedure will be quite different. Some form of logical logging and compensation transactions will have to be used. Also to perform forward recovery of a failed workflow, the recovery routines need to restore the state information of the scheduler and tasks, not just the updated data items. Thus simple WAL cannot be used.

26.2 Consider a main-memory database system recovering from a system crash. Explain the relative merits of:

- Loading the entire database back into main memory before resuming transaction processing.
- Loading data as it is requested by transactions.

Answer:

- Loading the entire database into memory in advance can provide transactions which need high-speed or realtime data access the guarantee that once they start they will not have to wait for disk accesses to fetch data. However no transaction can run till the entire database is loaded.
- The advantage in loading on demand is that transaction processing can start rightaway; however transactions may see long and unpredictable delays in disk access until the entire database is loaded into memory.

26.3 Is a high-performance transaction system necessarily a real-time system? Why or why not?

Answer: A high-performance system is not necessarily a real-time system. In a high performance system, the main aim is to execute each transaction as quickly as possible, by having more resources and better utilization. Thus average speed and response time are the main things to be optimized. In a real-time system, speed is not the central issue. Here *each* transaction has a deadline, and taking care that it finishes within the deadline or takes as little extra time as possible, is the critical issue.

26.4 Explain why it may be impractical to require serializability for long-duration transactions.

Answer: In the presence of long-duration transactions, trying to ensure serializability has several drawbacks:-

a. With a waiting scheme for concurrency control, long-duration transactions will force long waiting times. This means that response time will be high, concurrency will be low, so throughput will suffer. The probability of deadlocks is also increased.

b. With a time-stamp based scheme, a lot of work done by a long-running transaction will be wasted if it has to abort.

c. Long duration transactions are usually interactive in nature, and it is very difficult to enforce serializability with interactiveness.

Thus the serializability requirement is impractical. Some other notion of database consistency has to be used in order to support long duration transactions.

26.5 Consider a multithreaded process that delivers messages from a durable queue of persistent messages. Different threads may run concurrently, attempting to deliver different messages. In case of a delivery failure, the

message must be restored in the queue. Model the actions that each thread carries out as a multilevel transaction, so that locks on the queue need not be held until a message is delivered.

Answer: Each thread can be modeled as a transaction T which takes a message from the queue and delivers it. We can write transaction T as a multilevel transaction with subtransactions T_1 and T_2. Subtransaction T_1 removes a message from the queue and subtransaction T_2 delivers it. Each subtransaction releases locks once it completes, allowing other transactions to access the queue. If transaction T_2 fails to deliver the message, transaction T_1 will be undone by invoking a compensating transaction which will restore the message to the queue.

26.6 Discuss the modifications that need to be made in each of the recovery schemes covered in Chapter 16 if we allow nested transactions. Also, explain any differences that result if we allow multilevel transactions.

Answer: Consider the advanced recovery algorithm of Section 16.4. The redo pass, which repeats history, is the same as before. We discuss below how the undo pass is handled.

- **Recovery with nested transactions**:
  Each subtransaction needs to have a unique TID, because a failed subtransaction might have to be independently rolled back and restarted.

  If a subtransaction fails, the recovery actions depend on whether the unfinished upper-level transaction should be aborted or continued. If it should be aborted, all finished and unfinished subtransactions are undone by a backward scan of the log (this is possible because the locks on the modified data items are not released as soon as a subtransaction finishes). If the nested transaction is going to be continued, just the failed transaction is undone, and then the upper-level transaction continues.

  In the case of a system failure, depending on the application, the entire nested-transaction may need to be aborted, or, (for e.g., in the case of long duration transactions) incomplete subtransactions aborted, and the nested transaction resumed. If the nested-transaction must be aborted, the rollback can be done in the usual manner by the recovery algorithm, during the undo pass. If the nested-transaction must be restarted, any incomplete subtransactions that need to be rolled back can be rolled back as above. To restart the nested-transaction, state information about the transaction, such as locks held and execution state, must have been noted on the log, and must restored during recovery. Mini-batch transactions (discussed in Section 24.1.10) are an example of nested transactions that must be restarted.

- **Recovery with multi-level transactions**:
  In addition to what is done in the previous case, we have to handle the problems caused by exposure of updates performed by committed subtransactions of incomplete upper-level transactions. A committed

subtransaction may have released locks that it held, so the compensating transaction has to reacquire the locks. This is straightforward in the case of transaction failure, but is more complicated in the case of system failure.

The problem is, a lower level subtransaction a of a higher level transaction A may have released locks, which have to be reacquired to compensate A during recovery. Unfortunately, there may be some other lower level subtransaction b of a higher level transaction B that started and acquired the locks released by a, before the end of A. Thus undo records for b may precede the operation commit record for A. But if b had not finished at the time of the system failure, it must first be rolled back and its locks released, to allow the compensating transaction of A to reacquire the locks.

This complicates the undo pass; it can no longer be done in one backward scan of the log. Multilevel recovery is described in detail in David Lomet, "MLR: A Recovery Method for Multi-Level Systems", ACM SIGMOD Conf. on the Management of Data 1992, San Diego.

CHAPTER 26

Advanced Transaction Processing

In this chapter, we go beyond the basic transaction processing schemes discussed previously, and cover more advanced transaction-processing concepts, including transaction-processing monitors, workflow systems, main-memory databases, real-time transaction systems, and handling of long-duration transactions by means of nested transactions, multi-level transactions and weak degrees of consistency. We end the chapter by covering weak degrees of consistency used to handle multidatabase systems.

This chapter is suited to an advanced course. The sections on TP monitors and workflows may also be covered in an introductory course as independent-study material.

Coverage of remote backup systems has been moved from this chapter to the chapter on recovery, while coverage of transaction processing in multidatabases has been moved into this chapter from its earlier position in the distributed database chapter.

Exercises

26.7 Explain how a TP monitor manages memory and processor resources more effectively than a typical operating system.

Answer: In a typical OS, each client is represented by a process, which occupies a lot of memory. Also process multi-tasking over-head is high.

A TP monitor is more of a service provider, rather than an environment for executing client processes. The client processes run at their own sites, and they send requests to the TP monitor whenever they wish to avail of some service. The message is routed to the right server by the TP monitor, and the results of the service are sent back to the client.

The advantage of this scheme is that the same server process can be serving several clients simultaneously, by using multithreading. This saves memory space, and reduces CPU overheads on preserving ACID prop-

erties and on scheduling entire processes. Even without multi-threading, the TP monitor can dynamically change the number of servers running, depending on whatever factors affect good performance. All this is not possible with a typical OS setup.

26.8 Compare TP-monitor features with those provided by Web servers supporting servlets (such servers have been nicknamed *TP-lite*).

Answer: Web application servers supporting servlets (as also Web application servers providing similar functionality through other language APIs) have many features of TP monitors. For example, they allow a single process, or a few processes, to serve a large number of requests by exploiting multi-threading. Systems built to handle large numbers of requests usually have routers that divide up incoming traffic between a large number of Web application servers. Web application servers typically lack a few features such as support for transaction coordination (using two-phase commit), which TP-monitors support, i although some application servers do have addon features that support transaction coordination.

26.9 Consider the process of admitting new students at your university (or new employees at your organization).

a. Give a high-level picture of the workflow starting from the student application procedure.

b. Indicate acceptable termination states and which steps involve human intervention.

c. Indicate possible errors (including deadline expiry) and how they are dealt with.

d. Study how much of the workflow has been automated at your university.

Answer:

a. Students typically apply online to the university. Once they have filled in all details of their form, they submit it, and pay application fees. The fee payment is itself a sub-workflow, involving an external party such as a credit card company or bank, and possibly a third-party payment gateway as an intermediary to the credit card company or bank.

   At this point the university has to process the form. The first step of processing by the university is to check that the applications are complete; if anything is incomplete, the student has to be informed and given a chance to correct the form. Typically there is date for starting the decision process; all forms received up to that date are taken together, and after filtering on some criteria, and inputs from various people involved in the admission process, an admission decision is made. Subsequently forms may be placed in accepted, rejected, or wait-listed state.

This decision is conveyed back to students. The notification is done by email and SMS, and the deliver of email/SMS can itself be considered as an automated workflow since it involves external services, with persistent message delivery queues.

b. Acceptable termination states include: student admitted and student rejected; intermediate states may include student wait-listed, which will eventually end up in either admitted or reject state. The eventual decision on admission or rejection (or temporary wait-listing) will require human decisions. Applications may be rejected early if some requirements are not met, such as application fees not paid, without human intervention.

c. At IIT Bombay, as in most institutions today, applications, including fee payment are online and automated. Humans are involved in the decision making process, but there is a fair degree of bulk processing, for example cutoff marks are specified for standardized exams, and all students below the cutoff are automatically rejected. Once the decision is made, the notification by email/SMS is itself a workflow, which is automated.

26.10 Answer the following questions regarding electronic payment systems:

a. Explain why electronic transactions carried out using credit-card numbers may be insecure.

b. An alternative is to have an electronic payment gateway maintained by the credit-card company, and the site receiving payment redirects customers to the gateway site to make the payment.

   i. Explain what benefits such a system offers if the gateway does not authenticate the user.

   ii. Explain what further benefits are offered if the gateway has a mechanism to authenticate the user.

c. Some credit-card companies offer a one-time-use credit-card number as a more secure method of electronic payment. Customers connect to the credit-card company's Web site to get the one-time-use number. Explain what benefit such a system offers, as compared to using regular credit-card numbers. Also explain its benefits and drawbacks as compared to electronic payment gateways with authentication.

d. Does either of the above systems guarantee the same privacy that is available when payments are made in cash? Explain your answer.

Answer:

a. Credit card numbers can be easily stolen by someone who handles the card (for offline transactions) or may be stolen in bulk from computers where they are stored in (online) shops where a purchase was made. Some who has got the number (including verification

numbers such as the widely used CVV number) can make purchases online using these numbers.

b. i. Even if the gateway does not authorize the user, as long as the user can check the authenticity of the gateway (using HTTPS protocol certificates), the user knows that the credit card numbers are only handled by a trusted party; the site receiving the payment never gets to even see the card number.

ii. If the gateway further authenticates the user, using additional passwords (usually this is done by further redirecting the user to the bank which issued the credit card, so the user need not trust the gateway with passwords), even someone who has access to the credit card number cannot make online purchases using the credit card. Such authentication is mandatory for online payments in some countries.

c. One-time-use credit card numbers cannot be used again, so even if it is stored at the site receiving the payment, and subsequently compromised, it cannot be used to make any further purchases online. Thus, it has similar benefits to a system using authentication. However, its drawback is that the user has to perform extra actions to get a one-time-use number.

d. With cash, it is quite possible to make completely anonymous purchases, with complete privacy. None of the above systems guarantees the same privacy as cash. First, the credit card company knows who you bought things from, so there is not question of a fully anonymous purchase. Second, even though your identity can be concealed from the site where you make a purchase, law enforcement officials can get information from both the site and the credit card company, compromising your privacy completely.

26.11 If the entire database fits in main memory, do we still need a database system to manage the data? Explain your answer.

Answer: Even if the entire database fits in main memory, a DBMS is needed to perform tasks like concurrency control, recovery, logging etc, in order to preserve ACID properties of transactions.

26.12 In the group-commit technique, how many transactions should be part of a group? Explain your answer.

Answer: As log-records are written to stable storage in multiples of a block, we should group transaction commits in such a way that the last block containing log-records for the current group is almost full.

26.13 In a database system using write-ahead logging, what is the worst-case number of disk accesses required to read a data item from a specified disk page. Explain why this presents a problem to designers of real-time database systems. Hint: consider the case when the disk buffer is full.

Answer: In the worst case, a read can cause a buffer page to be written to disk (preceded by the corresponding log records), followed by the reading from disk of the page containing the data to be accessed. This takes two or more disk accesses, and the time taken is several orders of magnitude more than the main-memory reference required in the best case. Hence transaction execution-time variance is very high and can be estimated only poorly. It is therefore difficult to plan schedules which need to finish within a deadline.

26.14 What is the purpose of compensating transactions? Present two examples of their use.

Answer: A compensating transaction is used to perform a semantic undo of changes made previously by committed transactions. For example, a person might deposit a check in their savings account. Then the database would be updated to reflect the new balance. Since it takes a few days for the check to clear, it might be discovered later that the check bounced, in which case a compensating transaction would be run to subtract the amount of the bounced check from the depositor's account. Another example of when a compensating transaction would be used is in a grading program. If a student's grade on an assignment is to be changed after it is recorded, a compensating program (usually an option of the grading program itself) is run to change the grade and redo averages, etc.

26.15 Explain the connections between a workflow and a long-duration transaction.

Answer: A long-duration transaction is still expected to finish within the time of a user interaction, whereas a workflow may last for a very long time. Long duration transactions can be aborted at any point, whereas a workflow cannot simply be rolled back, instead it has to be moved to an acceptable termination state.