COMPLETE VERSION

Database Systems

An Application-Oriented Approach

SECOND EDITION

Michael Kifer, Arthur Bernstein, Philip M. Lewis

Solutions Manual

Contents

PART ONE

Introduction

1

Overview of Databases and Transactions

EXERCISES

This chapter has no exercises.

2

The Big Picture

EXERCISES

2.1 Design the following two tables (in addition to that in Figure 2.1) that might be used in the Student Registration System. Note that the same student Id might appear in many rows of each of these tables.

a. A table implementing the relation CoursesRegisteredFor, relating a student's Id and the identifying numbers of the courses for which she is registered

Solution:

Id	CrsCode
111111111	CSE515
111111111	CSE505
111111111	CSE532
666666666	CSE532
666666666	CSE541
111223344	CSE504
987654321	CSE504
023456789	CSE515
123454321	CSE505

b. A table implementing the relation CoursesTaken, relating a student's Id, the identifying numbers of the courses he has taken, and the grade received in each course

Solution:

Id	CrsCode	Grade
111111111	CSE501	A

111111111	CSE533	B+
666666666	CSE505	A-
666666666	CSE541	C
111223344	CSE533	B-
987654321	CSE515	B+
023456789	CSE505	A
123454321	CSE532	B+

Specify the predicate corresponding to each of these tables.

Solution:
For the first table: Student X is registered for Course Y
For the second table: Student X has taken Course Y and gotten Grade Z

2.2 Write an SQL statement that

a. Returns the Ids of all seniors in the table STUDENT

Solution:

```
SELECT S.Id
FROM Student
WHERE S.Status = 'senior'
```

b. Deletes all seniors from STUDENT

Solution:

```
DELETE
FROM Student S
WHERE S.Status = 'senior'
```

c. Promotes all juniors in the table STUDENT to seniors

Solution:

```
UPDATE Student S
SET S.Status = 'senior'
WHERE S.Status = 'junior'
```

2.3 Write an SQL statement that creates the TRANSCRIPT table.

Solution:

```
CREATE TABLE Transcript  (
StudId INTEGER,
CrsCode CHAR(6),
Semester CHAR(6),
Grade CHAR(1),
PRIMARY KEY (StudId, CrsCode, Semester) )
```

2.4 Using the TRANSCRIPT table, write an SQL statement that

a. Deregisters the student with `Id = 123456789` from the course `CS305` for the fall of 2001

Solution:

```
DELETE
FROM Transcript
WHERE StudId = '123456789'
    AND CrsCode = 'CS305' AND Semester = 'F2001'
```

b. Changes to an A the grade assigned to the student with `Id = 123456789` for the course `CS305` taken in the fall of 2000

Solution:

```
UPDATE Transcript
SET Grade = 'A'
WHERE StudId = '123456789'
    AND CrsCode = 'CS305' AND Semester ='F2000'
```

c. Returns the Id of all students who took `CS305` in the fall of 2000

Solution:

```
SELECT StudId
FROM Transcript
WHERE CrsCode = 'CS305' AND Semester = 'F2000'
```

⋆**2.5** Given the relation MARRIED that consists of tuples of the form $\langle a, b\rangle$, where a is the husband and b is the wife, the relation BROTHER that has tuples of the form $\langle c, d\rangle$, where c is the brother of d, and the relation SIBLING, which has tuples of the form $\langle e, f\rangle$, where e and f are siblings, use SQL to define the relation BROTHER-IN-LAW, where tuples have the form $\langle x, y\rangle$ with x being the brother-in-law of y.
(*Hint*: This query can be represented as a union of three separate SQL queries. SQL provides the operator `UNION` to achieve this effect.)

Solution:
The first SQL query, below, describes the situation where someone is the brother of the wife and hence the brother-in-law of the husband. The second disjunct describes the situation where someone is the brother of the husband and hence the brother-in-law of the wife. The third disjunct describes the situation where, someone is the husband and hence the brother-in-law of all the wife's brothers and sisters.

```
(SELECT Brother.col1, Married.col1
 FROM MARRIED, BROTHER
 WHERE  Brother.col2 = Married.col2)
UNION
(SELECT Brother.col1, Married.col2
 FROM MARRIED, BROTHER
 WHERE  Brother.col2 = Married.col1)
UNION
(SELECT Married.col1, Sibling.col2
 FROM MARRIED, SIBLING
 WHERE  Sibling.col1 = Married.col2)
```

2.6 Write an SQL statement that returns the names (not the Ids) of all students who received an A in `CS305` in the fall of 2000.

Solution:

```
SELECT Name
FROM Student, Transcript
WHERE StudId = Id AND Grade = 'A'
     AND CrsCode = 'CS305' AND Semester = 'F2000'
```

2.7 State whether or not each of the following statements could be an integrity constraint of a checking account database for a banking application. Give reasons for your answers.

a. The value stored in the `balance` column of an account is greater than or equal to `$0`.

Solution:
Yes. It describes a constraint on a snapshot of the database.

b. The value stored in the `balance` column of an account is greater than it was last week at this time.

Solution:
No. It does not describe a snapshot.

c. The value stored in the `balance` column of an account is `$128.32`.

Solution:
No. The balance will change.

d. The value stored in the `balance` column of an account is a decimal number with two digits following the decimal point.

Solution:
Yes. It is a domain constraint.

e. The `social_security_number` column of an account is defined and contains a nine-digit number.

Solution:
Yes. It is a domain constraint.

f. The value stored in the `check_credit_in_use` column of an account is less than or equal to the value stored in the `total_approved_check_credit` column. (These columns have their obvious meanings.)

Solution:
Yes. It describes a constraint on a snapshot

2.8 State five integrity constraints, other than those given in the text, for the database in the Student Registration System.

Solution:

1. The courses for which the student enrolled (registered) must be offered this semester(next semester).
2. An instructor cannot be assigned to two courses taught at the same time in the same semester.
3. Two courses are not taught in the same room at the same time in a given semester.
4. No student must be registered (enrolled) in two courses taught at the same hour.
5. No student must be allowed to register for more than 20 credits in a given semester.
6. The room assigned to a course must have at least as many seats as the maximum allowed enrollment for the course.

2.9 Give an example in the Student Registration System where the database satisfies the integrity constraints IC0–IC3 but its state does not reflect the state of the real world.

Solution:
We register a student, but do not change the database.

2.10 State five (possible) integrity constraints for the database in an airline reservation system.

Solution:

1. The flight for which a person makes a reservation must be on the schedule.
2. The number of reservations on each flight must not exceed the number of seats on the plane.
3. A passenger cannot order two meals
4. The number of meals ordered must equal the number of passengers who ordered meals
5. The same seat on the plane must not be reserved for two passengers.

2.11 A reservation transaction in an airline reservation system makes a reservation on a flight, reserves a seat on the plane, issues a ticket, and debits the appropriate credit card account. Assume that one of the integrity constraints of the reservation database is that the number of reservations on each flight does not exceed the number of seats on the plane. (Of course, many airlines purposely over-book and so do not use this integrity constraint.) Explain how transactions running on this system might violate

a. Atomicity

Solution:
A passenger makes a reservation and reserves a seat. The transaction records the reservation, but the system crashes before it records the seat reservation.

b. Consistency

Solution:
Flight is over-booked (More reservations are made than there are seats on the airplane.)

c. Isolation

Solution:
The same seat is given to two people because of a particular interleaving of the reservation transactions.

d. Durability

Solution:
A reservation is made; the system crashes; the system forgets the reservation

2.12 Describe informally in what ways the following events differ from or are similar to transactions with respect to atomicity and durability.

a. A telephone call from a pay phone (Consider line busy, no answer, and wrong number situations. When does this transaction "commit?")

Solution:
Commit occurs when caller hangs up. Billing information is durable. For line busy or no answer, the transaction aborts. For a wrong number the transaction commits, but later is compensated for by returning the callers money (Read about compensation later in the book,)

b. A wedding ceremony (Suppose that the groom refuses to say "I do." When does this transaction "commit?")

Solution:
Commit occurs when license is signed. Marriage is durable (hopefully).

c. The purchase of a house (Suppose that, after a purchase agreement is signed, the buyer is unable to obtain a mortgage. Suppose that the buyer backs out during the closing. Suppose that two years later the buyer does not make the mortgage payments and the bank forecloses.)

Solution:
Various levels of commit; Every time someone signs something. For example, when purchaser makes an offer to purchase and includes a deposit, he is committed to

either purchase the house at that price (assuming he is approved for the mortgage) or forfeit the deposit. If he is not approved for the mortgage, he is no longer committed to purchase the house and gets his deposit back. If he does not pay his mortgage payment the transaction is compensated for when the bank forecloses.

d. A baseball game (Suppose that it rains.)

Solution:
Commit occurs after game is official. If it rains before the game is official, the game is aborted.

2.13 Assume that, in addition to storing the grade a student has received in every course he has completed, the system stores the student's cumulative GPA. Describe an integrity constraint that relates this information. Describe how the constraint would be violated if the transaction that records a new grade were not atomic.

Solution:
The integrity constraint is that the GPA stored in the database is the GPA of the course grades stored. That constraint could be violated if a transaction updated a course grade and aborted before it could update the GPA.

2.14 Explain how a lost update could occur if, under the circumstances of the previous problem, two transactions that were recording grades for a particular student (in different courses) were run concurrently.

Solution:
The first transaction reads the course grades before the second updated its grade and then updates the GPA. The second transaction reads the course grades before the first updated its grade and then updates the GPA. The first update of the GPA is lost and the final one is incorrect.

PART TWO

Database Management

3

The Relational Data Model

EXERCISES

3.1 Define data atomicity as it relates to the definition of relational databases. Contrast data atomicity with transaction atomicity as used in a transaction processing system.

Solution:
These concepts are not related. Data atomicity means that the relational model does not specify any means for looking into the internal structure of the values, so they appear as indivisible to the relational operators. Transaction atomicity means that the system must ensure that either the transaction runs to completion or, if it does not complete, it has no effect at all. It does not mean that the transaction must be indivisible, but it is a type of all-or-none execution.

3.2 Prove that every relation has a key.

Solution:
Since relations are sets and, thus, cannot have identical elements, the set of all attributes in a relation must be a superkey. If this is not a minimal superkey, some strict subset of it must also be a superkey. Since the number of the attributes in every relation is finite, we will eventually get a minimal superkey, i.e., a key of the relation.

3.3 Define the following concepts:

a. Key

Solution:
A *key*, $\texttt{key}(\bar{K})$, associated with a relation schema, $\mathbf{S}$, is a minimal (by inclusion) subset $\bar{K}$ of attributes of $\mathbf{S}$ with the following property: An instance $\mathbf{s}$ of $\mathbf{S}$ *satisfies* $\texttt{key}(\bar{K})$ if it does not contain a pair of distinct tuples whose values agree on *all* of the attributes in $\bar{K}$.

b. Candidate key

Solution:
Every key of a relation is also called a candidate key for that relation

c. Primary key

Solution:
One of the keys of a relation is designated as primary key.

d. Superkey

Solution:
A superkey is a set of attributes in a relation that contains a key of that relation

3.4 Define

a. Integrity constraint

Solution:
An integrity constraint is an application-specific restriction on the tuples in one or several relations.

b. Static, as compared with dynamic, integrity constraint

Solution:
Static integrity constraints restrict the legal instances of a database. Examples of static ICs are domain constraints, key constraints, foreign-key constraints, etc. Dynamic integrity constraints restrict the evolution (over time) of legal instances of a database, for instance, a salary increase should not exceed 5%.

c. Referential integrity

Solution:
A referential integrity constraint is a requirement that the referenced tuple must exist.

d. Reactive constraint

Solution:
A reactive constraint is a static constraint with a trigger attached. The trigger specifies what to do if the constraint is violated by an update.

e. Inclusion dependency

Solution:
An inclusion dependency is a statement "$\mathbf{S}(\bar{F})$ references $\mathbf{T}(\bar{K})$", which states that for every tuple $s \in \mathbf{s}$, there is a tuple $t \in \mathbf{t}$ that has the same values over the attributes in $\bar{K}$ as does s over the corresponding attributes in $\bar{F}$.

f. Foreign-key constraint

Solution:
A foreign-key constraint is an inclusion dependency in which the set of attributes referred to is a candidate key in the referenced relation.

3.5 Looking at the data that happens to be stored in the tables for a particular application at some particular time, explain whether or not you can tell

a. What the key constraints for the tables are

Solution:
No, in one particular instance of the table, a particular attribute might uniquely identify the rows (and thus appear to be a key), but in other instances it might not.

b. Whether or not a particular attribute forms a key for a particular table

Solution:
You cannot tell whether a particular attribute is a key (for the reasons mentioned in (a)), but you can sometimes tell that a particular attribute cannot form a key by itself (if two distinct rows have the same value over that attribute).

c. What the integrity constraints for the application are

Solution:
No. Again, one cannot determine integrity constraints just by looking at database instances.

d. Whether or not a particular set of integrity constraints is satisfied

Solution:
Yes. Simply check if every constraint in the set is satisfied in the given instance.

3.6 We state in the book that once constraints have been specified in the schema, it is the responsibility of the DBMS to make sure that they are not violated by the execution of any transactions. SQL allows the application to control when each constraint is checked. If a constraint is in *immediate mode*, it is checked immediately after the execution of any SQL statement in a transaction that might make it false. If it is in *deferred mode*, it is not checked until the transaction requests to commit. Give an example where it is necessary for a constraint to be in deferred mode.

Solution:
Suppose the constraint states that the value of one attribute, A, is the sum of the values of two other attributes, B and C. If we want to increment the value of B, we must also increment the value of A in the same transaction. But no matter in what order we do the incrementing, the constraint will be false between the two statements that do the incrementing, and if we used immediate mode checking, the transaction would abort when the first increment statement was attempted.

3.7 Suppose we do not require that all attributes in the primary key are non-null and instead request that, in every tuple, at least one key (primary or candidate) does not have nulls in it. (Tuples can have nulls in other places and the non-null key can be different for different tuples.) Give an example of a relational instance that has two distinct tuples that *might* become one once the values for all nulls become known (that is, are replaced with real values). Explain why this is not possible when one key (such as the primary key) is designated to be non-null for all tuples in the relation.

Solution:
Let the relation have attributes `A` and `B`, each of which is a key. The tuples can be $\langle a, \texttt{NULL}\rangle$ and $\langle \texttt{NULL}, b\rangle$. If the first `NULL` becomes b and the second a then these tuples become the same.

If all tuples are non-`NULL` over the same key, then they must differ over that key somewhere and thus they cannot become the same regardless of what is substituted for nulls in other places.

3.8 Use SQL DDL to specify the schema of the Student Registration System fragment shown in Figure 3.4, including the constraints in Figure 3.6 and Example 3.2.2. Specify SQL domains for attributes with small numbers of values, such as `DeptId` and `Grade`.

Solution:

```
CREATE TABLE  STUDENT (
     Id            INTEGER,
     Name          CHAR(20),
     Address       CHAR(50),
     Status        CHAR(10)
     PRIMARY KEY (Id) )
```

```
CREATE TABLE  PROFESSOR (
     ProfId        INTEGER,
     Name          CHAR(20),
     DeptId        DEPARTMENTS,
     PRIMARY KEY (ProfId) )
```

```
CREATE TABLE  COURSE (
     CrsCode       CHAR(6),
     DeptId        DEPARTMENTS,
     CrsName       CHAR(20),
     Descr         CHAR(100),
     PRIMARY KEY (CrsCode),
     UNIQUE (DeptId, CrsName) )
```

```
CREATE TABLE  TRANSCRIPT (
     StudId        INTEGER,
     CrsCode       CHAR(6),
     Semester      SEMESTERS,
     Grade         GRADES,
     PRIMARY KEY (StudId, CrsCode, Semester),
     FOREIGN KEY (StudId) REFERENCES STUDENT (Id)
          ON DELETE NO ACTION
          ON UPDATE CASCADE,
     FOREIGN KEY (CrsCode) REFERENCES COURSE (CrsCode)
          ON DELETE NO ACTION
          ON UPDATE CASCADE
     FOREIGN KEY (CrsCode, Semester) REFERENCES
                              TEACHING (CrsCode, Semester)
          ON DELETE NO ACTION
          ON UPDATE CASCADE )
```

```
CREATE TABLE  Teaching (
    ProfId        INTEGER,
    CrsCode       CHAR(6),
    Semester      Semesters,
    PRIMARY KEY (CrsCode, Semester),
    FOREIGN KEY (ProfId) REFERENCES Professor(Id)
        ON DELETE NO ACTION
        ON UPDATE CASCADE,
    FOREIGN KEY (CrsCode) REFERENCES Course (CrsCode)
        ON DELETE SET NULL
        ON UPDATE CASCADE )
```

The Grades domain is defined in Section 3.3.6. The domain of departments is defined below.

```
CREATE DOMAIN  Departments CHAR(3)
    CHECK ( VALUE IN ('CS','MAT','EE','MUS','PHY','CHE') )
```

```
CREATE DOMAIN  Semesters CHAR(6)
    CHECK ( VALUE IN ('fall', 'spring', 'summer') )
```

3.9 Consider a database schema with four relations: Supplier, Product, Customer, and Contracts. Both the Supplier and the Customer relations have the attributes `Id`, `Name`, and `Address`. An `Id` is a nine-digit number. Product has `PartNumber` (an integer between 1 and 999999) and `Name`. Each tuple in the Contracts relation corresponds to a contract between a supplier and a customer for a specific product in a certain quantity for a given price.

a. Use SQL DDL to specify the schema of these relations, including the appropriate integrity constraints (primary, candidate, and foreign key) and SQL domains.

Solution:

```
CREATE TABLE  Supplier (
    Id            Suppliers,
    Name          CHAR(20),
    Address       CHAR(50),
    PRIMARY KEY (Id) )
```

```
CREATE TABLE  Customer (
    Id            Customers,
    Name          CHAR(20),
```

```
        Address       CHAR(50),
        PRIMARY KEY (Id) )
```

```
    CREATE TABLE  Product (
        PartNumber    Products,
        Name          CHAR(50),
        PRIMARY KEY (PartNumber) )
```

```
    CREATE TABLE  Contract (
        Customer      Customers,
        Supplier      Suppliers,
        Product       Products,
        Quantity      INTEGER,
        Price         INTEGER,
        PRIMARY KEY (Customer, Supplier, Product),
        FOREIGN KEY (Customer) REFERENCES Customer(Id)
            ON DELETE NO ACTION
            ON UPDATE CASCADE,
        FOREIGN KEY (Supplier) REFERENCES Supplier(Id)
            ON DELETE NO ACTION
            ON UPDATE CASCADE ),
        FOREIGN KEY (Product) REFERENCES Product(PartNumber)
            ON DELETE NO ACTION
            ON UPDATE CASCADE )
```

```
    CREATE DOMAIN  Suppliers INTEGER
        CHECK ( VALUE =< 999999999 AND VALUE > 0 )
```

The domain Customers is defined identically. The domain Products is similar, except that 999999 is used instead of 999999999.

b. Specify the following constraint as an SQL assertion: *there must be more contracts than suppliers.*

Solution:

```
    CREATE ASSERTION  ContractsShaltExceedSuppliers
        CHECK ( (SELECT COUNT(*) FROM Supplier)
                    < (SELECT COUNT(*) FROM Contract) ) )
```

3.10 You have been hired by a video store to create a database for tracking DVDs and videocassettes, customers, and who rented what. The database includes these relations: RentalItem, Customer, and Rentals. Use SQL DDL to specify the schema for this database, including all the applicable constraints. You are free to choose reasonable attributes for the first two relations. The relation Rentals is intended to describe who rented what and should have these attributes: `CustomerId`, `ItemId`, `RentedFrom`, `RentedUntil`, and `DateReturned`.

Solution:

3.11 You are in a real estate business renting apartments to customers. Your job is to define an appropriate schema using SQL DDL. The relations are Property(`Id`, `Address`, `NumberOfUnits`), Unit(`ApartmentNumber`, `PropertyId`, `RentalPrice`, `Size`), Customer (choose appropriate attributes), Rentals (choose attributes; this relation should describe who rents what, since when, and until when), and Payments (should describe who paid for which unit, how much, and when). Assume that a customer can rent more than one unit (in the same or different properties) and that the same unit can be co-rented by several customers.

Solution:
The students must recognize that the key of Unit is ⟨`ApartmentNumber`, `PropertyId`⟩ and not just `ApartmentNumber`. The students should also recognize that both Rentals and Payments should include attributes that reference ⟨`ApartmentNumber`, `PropertyId`⟩. In addition, neither ⟨`ApartmentNumber`, `PropertyId`⟩ nor customer should be a key in Payments.

3.12 You love movies and decided to create a personal database to help you with trivia questions. You chose to have the following relations: Actor, Studio, Movie, and PlayedIn (which actor played in which movie). The attributes of Movie are `Name`, `Year`, `Studio`, and `Budget`. The attributes of PlayedIn are `Movie` and `Actor`. You are free to choose the attributes for the other relations as appropriate. Use SQL DDL to design the schema and all the applicable constraints.

Solution:

3.13 You want to get rich by operating an auction Web site, similar to eBay, at which students can register used textbooks that they want to sell and other students can bid on purchasing those books. The site is to use the same proxy bidding system used by eBay (*http://www.ebay.com*).

Design a schema for the database required for the site. In the initial version of the system, the database must contain the following information:

1. For each book being auctioned: name, authors, edition, ISBN number, bookId (unique), condition, initial offering price, current bid, current maximum bid, auction start date and time, auction end date and time, userId of the seller, userId of the current high bidder, and an indication that the auction is either currently active or complete
2. For each registered user: name, userId (unique), password, and e-mail address

Solution:

3.14 You want to design a room-scheduling system that can be used by the faculty and staff of your department to schedule rooms for events, meetings, classes, etc. Design a schema for the database required for the system. The database must contain the following information:

1. For each registered user: name, userId (unique), password, and e-mail address
2. For each room: room number, start date of the event, start time of the event, duration of the event, repetition of the event (once, daily, weekly, monthly, mon-wed-fri, or tues-thurs), and end date of repetitive event

Solution:

3.15 Design the schema for a library system. The following data should either be contained directly in the system or it should be possible to calculate it from stored information:

1. About each patron: name, password, address, Id, unpaid fines, identity of each book the patron has currently withdrawn, and each book's due date
2. About each book: ISBN number, title, author(s), year of publication, shelfId, publisher, and status (on-shelf, on-loan, on-hold, or on-loan-and-on-hold). For books on-loan the database shall contain the Id of the patron involved and the due date. For books on hold the database shall contain a list of Ids of patrons who have requested the book.
3. About each shelf: shelfId and capacity (in number of books)
4. About each author: year of birth

The system should enforce the following integrity constraints. You should decide whether a particular constraint will be embedded in the schema, and, if so, show how this is done or will be enforced in the code of a transaction.

1. The number of books on a shelf cannot exceed its capacity.
2. A patron cannot withdraw more than two books at a time.
3. A patron cannot withdraw a book if his/her unpaid fines exceed $5. Assume that a book becomes overdue after two weeks and that it accumulates a fine at the rate of $.10 a day.

Solution:

3.16 Suppose that the fragment of the Student Registration System shown in Figure 3.4 has two user accounts: `Student` and `Administrator`. Specify the permissions appropriate for these user categories using the SQL GRANT statement.

Solution:
An administrator should be able to read and modify all information in STUDENT, PROFESSOR, COURSE, TRANSCRIPT, and TEACHING — if your university is like ours, only administrators can change information. Here is an example:

```
GRANT SELECT, INSERT, DELETE, UPDATE ON TRANSCRIPT
    TO Administrator WITH GRANT OPTION
```

Assuming that Ids and addresses are considered private, a student should be able to see names and student status in the STUDENT relation, professors and departments in the PROFESSOR relation, everything in the COURSE relation, nothing in TRANSCRIPT, and, perhaps, `CrsCode` and `Semester` in TEACHING. Here is an example:

```
GRANT SELECT (CrsCode, Semester) ON TEACHING
    TO Student
```

3.17 Suppose that the video store of Exercise 3.10 has the following accounts: `Owner`, `Employee`, and `User`. Specify GRANT statements appropriate for each account.

Solution:
This question is formulated with a pitfall: the student is supposed to recognize that users should not be granted any privileges, because a user should be allowed to see only the data pertaining to that particular user.

`Employee` and `Owner` can have the same permissions. In any case, `Employee` should not be allowed to delete records. `Owner` might be allowed all privileges.

3.18 Explain why the REFERENCES privilege is necessary. Give an example of how it is possible to obtain partial information about the contents of a relation by creating foreign-key constraints referencing that relation.

Solution:
Suppose `Phone` number is a candidate key in the LOAN relation. One can then find out who has loans by creating a relation, PROBE, with the attribute `Phone`, which REFERENCES the `Phone` attribute in LOAN. Then, by exhaustively inserting all possible phone numbers and recording which ones produce errors, such a user would find out the phone numbers of all people who have loans. Reverse telephone lookup using a number of services on the Web will make it possible to also find names and addresses of those people.

4

Conceptual Modeling of Databases with Entity-Relationship Diagrams and the Unified Modeling Language

EXERCISES

4.1 Suppose that you decide to convert IsA hierarchies into the relational model by adding a new attribute (such as `Status` in the case of STUDENT entities, as described on page 91—the second option for representing IsA hierarchies). What kind of problems exist if subentities are not disjoint (e.g., if a secretary can also be a technician)? What problems exist if the covering constraint does not hold (e.g., if some employees are not classified as either secretary or technician)?

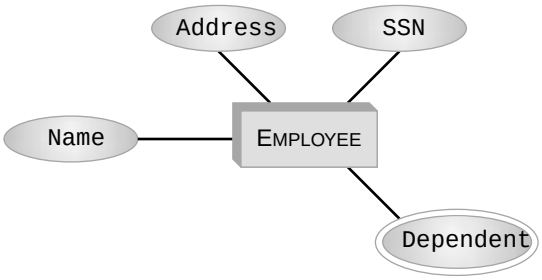
Solution:
If sub-entities are not disjoint then some tuples would be duplicated (except for the value of the new attribute that simulates the hierarchy). If the covering constraint does not hold, then we would have to use NULs to represent the tuples that do not belong to any one of the subentities.

4.2 Construct your own example of an E-R or UML diagram whose direct translation into the relational model has an anomaly similar to that of the PERSON entity (see the discussion regarding Figure 4.13 on page 86).

Solution:
An example is the EMPLOYEE relation, depicted below, in which an employee can have several dependents. The E-R diagram is depicted in Figure 4.1.

SSN	Name	Address	Dependent
111111111	John Doe	123 Main St.	Joan
111111111	John Doe	123 Main St.	Kim
111111111	John Doe	123 Main St.	Kathy
555666777	Mary Doe	7 Lake Dr.	Peter
555666777	Mary Doe	7 Lake Dr.	Tom
987654321	Bart Simpson	Fox 5 TV	Mary

FIGURE 4.1



4.3 Represent the IsA hierarchy in Figure 4.6, page 80, in the relational model. For each IsA relationship discuss your choice of the representation technique Discuss the circumstances in which an alternative representation (to the one you have chosen) would be better.

Solution:
To represent the relationship IsA EMPLOYEE, we would add an additional status attribute to the schema of EMPLOYEE. This is because the entity types SECRETARY and TECHNICIAN are likely to be disjoint. Similar considerations apply to IsA STUDENT.

The downside of this representation is that some tuples will have null values over the attribute `Specialization` (in case of secretaries) and `Major` and `Advisor` (in case, say, of freshmen).

The relationship IsA PERSON is different from the above two in that its subentities, EMPLOYEE and STUDENT, can have fairly large intersection. Thus, to avoid the duplication associated with the use of the `Status` attribute, we would use three relations, PERSON`(SSN,Name,DOB)`, EMPLOYEE`(SSN,Department,Salary)`, and STUDENT`(SSN,GPA,StartDate)`.

4.4 Suppose, in Figure 4.8, the PROFESSOR entity did not participate in the relationship WORKSIN, but the arrow between them was still present. Would it make sense to merge PROFESSOR with WORKSIN during translation into the relational model? What kind of problems can arise here? Are they serious problems?

Solution:
The problem with such a merge is that there might be null values over the attributes of WORKSIN. This is most likely not a serious problem in this example.

4.5 Translate the brokerage example of Section 4.7 into an SQL schema. Use the necessary SQL machinery to express all constraints specified in the E-R model.

Solution:

We will start with the diagram in Figure 4.30. In translating this E-R diagram, we choose to combine ISHANDLEDBY with ACCOUNT, WORKSIN with BROKER, and OFFICE with MANAGEDBY. The rationale for these combinations is that in each case the entity type is a key role in the corresponding relationship. The reason why MANAGEDBY is combined with OFFICE rather than BROKER (which also happens to be a key role in the MANAGEDBY relationship) is that it is more likely for an office to have a manager than it is for a broker to be a manager. Therefore, combining OFFICE

with ManagedBy into one table is unlikely to lead to many null values being stored with the Office table. In contrast, combining ManagedBy with Broker is likely to cause a situation where the value of the attribute `Manages` of the table Broker would be mostly `NULL`.

```
CREATE TABLE Client (
Name CHAR(20) NOT NULL,
Id         INTEGER,
WorkPhone# CHAR(10),
PRIMARY KEY (Id)
)

CREATE TABLE Account (
Account#   INTEGER,
DateOpened DATE,
Status     CHAR(1),
-- We choose to keep Office here rather than with HasAccount
Office     CHAR(10) NOT NULL,
Broker     INTEGER,
PRIMARY KEY (Account#),
FOREIGN KEY Office REFERENCES Office(Phone#)
 ON UPDATE CASCADE
 OD DELETE NO ACTION,
FOREIGN KEY Broker REFERENCES Broker(Id)
 ON UPDATE CASCADE
 ON DELETE SET NULL
)

CREATE TABLE Office (
Phone#   CHAR(10),
Address  CHAR(50) NOT NULL,
Manager  INTEGER,
ManagedSince DATE,
PRIMARY KEY (Phone#),
UNIQUE (Address),
FOREIGN KEY Manager REFERENCES Broker(Id)
ON UPDATE CASCADE
ON DELETE SET NULL
)

CREATE TABLE HasAccount (
Client  INTEGER NOT NULL,
Account INTEGER NOT NULL,
FOREIGN KEY Client REFERENCES Client(Id)
 ON UPDATE CASCADE
```

```
 ON DELETE NO ACTION,
FOREIGN KEY Account REFERENCES Account(Account#)
 ON UPDATE CASCADE
 ON DELETE CASCADE
)

CREATE TABLE Broker (
Id       INTEGER,
Name     CHAR(20) NOT NULL,
WorksIn  CHAR(10) NOT NULL,
WorksSince DATE,
PRIMARY KEY (Id)
FOREIGN KEY WorksIn REFERENCES Office(Phone#)
ON UPDATE CASCADE
ON DELETE CASCADE
)

CREATE TABLE BrokerPhone (
Broker         INTEGER NOT NULL,
PhoneExtension CHAR(10) NOT NULL,
FOREIGN KEY Broker REFERENCES Broker(Id)
 ON UPDATE CASCADE
 ON DELETE CASCADE
)
```

Note that there is a reason why we split off the `PhoneExtension` attribute from the Broker relation. If we join this attribute to Broker, then `Id` would no longer be a candidate key, so one of the foreign key constraints in Accounts would not be possible.

We still need to write a constraint to ensure that a client cannot have two separate accounts in the same office. We will use an SQL assertion for that. This last constraint might be a little hard to write with the amount of knowledge of SQL that the students have at this point, so this part can be used for extra credit or it can be postponed until Chapter 5.

```
CREATE ASSERTION OneAcctPerClientOffice
CHECK NOT EXISTS (
      SELECT C.Id
      FROM  Client C,
            Account A1, HasAccount H1,
            Account A2, HasAccount H2
      WHERE  C.Id = H1.Client AND C.Id = H2.Client
             AND H1.Account = A1.Account#
             AND H2.Account = A2.Account#
             AND A1.Office <> A2.Office
```

```
)
```

We now turn to the representation of the trading information in Figure 4.31. Because TRANSACTION is a key role in TRADE, it is convenient to combine the two. Thus, we end up with the following tables:

```
CREATE TABLE TRANSACTION (
Id     INTEGER,
Date   DATE NOT NULL,
Time   TIME NOT NULL,
PricePerShare  DECIMAL(6) NOT NULL,
NumberOfShares INTEGER NOT NULL,
StockSymbol    CHAR(5) NOT NULL,
Account        INTEGER NOT NULL,
PRIMARY KEY (Id),
FOREIGN KEY StockSymbol REFERENCES STOCK
   ON UPDATE NO ACTION
   ON DELETE NO ACTION,
FOREIGN KEY Account REFERENCES ACCOUNT(Account#)
   ON UPDATE CASCADE
   ON DELETE NO ACTION
)

CREATE TABLE POSITION (
Account     INTEGER,
StockSymbol CHAR(5),
Amount      INTEGER NOT NULL,
PRIMARY KEY (Account, Stock),
FOREIGN KEY Account REFERENCES ACCOUNT(Account#)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
FOREIGN KEY StockSymbol REFERENCES STOCK
    ON UPDATE CASCADE
    ON DELETE CASCADE
)

CREATE TABLE STOCK (
StockSymbol  CHAR(5),
CompanyName  CHAR(50) NOT NULL,
CurrentPrice DECIMAL(6) NOT NULL,
PRIMARY KEY (StockSymbol)
)
```

4.6 Identify the navigation traps present in the diagram of Figure 4.29, page 107.

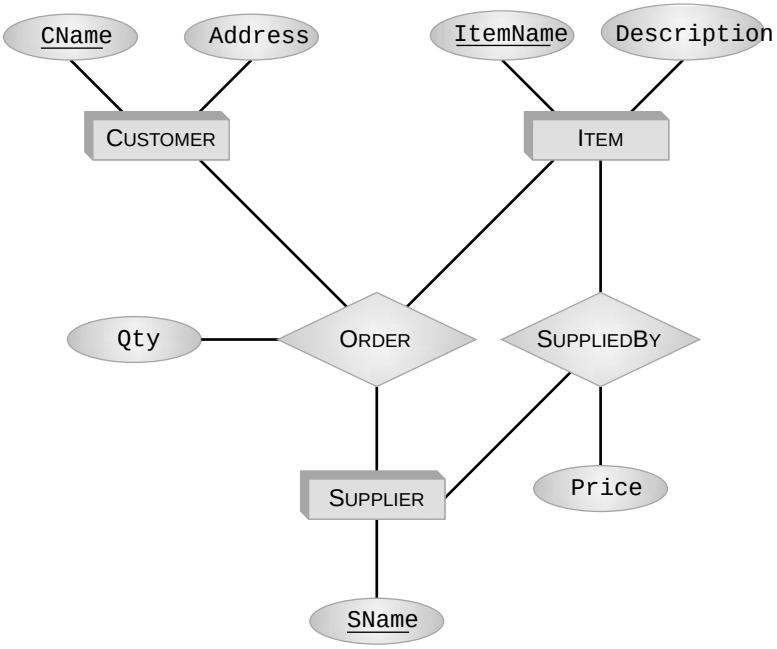


FIGURE 4.2

Solution:
One navigation trap is when we follow the links ACCOUNT to ISHANDLEDBY to OFFICE to WORKSIN to BROKER, which can associate accounts with wrong brokers. Another trap occurs when we travel through MANAGEDBY instead of WORKSIN.

4.7 Consider the following database schema:

- SUPPLIER(`SName`, `ItemName`, `Price`)—supplier `SName` sells item `ItemName` at `Price`
- CUSTOMER(`CName`, `Address`)—customer `CName` lives at `Address`.
- ORDER(`CName`, `SName`, `ItemName`, `Qty`)—customer `CName` has ordered `Qty` of item `ItemName` from supplier `SName`.
- ITEM(`ItemName`, `Description`)—information about items.

(a) Draw the E-R diagram from which the above schema might have been derived. Specify the keys.

Solution:
See the E-R diagram in Figure 4.2.

The key of SUPPLIER is `SName`, `ItemName`; of CUSTOMER is `CName`; of ITEM is `ItemName`; and of ORDER is `CName`, `SName`, `ItemName` (and, possibly, `Qty`, if multiple orders for the same item from the same supplier are allowed).

(b) Suppose now that you want to add the following constraint to this diagram: *Every item is supplied by some supplier*. Modify the diagram to accommodate

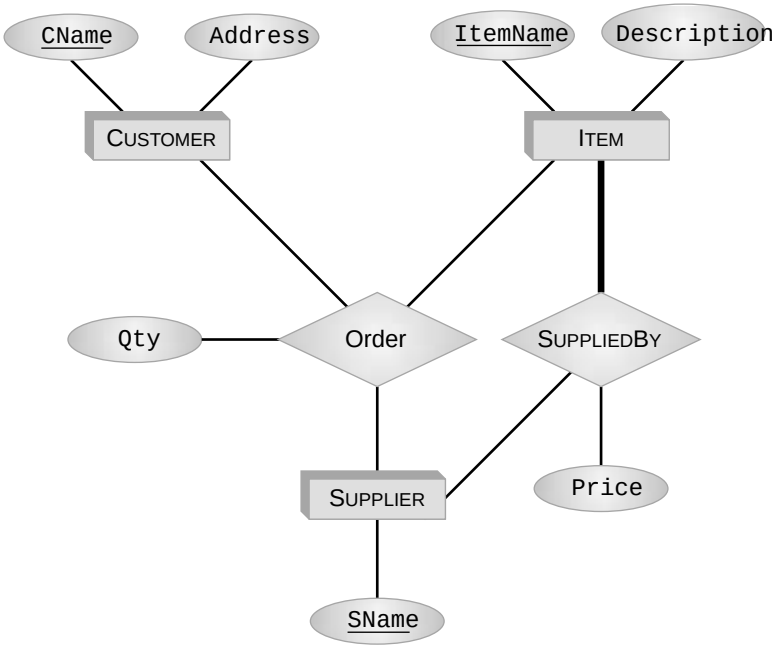


FIGURE 4.3

this constraint. Also show how this new diagram can be translated back to the relational model.

Solution:
See the E-R diagram in Figure 4.3.

One possible relational tables corresponding to this E-R diagram:

```
CREATE TABLE  CUSTOMER (
     CName          CHAR(10) NOT NULL,
     Address        CHAR(20),
     PRIMARY KEY(CName));

CREATE TABLE  SUPPLIER (
     SName          CHAR(10) NOT NULL,
     PRIMARY KEY(SName));

CREATE TABLE  ITEM (
     ItemName       CHAR(10) NOT NULL,
     Description    CHAR(20),
     Price          FLOAT,
     SName          CHAR(10) NOT NULL,
```

```
    PRIMARY KEY(ItemName, SName),
    FOREIGN KEY(SName) references SUPPLIER);
CREATE TABLE  ORDER (
    CName         CHAR(10) NOT NULL,
    SName         CHAR(10) NOT NULL,
    ItemName      CHAR(10) NOT NULL,
    Qty           INTEGER,
    PRIMARY KEY(CName, SName, ItemName),
    FOREIGN KEY(CName) REFERENCES CUSTOMER,
    FOREIGN KEY(SName) REFERENCES SUPPLIER,
    FOREIGN KEY(ItemName) REFERENCES ITEM);
```

(c) Repeat parts (a) and (b) in UML.

Solution:

4.8 Perform conceptual design of the operations of your local community library. The library has books, CDs, tapes, and so forth, which are lent to library patrons. The latter have accounts, addresses, and so forth. If a loaned item is overdue, it accumulates penalty. Some patrons are minors, so they must have sponsoring patrons who are responsible for paying penalties (or replacing a book in case of a loss).

a. Use the E-R approach.
b. Use UML.

Solution:
a. See the E-R diagram in Figure 4.4.

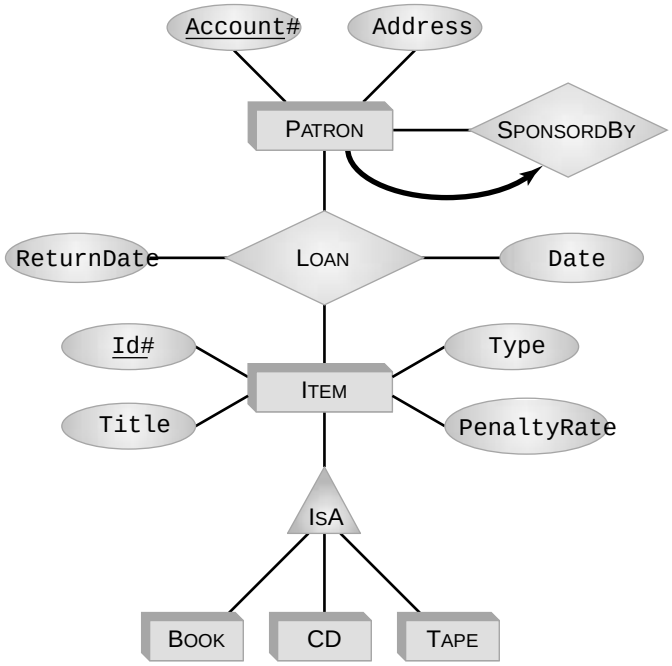
4.9 A real estate firm keeps track of the houses for sale and customers looking to buy houses. A house for sale can be *listed* with this firm or with a different one. Being "listed" with a firm means that the house owner has a contract with an agent who works for that firm. Each house on the market has price, address, owner, and a list of features, such as the number of bedrooms, bathrooms, type of heating, appliances, size of garage, and the like. This list can be different for different houses, and some features can be present in some houses but missing in others. Likewise, each customer has preferences that are expressed in the same terms (the number of bedrooms, bathrooms, etc.). Apart from these preferences, customers specify the price range of houses they are interested in. Perform conceptual design for this enterprise.

a. Use the E-R approach.
b. Use UML.

Solution:
a. See the E-R diagram in Figure 4.5.

4.10 A supermarket chain is interested in building a decision support system with which they can analyze the sales of different products in different supermarkets at different times. Each supermarket is in a city, which is in a state, which is in a region. Time can be measured in days, months, quarters, and years. Products have names and categories (produce, canned goods, etc.).

FIGURE 4.4



a. Design an E-R diagram for this application.
b. Do the same in UML.

Solution:
a. See the E-R diagram in Figure 4.6.

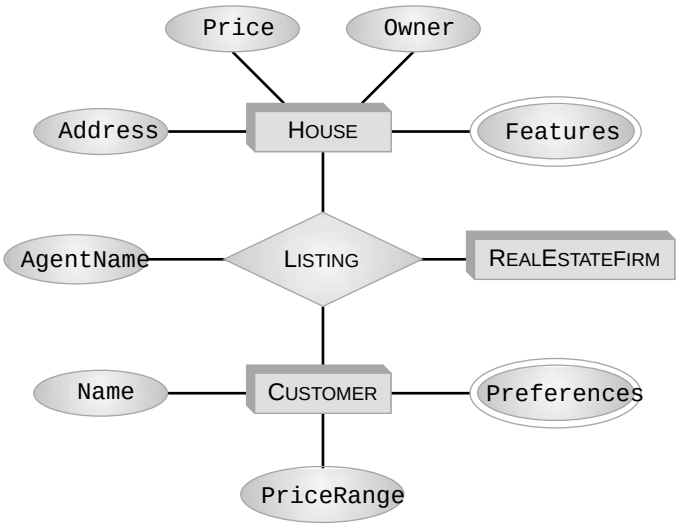
4.11 Modify the E-R diagram for the Student Registration System in Figure 4.33 on page 114 to include co-requisite and prerequisite relationships that exist over multiple periods of time. Each period begins in a certain semester and year and ends in a certain semester and year, or it continues into the present. Modify the translation into the relational model appropriately.

Solution:
Instead of the relationship REQUIRES with a single attribute `EnforcedSince`, we use two attributes, `EnforcedSince` and `EnforcedUntil`. If the latter is `NULL`, then the prerequisite is in force at present time. A similar relationship is needed for co-requisites. The corresponding relational representation requires that we add the attribute `EnforcedUntil` to the definition of the table REQUIRES and create a similar table for co-requisites.

```
    EnforcedUntil DATE,
```

Note that unlike `EnforcedSince`, we do not use the `NOT NULL` specification, because `NULL` is meaningful — it represents prerequisites that are valid at present time.

FIGURE 4.5



4.12 Modify the E-R diagram for the Student Registration System in Figure 4.33 on page 114 to include information about the student majors and the majors allowed in courses. A student can have several majors (which are codes of the various programs in the university, such as `CSE`, `ISE`, `MUS`, `ECO`). A course can also have several admissible majors, or the list of admissible majors can be empty. In the latter case, anyone is admitted into the course. Express the constraint that says that a course with restrictions on majors can have only those students who hold one of the allowed majors.

Alas, in full generality this constraint can be expressed only as an SQL assertion (introduced in Section 3.3) that uses features of Section 5.2 (which we have yet to study). However, it is possible to express this constraint under the following simplifying assumption: when a student registers for a course, she must declare the major toward which the course is going to be taken, and this declared major is checked against the admissible majors.

Modify the relation schema in Figures 4.34 and 4.35 to reflect this simplifying assumption and then express the aforesaid integrity constraint.

Solution:
The STUDENT entity type gets a new set-valued attribute `Major`, and similarly for COURSES.

Under the simplifying assumption mentioned above, the relations STUDENT, COURSES, and TRANSCRIPT each gets a new attribute, `Major`. The constrain then can be expressed as a foreign-key constraint on the TRANSCRIPT relation:

```
FOREIGN KEY (StudId, Major)
    REFERENCES STUDENT(Id, Major)
```

4.13 Redo the E-R diagram of the Student Registration System (Figure 4.33) in UML.

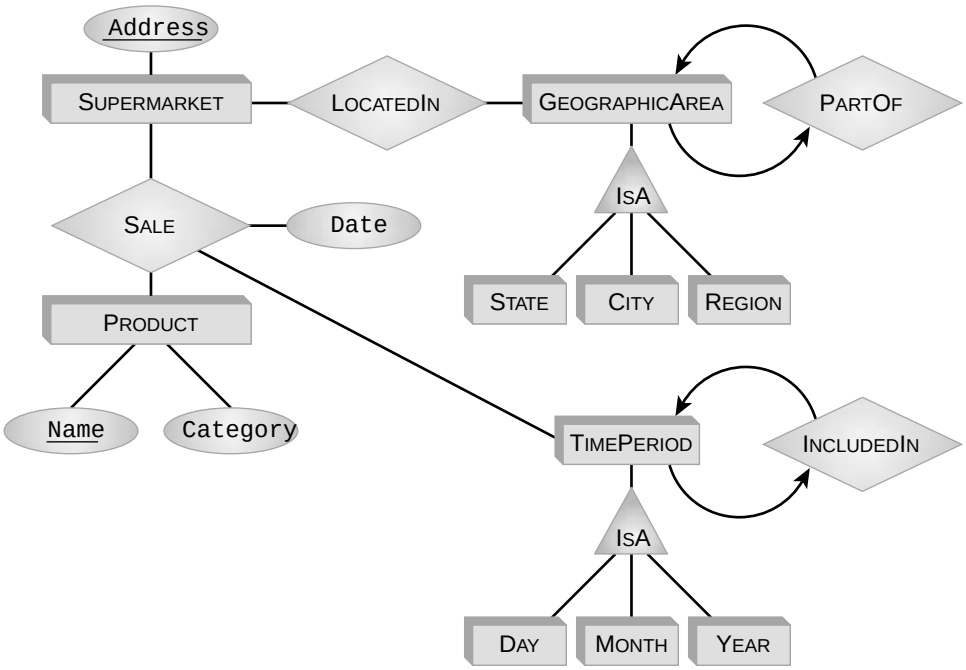


FIGURE 4.6

Solution:

4.14 Make the necessary modifications to the schema of the Student Registration System to reflect the design that uses the SQL domain SEMESTERS, as discussed at the end of Section 4.8. Express the constraint that a class can be taught only during the semesters in which the corresponding course is offered. For instance, if the value of the attribute `SemestersOffered` for the course `CS305` is `Both`, then the corresponding classes can be taught in the spring and the fall semesters. However, if the value of that attribute is `Spring` then these classes can be taught only in the spring.

Solution:
The change to the schema was described at the end of Section 4.8: We no longer need the relation WHENOFFERED. Instead, COURSE gets the following attribute:

`SemestersOffered`	SEMESTERS

The aforesaid integrity constraint can then be expressed as a `CHECK` constraint on CLASS as follows:

```
CHECK (Semester = (SELECT C.SemestersOffered
                   FROM COURSE C
```

```
                WHERE C.CrsCode = CrsCode)
OR
Both = (SELECT C.SemestersOffered
          FROM COURSE C
          WHERE C.CrsCode = CrsCode) )
```

4.15 Design an E-R model for the following enterprise. Various organizations make business deals with various other organizations. (For simplicity, let us assume that there are only two parties to each deal.) When negotiating (and signing) a deal, each organization is represented by a lawyer. The same organization can have deals with many other organizations, and it might use different lawyers in each case. Lawyers and organizations have various attributes, like address and name. They also have their own unique attributes, such as specialization and fee, in the case of a lawyer, and budget, in the case of an organization.

Show how information loss can occur if a relationship of degree higher than two is split into a binary relationship. Discuss the assumption under which such a split does not lead to a loss of information.

Solution:
Two types of entities: ORGANIZATION and LAWYER. DEAL is a quadruple relationship with roles `Party1`, `Party2`, `Lawyer1` and `Lawyer2`.

We can split this into three binary relationships: Two representing a contract between an ORGANIZATION entity and a LAWYER entity (one for each party in the deal) and one that represents negotiations between the two lawyers. The information loss here is obvious, nothing connects the three binary relationships to indicate that they are part of the same deal.

One assumption that ensures that information loss will not occur is that no organization uses the same lawyer twice.

4.16 Design an E-R model for the library system described in Exercise 3.15. Do the same with UML.

Solution:

4.17 Consider the E-R diagram depicted in Figure 4.38. Write down the corresponding relational schema using SQL. Include all keys and other applicable constraints.

Solution:

```
CREATE TABLE  STUDENT  (
     Id       INTEGER,
     Name     CHAR(20),
     Major    CHAR(3),
     PRIMARY KEY (Id, Major) )
```

```
CREATE TABLE  CLASS  (
```

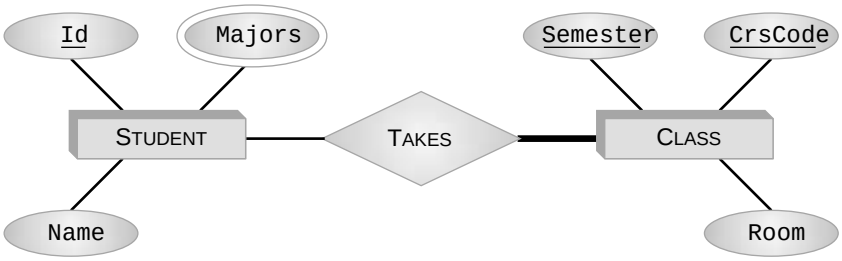


FIGURE 4.7 E-R diagram for Exercise 4.17.

```
        Semester CHAR(5),
        CrsCode  CHAR(6),
        Room     CHAR(15),
        PRIMARY KEY (Semester, CrsCode) )
```

```
    CREATE TABLE  TAKES  (
        StudId   INTEGER NOT NULL,
        Semester CHAR(5) NOT NULL,
        CrsCode  CHAR(6) NOT NULL,
        FOREIGN KEY (Semester,CrsCode) REFERENCES CLASS,
        FOREIGN KEY (StudId) REFERENCES STUDENT (Id) )
```

4.18 Consider the partial translation of an E-R participation constraint into UML (shown in Figure 4.24). Show that this is, indeed, an imprecise translation in that there are instances of the database (i.e., collections of objects and associations) that comply with the E-R constraint in Figure 4.24(b) such that it does not satisfy the multiplicity constraint in UML.

5

Relational Algebra and SQL

EXERCISES

5.1 Assume that **R** and **S** are relations containing n_R and n_S tuples, respectively. What is the maximum and minimum number of tuples that can possibly be in the result of each of the following expressions (assuming appropriate union compatibilities)?

a. $\mathbf{R} \cup \mathbf{S}$

Solution:
Maximum number of tuples: $(n_R + n_S)$;
Minimum number of tuples: $\texttt{max}(n_R, n_S)$.

b. $\mathbf{R} \cap \mathbf{S}$

Solution:
Maximum number of tuples: $\texttt{min}(n_R, n_S)$;
Minimum number of tuples: 0.

c. $\mathbf{R} - \mathbf{S}$

Solution:
Maximum number of tuples: n_R;
Minimum number of tuples: 0.

d. $\mathbf{R} \times \mathbf{S}$

Solution:
Maximum number of tuples: $n_R \times n_S$;
Minimum number of tuples: $n_R \times n_S$.

e. $\mathbf{R} \bowtie \mathbf{S}$

Solution:
Maximum number of tuples: $n_R \times n_S$;
Minimum number of tuples: 0.

f. $\mathbf{R} \,/\, \mathbf{S}$

Solution:
Maximum number of tuples: $n_{\mathbf{R}}/n_{\mathbf{S}}$;
Minimum number of tuples: 0.

g. $\sigma_{s=4}(\mathbf{R}) \times \pi_{s,t}(\mathbf{S})$

Solution:
Maximum number of tuples: $n_R \times n_S$;
Minimum number of tuples: 0. (Selection can be empty.)

5.2 Assume that **R** and **S** are tables representing the relations of the previous exercise. Design SQL queries that will return the results of each of the expressions of that exercise.

Solution:

a.

```
SELECT * FROM R
UNION
SELECT * FROM S
```

b.

```
SELECT * FROM R
INTERSECT
SELECT * FROM S
```

c.

```
SELECT * FROM R
EXCEPT
SELECT * FROM S
```

d.

```
SELECT * FROM R,S
```

e.

```
SELECT   All attributes of R plus all attributes of S
FROM     R, S
WHERE    R.A1 = S.A1 AND ... AND R.An = S.An
```

where $A_1, \ldots, A_n$ are all the attributes that R and S have in common.

f. Assume that S has attributes $B_1, \ldots, B_m$ and that R has *in addition* the attributes $A_1, \ldots, A_n$.

```
SELECT   A1, ..., An
```

```
FROM     R r
WHERE    NOT EXISTS (
              (SELECT  * FROM  S)
              EXCEPT
              (SELECT  r2.B1, ..., r2.Bm
               FROM    R r2
               WHERE   r.A1 = r2.A1 AND ...
                       AND r.An = r2.An) )
```

g. Assume that R has s among its attributes and S has s and t:

```
SELECT   All attributes of R, S.s, S.t
FROM     R, S
WHERE    R.s = 4
```

5.3 Verify that the Cartesian product is an associative operator—that is,

$$\mathbf{r} \times (\mathbf{s} \times \mathbf{t}) = (\mathbf{r} \times \mathbf{s}) \times \mathbf{t}$$

for all relations **r**, **s**, and **t**.

Solution:
$\mathbf{r} \times (\mathbf{s} \times \mathbf{t}) = \mathbf{r} \times \{< s, t > | \ s \in \mathbf{s}, \ t \in \mathbf{t}\} = \{< r, s, t > | \ r \in \mathbf{r}, \ s \in \mathbf{s}, t \in \mathbf{t}\}$
$(\mathbf{r} \times \mathbf{s}) \times \mathbf{t} = \{< r, s > | \ r \in \mathbf{r}, \ s \in \mathbf{s}\} \times \mathbf{t} = \{< r, s, t > | \ r \in \mathbf{r}; \ s \in \mathbf{s}; \ t \in \mathbf{t}\}$
Thus, $\mathbf{r} \times (\mathbf{s} \times \mathbf{t}) = (\mathbf{r} \times \mathbf{s}) \times \mathbf{t}$

5.4 Verify that selections commute—that is, for any relation **r** and any pair of selection conditions $cond_1$ and $cond_2$, $\sigma_{cond_1}(\sigma_{cond_2}(\mathbf{r})) = \sigma_{cond_2}(\sigma_{cond_1}(\mathbf{r}))$.

Solution:
$\sigma_{cond_1}(\sigma_{cond_2}(\mathbf{r})) = \sigma_{cond_1 \text{AND} cond_2}(\mathbf{r}))$
$\sigma_{cond_2}(\sigma_{cond_1}(\mathbf{r})) = \sigma_{cond_1 \text{AND} cond_2}(\mathbf{r}))$
Thus, $\sigma_{cond_1}(\sigma_{cond_2}(\mathbf{r})) = \sigma_{cond_2}(\sigma_{cond_1}(\mathbf{r}))$.

5.5 Verify that, for any pair of relations **r** and **s**, $\sigma_{cond}(\mathbf{r} \times \mathbf{s}) = \mathbf{r} \times \sigma_{cond}(\mathbf{s})$ if the selection condition *cond* involves *only* the attributes mentioned in the schema of relation **s**.

Solution:
$\sigma_{cond}(\mathbf{r} \times \mathbf{s}) = \{< r, s > | r \in \mathbf{r}, \ s \in \mathbf{s}, \ < r, s > \text{ satisfies } cond\}$. Since *cond* involves only the attribute in **s**, the requirement that $< r, s >$ satisfies *cond* can be replaced with s satisfies *cond*. By definition of selection and cross product, $\mathbf{r} \times \sigma_{cond}(\mathbf{s}) = \{< r, s > | r \in \mathbf{r}, \ s \in \mathbf{s}$ and s satisfies $cond\}$. Hence, the two expressions are equivalent.

5.6 Prove that, if **r** and **s** are union-compatible, then $\mathbf{r} \cap \mathbf{s} = \mathbf{r} \bowtie \mathbf{s}$.

Solution:
Because **r** and **s** are union-compatible, the attributes of **r** and **s** are the same. Let them be $A_1, \ldots, A_n$. Thus:
$\mathbf{r} \bowtie \mathbf{s} = \pi_{A_1,\ldots,A_n}(\sigma_{A_1=A_1 \text{ AND } A_2=A_2 \text{ AND } \ldots \text{ AND } A_n=A_n}(\mathbf{r} \times \mathbf{s}))$

Since **r** and **s** are relations over $A_1, \ldots, A_n$, it means that only the tuples that are both in **r** and **s** survive the selection, i.e., the result is the intersection of the two relations.

5.7 Using division, write a relational algebra expression that produces all students who have taken all courses offered in every semester (this implies that they might have taken the same course twice).

Solution:

$\pi_{\texttt{StudId,CrsCode,Semester}}(\textsc{Transcript}) \;/\; \pi_{\texttt{CrsCode,Semester}}(\textsc{Teaching})$

5.8 Using division, write a relational algebra expression that produces all students who have taken all courses that have been offered. (If a course has been offered more than once, they have to have taken it at least once.)

Solution:

$\pi_{\texttt{StudId,CrsCode}}\ (\textsc{Transcript}) \;/\; \pi_{\texttt{CrsCode}}\ (\textsc{Teaching})$

5.9 Construct a relational algebra query that produces the same result as the outer join $\mathbf{r} \bowtie^{outer}_{cond} \mathbf{s}$ using only these operators: union, difference, Cartesian product, projection, general join (not outer join). You can also use constant relations, that is, relations with fixed content (e.g., one that has tuples filled with nulls or other predefined constants).

Solution:

Let $\mathbf{null_R}$ be the relation over **R** that contains only one tuple, which entirely consists of NULLs. Similarly, let $\mathbf{null_S}$ be a relation over **S** that has a single NULL-tuple.

Then $\mathbf{r} \bowtie^{outer}_{cond} \mathbf{s}$ is:

$$\begin{array}{l} \mathbf{r} \bowtie_{cond} \mathbf{s} \\ \cup\ (\mathbf{r} - \pi_{\mathbf{R}}(\mathbf{r} \bowtie_{cond} \mathbf{s})) \times \mathbf{null_S} \\ \cup\ \mathbf{null_R} \times (\mathbf{s} - \pi_{\mathbf{S}}(\mathbf{r} \bowtie_{cond} \mathbf{s})) \end{array}$$

5.10 Express each of the following queries in (*i*) relational algebra and (*ii*) SQL using the Student Registration System schema of Figure 3.4.

a. List all courses that are taught by professors who belong to the `EE` or `MGT` departments.

Solution:

– *Algebra*:

$$\pi_{\texttt{CrsCode,Semester}}(\sigma_{\texttt{DeptId='EE'}\ \texttt{OR}\ \texttt{DeptId='MGT'}}(\textsc{Professor}) \bowtie_{\texttt{Id=ProfId}} \textsc{Teaching})$$

– *SQL*:

```
SELECT   T.CrsCode, T.Semester
FROM     Teaching T,  Professor P
WHERE    T.ProfId=P.Id
```

```
                AND (P.DeptId = 'EE' OR P.DeptId = 'EE')
```

b. List the names of all students who took courses *both* in spring 1997 and fall 1998.

Solution:

– *Algebra*: Let SEMS be a relation with `Semester` as the only column and $\langle$`'S1997'`$\rangle$ and $\langle$`'F1998'`$\rangle$ as the only tuples. Then the answer is:

$$\pi_{\texttt{Name}}(\text{STUDENT} \bowtie_{\texttt{Id=StudId}} \pi_{\texttt{StudId,Semester}}(\text{TRANSCRIPT})/\text{SEMS})$$

– *SQL*:

```
SELECT   S.Name
FROM     STUDENT S
WHERE    NOT EXISTS (
              (SELECT * FROM SEMS)
              EXCEPT
              (SELECT T.Semester
               FROM TEACHING T
               WHERE T.StudId = S.Id ) )
```

Note that if the query required only the students who took courses in either of the two semesters, then the solution would not have required the division operator and NOT EXISTS and it would have been much simpler.

c. List the names of all students who took courses from at least two professors in different departments.

Solution:

– *Algebra*: We construct the requisite expression in two stages. First, we construct STUDDEPT(`StudId`, `DeptId`) as

$$\pi_{\texttt{StudId,DeptId}}(\text{TRANSCRIPT} \bowtie \text{TEACHING} \bowtie_{\texttt{ProfId=Id}} \text{PROFESSOR})$$

Tuples in this relation represent instances when a given student took a course from a professor in the given department. Constructing the final answer is now easy:

$$\pi_{\texttt{Name}}(\text{STUDENT} \bowtie_{\texttt{Id=StudId}} \sigma_{\texttt{DeptId}\neq\texttt{DeptId2}}(\text{STUDDEPT} \bowtie \text{STUDDEPT}[\texttt{StudId,DeptId2}]))$$

– *SQL*: The SQL solution is also in two stages, for clarity. First, we create an auxiliary view:

```
CREATE VIEW    STUDDEPT(StudId, DeptId) AS
```

```
SELECT   R.StudId, P.DeptId
FROM     Transcript R, Teaching T, Professor P
WHERE    R.CrsCode = T.Crscode
         AND R.Semester = T.Semester
         AND T.ProfId = P.Id
```

```
SELECT   S.Name
FROM     Student S, StudDept SD1, StudDept SD2
WHERE    S.Id = SD1.StudId AND SD1.StudId = SD2.StudId
         AND SD1.DeptId ≠ SD2.DeptId
```

d. List all courses that are offered by the MGT Department and that have been taken by all students.

Solution:

– *Algebra*:

$$\pi_{\texttt{CrsCode,StudId}}(\sigma_{\texttt{DeptId='MGT'}}\textsc{Course} \bowtie \textsc{Transcript}) \;/\; \pi_{\texttt{StudId}}(\textsc{Student})$$

– *SQL*:

```
SELECT   C.CrsCode
FROM     Course C
WHERE    C.DeptId = 'MGT' AND
         NOT EXISTS (
              (SELECT  S.StudId
               FROM    Student S)
               EXCEPT
              (SELECT  T.StudId
               FROM    Transcript T
               WHERE   C.CrsCode = T.CrsCode) )
```

*e. Find every department that has a professor who taught all courses ever offered by that department.

Solution:

– *Algebra*: The algebraic solution is trickier than SQL's. Let ProfCrsAux(ProfId, CrsCode) be defined as follows:

$$\pi_{\texttt{ProfId,CrsCode}}(\textsc{Professor} \bowtie_{\texttt{DeptId}\neq\texttt{DeptId}} \textsc{Course})$$

A tuple, $\langle p, c \rangle$ here means that c is a course that is offered by a department *other than* the one where p works. Next, consider ProfCrs(ProfId, CrsCode), which is defined as follows:

ProfCrsAux $\cup$ $\pi_{\texttt{ProfId,CrsCode}}$(Teaching)

A tuple, $\langle p, c \rangle$ in this relation means that c is a course that was either taught by p or it is offered by a department *other than* the one where p works.

Finally, the expression for the requisite query can be constructed as follows:

ProfCrs / $\pi_{\texttt{CrsCode}}$(Course)

Note that constructing ProfCrsAux was essential. The expression

---

$\pi_{\texttt{ProfId,CrsCode}}$(Teaching) / $\pi_{\texttt{CrsCode}}$(Course)

---

is *not* correct, because it finds every professor who have taught every course in every department rather than the courses that are offered by the department where that professor works.

The solution given above uses the division operator. There is a more straightforward solution where one would first find professors who did not teach one of the courses offered by the professor's department, and then take the complement.

– *SQL*:

```
SELECT   P.DeptId
FROM     Professor P
WHERE    EXISTS (
         ( SELECT  P2.Id
           FROM    Professor P2
           WHERE   NOT EXISTS (
                        (SELECT  C.CrsCode
                         FROM    Course C
                         WHERE   C.DeptId = P2.DeptId )
                        EXCEPT
                        (SELECT  T.CrsCode
                         FROM    Teaching T
                         WHERE   T.ProfId = P2.Id ) )
                   AND  P.Id = P2.Id
```

In fact, the following simplified query is also correct:

```
SELECT   P.DeptId
FROM     Professor P
WHERE    NOT EXISTS (
```

```
(SELECT   C.CrsCode
 FROM     COURSE C
 WHERE    C.DeptId = P.DeptId )
EXCEPT
(SELECT   T.CrsCode
 FROM     TEACHING T
 WHERE    T.ProfId = P.Id ) )
```

5.11 Use the relational algebra to find the list of all "problematic" classes (i.e., course-semester pairs) where the failure rate is higher than 20%. (Assume, for simplicity, that grades are numbers between 1 and 4, and that a failing grade is anything less than 2.)

Because the relational algebra does not have aggregate operators, we must add them to be able to solve the above problem. The additional operator you should use is $count_{A/B}(\mathbf{r})$.

The meaning of this operator is as follows: A and B must be lists of attributes in $\mathbf{r}$. The *schema* of $count_{A/B}(\mathbf{r})$ consists of all attributes in B plus one additional attribute, which represents the counted value. The *contents* of $count_{A/B}(\mathbf{r})$ are defined as follows: for each tuple, $t \in \pi_B(\mathbf{r})$ (the projection on B), take $\pi_A(\sigma_{B=t}(\mathbf{r}))$ and count the number of tuples in the resulting relation (where $\sigma_{B=t}(\mathbf{r})$ stands for the set of all tuples in $\mathbf{r}$ whose value on the attributes in B is t). Let us denote this number by $c(t)$. Then the relation $count_{A/B}(\mathbf{r})$ is defined as $\{< t, c(t) > | \, t \in \pi_B(\mathbf{r})\}$.

You should be able to recognize the above construction as a straightforward adaptation of GROUP BY of SQL to the relational algebra.

Solution:
First compute the total number of students in each course and the number of failures:

$$\begin{aligned}&\textsc{AuxCounts} = \\ &\quad count_{\texttt{StudId}/\{\texttt{CrsCode,Semester}\}}(\textsc{Transcript}) \\ &\qquad\qquad [\texttt{CrsCode,Semester,TotCount}] \bowtie \\ &\quad count_{\texttt{StudId}/\{\texttt{CrsCode,Semester}\}}(\sigma_{\texttt{Grade}<2}(\textsc{Transcript})) \\ &\qquad\qquad [\texttt{CrsCode,Semester,FailCount}]\end{aligned}$$

Computing the "problematic" classes is now easy:

$$\pi_{\texttt{CrsCode}}(\sigma_{\texttt{FailCount/TotCount}>0.2}(\textsc{AuxCounts}))$$

5.12 State the English meaning of the following algebraic expressions (some of these queries are likely to yield empty results in a typical university, but this is beside the point):

a. $\pi_{\texttt{CrsCode,Semester}}(\textsc{Transcript}) / \pi_{\texttt{CrsCode}}(\textsc{Transcript})$

Solution:
Semesters in which all courses (that were ever taught) were offered.

b. $\pi_{\texttt{CrsCode,Semester}}(\textsc{Transcript}) / \pi_{\texttt{Semester}}(\textsc{Transcript})$

Solution:
Courses offered in every semester.

c. $\pi_{\texttt{CrsCode,StudId}}(\textsc{Transcript})/\ (\pi_{\texttt{Id}}(\textsc{Student}))[\texttt{StudId}]$

Solution:
Courses taken by every student.

d. $\pi_{\texttt{CrsCode,Semester,StudId}}(\textsc{Transcript})/\ (\pi_{\texttt{Id}}(\textsc{Student}))[\texttt{StudId}]$

Solution:
Classes (courses + semester) taken by all students.

5.13 Consider the following query:

```
SELECT   S.Name
FROM     STUDENT S, TRANSCRIPT T
WHERE    S.Id = T.StudId
         AND T.CrsCode IN ('CS305','CS315')
```

What does this query mean (express the meaning in one short English sentence)? Write an equivalent SQL query without using the IN operator and the set construct.

Solution:
The equivalent SQL query to query (7.33) is:

```
SELECT   S.Name
FROM     STUDENT S, TRANSCRIPT T
WHERE    S.Id = T.StudId
         AND (T.CrsCode = 'CS305' OR T.CrsCode = 'CS315')
```

The meaning of the query is: the names of all students who took CS305 or CS315.

5.14 Explain the conceptual difference between views and bulk insertion of tuples into base relations using the INSERT statement with an attached query.

Solution:
Tuples in a view are not materialized, i.e., they are computed on demand. In contrast, when tuples are inserted into a relation using bulk INSERT they physically exist on the disk.

5.15 Explain why a view is like a subroutine.

Solution:
A view represents a meaningful query, which is often used as a black box within several other frequently asked queries. This is similar to subroutines in conventional programming languages.

5.16 Write query (5.38) on page 175 without the use of the views.

Solution:

```
SELECT   A.Dept
```

```
FROM     (SELECT  P.DeptId, AVG(P.Age) AS AvgAge
          FROM    PROFESSOR P
          GROUP BY  P.DeptId ) AS AVGDEPTAGE A
WHERE    A.AvgAge = (SELECT MIN(A1.AvgAge)
                     FROM AVGDEPTAGE A1)
```

5.17 Express the following queries using SQL. Assume that the STUDENT table is augmented with an additional attribute, `Age`, and that the PROFESSOR table has additional attributes, `Age` and `Salary`.

a. Find the average age of students who received an `A` for *some* course.

Solution:
Note: the following is *incorrect*, because some students' ages might be counted more than once:

```
SELECT   AVG(S.Age)
FROM     STUDENT S, TRANSCRIPT T
WHERE    S.Id =T.StudId AND T.Grade = 'A'
```

The correct solution requires a subquery with a DISTINCT clause:

```
SELECT   AVG(S.Age)
FROM     STUDENT S
WHERE    S.Id  IN
     (SELECT DISTINCT  S.Id
      FROM    STUDENT S, TRANSCRIPT T
      WHERE   S.Id =T.StudId AND T.Grade = 'A' )
```

b. Find the minimum age among straight `A` students *per course*.

Solution:

```
SELECT   T.CrsCode, MIN(S.Age)
FROM     STUDENT S, TRANSCRIPT T
WHERE    S.Id = T.StudId AND
         NOT EXISTS (SELECT T.Grade
                     FROM TRANSCRIPT T
                     WHERE S.Id = T.StudId AND T.Grade ≠ A )
GROUP BY T.CrsCode
```

c. Find the minimum age among straight `A` students per course among the students who have taken `CS305` or `MAT123`. (Hint: a HAVING clause might help.)

Solution:
Add the following HAVING clause to the above query:

```
HAVING   T.CrsCode IN ('CS305','MAT123')
```

d. Raise by 10% the salary of every professor who is now younger than 40 and who taught MAT123 in the spring 1997 or fall 1997 semester. (*Hint*: Try a nested subquery in the WHERE clause of the UPDATE statement.)

Solution:

```
UPDATE   PROFESSOR
SET      Salary = Salary * 1.1
WHERE    Id IN
              (SELECT  P.Id
               FROM    PROFESSOR P, TEACHING T
               WHERE   P.Id = T.ProfId
                   AND P.Age < 40
                   AND T.CrsCode = 'MAT123'
                   AND (T.Semester = 'S1997'
                        OR T.Semester = 'F1997') )
```

e. Find the professors whose salaries are at least 10% higher than the average salary of all professors. (*Hint*: Use views, as in the HARDCLASS example (5.39), page 183.)

Solution:

```
CREATE VIEW  AVGSAL(Avgsal) AS
     SELECT   AVG(P.Salary)
     FROM     PROFESSOR P
```

The actual query that uses the above view:

```
SELECT   P.Id
FROM     PROFESSOR P, AVGSAL A
WHERE    P.Salary > 1.1 * A.Avgsal
```

f. Find all professors whose salaries are at least 10% higher than the average salary of all professors *in their departments*. (*Hint*: Use views, as in (5.39).)

Solution:

```
CREATE VIEW  DEPTAVGSAL(DeptId, Avgsal) AS
     SELECT   P.DeptId, AVG(P.Salary)
     FROM     PROFESSOR P
```

```
        GROUP BY P.DeptId
```

The actual query, which uses the above view:

```
SELECT   P.Id
FROM     PROFESSOR P, DEPTAVGSAL D
WHERE    P.DeptId = D.DeptId AND P.Salary > 1.1 * D.Avgsal
```

5.18 Express the following queries in relational algebra.

a. (5.16), page 154

Solution:

$\pi_{\texttt{Name}}(\sigma_{\texttt{DeptId='CS' OR DeptId='EE'}}(\textsc{Professor}))$

b. (5.20), page 156

Solution:

$$\pi_{\texttt{Name}}(\sigma_{\texttt{CrsCode='CS305' AND CrsCode2='CS315'}}(\textsc{BigJoin})$$

where BIGJOIN is defined as:

$$\textsc{Student} \bowtie_{\texttt{Id=StudId}} \textsc{Transcript} \bowtie \textsc{Transcript}\texttt{[StudId,CrsCode2,S2,G2]}$$

Observe that a careful renaming of the attributes has allowed us to use the natural join between the two occurrences of TRANSCRIPT and to make the expression shorter.

c. (5.23), page 158

Solution:

$\pi_{\texttt{Id}}(\textsc{Student}) - \pi_{\texttt{StudId}}(\textsc{Transcript})$

5.19 Write an equivalent expression in relational algebra for the following SQL query:

```
SELECT   P.Name, C.Name
FROM     PROFESSOR P, COURSE C, TAUGHT T
WHERE    P.Id = T.ProfId AND T.Semester = 'S2002'
             AND T.CrsCode = C.CrsCode
```

Solution:

$$\pi_{\textsc{Professor}.\texttt{Name},\textsc{Course}.\texttt{Name}}\ (\sigma_{\texttt{Semester='S2002'}}(\textsc{Professor} \bowtie_{\texttt{Id=ProfId}} \textsc{Taught} \bowtie_{\texttt{CrsCode=CrsCode}} \textsc{Course}))$$

5.20 Consider the following schema:

```
TRANSCRIPT(StudId, CrsCode, Semester, Grade)
TEACHING(ProfId, CrsCode, Semester)
PROFESSOR(Id, ProfName, Dept)
```

Write the following query in relational algebra and in SQL: *Find all student Ids who have taken a course from* each *professor in the* MUS *Department*.

Solution:

- *Algebra*:
  $\pi_{\texttt{StudId,ProfId}}(\textsc{Transcript} \bowtie \textsc{Teaching}) \;/\; \pi_{\texttt{ProfId}}(\sigma_{\texttt{Dept='MUS'}}(\textsc{Professor}))$
- *SQL*:

```
SELECT   R.StudId
FROM     TRANSCRIPT R
WHERE    NOT EXISTS (
              (SELECT  P.ProfId
               FROM    PROFESSOR P
               WHERE   P.Dept = 'MUS')
              EXCEPT
              (SELECT  T.ProfId
               FROM    TEACHING T
               WHERE   T.CrsCode = R.CrsCode
                        AND T.Semester = R.Semester)
```

5.21 Define the above query as an SQL view and then use this view to answer the following query: *For each student who has taken a course from every professor in the* MUS *Department, show the number of courses taken, provided that this number is more than* 10.

Solution:

```
CREATE VIEW   MUSICIAN(StudId) AS
     SELECT   R.StudId
     FROM     TRANSCRIPT R
     WHERE    NOT EXISTS (
                   (SELECT  P.ProfId
                    FROM    PROFESSOR P
                    WHERE   P.Dept = 'MUS')
                   EXCEPT
                   (SELECT  T.ProfId
                    FROM    TEACHING T
```

```
                    WHERE   T.CrsCode = R.CrsCode
                            AND T.Semester = R.Semester))
```

A query that uses MUSICIAN:

```
SELECT   M.StudId, COUNT (*)
FROM     MUSICIAN M, TRANSCRIPT T
WHERE    M.StudId = T.StudId
GROUP BY M.StudId
HAVING   COUNT(*) > 10
```

5.22 Consider the following schema:

BROKER(Id, Name) ACCOUNT(Acct#, BrokerId, Gain)

Write the following query in relational algebra and in SQL: *Find the names of all brokers who have made money in all accounts assigned to them* (*i.e.,* Gain > 0).

Solution:

- *Algebra*: The algebraic solution has a pitfall and is harder than the SQL-based solution. Note that the following is *incorrect*:

  $\pi_{\texttt{Name}}(\text{BROKER} \bowtie (\pi_{\texttt{Acct\#,BrokerId}}(\sigma_{\texttt{Gain}>0}(\text{ACCOUNT})) \ / \ \pi_{\texttt{Acct\#}}(\text{ACCOUNT})))$

  because it finds brokers who manage *all* accounts and made money for all of them.

  For a correct solution, first find all broker-account associations where the broker lost money:

  $$\text{LOOSINGACCOUNTS} = \pi_{\texttt{Acct\#,BrokerId}}(\text{ACCOUNT}) - \pi_{\texttt{Acct\#,BrokerId}}(\sigma_{\texttt{Gain}>0}(\text{ACCOUNT}))$$

  The requisite query is then

  $$\sigma_{\texttt{Name}}(\text{BROKER} \bowtie (\pi_{\texttt{Id}}(\text{BROKER}) - \pi_{\texttt{BrokerId}}(\text{LOOSINGACCOUNTS})))$$

- *SQL*:

```
SELECT   B.Name
FROM     BROKER B
WHERE    NOT EXISTS (
             SELECT   A.Acct#
             FROM     ACCOUNT A
             WHERE    A.BrokerId = B.Id AND A.Gain <= 0)
```

5.23 Write an SQL statement (for the database schema given in Exercise 5.22) to fire all brokers who lost money in at least 40% of their accounts. Assume that every broker has at least one account. (*Hint*: Define intermediate views to facilitate formulation of the query.)

Solution:

```
CREATE VIEW   BrokerTotalNumber(Id, Number) AS
     SELECT   A.BrokerId, COUNT(*)
     FROM     Account A
     GROUP BY A.BrokerId
```

```
CREATE VIEW   BrokerLoosingNumber(Id, Number) AS
     SELECT   A.BrokerId, COUNT(*)
     FROM     Account A
     WHERE    A.Gain < 0
     GROUP BY A.BrokerId
```

```
DELETE FROM   Broker
WHERE  Id IN
     (SELECT  B1.Id
      FROM    BrokerTotalNumber B1,
              BrokerLoosingNumber B2
      WHERE   B1.Id = B2.Id
              AND B2.Number >= 0.4 * B1.Number)
```

5.24 Consider the following schema that represents houses for sale and customers who are looking to buy:

```
Customer(Id, Name, Address)
Preference(CustId, Feature)
Agent(Id, AgentName)
House(Address, OwnerId, AgentId)
Amenity(Address, Feature)
```

Preference is a relation that lists all features requested by the customers (one tuple per customer/feature; e.g., ⟨`123, '5BR'`⟩, ⟨`123,'2BATH'`⟩, ⟨`432,'pool'`⟩), and Amenity is a relation that lists all features of each house (one tuple per house/feature).

A customer is *interested* in buying a house if the set of all features specified by the customer is a subset of the amenities the house has. A tuple in the House relation states who is the owner and who is the real estate agent listing the house. Write the following queries in SQL:

a. Find all customers who are interested in every house listed with the agent with Id 007.

Solution:

```
SELECT   C.Id
FROM     Customer C
WHERE    NOT EXISTS (
              (SELECT  P.Feature
               FROM    Preference P
               WHERE   P.CustId = C.Id)
              EXCEPT
              (SELECT  A.Feature
               FROM    Amenity A, House H
               WHERE   A.Address = H.Address
                       AND H.AgentId = '007') )
```

b. Using the previous query as a view, retrieve a set of tuples of the form $\langle feature, number\_of\_customers \rangle$, where each tuple in the result shows a feature and the number of customers who want this feature such that
   - Only the customers who are interested in every house listed with Agent 007 are considered.
   - The number of customers interested in *feature* is greater than three. (If this number is not greater than three, the corresponding tuple $\langle feature, number\_of\_customers \rangle$ is not added to the result.)

Solution:

```
CREATE VIEW   ClientOf007(CustId) AS
     SELECT    C.Id
     FROM      Customer C
     WHERE     NOT EXISTS (
                    (SELECT  P.Feature
                     FROM    Preference P
                     WHERE   P.CustId = C.Id)
                    EXCEPT
                    (SELECT  A.Feature
                     FROM    Amenity A, House H
                     WHERE   A.Address = H.Address
                             AND H.AgentId = '007' )
```

```
SELECT   P.Feature, COUNT(*)
FROM     Preference P, ClientOf007 C
```

```
WHERE    P.CustId = C.CustId
GROUP BY P.Feature
HAVING   COUNT(*) > 3
```

5.25 Consider the schema PERSON(Id, Name, Age). Write an SQL query that finds the 100th oldest person in the relation. A 100th oldest person is one such that there are 99 people who are strictly older. (There can be several such people who might have the same age, or there can be none.)

Solution:

```
SELECT P.Id, P.Name
FROM PERSON P
WHERE
     99 = (SELECT COUNT(*)
           FROM PERSON PP
           WHERE PP.Age > P.Age)
```

5.26 The last section of this chapter presented four main rules that characterize updatable views in SQL. The third rule states that if the WHERE clause contains a nested subquery, none of the tables mentioned in that subquery (explicitly or implicitly) can be the table used in the FROM clause of the view definition.

Construct a view that violates condition 3 but satisfies conditions 1, 2, and 4 for updatability, such that there is an update to this view that has two different translations into the updates on the underlying base relation.

Solution:
Consider two relation schemas, **R**(Att1,Att2) and **S**(Att1,Att2), and a view:

```
CREATE VIEW V(Att1)
     SELECT   A.Att1
     FROM     V  A
     WHERE    EXISTS (
                   SELECT  *
                   FROM   S  B
                   WHERE  B.Att1 = A.Att1 AND B.Att2 = A.Att2 )
```

Here the base table of the view, **R**, is implicitly mentioned in the subquery (through the tuple variable A). Now, suppose we want to delete a tuple of the form $\langle a, b\rangle$ from **V**. There are two translations: to delete all tuples of the form $\langle a, *\rangle$ from **R** or from **S**.

5.27 Using the relations TEACHING and PROFESSOR, create a view of TRANSCRIPT containing only rows corresponding to classes taught by John Smyth. Can this view be used in a GRANT statement whose purpose is to allow Smyth to update student grades, but only those he has assigned? Explain.

Solution:

```
CREATE VIEW SmythCourses(StudId, CrsCode, Semester, Grade) AS
    SELECT   T.StudId, T.CrsCode, T.Semester, T.Grade
    FROM     Teaching R, Transcript T, Professor P
    WHERE    T.CrsCode = R.CrsCode
                 AND T.Semester = R.Semester
                 AND R.ProfId = P.Id
                 AND P.Name = 'John Smyth'
```

The view SmythCourses allows Professor Smyth to see only those rows of Transcript corresponding to courses he has taught. However, since the view involves several tables it would not be updatable although a change to a grade visible in the view uniquely corresponds to a change in Transcript.

6

Database Design with the Relational Normalization Theory

EXERCISES

6.1 The definition of functional dependencies does not preclude the case in which the left-hand side is empty—that is, it allows FDs of the form $\{\} \rightarrow A$. Explain the meaning of such dependencies.

Solution:
This means that all tuples have the same value for the attribute A.

6.2 Give an example of a schema that is not in 3NF and has just two attributes.

Solution:
$(AB;\ \{\} \rightarrow A,\ \ B \rightarrow A)$. Here B is a key, but $\{\} \rightarrow A$ does not satisfy the 3NF requirements.

6.3 What is the smallest number of attributes a relation key can have?

Solution:
Zero. In this case, the FD $\{\} \rightarrow X$ (please refer to exercise 6.1) holds, where X is the set of all attributes in the schema.

6.4 A table, ABC, has attributes A, B, and C, and a functional dependency $A \rightarrow BC$. Write an SQL CREATE ASSERTION statement that prevents a violation of this functional dependency.

Solution:

```
CREATE ASSERTION  FD
CHECK    (1 >= ALL (
              SELECT COUNT(DISTINCT *)
              FROM ABC
              GROUP BY A ) )
```

6.5 Prove that every 3NF relation schema with just two attributes is also in BCNF. Prove that every schema that has *at most* one nontrivial FD is in BCNF.

Solution:

a. Suppose the schema **R** is in 3NF and that its attributes are AB. Then for every FD $X \rightarrow A \in F$, either (1)$A \in X$, or (2) X is a superkey of **R**, or (3) $A \notin X$, X is not a superkey, and $A \in K$, where K is a key of **R**. We will show that (3) is impossible in our situation, which implies that **R** is in BCNF.

Indeed, if (3) holds, then either $\{A\} = K$ or X is empty and $K = AB$. The first case is not possible because $X \rightarrow A$ and A being a key implies that X is a superkey, contrary to the assumption that it is not (this assumption is part of (3)). The second case is impossible because $\{\ \} \rightarrow A$ implies that $B \rightarrow A$, contrary to the assumption that AB is a key (because it means that B alone is a key).

b. If **R** has exactly one non-trivial FD, $X \rightarrow Y$, X must be a superkey of **R** (if it is not, then by augmentation it is possible to make another non-trivial FD, violating the assumption). If **R** has no non-trivial FDs, the set of all attributes is its key and the schema is, again, in BCNF.

6.6 If the functional dependency $X \rightarrow Y$ is a key constraint, what are X and Y?

Solution:
X is the key and Y is the set of all attributes.

6.7 Can a key be the set of all attributes if there is at least one nontrivial FD in a schema?

Solution:
No. Suppose a key k contained all attributes and there is a non-trivial FD $X \rightarrow Y$. k contains X and Y. But the FD implies that X determines Y. Hence we can delete $Y - (X \cap Y)$ from k and the remaining attributes still form a key. This means that k was not minimal, a contradiction.

6.8 The following is an instance of a relation schema. Can you tell whether the schema includes the functional dependencies $A \rightarrow B$ and $BC \rightarrow A$?

A	B	C
1	2	3
2	2	2
1	3	2
4	2	3

Solution:
$A \rightarrow B$ is not a FD; we cannot determine whether $BC \rightarrow A$ is a FD.

6.9 Prove that Armstrong's transitivity axiom is sound—that is, every relation that satisfies the FDs $\overline{X} \rightarrow \overline{Y}$ and $\overline{Y} \rightarrow \overline{Z}$ must also satisfy the FD $\overline{X} \rightarrow \overline{Z}$.

Solution:
Consider a relation **r** and an arbitrary pair of its tuples $t, s \in \mathbf{r}$.

If $\overline{X} \rightarrow \overline{Y}$ and t, s agree on every attribute of $\overline{X}$, then t, s agree on every attribute of $\overline{Y}$. Because of $\overline{Y} \rightarrow \overline{Z}$ and since t, s agree on every attribute of $\overline{Y}$, they must agree on every attribute of $\overline{Z}$. Thus, every pair of tuples that agrees on $\overline{X}$ must also agree on $\overline{Z}$, so $\overline{X} \rightarrow \overline{Z}$ holds.

6.10 Prove the following *generalized transitivity rule*: If $\overline{Z} \subseteq \overline{Y}$, then $\overline{X} \rightarrow \overline{Y}$ and $\overline{Z} \rightarrow \overline{W}$ entail $\overline{X} \rightarrow \overline{W}$. Try to prove this rule in two ways:

- Using the argument that directly appeals to the definition of FDs, as in Section 6.4

  Solution:
  Let t, s be arbitrary tuples in a relation that satisfies the above conditions, and suppose t, s agree on $\overline{X}$. Then, due to $\overline{X} \rightarrow \overline{Y}$, they agree on $\overline{Y}$ and thus also on $\overline{Z}$. But then, due to $\overline{Z} \rightarrow \overline{W}$, t, s must agree on $\overline{W}$.

  Therefore, any pair of tuples that agrees on $\overline{X}$ in such a relation must also agree on $\overline{W}$, i.e., $\overline{X} \rightarrow \overline{W}$ must hold.

- By deriving $\overline{X} \rightarrow \overline{W}$ from $\overline{X} \rightarrow \overline{Y}$ and $\overline{Z} \rightarrow \overline{W}$ via a series of steps using Armstrong's axioms

  Solution:

(1)	$\overline{Z} \rightarrow \overline{W}$	Given
(2)	$\overline{Y} \rightarrow \overline{WY}$	By Augmentation of (1) and because $\overline{Z} \subseteq \overline{Y}$
(3)	$\overline{X} \rightarrow \overline{Y}$	Given
(4)	$\overline{X} \rightarrow \overline{WY}$	From (3),(2) by transitivity
(5)	$\overline{WY} \rightarrow \overline{W}$	By the Reflexivity rule
(6)	$\overline{X} \rightarrow \overline{W}$	From (4),(5) by Transitivity

⋆**6.11** We have shown the *soundness* of the algorithm in Figure 6.3—that if $A \in$ *closure* then $A \in \overline{X}^+_{\mathcal{F}}$. Prove the *completeness* of this algorithm; that is, if $A \in \overline{X}^+_{\mathcal{F}}$, then $A \in$ *closure* at the end of the computation. *Hint:* Use induction on the length of derivation of $X \rightarrow A$ by Armstrong's axioms.

Solution:
Since $A \in \overline{X}^+_{\mathcal{F}}$ and because of the completeness of the Armstrong's axioms, there must be a derivation of $\overline{X} \rightarrow A$. We prove that $A \in$ *closure* by induction on the length, n, of this derivation. More specifically, the induction will show a more general claim: if $\overline{W} \rightarrow \overline{V}$, where $\overline{W} \subseteq$ *closure*, can be derived by Armstrong's axioms, then $\overline{V} \subseteq$ *closure*. Since $\overline{X} \subseteq$ *closure*, the above would imply that if $\overline{X} \rightarrow A \in \mathcal{F}^+$ then $A \in$ *closure*.

Case $n = 1$: This is only possible if the rule used in such a 1-step derivation is the Reflexivity axiom or is an FD from $\mathcal{F}$. In case of the Reflexivity axiom, we must already have $\overline{V} \subseteq \overline{W}$, for some $\overline{W} \subseteq$ *closure*, and so we conclude that $\overline{V} \in$ *closure*. In case $\overline{W} \rightarrow \overline{V} \in \mathcal{F}$, where $\overline{W} \subseteq$ *closure*, $\overline{V}$ must be a subset of *closure* by construction: during the iteration when $\overline{W}$ was determined to be in *closure*, the right-hand side of the FD (i.e., $\overline{V}$) will be added.

Inductive step: Assume that, for any set $\overline{V}$, if $\overline{W} \rightarrow \overline{V}$ (where $\overline{W} \subseteq$ *closure*) can be derived via n steps, then $\overline{V} \subseteq$ *closure*.

Suppose $\overline{W} \rightarrow \overline{V}$, where $\overline{W} \subseteq$ *closure*, was derived using $n + 1$ derivation steps (and no less). If the last rule in the derivation was Reflexivity, then $\overline{V} \subseteq \overline{W}$ and, since $\overline{W} \subseteq$ *closure*, it follows that $\overline{V} \subseteq$ *closure*.

If the last step was Augmentation, then $\overline{W'} \rightarrow \overline{U}$ ($\overline{U} \neq \emptyset$) was derived at step n or earlier, and then augmented with a nonempty set of attributes $\overline{Z}$, such that $\overline{W'Z} = \overline{W}$

and $\overline{UZ} = \overline{V}$. Since $\overline{W'} \subseteq \overline{W} \subseteq$ *closure*, it follows by the inductive assumption that $\overline{U} \subseteq$ *closure*. Since $\overline{Z} \subseteq \overline{W} \subseteq$ *closure*, it follows that $\overline{V} \subseteq$ *closure*.

If the last step in the derivation was transitivity, it means that the following two FDs were derived earlier: $\overline{W} \rightarrow \overline{U}$ and $\overline{U} \rightarrow \overline{V}$. By the inductive assumption, $\overline{U} \subseteq$ *closure* and, hence, $\overline{V} \subseteq$ *closure*.

6.12 Suppose that $\mathbf{R} = (\overline{R}, \mathcal{F})$ is a relation schema and $\mathbf{R}_1 = (\overline{R}_1; \mathcal{F}_1), \ldots, \mathbf{R}_n = (\overline{R}_n; \mathcal{F}_n)$ is its decomposition. Let $\mathbf{r}$ be a valid relation instance over $\mathbf{R}$ and $\mathbf{r}_i = \pi_{\overline{R}_i}(\mathbf{r})$. Show that $\mathbf{r}_i$ satisfies the set of FDs $\mathcal{F}_i$ and is therefore a valid relation instance over the schema $\mathbf{R}_i$.

Solution:
Since $\mathcal{F}_i$ is entailed by $\mathcal{F}$ and all of the attributes mentioned in $\mathcal{F}_i$ are contained within $\overline{R}_i$, it follows that the projection of $\mathbf{r}$ on $\mathbf{R}_i$ satisfies every FD in $\mathcal{F}_i$.

6.13 Let $\overline{R}_1$ and $\overline{R}_2$ be sets of attributes and $\overline{R} = \overline{R}_1 \cup \overline{R}_2$. Let $\mathbf{r}$ be a relation on $\overline{R}$. Prove that $\mathbf{r} \subseteq \pi_{\overline{R}_1}(\mathbf{r}) \bowtie \pi_{\overline{R}_2}(\mathbf{r})$. Generalize this result to decompositions of $\overline{R}$ into $n > 2$ schemas.

Solution:
If $t \in \mathbf{r}$, then $t_1 = \pi_{\overline{R}_1}(t)$ and $t_2 = \pi_{\overline{R}_2}(t)$ match over the attributes in $\overline{R}_1 \cap \overline{R}_2$, so this tuple will be in the natural join of $\pi_{\overline{R}_1}(\mathbf{r})$ and $\pi_{\overline{R}_2}(\mathbf{r})$.

For the general case, it is easy to see that

$$\pi_{\overline{R}_1}(\mathbf{r}) \bowtie \cdots \bowtie \pi_{\overline{R}_n}(\mathbf{r}) = \{t \mid t \text{ is a tuple over } \overline{R} \text{ and } \pi_{\overline{R}_1}(t) \in \pi_{\overline{R}_1}(\mathbf{r}), \ldots, \pi_{\overline{R}_n}(t) \in \pi_{\overline{R}_n}(\mathbf{r})\}$$

Clearly, every tuple in $\mathbf{r}$ satisfies the above conditions and thus is in the above set.

6.14 Suppose that $\mathbf{R} = (\overline{R};\ \mathcal{F})$ is a schema and that $\mathbf{R}_1 = (\overline{R}_1;\ \mathcal{F}_1)$, $\mathbf{R}_2 = (\overline{R}_2;\ \mathcal{F}_2)$ is a binary decomposition such that neither $(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_1$ nor $(\overline{R}_1 \cap \overline{R}_2) \rightarrow \overline{R}_2$ is implied by $\mathcal{F}$. Construct a relation, $\mathbf{r}$, such that $\mathbf{r} \subset \pi_{\overline{R}_1}(\mathbf{r}) \bowtie \pi_{\overline{R}_2}(\mathbf{r})$, where $\subset$ denotes strict subset. (This relation, therefore, shows that at least one of these FDs is necessary for the decomposition of $\mathbf{R}$ to be lossless.)

Solution:
Let $\overline{R} = ABC$, $\overline{R}_1 = AB$, and $\overline{R}_2 = BC$. Let $\mathcal{F}$ be empty and $\mathbf{r}$ be the following relation:

A	B	C
1	0	2
3	0	4

Clearly, this relation satisfies the conditions in the statement of the problem. It is easy to see that the join $\pi_{\overline{R}_1}(\mathbf{r}) \bowtie \pi_{\overline{R}_2}(\mathbf{r})$ contains (among others) the tuple $\langle 1, 0, 4 \rangle$, which is not in $\mathbf{r}$.

6.15 Suppose that $\mathbf{R}_1, \ldots, \mathbf{R}_n$ is a decomposition of schema $\mathbf{R}$ obtained by a sequence of binary lossless decompositions (beginning with a decomposition of $\mathbf{R}$). Prove that $\mathbf{R}_1, \ldots, \mathbf{R}_n$ is a lossless decomposition of $\mathbf{R}$.

Solution:
Proof by induction on n.

- Base case: $n = 2$. The decomposition $\mathbf{R}_1$, $\mathbf{R}_2$ of $\mathbf{R}$ is lossless, but construction.
- Inductive assumption. Suppose $\mathbf{R}_1, \ldots, \mathbf{R}_n$ is a lossless decomposition of $\mathbf{R}$.

  We need to prove that $\mathbf{R}_1, \ldots, \mathbf{R}_{n-1}$, $\mathbf{R}'_n$, $\mathbf{R}''_n$ is a lossless decompositions of $\mathbf{R}$, if $\mathbf{R}'_n$, $\mathbf{R}''_n$ is a lossless binary decomposition of $\mathbf{R}_n$,

  Suppose $\mathbf{r}$ is a relation over $\mathbf{R}$ and t is an arbitrary tuple over the attribute set of $\mathbf{R}$ such that

$$\pi_{\mathbf{R}_1}(t) \in \pi_{\mathbf{R}_1}(\mathbf{r}), \ldots, \pi_{\mathbf{R}_{n-1}}(t) \in \pi_{\mathbf{R}_{n-1}}(\mathbf{r}),\ \pi_{\mathbf{R}'_n}(t) \in \pi_{\mathbf{R}'_n}(\mathbf{r}), \pi_{\mathbf{R}''_n}(t) \in \pi_{\mathbf{R}''_n}(\mathbf{r})$$

  Because $\mathbf{R}'_n$ $\mathbf{R}''_n$ is a lossless binary decomposition of $\mathbf{R}_n$, it follows that $\pi_{\mathbf{R}_n}(t) \in \pi_{\mathbf{R}_n}(\mathbf{r})$. Hence, by the definition of a join, $t \in (\pi_{\mathbf{R}_1}(\mathbf{r}) \bowtie \cdots \bowtie \pi_{\mathbf{R}_n}(\mathbf{r})) = \mathbf{r}$. The last equality here holds because $\mathbf{R}_1, \ldots, \mathbf{R}_n$ is a lossless decompositions of $\mathbf{R}$, by the inductive assumption.

6.16 Prove that the loop in the BCNF decomposition algorithm of Figure 6.9 has the property that the database schema at each subsequent iteration has strictly fewer FDs that violate BCNF than has the schema in the previous iteration.

Solution:
Let $\mathbf{S} = (\overline{S}; \mathcal{F})$ be a schema that is not in BCNF. First, we have to clarify *which* sets of FDs possess the properties stated in the exercise. Here we are talking about the set of all *entailed* FDs, not the FDs in $\mathcal{F}$. It is easy to find examples where the number of violating FDs can increase temporarily if we were just looking at the FDs in $\mathcal{F}$ rather than $\mathcal{F}^+$.

To prove the required result, observe that during each iteration, when there is an FD $X \rightarrow Y$, which violates BCNF, the algorithm decomposes the original schema $\mathbf{S} = (\overline{S}; \mathcal{F})$ into $\mathbf{S}_1 = (XY; \mathcal{F}_1)$ and $\mathbf{S}_2 = (\overline{S} - (Y - X); \mathcal{F}_2)$. The set of FDs in $\mathcal{F}_1^+ \cup \mathcal{F}_2^+$ is a subset of $\mathcal{F}^+$, so it does not have any additional FDs that violate BCNF. Moreover, $X \rightarrow Y$ is in $\mathcal{F}_1$, but not in $\mathcal{F}_2$, and it no longer violates BCNF, since X is a superkey in $\mathbf{S}_1$. Therefore the number of violating FDs has decreased.

Note that both $\mathbf{S}_1$ and $\mathbf{S}_2$ can have other FDs that still violate BCNF, so the decomposition process must continue until none is left.

⋆**6.17** Prove that the algorithm for synthesizing 3NF decompositions in Section 6.8.2 yields schemas that satisfy the conditions of 3NF. (*Hint*: Use the proof-by-contradiction technique. Assume that some FD violates 3NF and then show that this contradicts the fact that the algorithm synthesized schemas out of a minimal cover.)

Solution:
Suppose, to the contrary, that one of the schemas, say $(\overline{X}A; \{\overline{X} \rightarrow A, \ldots\})$, which results from the synthesis using the FD set $\mathcal{F}$, is not in 3NF. Then there must be $\overline{Y} \rightarrow B \in \mathcal{F}^+$, where $\overline{Y}B \subset \overline{X}A$, which violates the 3NF conditions.

If $A = B$, then $\overline{Y} \subset \overline{X}$. However, this means that the FD $\overline{X} \rightarrow A$ cannot be in the minimal cover of $\mathcal{F}$, because $\overline{Y} \subset \overline{X}$ and so $\overline{Y} \rightarrow A$ holds and has a smaller left-hand side.

If $A \neq B$, then $B \in \overline{X}$ and B but it cannot be part of a key (because otherwise $\overline{Y} \rightarrow B$ would not have violated 3NF). Thus, there is a key, $\overline{W}$, of $\overline{X}A$ such that

$\overline{W} \subseteq \overline{X} - \{A\}$. In particular, it implies that $\overline{W} \rightarrow A \in \mathcal{F}^+$. Again, this means that $\overline{X} \rightarrow A$ cannot be in a minimal cover of $\mathcal{F}$.

6.18 Consider a database schema with attributes A, B, C, D, and E and functional dependencies $B \rightarrow E$, $E \rightarrow A$, $A \rightarrow D$, and $D \rightarrow E$. Prove that the decomposition of this schema into AB, BCD, and ADE is lossless. Is it dependency preserving?

Solution:
The projection of the dependencies on AB includes $B \rightarrow A$, on BCD includes $B \rightarrow D$, and on ADE includes $A \rightarrow D$, $E \rightarrow A$, $D \rightarrow E$. It is easy to see that this set entails all the original dependencies, so the decomposition is dependency-preserving.

Regarding losslessness, one solution is to note that this decomposition can be obtained using a sequence of binary lossless decompositions. The other solution is to use the definition of a lossless join.

Let $\mathbf{r}$ be a relation over $ABCDE$, and $\mathbf{r}_{\mathtt{AB}}$, $\mathbf{r}_{\mathtt{BCD}}$, $\mathbf{r}_{\mathtt{ADE}}$ be its projections on the corresponding sets of attributes. We need to show that $\mathbf{r}_{\mathtt{AB}} \bowtie \mathbf{r}_{\mathtt{BCD}} \bowtie \mathbf{r}_{\mathtt{ADE}} \subseteq \mathbf{r}$.

Every tuple in the join can be represented as $abcde$, where $ab \in \mathbf{r}_{\mathtt{AB}}$, $bcd \in \mathbf{r}_{\mathtt{BCD}}$, and $ade \in \mathbf{r}_{\mathtt{ADE}}$. We need to show that then $abcde$ must be a tuple in $\mathbf{r}$. Since the relations in the join are projections of $\mathbf{r}$, the tuple ab must be a projection of some tuple $abc_1d_1e_1$ in $\mathbf{r}$; the tuple bcd must be a projection of some a_2bcde_2 in $\mathbf{r}$, and ade must come from some tuple ab_3c_3de.

But because $B \rightarrow E$ holds in $\mathbf{r}$, $e_1 = e_2$. Similarly, because of $E \rightarrow A$ and since tuples $abc_1d_1e_1$ and a_2bcde_2 agree on the attribute E (as we just proved that $e_1 = e_2$), they must agree on A. Hence, $a = a_2$. Likewise, $A \rightarrow D$ implies $d_1 = d$. Finally, $D \rightarrow E$ implies $e = e_1 = e_2$.

If we now apply all these equalities to the first tuple, $abc_1d_1e_1$, we obtain $abcde$. Thus, $abcde$ is in $\mathbf{r}$.

6.19 Consider a relation schema with attributes $ABCGWXYZ$ and the set of dependencies $\mathcal{F} = \{XZ \rightarrow ZYB,\ YA \rightarrow CG,\ C \rightarrow W,\ B \rightarrow G,\ XZ \rightarrow G\}$. Solve the following problems using the appropriate algorithms.

a. Find a minimal cover for $\mathcal{F}$.

Solution:
Minimal cover: After splitting the right-hand sides, we get $XZ \rightarrow G$ plus the following set:

$XZ \rightarrow Y$
$XZ \rightarrow B$
$YA \rightarrow C$
$YA \rightarrow G$
$C \rightarrow W$
$B \rightarrow G$

It is easy to see that the left-hand sides XZ and YA (as well as C and B) cannot be reduced because there are no FDs of the form $X \rightarrow \ldots$, $Y \rightarrow \ldots$, $Z \rightarrow \ldots$, $A \rightarrow \ldots$, or $\{\,\} \rightarrow \ldots$.

Then, we notice that $XZ \rightarrow G$ is redundant and none of the other FDs is, so the minimal cover consists of the above siz FDs.

b. Is the dependency $XZA \rightarrow YB$ implied by $\mathcal{F}$?

Solution:
Yes, because $XZA^+ = XZAYBCWG$, which includes YB.

c. Is the decomposition into $XZYAB$ and $YABCGW$ lossless?

Solution:
Yes, because $XZYAB \cap YABCGW = YAB$ and $YAB^+ = YAMCGW$.

d. Is the above decomposition dependency preserving?

Solution:
Yes, because every FD in the above minimal cover is embedded into either $XZYAB$ or $YABCGW$.

6.20 Consider the following functional dependencies over the attribute set $ABCDEFGH$:

$A \rightarrow E$	$BE \rightarrow D$
$AD \rightarrow BE$	$BDH \rightarrow E$
$AC \rightarrow E$	$F \rightarrow A$
$E \rightarrow B$	$D \rightarrow H$
$BG \rightarrow F$	$CD \rightarrow A$

Find a minimal cover, then decompose into lossless 3NF. After that, check if all the resulting relations are in BCNF. If you find a schema that is not, decompose it into a lossless BCNF. Explain all steps.

Solution:
Minimal cover: First split $AD \rightarrow BE$ into $AD \rightarrow B$ and $AD \rightarrow E$. The next step is to reduce the left-hand sides of the FDs. Since $A \rightarrow B$ and $A \rightarrow E$ are implied by the given set of FDs, we can replace $AD \rightarrow B$, $AD \rightarrow E$, and $AC \rightarrow E$ in the original set with $A \rightarrow B$ ($A \rightarrow E$ already exists in the original set). Likewise $E \rightarrow D$ and $BD \rightarrow E$ can be derived from the original set so we can replace $BE \rightarrow D$ and $BDH \rightarrow E$ with $E \rightarrow D$ and $BD \rightarrow E$.

In the next step, we remove redundant FDs, of which we find only $A \rightarrow B$. The final result is therefore

---

$A \rightarrow E$
$E \rightarrow B$
$BG \rightarrow F$
$E \rightarrow D$
$BD \rightarrow E$
$F \rightarrow A$
$D \rightarrow H$
$CD \rightarrow A$

Lossless 3NF:
$(AE;\ \{A \rightarrow E\}), (EBD;\ \{BD \rightarrow E,\ E \rightarrow D,\ E \rightarrow B\}),$
$(BGF;\ \{BG \rightarrow F\}), (FA;\ \{F \rightarrow A\}), (DH;\ \{D \rightarrow H\}), (CDA;\ \{CD \rightarrow A).$

Since BGF is a superkey of the original schema, we do not need to add anything to this decomposition.

The schema $(BGF;\ \{BG \rightarrow F\})$ is not in BCNF because $F \rightarrow B$ is entailed by the original set of FDs and $(CDA;\ \{CD \rightarrow A)$ is not in BCNF because $A \rightarrow D$ is entailed.

We decompose BGF with respect to $F \rightarrow B$ into $(BF; \{F \rightarrow B\})$ and $(GF; \{\})$, thereby loosing the FD $BG \rightarrow F$.

Similarly CDA is decomposed using $A \rightarrow D$ into $(AD; \{A \rightarrow D\})$ and $(AC; \{\})$, loosing $CD \rightarrow A$.

6.21 Find a projection of the following set of dependencies on the attributes AFE:

$A \rightarrow BC$ $\qquad$ $E \rightarrow HG$

$C \rightarrow FG$ $\qquad$ $G \rightarrow A$

Solution:

A → F
E → AF

6.22 Consider the schema with the attribute set $ABCDEFH$ and the FDs depicted in (6.12), page 224. Prove that the decomposition $(AD;\ A \rightarrow D)$, $(CE;\ C \rightarrow E)$, $(FA;\ F \rightarrow A)$, $(EF;\ E \rightarrow F)$, $(BHE;\ BH \rightarrow E)$ is not lossless by providing a concrete relation instance over $ABCDEFH$ that exhibits the loss of information when projected on this schema.

Solution:

A	B	C	D	E	F	H
a	b1	b1	a	b1	b1	b1
a	b2	a	a	a	a	b2
a	b3	b3	a	b3	a	b3
a	b4	b4	a	a	a	b4
a	a	b5	a	a	a	a

This relation satisfies all the FDs. It is easy to verify, however, that the tuple a a a a a a a belongs to the join of the projections of this relation on the above schemas, but it is not one of the tuples in this relation. Therefore, this relation exhibits a case where information loss occurs due to the decomposition.

6.23 Consider the schema $BCDFGH$ with the following FDs: $BG \rightarrow CD$, $G \rightarrow F$, $CD \rightarrow GH$, $C \rightarrow FG$, $F \rightarrow D$. Use the 3NF synthesis algorithm to obtain a lossless, dependency-preserving decomposition into 3NF. If any of the resulting schemas is not in BCNF, proceed to decompose them into BCNF.

Solution:
Minimal cover: First, split the right-hand sides into $BG \rightarrow C$, $BG \rightarrow D$, $CD \rightarrow G$, $CD \rightarrow H$, $C \rightarrow F$, and $C \rightarrow G$. Next reduce the left-hand sides. $CD \rightarrow G$ reduces to

$C \to G$ (and is eliminated, since it is a duplicate), $CD \to H$ reduces to $C \to H$, and $BG \to D$ reduces to $G \to D$. Thus, after the second step we end up with

---

BG → C
G → D
G → F
C → H
C → F
C → G
F → D

The last stage eliminates the redundant FDs and we end up with:

BG → C
G → F
C → H
C → G
F → D

A lossless, dependency-preserving decomposition into 3NF: $(BGC;\ \{BG \to C,\ C \to G\}), (GF;\ \{G \to F\}), (CGH;\ \{C \to GH\}), (FD;\ \{F \to D\})$.

The schema $(BGC;\ \{BG \to C,\ C \to G\})$ is not in BCNF. A decomposition with respect to $C \to G$ into $(CG; \{C \to G\})$ and $(CB; \{\})$ brings it into BCNF, but looses the FD $BG \to C$.

⋆**6.24** Prove that all rules for inferring FDs and MVDs given in Section 6.10 are sound. In other words, for every relation, **r**, which satisfies the dependencies in the premise of any rule, R, the conclusion of R is also satisfied by **r** (e.g., for the augmentation rule, prove that if $\overline{X} \twoheadrightarrow \overline{Y}$ holds in **r** then $\overline{X}\,\overline{Z} \twoheadrightarrow \overline{Y}$ also holds in **r**).

Solution:
The proof uses a technique, which we used for proving the soundness of Armstrong's axioms. However, the reasoning is much more involved. To illustrate, we prove the soundness of the transitivity axiom

$$\overline{X} \twoheadrightarrow \overline{Y} \text{ and } \overline{Y} \twoheadrightarrow \overline{Z} \text{ entail } \overline{X} \twoheadrightarrow \overline{Z} - \overline{Y}.$$

Most of the other proofs are every bit as tedious.

Let $\overline{R}$ be the set of all attributes in the schema. Then we have the following binary join dependencies:

JD1: $\overline{R} = \overline{XY} \bowtie \overline{X}(\overline{R} - \overline{Y})$
JD2: $\overline{R} = \overline{YZ} \bowtie \overline{Y}(\overline{R} - \overline{Z})$

We need to show that every relation, **r**, that satisfies these two JDs also satisfies the following JD:

$$\overline{X}(\overline{Z}-\overline{Y}) \bowtie \overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))$$

To see this, let t be a tuple over $\overline{R}$ such that $\pi_{\overline{X}(\overline{Z}-\overline{Y})}(t) \in \pi_{\overline{X}(\overline{Z}-\overline{Y})}(\mathbf{r})$ and $\pi_{\overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))}(t) \in \pi_{\overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))}(\mathbf{r})$. To prove the soundness of the inference rule we need to show that $t \in \mathbf{r}$.

The proof relies on the result of exercise 6.31(a). According to this result, the projection of JD1 on $\overline{XYZ}$ holds in $\pi_{\overline{XYZ}}(\mathbf{r})$.

Now notice that the intersection of $\overline{X}(\overline{Z}-\overline{Y})$ and $\overline{XYZ}$ is $\overline{X}(\overline{Z}-\overline{Y})$ and the intersection of $\overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))$ with $\overline{XYZ}$ is $\overline{XY}$. Also notice that $\overline{X}(\overline{Z}-\overline{Y})$ is the intersection of $\overline{XYZ}$ with $\overline{X}(\overline{R}-\overline{Y})$.

This means that $\pi_{\overline{X}(\overline{Z}-\overline{Y})}(t) \in \pi_{\overline{X}(\overline{Z}-\overline{Y})}(\mathbf{r})$ (as we saw earlier) and $\pi_{\overline{XY}}(t) \in \pi_{\overline{XY}}(\mathbf{r})$. Since the projection of JD1 on $\overline{XYZ}$ holds in $\pi_{\overline{XYZ}}(\mathbf{r})$, it means that $\pi_{\overline{XYZ}}(t) \in \pi_{\overline{XYZ}}(\mathbf{r})$. In particular, $\pi_{\overline{YZ}}(t) \in \pi_{\overline{YZ}}(\mathbf{r})$.

Since $\pi_{\overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))}(t) \in \pi_{\overline{X}(\overline{R}-(\overline{Z}-\overline{Y}))}(\mathbf{r})$ (by assumption) and $\overline{R}-(\overline{Z}-\overline{Y}) = \overline{Y}(\overline{R}-\overline{Z})$, it follows that both $\pi_{\overline{YZ}}(t) \in \pi_{\overline{YZ}}(\mathbf{r})$ and $\pi_{\overline{Y}(\overline{R}-\overline{Z})}(t) \in \pi_{\overline{Y}(\overline{R}-\overline{Z})}(\mathbf{r})$. Because JD2 holds in $\mathbf{r}$, it follows that $t \in \mathbf{r}$, which completes the proof.

The above proof (and the proofs for other rules) can be simplified somewhat by making the left and right hand sides of the MVDs disjoint. This is possible because it is easy to show (directly from the definitions) that the MVD $X \twoheadrightarrow Y$ is equivalent to $X \twoheadrightarrow (Y - X)$.

6.25 Prove that if a schema, $\mathbf{S} = (\overline{S}, \mathcal{F})$, is in 3NF, then every FD in $\mathcal{F}^+$ (not only those that are in $\mathcal{F}$) satisfies the 3NF requirements.

Solution:
Suppose to the contrary, that there is an FD $\overline{X} \rightarrow A \in (\mathcal{F}^+ - \mathcal{F})$ such that $\overline{X}$ is not a superkey, A does not belong to any key of $\mathbf{S}$, and the FD is non-trivial, i.e., $A \notin \overline{X}$. As we know, it must be the case that $A \in \overline{X}^+_{\mathcal{F}}$ and $\overline{X}^+_{\mathcal{F}}$ should be computable by the attribute closure algorithm in Figure 6.3. The main step in that algorithm hinges on being able to find an FD $\overline{Z} \rightarrow \overline{V} \in \mathcal{F}$ such that $\overline{Z} \subseteq \mathit{closure}$. Since $\mathit{closure} \subseteq \overline{X}^+_{\mathcal{F}}$ and $\overline{X}$ is not a superkey, none of the FDs in $\mathcal{F}$ whose left-hand side is a superkey can be used in that algorithm. Therefore, $\overline{Z} \rightarrow \overline{V}$ must be one of those FDs where the attributes of $\overline{V}$ belong to the various keys of $\mathbf{S}$. But in order to add A to $\mathit{closure}$, it must belong to $\overline{V}$ for some such FD, i.e., A itself must be an attribute of one of the keys in $\mathbf{S}$ — a contradiction.

6.26 If $X = \{A, B\}$ and $F = \{A \rightarrow D,\ BC \rightarrow EJ,\ BD \rightarrow AE,\ EJ \rightarrow G,\ ADE \rightarrow H,\ HD \rightarrow J\}$, what is X^+_F? Are the FDs $AB \rightarrow C$ and $AE \rightarrow G$ entailed by F?

Solution:
$AB \rightarrow C$ is not entailed; $AE \rightarrow G$ is entailed by F.

6.27 Using only Armstrong's axioms and the FDs

(a) $AB \rightarrow C$
(b) $A \rightarrow BE$
(c) $C \rightarrow D$

give a complete derivation of the FD $A \rightarrow D$.

Solution:

---

(1) $A \rightarrow BE$ using (b)
(2) $BE \rightarrow B$ using Reflexivity
(3) $A \rightarrow B$ using Transitivity on (1) and (2)
(4) $A \rightarrow AB$ using Augmentation on (3)
(5) $A \rightarrow C$ using Transitivity on (4) and (a)
(6) $A \rightarrow D$ using Transitivity on (5) and (c)

6.28 Consider a decomposition $\mathbf{R}_1, \ldots, \mathbf{R}_n$ of $\mathbf{R}$ obtained via steps 1, 2, and 3 (but not step 4) of the 3NF synthesis algorithm on page 225. Suppose there is an attribute A in $\mathbf{R}$ that does not belong to any of the $\mathbf{R}_i$, $i = 1, \ldots, n$. Prove that A must be part of every key of $\mathbf{R}$.

Solution:
If A does not occur in any of the schemas constructed via the steps 1,2,3 of the 3NF synthesis algorithm, it means that A does not occur in any of the FDs in the minimal cover of the FDs of the schema. Therefore, there is no way by which A might be in the attribute closure of an attribute set that does not contain A.

6.29 Consider the schema $\mathbf{R} = (ABCDEFGH, \{BE \rightarrow GH,\ G \rightarrow FA,\ D \rightarrow C,\ F \rightarrow B\})$.

a. Can there be a key that does not contain D? Explain.

Solution:
No - Since D is not uniquely determined by any other attributes, it must be part of all keys.

b. Is the schema in BCNF? Explain.

Solution:
No - None of the left hand sides are superkeys.

c. Use one cycle of the BCNF algorithm to decompose $\mathbf{R}$ into two subrelations. Are the subrelations in BCNF?

Solution:
We can choose any of the FD as a basis for a decomposition. For example, **R1** $= (ABDEFGH, \{BE \rightarrow GH,\ G \rightarrow FA,\ F \rightarrow B\})$, **R2** $= (DC, \{D \rightarrow C\})$. In this case **R1** is not in BCNF (since the LHS of all its FDs are not superkeys), but **R2** is.

d. Show that your decomposition is lossless.

Solution:
$\{ABDEFGH\} \cap \{CD\}$ is a key of **R2**.

e. Is your decomposition dependency preserving? Explain.

Solution:
The decomposition is dependency preserving because each of the FDs of **R** is a FD of one of the component relations.

6.30 Find a minimal cover of the following set of FDs: $AB \rightarrow CD,\ BC \rightarrow FG,\ A \rightarrow G,\ G \rightarrow B,\ C \rightarrow G$. Is the decomposition of $ABCDFG$ into $ABCD$ and $ACFG$ lossless? Explain.

Solution:
Splitting the RHSs replaces the first two FDs with: $AB \rightarrow C,\ AB \rightarrow D,\ BC \rightarrow F,\ BC \rightarrow G$.

Since we can derive $A \rightarrow B$, we can delete B from the LHSs of the first two of these: $A \rightarrow C$, $A \rightarrow D$. Because of $C \rightarrow G \rightarrow B$, we can eliminate B from $BC \rightarrow G$ and $BC \rightarrow F$. We are left with: $A \rightarrow C,\ A \rightarrow D,\ C \rightarrow F,\ C \rightarrow G,\ A \rightarrow G,\ G \rightarrow B$.

$A \rightarrow G$ is redundant, so we are left with $A \rightarrow C,\ A \rightarrow D,\ C \rightarrow F,\ C \rightarrow G,\ G \rightarrow B$.

The decomposition of $ABCDFG$ into $ABCD$ and $ACFG$ is lossless because $AC = ABCD \cap ACFG$ and $AC \rightarrow FG$ is implied by the given FDs.

6.31 Let $\overline{X}, \overline{Y}, \overline{S}, \overline{R}$ be sets of attributes such that $\overline{S} \subseteq \overline{R}$ and $\overline{X} \cup \overline{Y} = R$. Let $\mathbf{r}$ be a relation over $\overline{R}$ that satisfies the nontrivial MVD $\overline{R} = \overline{X} \bowtie \overline{Y}$ (i.e., neither set $\overline{X}$ or $\overline{Y}$ is a subset of the other).

a. Prove that if $\overline{X} \cap \overline{Y} \subseteq \overline{S}$, then the relation $\pi_{\overline{S}}(\mathbf{r})$ satisfies the MVD $\overline{S} = (\overline{S} \cap \overline{X}) \bowtie (\overline{S} \cap \overline{Y})$.

Solution:
If t is a tuple over $\overline{S}$ such that $t_X = \pi_{\overline{S} \cap \overline{X}}(t) \in \pi_{\overline{S} \cap \overline{X}}(\mathbf{r})$ and $t_Y = \pi_{\overline{S} \cap \overline{Y}}(t) \in \pi_{\overline{S} \cap \overline{Y}}(\mathbf{r})$ then t_X and t_Y match over the attributes in $\overline{X} \cap \overline{Y}$. We can extend these tuples to $t'_X \in \pi_{\overline{X}}(\mathbf{r})$ and $t'_Y \in \pi_{\overline{Y}}(\mathbf{r})$. Being extensions of t_X and t_Y, these tuples also match on $\overline{X} \cap \overline{Y}$. Since the MVD $\overline{R} = \overline{X} \bowtie \overline{Y}$ holds, t'_X and t'_Y join over $\overline{X} \cap \overline{Y}$ into a tuple $t' \in \mathbf{r}$. Since $t = \pi_{\overline{S}}(t')$, it follows that $t \in \pi_{\overline{S}}(\mathbf{r})$. This proves that $\pi_{\overline{S}}(\mathbf{r}) = \pi_{\overline{S} \cap \overline{X}}(\mathbf{r}) \bowtie \pi_{\overline{S} \cap \overline{Y}}(\mathbf{r})$.

b. Suppose $\overline{X}$, $\overline{Y}$, $\overline{S}$, and $\overline{R}$ satisfy all the above conditions, except that $\overline{X} \cap \overline{Y} \not\subseteq \overline{S}$. Give an example of $\mathbf{r}$ that satisfies $\overline{R} = \overline{X} \bowtie \overline{Y}$ but does not satisfy $\overline{S} = (\overline{S} \cap \overline{X}) \bowtie (\overline{S} \cap \overline{Y})$.

Solution:
Let $A \in (\overline{X} \cap \overline{Y}) - \overline{S}$.

Let $\mathbf{r}$ consist of just two tuples, t and s, such that:

- t has the value 1 over the attributes in $\overline{X} - \overline{Y}$ and $\overline{Y} - \overline{X}$, the value 2 over the attributes in $(\overline{X} \cap \overline{Y}) - A$, and the value 3 over the attribute A
- s has the value 4 over the attributes in $\overline{X} - \overline{Y}$ and $\overline{Y} - \overline{X}$, the value 2 over the attributes in $(\overline{X} \cap \overline{Y}) - A$, and the value 5 over the attribute A

It is easy to see that projecting this relation on $\overline{X}$ and $\overline{Y}$ and then joining back yields the original relation. However, if we project $\mathbf{r}$ on $\overline{S}$, then the attribute A is projected out and thus the tuples s and t match on $\overline{X} \cap \overline{Y} \cap \overline{S}$. Therefore,

$$\pi_{\overline{S} \cap \overline{X}}(\mathbf{r}) \bowtie \pi_{\overline{S} \cap \overline{Y}}(\mathbf{r})$$

contains the extraneous tuples of the form 111222444 and 444222111, which are not in $\pi_{\overline{S}}(\mathbf{r})$.

6.32 Consider a relation schema over the attributes $ABCDEFG$ and the following MVDs:

$$ABCD \bowtie DEFG$$
$$CD \bowtie ABCEFG$$
$$DFG \bowtie ABCDEG$$

Find a lossless decomposition into 4NF.

Solution:
Use the first MVD to obtain the following decomposition: $ABCD$, $DEFG$. The second MVD still applies to $ABCD$ and decomposes it into CD, ABC. The third MVD applies to $DEFG$ and decomposes it into DFG, DGE. Thus, the result is CD, ABC, DFG, DGE.

6.33 For the attribute set $ABCDEFG$, let the MVDs be:

$$ABCD \bowtie DEFG$$
$$ABCE \bowtie ABDFG$$
$$ABD \bowtie CDEFG$$

Find a lossless decomposition into 4NF. Is it unique?

Solution:
Use the first MVD to obtain the following decomposition: $ABCD$, $DEFG$. The second MVD still applies to $ABCD$ and yields ABC, ABD. The third MVD cannot be used to decompose ABC because the join attribute, D, is not in this attribute set. It cannot be used to decompose ABD or $DEFG$ because these sets are subsets of one of the components of that MVD, so the MVD yields trivial decompositions in these cases. Thus, the result is ABC, ABD, $DEFG$.

The above decomposition is not unique. If we apply the third MVD first, we would obtain the following result: ABD, CD, $DEFG$.

6.34 Consider a decomposition $\mathbf{R}_1, \ldots, \mathbf{R}_n$ of $\mathbf{R}$ obtained through 3NF synthesis. Suppose that $\mathbf{R}_i$ is *not* in BCNF and let $X \rightarrow A$ be a violating FD in $\mathbf{R}_i$. Prove that $\mathbf{R}_i$ must have another FD, $Y \rightarrow B$, which will be lost if $\mathbf{R}_i$ is further decomposed with respect to $X \rightarrow A$.

Solution:
If $X \rightarrow A$ is a violating FD of $\mathbf{R}_i$, there must be a key K in $\mathbf{R}_i$ and an FD $K \rightarrow B$ in the minimal cover, such that $A \in K$, $X \not\subseteq K$, and $B \notin K$. Since the decomposition with respect to $X \rightarrow A$ splits the attributes of K between the schemas XA and $\overline{R_i} - A$, where $\overline{R_i}$ is the set of attributes of $\mathbf{R}_i$, the FD $K \rightarrow B$ is not embedded in any schema.

If it were possible to derive $K \rightarrow B$ using the remaining FDs, then it would mean that $K \rightarrow B$ is redundant, contrary to the fact that it was chosen from a minimal cover.

***6.35** This exercise relies on a technique explained in the optional Section 6.10. Consider a relation schema over the attributes $ABCDEFGHI$ and the following MVDs and FDs:

$$D \rightarrow AH \qquad D \twoheadrightarrow BC$$

$$G \rightarrow I \qquad C \twoheadrightarrow B$$
$$G \twoheadrightarrow ABCE$$

Find a lossless and dependency-preserving decomposition into 4NF.

Solution:
The first step is to close the left-hand sides of the MVDs with respect to the FDs:

$$\begin{aligned} D &\rightarrow AH \\ G &\rightarrow I \\ DAH &\twoheadrightarrow BC \\ C &\twoheadrightarrow B \\ GI &\twoheadrightarrow ABCE \end{aligned}$$

The 4NF decomposition algorithm first yields $ABCDH$, $ADHEFGI$. Then $ABCDH$ is split into CB, $CADH$. $ADHEFGI$ is split by the third MVD into $GIAE$, $GIDHF$. Thus, after the 4NF decomposition stage we get: CB, $CADH$, $GIAE$, $GIDHF$.

The last three schemas are not in 3NF, due to the FDs. A lossless, dependency-preserving 3NF decomposition of $CADH$ is DAH, DC. A 3NF decomposition of $GIAE$ is GI, GAE, and A 3NF decomposition of $GIDHF$ is DH, GI, DGF. Since DAH and DH have the same key, we can eliminate DH and obtain the following final result: CB, DC, DAH, GI, GAE, DGF.

6.36 Let $\mathbf{R} = (\overline{R};\ \mathcal{F})$ be a schema and $\overline{S}$ a set of attributes, such that $\overline{S} \subseteq \overline{R}$. Let $\overline{X} \rightarrow \overline{Y}$ be an FD such that $\overline{X}, \overline{Y} \subseteq S$. Prove that

1. If $\overline{X} \rightarrow \overline{Y} \in \mathcal{F}^+$ then for *every* legal instance $\mathbf{r}$ of $\mathbf{R}$ its projection $\pi_{\overline{S}}(\mathbf{r})$ satisfies $\overline{X} \rightarrow \overline{Y}$.
2. Conversely, if $\overline{X} \rightarrow \overline{Y} \notin \mathcal{F}^+$ then there is a legal instance $\mathbf{r}$ of $\mathbf{R}$ such that $\pi_{\overline{S}}(\mathbf{r})$ violates $\overline{X} \rightarrow \overline{Y}$.

Solution:

1. Since $\overline{X} \rightarrow \overline{Y} \in \mathcal{F}^+$, this FD must be satisfied by $\mathbf{r}$. Since the attributes of this FD belong to $\overline{S}$, it follows directly from the definition of FDs that $\overline{X} \rightarrow \overline{Y}$ is satisfied by $\pi_{\overline{X}}(\mathbf{r})$.
2. If $\overline{X} \rightarrow \overline{Y} \notin \mathcal{F}^+$ then there is a legal instance of $\mathbf{R}$, $\mathbf{r}$, which satisfies $\mathcal{F}$, but violates $\overline{X} \rightarrow \overline{Y}$. Again, by the definition of FDs, $\pi_{\overline{X}}(\mathbf{r})$ must violate this dependency.

7

Triggers and Active Databases

EXERCISES

7.1 Explain the semantics of the triggers that are available in the DBMS that is used for your course project. Describe the syntax for defining these triggers.

Solution:
Depends on the system. For instance, IBM's DB/2 has the same semantics as the SQL standard.

7.2 Give the exact syntactic rules for constructing the triggering graphs from the sets of SQL triggers and foreign-key constraints.

Solution:
For each SQL trigger the *activating* event is an update, deletion, or insertion of/from/into the relation specified in the CREATE TRIGGER . . . BEFORE/AFTER . . . ON *Table* clause. Whether the actually event must be an update, deletion or insertion depends on what is specified in the BEFORE/AFTER clause. The events *produced* by the trigger are updates, insertions, and deletions on the relations specified in the action part of the trigger. (The exact type of the event depends on the form of the action specified.)

Note that a trigger with the FOR EACH STATEMENT clause produces only one event, while a trigger with the FOR EACH ROW clause produces one event per updated/inserted/deleted tuple (whichever applies).

For a foreign-key constraint, the *activating* event is any update or deletion that involves the relation mentioned in the REFERENCES clause and the events produced are updates or deletions on the relation mentioned in the CREATE TABLE statement, which this foreign key is part of.

The triggering graph then consists of the edges among all these triggers and foreign-key constraints. An edge is drawn from one trigger/constraint T to another, S, if T produces an event that activates S.

7.3 Design a trigger that complements the trigger MaintainCoursesNonEmpty (see (7.1) on page 260) by precluding the insertion of tuples into the relation Teaching when there are no corresponding tuples in the Transcript relation.

Solution:

```
CREATE TRIGGER  PrecludeIdleTeaching
     BEFORE INSERT OF CrsCode, Semester ON Teaching
     REFERENCING OLD AS  O
     FOR EACH ROW
     WHEN  (NOT EXISTS  (
                  SELECT   *
                  FROM     Transcript
                  WHERE    Semester = O.Semester AND
                           CrsCode = O.CrsCode ) )
          ROLLBACK
```

7.4 Design a trigger that works like MaintainCoursesNonEmpty but is a row-level trigger.

Solution:

```
CREATE TRIGGER  MaintainCoursesNonEmpty2
     AFTER DELETE, UPDATE OF CrsCode, Semester ON Transcript
     REFERENCING OLD AS O
     FOR EACH ROW
     WHEN  (NOT EXISTS  (
                  SELECT   *
                  FROM     Transcript
                  WHERE    Semester = O.Semester AND
                           CrsCode = O.CrsCode ) )
          DELETE FROM  Teaching
          WHERE  Semester = O.Semester AND
                       CrsCode = O.CrsCode
```

7.5 Define the trigger WatchCourseHistory that uses a table Log to record all changes that transactions make to the various courses in the Course relation.

Solution:

```
CREATE TRIGGER  WatchCourseHistory
     AFTER UPDATE, DELETE, INSERT  ON  Course
     REFERENCING OLD TABLE AS  OldT
               NEW TABLE AS  NewT
     FOR EACH STATEMENT
     BEGIN
          INSERT INTO  Log
               SELECT  CURRENT_DATE, 'OLD', * FROM  OldT;
          INSERT INTO  Log
```

```
            SELECT CURRENT_DATE, 'NEW', * FROM NewT
    END
```

7.6 Define triggers that fire when a student drops a course, changes her major, or when her grade average drops below a certain threshold. (For simplicity, assume that there is a function, `grade_avg()`, which takes a student `Id` and returns the student average grade.)

Solution:
The action part of these triggers is uninteresting—just an insertion of a record into the log table, as we saw in earlier problems. We give an example of the last trigger.

```
CREATE TRIGGER WatchGPA
    AFTER UPDATE, DELETE, INSERT ON Transcript
    FOR EACH STATEMENT
    WHEN (EXISTS (
            SELECT StudId FROM Transcript T
            WHERE grade_avg(StudId) < 3.0 ) )
        INSERT INTO Log
            SELECT DISTINCT StudId, grade_avg(StudId)
            FROM Transcript T
            WHERE grade_avg(StudId) < 3.0
```

7.7 Consider the IsA relationship between Student(`Id`,`Major`) and Person(`Id`, `Name`). Write the triggers appropriate for maintaining this relationship: when a tuple is deleted from Person, the tuple with the same Id must be deleted from Student; when a tuple is inserted into Student, check whether a corresponding tuple exists in Person and abort if not. (Do not use the ON DELETE and ON INSERT clauses provided by the FOREIGN KEY statement.)

Solution:

```
CREATE TRIGGER MaintainStudIsaPerson1
    AFTER DELETE ON Person
    REFERENCING OLD AS O
    FOR EACH ROW
        DELETE FROM Student
            WHERE Id = O.Id
```

```
CREATE TRIGGER MaintainStudIsaPerson2
    BEFORE INSERT ON Student
    REFERENCING NEW AS N
    FOR EACH ROW
    WHEN (NOT EXISTS(
```

```
            SELECT * FROM PERSON P
            WHERE P.Id = N.Id ) )
        ROLLBACK
```

7.8 Consider a brokerage firm database with relations HOLDINGS(`AccountId`, `StockSymbol`, `CurrentPrice`, `Quantity`) and BALANCE(`AccountId`, `Balance`). Write the triggers for maintaining the correctness of the account balance when stock is bought (a tuple is added to HOLDINGS or `Quantity` is incremented), sold (a tuple is deleted from HOLDINGS or `Quantity` is decremented), or a price change occurs.

Solve the problem using both row-level and statement-level triggers. Give an example of a situation when row-level triggers are more appropriate for the above problem and when statement-level triggers are more appropriate.

Solution:

```
CREATE TRIGGER UPDATEBALANCEREALTIME
    BEFORE AFTER INSERT, DELETE, UPDATE ON HOLDINGS
    REFERENCING NEW AS N
    FOR EACH ROW
        UPDATE BALANCE
        SET Balance =
            (SELECT SUM(H.CurrentPrice*H.Quantity)
             FROM HOLDINGS H
             WHERE H.AccountId = N.AccountId )
```

The above trigger is appropriate for real-time updates of HOLDINGS, so the balances are also updated in real time. If HOLDINGS is updated only periodically (e.g., every day), then a statement level trigger would be more efficient. This trigger can work by erasing the old contents of BALANCE and then recomputing it from scratch:

```
CREATE TRIGGER UPDATEBALANCEALLATONCE
    BEFORE AFTER INSERT, DELETE, UPDATE ON HOLDINGS
    FOR EACH STATEMENT
    BEGIN
      DELETE FROM BALANCE;   -- Erase
      INSERT INTO BALANCE
        SELECT DISTINCT H.AccountId,SUM(H.CurrentPrice*H.Quantity)
        FROM HOLDINGS H
        GROUP BY H.AccountId
    END
```

7.9 Consider an enterprise in which different projects use parts supplied by various suppliers. Define the appropriate tables along with the corresponding foreign-key constraints. Define triggers that fire when a project changes a supplier for a part; when a supplier discontinues a part; or when a project stops using a part.

Solution:

```
CREATE TABLE  Contract  (
     PartId        INTEGER,
     SupplierId    INTEGER,
     ProjectId     INTEGER,
     PRIMARY KEY (PartId, SupplierId, ProjectId),
     FOREIGN KEY (PartId, SupplierId) REFERENCES Supplies
     FOREIGN KEY (PartId) REFERENCES Part
     FOREIGN KEY (ProjectId) REFERENCES Project
     FOREIGN KEY (SupplierId) REFERENCES Supplier )
```

In addition, we have a table Supplies, which tells which supplier supplies what, and the relations Part, Project, and Supplier, which describe the corresponding entities.

```
CREATE TRIGGER  ProjectChangedSupplier
     BEFORE UPDATE OF SupplierId ON Contract
     REFERENCING NEW AS  N
     FOR EACH ROW
          INSERT INTO    Log
          VALUES    (CURRENT_DATE, 'Supplier Change', ProjectId,
                      PartId, SupplierId, N.SupplierId)
```

```
CREATE TRIGGER  SupplierDiscontinuedPart
     AFTER DELETE ON Supplies
     REFERENCING OLD AS  O
     FOR EACH ROW
     UPDATE  Contract
     SET       SupplierId = NULL
     WHERE     PartId = O.PartId AND SupplierId=O.SupplierId
```

```
CREATE TRIGGER  ProjectStoppedUsingPart
     AFTER DELETE ON Contract
     REFERENCING OLD AS  O
     FOR EACH ROW
     WHEN (NOT EXISTS (
               SELECT *
               FROM Contract C
               WHERE C.PartId = O.PartId ))
          INSERT INTO  Log
```

```
        VALUES  (CURRENT_DATE, 'Stopped part use', ProjectId,
                 PartId, NULL, O.SupplierId)
```

7.10 Consider triggers with immediate consideration and deferred execution. What do OLD AS and NEW AS refer to during consideration and during execution?

Solution:
They refer to the state of the updated tuple just before the triggering event and just after it. It does not matter whether the execution is immediate or deferred.

7.11 Give an example of an application where SQL triggers could be used for a purpose other than just maintaining integrity constraints.

Solution:
When something unusual happens during the execution of a transaction, a trigger can be used to initiate another transaction, which will report the unusual event to the system administrator or take a corrective action.

8

Using SQL in an Application

EXERCISES

8.1 Explain why a precompiler is needed for embedded SQL and SQLJ but not for ODBC and JDBC.

Solution:
In embedded SQL and SQLJ, SQL statements are included as statements that would not be recognized by the host language compiler. In ODBC and JDBC, SQL statements are included as arguments to procedure calls, which can be processed by the host language compiler.

8.2 Since the precompiler for embedded SQL translates SQL statements into procedure calls, explain the difference between embedded SQL and call-level interfaces, such as ODBC and JDBC, where SQL statements are specified as the arguments of procedure calls.

Solution:
With embedded SQL, the actual SQL statements to be executed are known at compile time while with the call-level interfaces, the SQL statement might not be known at compile time. They are just string variables that can be computed at run time and passed as arguments to host language procedures provided by the CLI. Thus in embedded SQL, the syntax of the SQL statements can be checked at compile time, and host language variables can be used as parameters of the SQL statements.

8.3 Explain why constraint checking is usually deferred in transaction processing applications.

Solution:
Because in many transactions, the intermediate database states violate the integrity constraints but the final states do not.

8.4 Give an example where immediate constraint checking is undesirable in a transaction processing application.

Solution:
The registration transaction for the Student Registration System executes two SQL update statements, and the intermediate state might violate an integrity constraint.

8.5 Explain the advantages and disadvantages of using stored procedures in transaction processing applications.

Solution:

1. Stored procedures can reduce the communication between server and client because only the arguments and the results need be communicated.
2. The statements within a stored procedure will have been prepared before execution of that procedure and will execute more efficiently.
3. The checking of authorization privileges can be performed by the DBMS at the level of the stored procedure.
4. The application programmer need not know the details of the database schema, since all database accesses can be encapsulated within the procedure body.
5. Maintenance of the system is simplified since only one copy of the procedure, stored on the server, need be maintained and updated.
6. The physical security of the code for the procedure is higher because it is stored on the server rather than with the application program.
7. A possible disadvantage is that since preparation is done in advance, the database might have changed and the selected query execution plan might be inefficient. However, most systems allow preparation to be done over again at the discretion of the programmer or system administrator.

8.6 Write transaction programs in

a. Embedded SQL
b. JDBC

Solution:
See Section C.7.3.

c. ODBC
d. SQLJ

that implement the registration transaction in the Student Registration System. Use the database schema from Figures 4.34 and 4.35.

8.7 Write transaction programs in

a. Embedded SQL
b. JDBC
c. ODBC
d. SQLJ

that use a cursor to print out a student's transcript in the Student Registration System. Use the database schema from Figures 4.34 and 4.35.

8.8 Explain the following:

a. Why do embedded SQL and SQLJ use host language variables as parameters, whereas dynamic SQL, JDBC, and ODBC use the ? placeholders?

Solution:
With embedded SQL, the precompiler and compiler generate code for fetching and storing arguments between host language variables and SQL statements. Since the SQL statement is not known at compile time for dynamic SQL, JDBC, and ODBC, the compiler cannot generate that code and placeholders must be used.

b. What are the advantages of using host language variables as parameters in embedded SQL compared with ? placeholders?

Solution:
Syntax checking can be done at compile time, and less communication is required at run time.

8.9 Explain the advantages and disadvantages of using dynamic SQL compared with ODBC and JDBC.

Solution:

1. *Advantages*: Dynamic SQL can use all the proprietary extensions to SQL provided by the special DBMS, while JDBC and ODBC can not;
2. *Disadvantages*: In ODBC, the SQL statements use a generic dialect. Likewise, in JDBC, a common core of SQL is guaranteed to be supported by all drivers. If the application is moved to a different DBMS, only the the driver need be changed. (In JDBC, the application should be restricted to the common SQL core). In contrast, with dynamic SQL the actual SQL statements might have to be rewritten.

8.10 Write a transaction program in

a. Embedded SQL
b. JDBC
c. ODBC
d. SQLJ

that transfers the rows of a table between two DBMSs. Is your transaction globally atomic?

8.11 Write a Java program that executes in your local browser, uses JDBC to connect to your local DBMS, and makes a simple query against a table you have created.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

8.12 Suppose that, at compile time, the application programmer knows all of the details of the SQL statements to be executed, the DBMS to be used, and the database schema. Explain the advantages and disadvantages of using embedded SQL as compared with JDBC or ODBC as the basis for the implementation.

Solution:
Embedded SQL is more efficient than JDBC and ODBC because the syntax can be checked at compile time and parameter passing is more efficient. Also there is less communication involved.

8.13 Section 8.6 discusses KEYSET_DRIVEN cursors.

a. Explain the difference between STATIC and KEYSET_DRIVEN cursors.

Solution:
STATIC cursors effectively make a copy of the result set and scroll through that copy. Thus they cannot see any subsequent changes to the result set. KEYSET_DRIVEN cursors return pointers to the items in the result set and can see subsequent changes

to those tuples. Also KEYSET_DRIVEN cursers can be used to update data while STATIC cursors cannot.

b. Give an example of a schedule in which these cursors give different results even when the transaction is executing in isolation.

Solution:

OPEN a cursor, UPDATE not through the cursor, FETCH through the cursor

If the cursor is STATIC, the FETCH cannot see the effect of the UPDATE. If the cursor is KEYSET_DRIVEN. the FETCH could see the effect of the UPDATE.

c. Explain why updates and deletes can be made through KEYSET_DRIVEN cursors.

Solution:
The cursor returns a set of pointers to the rows in the result set. Hence updates and deletes can be made through these pointers.

8.14 Give an example of a transaction program that contains a cursor, such that the value returned by one of its FETCH statements depends on whether or not the cursor was defined to be INSENSITIVE. Assume that this transaction is the only one executing. We are not concerned about any effect that a concurrently executing transaction might have.

Solution:

```
EXEC SQL DECLARE GetEnrolled INSENSITIVE CURSOR FOR
     SELECT T.StudId, T.Grade
          FROM Transcript T
          WHERE T.CrsCode = :crs_code
               AND T.Semester = :semester;
EXEC SQL OPEN GetEnrolled;

INSERT INTO Transcript
     VALUES ('656565656', 'CS315', 'F1997', 'C');

EXEC SQL FETCH GetEnrolled INTO :stud_id, :grade;
EXEC SQL CLOSE GetEnrolled;
```

Since the cursor is INSENSITIVE, it does not see the effect of the INSERT. If it were not INSENSITIVE, it could see the effect.

8.15 Compare the advantages and disadvantages of the exception handling mechanisms in embedded SQL, SQL/PSM, JDBC, SQLJ, and ODBC.

PART THREE

Optimizing DBMS Performance

9

Physical Data Organization and Indexing

EXERCISES

9.1 State the storage capacity, sector size, page size, seek time, rotational latency, and transfer time of the disk

a. On your local PC
b. On the server provided by your university

Solution:
This is a hands-on problem. It's solution is not meant to be included in the manual.

9.2 Explain the difference between an equality search and a range search.

Solution:
Search with Equality Selection — fetch all records having attribute value(s) equal to a given value(s). For example, "Find the record of the student with sid 23." Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

Search with Range Selection – fetch all records having attribute values within a given range. For example, "Find all Students records with name alphabetically between 'Jones' and 'Smith'."

9.3 a. Give an upper bound on the number of pages that must be fetched to perform a binary search for a particular name in the phone book for your city or town.

Solution:
If the phone book has 1,000 pages, an upper bound on the number of pages that would have to be fetched is $log_2 1000 = 10$.

b. By how much is this number reduced if an index is prepared giving the name of the first entry on each page of the phone book? (Does that index fit on one page of the phone book?)

Solution:
Our local phone book has approximately 625 names on each page (in 5 columns), so if the phone book has 1,000 pages, the index would fit on 2 (phone book) pages. Hence, 2 page fetches (at most) would be needed to search the index and one additional fetch to get the desired page.

c. Conduct an experiment using your usual (informal) method to search for the name "John Lewis" in your local phone book, and compare the number of pages you look at with the number in Exercise 3(a).

Solution:
This is a hands-on problem. It's solution is not meant to be included in the manual.

9.4 Explain why a file can have only one clustered index.

Solution:
With a clustered index the index entries and the data records are sorted on the same search key. Hence the index entries of a second clustered index must be sorted in the same way as the index entries of the first clustered index: the two indices are the same.

9.5 Explain why a secondary, unclustered index must be dense.

Solution:
A dense index is an index whose entries are in 1-1 correspondence with the corresponding records in the data file. A secondary, unclustered index must be dense since data records are ordered differently from index entries and hence there is no way to find a record if no index entry refers to it.

9.6 Does the final structure of a B^+ tree depend on the order in which the items are added to it? Explain your answer and give an example.

Solution:
Yes, because when an insertion causes a split of a leaf page, the entry to be inserted in the parent node depends on the order in which items have been added to the tree.

9.7 Starting with an empty B^+ tree with up to two keys per node; show how the tree grows when the following keys are inserted one after another:

18, 10, 7, 14, 8, 9, 21

Solution:
See Figure 9.1.

9.8 Consider the partially specified B^+ tree in Figure 9.34.

a. Fill in the internal nodes without adding new keys.

Solution:
See Figure 9.2.

b. Add the key bbb. Show how the tree changes.

Solution:
See Figure 9.3.

c. Delete the key abc from the result of (b). Show how the tree changes.

Solution:
See Figure 9.4.

9.9 Consider the B^+ tree in Figure 9.35. Suppose that it was obtained by inserting a key into a leaf node of some other tree, *causing a node split.* What was the original tree and the inserted key? Is the solution unique? Explain your answer.

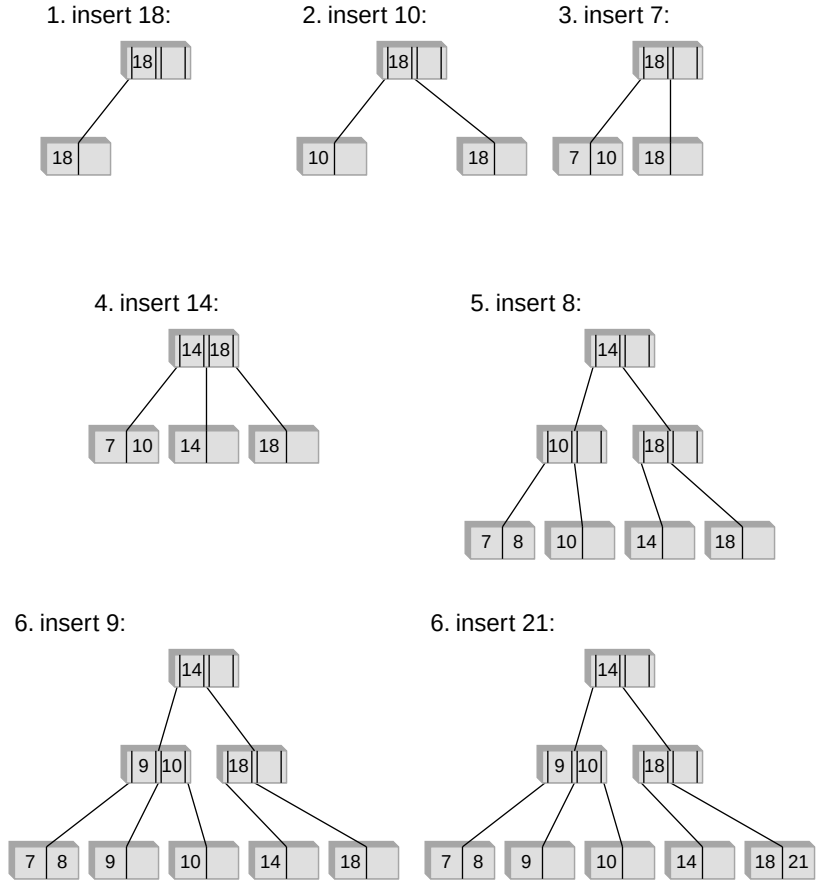


FIGURE **9.1**

Solution:
The solution is not unique. The inserted key could be 17, or 19. The possible original trees are depicted in Figures 9.5 and 9.6.

9.10 Describe a search algorithm for a B^+ tree in which the search key is not a candidate key. Assume that overflow pages are not used to handle duplicates.

Solution:
Descend through the tree using the target key value, k_0. At each level (including leaf) multiple entries in the page can contain the same key value — that is $k_i = k_{i+1} = \ldots = k_{i+r}$.

- case 1: If $k_{i+r} < k_0 < k_{i+r+1}$ then follow p_{i+r}.
- case 2: If $k_{i-1} \leq k_0 < k_i$ then follow p_{i-1}
- case 3: If $k_0 = k_i$ then follow p_{i-1}

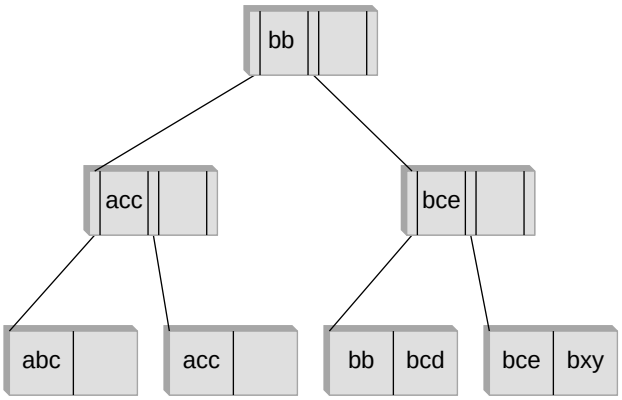


FIGURE 9.2

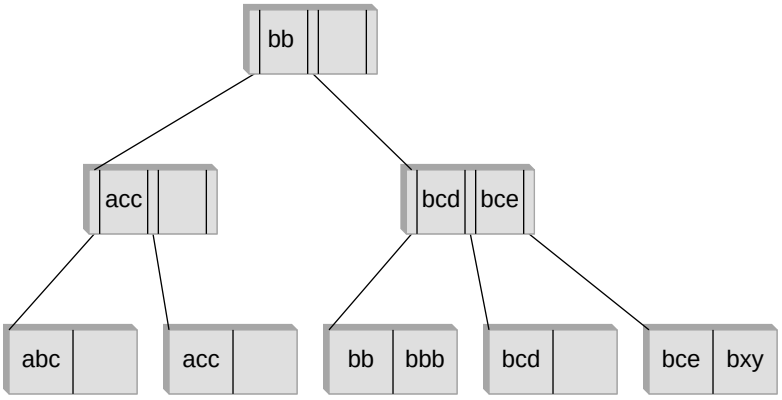


FIGURE 9.3

When the leaf level is reached, scan forward using sibling pointers until a key, k, is reached such that $k > k_0$.

9.11 Consider a hash function, h, that takes as an argument a value of a composite search key that is a sequence of r attributes, $a_1, a_2, \ldots, a_r$. If h has the form

$$h(a_1 \circ a_2 \circ \ldots \circ a_r) \;=\; h_1(a_1) \circ h_2(a_2) \circ \ldots \circ h_r(a_r)$$

where h_i is a hash of attribute a_i and $\circ$ is the concatenation operator, h is referred to as a **partitioned hash function**. Describe the advantages of such a function with respect to equality, partial-key, and range searches.

Solution:
No advantage with respect to equality searching. For partial key searches we can use the attributes that are provided to get part of the identification of the buckets in which target tuples can reside. All buckets satisfying this partial identification have to be searched since any might contain target tuples. This narrows the search since a

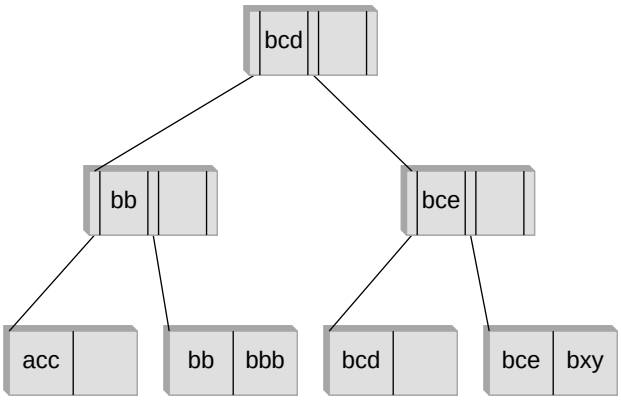


FIGURE 9.4

The inserted key is 17:

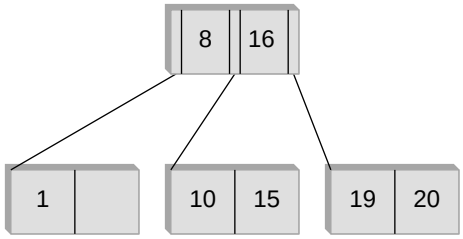


FIGURE 9.5

The inserted key is 19:

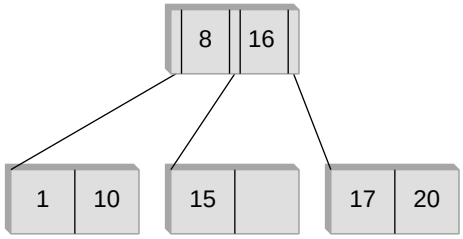


FIGURE 9.6

hash function that is not partitioned would require that all buckets be searched. The same benefit applies to range searches, although the range information for a particular attribute cannot be used.

9.12 Starting with the B^+ tree shown in Figure 9.22 show the successive B^+ trees that result from deleting `pete`, `phil`, `mike` and `nan`.

Solution:
Figure 9.7 shows the portion of the tree that has been changed at various stages in the

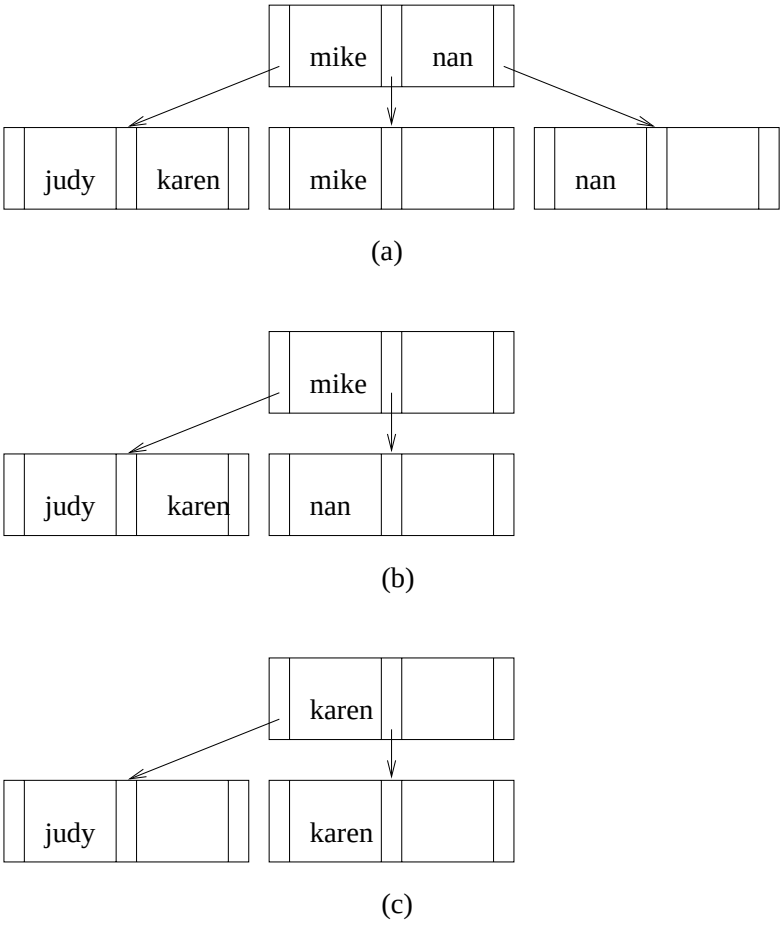


FIGURE 9.7 Various stages of the portion of the B^+ tree shown in Figure 9.22 that is changed when `pete`, `phil`, `mike`, and `nan` are deleted.

process: after `pete` and `phil` have been deleted (a), after `mike` has been deleted (b), and after `nan` has been deleted (c).

9.13 Give the pseudocode for the procedure `redistributeright` called by the procedure `delete` shown in Figure 9.24.

Solution:

```
proc redistributeright(parentptr, leftptr, rightptr)
//Let e1 be entry in *parentptr containing rightptr: e1 = < k1, rightptr >
//Let e2 be largest entry in *leftptr: e2 = < k2, ptr >
//Let P0 be the extra pointer in *rightptr
  add < k1, P0 > to *rightptr;
  delete e1 from *parentptr;
  add < k2, rightptr > to *parentptr;
  delete e2 from *leftptr;
  set extra pointer in *rightptr to ptr;
endproc
```

9.14 Express the algorithm for insertion and deletion of index entries in the extendable hashing schema using pseudocode.

Solution:

```
proc insert(ent)

// ent is the entry to be inserted;
// h is a hash function that that produces b bits;
// ch is the current hash level;
// D[i] is the i^th entry in the directory;
// bl[i] is the level of the i^th bucket

if bucket *D[h_ch(ent)] is not full then store ent in that bucket;
else   // *D[h_ch(ent)] is full, call that bucket B_i
  if bl[i] < ch then
    create a new bucket, B_{i+2^{ch-1}};
    D[i + 2^{ch-1}] = &(B_{i+2^{ch-1}});    // store pointer to new bucket in directory
    //rehash the elements of B_i to B_i and B_{i+2^{ch-1}}
    put each element, e, of B_i in *D[h_ch(e)];
    bl[i] and bl[i + 2^{ch-1}] are set to bl[i] + 1;
    insert(ent); return;
  else    //bl[i] = ch
    if ch = b then halt;
    else    // split directory
      ch := ch + 1;
      (∀i)(2^{ch-1} ≤ i < 2^{ch}) D[i] := D[i - 2^{ch-1}];
      (∀i)(2^{ch-1} ≤ i < 2^{ch}) bl[i] := bl[i - 2^{ch-1}];
insert(ent); return;
```

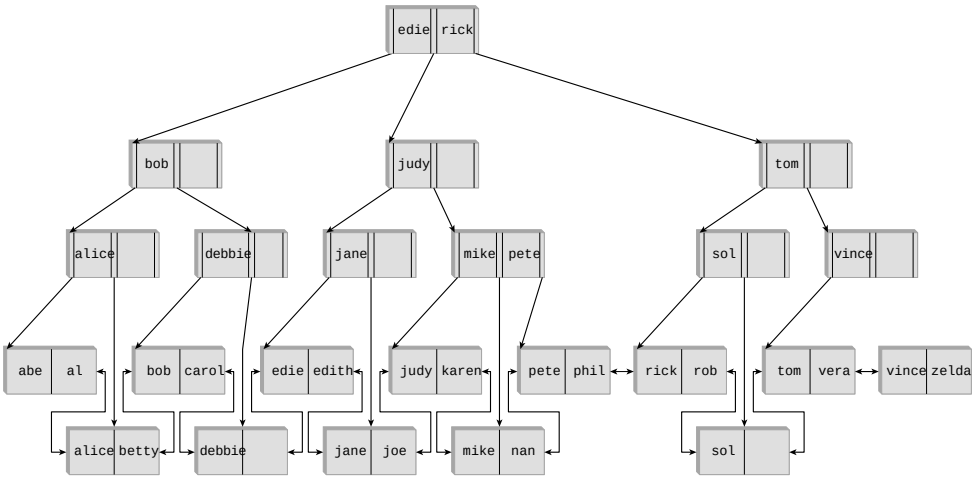


FIGURE 9.8

endproc

9.15 Give examples of select statements that are

a. Speeded up due to the addition of the B^+ tree index shown in Figure 9.22 on page 358.

Solution:
Find the record for `judy`.

b. Slowed down due to the addition of the B^+ tree index shown in that figure.

Solution:
Suppose each record contains an Id field and there already exists an index on Id. Then the insertion of a new record can be done efficiently with that index. The addition of the B^+ tree slows the insertion down since additional overhead is required in order to insert a new entry into the tree.

9.16 Draw the B^+ tree that results from inserting `alice`, `betty`, `carol`, `debbie`, `edith`, and `zelda` into the index of Figure 9.22 on page 358.

Solution:
See Figure 9.8.

9.17 A particular table in a relational database contains 100,000 rows, each of which requires 200 bytes of memory. A SELECT statement returns all rows in the table that satisfy an equality search on an attribute. Estimate the time in milliseconds to complete the query when each of the following indices on that attribute is used. Make realistic estimates for page size, disk access time, and so forth.

a. No index (heap file)

Solution:
Assume a page size of $4K$ bytes and a page access time of 20ms. The data file occupies 5000 pages and all pages will have to be scanned. Hence, the query will require 100 sec.

b. A static hash index (with no overflow pages)

Solution:
20ms.

c. A clustered, unintegrated B^+ tree index

Solution:
If we assume that each entry in the index occupies 100 bytes then an index page can hold 40 entries. Since the data file occupies 5000 pages, the leaf level of the tree must contain at least 5000/40, or 125 pages. Then the number of levels in the tree (assuming page 75% occupancy in the index) is $\lceil log_{30}125 \rceil + 1 = 3$. Assume that the index is clustered, not integrated with the data file and that all matching entries are in a single page, 4 I/O operations and 80ms are required to retrieve all matching records.

9.18 Estimate the time in milliseconds to insert a new row into the table of Exercise 9.17 when each of the following indices is used:

a. No index (file sorted on the search key)

Solution:
Use a binary search to locate the page in which the new row is to be inserted and assume that there is room in the target page. The data file occupies 5000 pages and the search will require retrieving $\lceil log_2 5000 \rceil = 13$. An additional I/O operation is required to write the page back to mass store, hence 14 I/O operations taking 280ms are required.

b. A static hash index (with no overflow pages)

Solution:
Assuming once again that the target bucket has room for the new row, 2 I/O operations and 40ms are required.

c. A clustered, unintegrated B^+ tree index (with no node splitting required)

Solution:
As in Exercise 9.17, 4 I/O operations are required to retrieve the page into which the new row must be inserted. Two additional I/O operations are required to update the leaf page of the index and the data page. Hence, the time to do the insertion is 120ms.

9.19 Estimate the time in milliseconds to update the search-key value of a row in the table of Exercise 9.17 when each of the following is used:

a. No index (file sorted on the search key)

Solution:
As in Exercise 9.18 it takes 280ms to locate the row to be updated and another 20ms to delete the row from the page. Another 280+20ms is required to locate and update the new page in which the row is to reside. Hence a total of 600ms is required.

b. A static hash index (where the updated row goes in a different bucket than the original row's page, but no overflow pages are required)

Solution:
40ms to update the old bucket and 40ms to update the new bucket.

c. A clustered, unintegrated B^+ tree index (where the updated row goes on a different page than the original row's pages, but no node splitting is required)

Solution:
Using the answer to Exercise 9.18 it takes 120ms to delete the row and update the index and another 120ms to insert the updated row and update the index. Hence, 240ms are required.

9.20 Estimate the amount of space required to store the B^+ tree of Exercise 9.17 and compare that with the space required to store the table.

Solution:
The minimum number of pages in the tree (assuming all pages are full) is 41 (for the first two levels) plus 125 leaf pages = 166 pages. The data file occupies 5000 pages.

9.21 Explain what index types are supported by your local DBMS. Give the commands used to create each type.

Solution:
Sybase supports B trees with the command:

```
create [unique] [clustered | nonclustered] index index_name
on table_name
     (column_name [asc | desc] [, column_name [asc | desc]]...)
[with [fillfactor = pct | max_rows_per_page=num_rows ] ]
```

The `unique` option indicates that the search key is a key of the table (and hence duplicate values of the search key are not allowed). The `asc | desc` option specifies whether index entries on the associated column are to be sorted in ascending or descending order. Either a fillfactor or (not both) the maximum number of rows per page can be specified. These constraints limit the number of rows on data pages or the number of index entries on the leaf pages of an unclustered index. The fillfactor specifies the initial percentage of the page that will be filled. However, the actual percentage will change as the data changes. A specification of the maximum number of rows per page, however, is maintained as the data changes.

9.22 Design the indices for the tables in the Student Registration System. Give the rationale for all design decisions (including those not to use indices in certain cases where they might be expected).

Solution:
This is a hands-on problem. It's solution is not meant to be included in the manual.

9.23 Explain the rationale for using fillfactors less than 1 in

a. Sorted files

Solution:
The page in which a row is placed depends on the value of the search key in the row. If that page is full, the file has to be reorganized. Using a fill factor of less than 1 increases the probability that there will be room in the page for the row.

b. ISAM indices

Solution:
By using a fill factor of less than 1, the probability that overflow pages will be needed is reduced.

c. B^+ tree indices

Solution:
By using a fill factor of less than 1, the probability that page splits will be needed is reduced.

d. Hash indices

Solution:
By using a fill factor of less than 1, the probability that a bucket will overflow is reduced.

9.24 Design an algorithm for maintaining join indices incrementally—that is, so that the addition of new tuples to the relations involved in the join do not require the entire join to be recomputed from scratch.

Solution:
Consider a join index for an equi-join of tables A and B. Implement the index as an integrated B^+ tree. The search key is the index into A of a particular row. Each leaf entry contains a pair $< p, q >$ where p is a search key value and q is the index of a row in B that satisfies the equi-join condition with respect to the row with index p in A. When a row with index a is added to A the index of each row, with index b, in B that joins with it is computed and the entry $< a, b >$ is added to the tree. Computing the equi-join involves scanning the leaf level once. Since all entries corresponding to a particular row of A are in sequence at that level, only a single scan of A is required. The number of accesses to B depends on the number of rows in the equi-join.

Adding a row to B also requires that we first find the rids of all the matching tuples in A and then insert the appropriate pairs of rids in the join index.

If a row is deleted from A, we can find all the index entries for that row in the B^+ tree and delete them. Deleting the rows of B requires more work because we cannot easily find the corresponding index entries in the join index, since the search key of the B^+ tree has the rid of A, not of B, as the first component. Thus, we might have to scan all the leaves of that tree in order to find the index entries to delete.

*** 9.25** Propose an improvement that permits bitmap indices to maintain only $n - 1$ bit vectors in order to represent attributes that can take n different values. Discuss the change that is needed to perform selection, especially in case of a multiattribute selection that requires a Boolean operation on multiple bit vectors.

Solution:
Consider an attribute, A, that can take n values. We can use $n - 1$ vectors for the first $n - 1$ values of A. For the n-th value, we can always reconstruct the corresponding

bit vector using the following Boolean expression: $\neg vector_1 \wedge \ldots \wedge \neg vector_{n-1}$. This vector has 1 exactly for the rows where A has the value that is different from any of the first $n-1$ values.

10

The Basics of Query Processing

EXERCISES

10.1 Consider the use of unclustered B^+ trees for external sorting. Let R denote the number of data records per disk block, and let F be the number of blocks in the data file. Estimate the cost of such a sorting procedure as a function of R and F. Compare this cost to merge-based external sorting. Consider the cases of $R = 1$, 10, and 100.

Solution:
The cost of using unclustered B^+ trees is $2R \times F$ because we have to scan the leaves of the B^+ tree and for each entry there to access the corresponding data file block twice (to read and then to write). When R increases, the cost increases.

Suppose we have M pages of main memory in which to do the sorting and that we use k-way merge. The cost of merge-based external sorting is $2F \times log_k\lceil F/M\rceil$. This cost does not depend on R.

When $R = 1$, B^+ tree sorting is slightly better. For $R = 10$, merge-sort is likely to be better even for large files. For instance, if $M = 10,000$ and $R = 100,000,000$ (i.e., 400G), then we can do 999-way merge and

$$log_{999}(100000000/10000) < 2 < R = 10$$

10.2 Estimate the cost of the sort-based projection assuming that, during the initial scan (where tuple components are deleted), the size of the original relation shrinks by the factor $\alpha < 1$.

Solution:
The technique first scans the original relation and removes the tuple components that are to be projected out. The cost of this operation is of the order of $(1 + \alpha)F$, where F is the number of blocks in the relation. (F to read and αF to write.)

Then the result is sorted at the cost of

$$2\alpha F \times log_{(M-1)}\lceil \alpha F/M\rceil$$

where M is the number of main memory pages available for sorting and F is the number of blocks in the file.

Finally, we scan the result again, at the cost of $2\alpha F$, and since identical tuples are right next to each other (because the relation is sorted), we can easily delete the duplicates during this last scan.

10.3 Consider hash-based evaluation of the projection operator. Assume that all buckets are about the same size but do not fit in main memory. Let N be the size of the hash table measured in memory pages, F be the size of the original relation measured in pages, and $\alpha < 1$ be the reduction factor due to projection. Estimate the number of page transfers to and from the disk needed to compute the projection.

Solution:
In the first phase, the original relation is scanned. During the scan, we chop off the tuple components that are to be projected out, and the rest of the tuple is hashed on the remaining attributes. Whenever a page of the hash table becomes full, it is flushed to the corresponding bucket on disk. The cost of this operation is of the order of $(1 + \alpha)F$, where F is the number of blocks in the relation.

Clearly, duplicate tuples are always hashed into the same bucket, so we can eliminate duplicates in each bucket separately. This elimination is done in the second phase of the algorithm. Because the individual buckets do not fit in the main memory, they must be sorted using external sorting. There are N buckets and for each bucket, the cost is: $2 \times log_{(M-1)}\lceil B/M \rceil$, where B is the average size of each bucket measured in memory pages, and M is the number of main memory pages available to sort the buckets. Since all buckets are approximately the same in size, $B = F/N$. The total cost of this step is: $2\alpha N \times log_{(M-1)}\lceil F/(MN) \rceil)$.

10.4 Give an example of an instance of the TRANSCRIPT relation (Figure 3.5) and a hash function on the attribute sequence $\langle$`StudId`, `Grade`$\rangle$ that sends two identical tuples in $\pi_{\texttt{StudId,Semester}}$(TRANSCRIPT) into *different* hash buckets. (This shows that such a hash-based access path cannot be used to compute the projection.)

Solution:
Consider the following two tuples: $\langle$`111111111`, `EE101`, `F1997`, `A`$\rangle$ and $\langle$111111111,MAT123,F1997,B$\rangle$. Since these tuples have identical projections on the attributes `StudId,Semester`, they become one in the projected relation $\pi_{\texttt{StudId,Semester}}$(TRANSCRIPT).

Consider now the following hash function on `StudId,Grade`:

$$f(t) \;=\; (t.\texttt{StudId} + (t.\texttt{Grade} - \texttt{'A'}))\ \texttt{mod}\ 1000$$

The first tuple will then be sent to the bucket 111111111 `mod` 1000 = 111 and the second to 111111112 `mod` 1000 = 112.

10.5 Clearly, the theoretical minimum for the selectivity of an access path is the number of pages that hold the output of the relational operator involved. What is the best theoretical upper bound on the selectivity of an access path when selection or projection operators are involved?

Solution:
The theoretical upper bound on the selectivity of an access path when selection or projection operators are involved is the number of pages in the original relation.

10.6 Based on the discussion in Section 10.4.2, give a precise definition of when an access path covers the use of projection, union, and set-difference operators.

Solution:

1. Consider a relational expression of the form $\pi_{\texttt{attr}_1,\ldots,\texttt{attr}_n}(R)$.
   The use of projection operator is covered by an access path if and only if one of the following conditions holds:
   - The access path is file scan.
   - The access path is a hash index whose search key is a *subset* of the attributes $\texttt{attr}_1, \ldots, \texttt{attr}_n$.
   - The access path is a B^+ tree index with the search key $\texttt{sk}_1, \ldots, \texttt{sk}_m$ *and* some prefix $\texttt{sk}_1, \ldots, \texttt{sk}_i$ of that search key is a *subset* of $\texttt{attr}_1, \ldots, \texttt{attr}_n$.
   - The access path is binary search and the relation R is sorted on the attributes $\texttt{sk}_1, \ldots, \texttt{sk}_m$. The definition of "covering" in this case is the same as for B^+ tree indices.
2. Consider relational expressions of the form $R \cup S$ or $R - S$. The use of union or set-difference is covered by an access path if and only if one of the following conditions holds:
   - The access path is file scan.
   - The access path is a hash index on R and S.
   - The access path is a B^+ tree index on R and S.
   - The access path is binary search and the relations R and S are sorted.

10.7 Consider the expression

$$\sigma_{\texttt{StudId=666666666}\wedge\texttt{Semester='F1995'}\wedge\texttt{Grade='A'}}(\textsc{Transcript})$$

Suppose the following access paths are available:

- An unclustered hash index on `StudId`
- An unclustered hash index on `Semester`
- An unclustered hash index on `Grade`

Which of these access paths has the best selectivity, and which has the worst? Compare the selectivity of the worst access path (among the above three) to the selectivity of the file scan.

Solution:
With the unclustered hash index on `StudId`, we will find exactly the bucket that contains all the transcript records for student with the Id 666666666. Since the index is unclustered, this access method will fetch (in the worst case) as many pages as the number of transcript records for that student. In our sample relation in Figure 3.5, this would be 3 pages. In a typical university, an undergraduate student would have to earn 120–150 credits. With 3 credits per course it would make 40-50 transcript records and, thus, the selectivity would be this many pages of data.

With the unclustered hash index on `Semester`, we jump to the bucket for the transcript records in the `F1995` semester and then we need to fetch all these records from the disk to check the other conditions. In a university with enrollment 20,000, selectivity of this access path can be as high as that. In our sample database, however, there are only two transcript records for Fall 1995.

With the unclustered hash index on `Grade`, we get into the bucket of the transcript records where the student received the grade `A`. If only 10% of the students get an `A`, the bucket would hold 2,000 records per semester. In 20 years (2 semesters a year), theuniversity might accumulate as many as 80,000 transcript records in that bucket. In our sample database, we have 5 transcript records where the student got an `A`.

Thus, in a typical university, the third access path has the worst selectivity and the first has the best. In the sample database of Figure 3.5, the second method has the best selectivity and the third the worst.

10.8 Compute the cost of $\mathbf{r} \bowtie_{\mathtt{A=B}} \mathbf{s}$ using the following methods:

- Nested loops
- Block-nested loops
- Index-nested loops with a hash index on B in $\mathbf{s}$ (consider both clustered and unclustered index)

where $\mathbf{r}$ occupies 2000 pages, 20 tuples per page; $\mathbf{s}$ occupies 5000 pages, 5 tuples per page; and the amount of main memory available for a block-nested loops join is 402 pages. Assume that at most 5 tuples of $\mathbf{s}$ match each tuple in $\mathbf{r}$.

Solution:

1. Nested loops: scan $\mathbf{r}$ and for each of its 40,000 tuples scan $\mathbf{s}$ once. The result is

$$2,000 + 40,000 \times 5,000 = 200,002,000 \text{ pages}$$

2. Block-nested loops: Scan $\mathbf{s}$ once per each 400-page block of $\mathbf{r}$, i.e., 5 times. The result therefore is:

$$2,000 + 5,000\lceil\frac{2,000}{402-2}\rceil = 27,000 \text{ pages}$$

3. Index-nested loops: The result depends on whether the index on `B` in $\mathbf{s}$ is clustered or not. For the clustered case, all tuples of $\mathbf{s}$ that match a tuple of $\mathbf{r}$ are in the same disk block and require 1 page transfer (since we assumed that at most 5 tuples of $\mathbf{s}$ match, they all fit in one disk block). We also need to search the index once per each tuple of $\mathbf{r}$. Suppose the later takes 1 disk access. Thus, the total is

$$2,000 + (1 + 1 * 1.2) \times 40,000 = 90,000 \text{ pages}$$

In case of an unclustered index, the matching tuples of $\mathbf{s}$ can be in different blocks. As before, assume that $\mathbf{s}$ has at most 5 matching tuples per tuple in $\mathbf{r}$. Thus, the cost would be

$$2,000 + (1 + 5 * 1.2) \times 40,000 = 282,000 \text{ pages}$$

10.9 In sort-based union and difference algorithms, the final scan—where the actual union or difference is computed—can be performed at no cost in I/O because this step can be combined with the last merge step during sorting of the relations involved. Work out the details of this algorithm.

⋆ **10.10** In the sort-merge join of $\mathbf{r} \bowtie \mathbf{s}$, the scan in the algorithm of Figure 10.7 can be performed at no cost in I/O because it can be combined with the final merging step of sorting $\mathbf{r}$ and $\mathbf{s}$. Work out the details of such an algorithm.

Solution:
In the last merging stage of the sorting algorithm we will have n runs, $\mathbf{r}_1, \ldots, \mathbf{r}_n$, of relation $\mathbf{r}$ sorted on the attribute `A` and m runs, $\mathbf{s}_1, \ldots, \mathbf{s}_m$, of relation $\mathbf{s}$ sorted on the attribute `B`. The merging stage of the join is combined with that last stage of sorting as follows. We assume that there is a function `getSmallest(X,[`$\mathbf{p}_1, \ldots, \mathbf{p}_k$`])`

which takes an attribute, `X`, and a list of union-compatible relations sorted on that attribute, and returns a tuple with the smallest value of `X` that exists in any of the $\mathbf{p}_1, \ldots, \mathbf{p}_k$. The tuple is then deleted from the corresponding relation so that the next call to `getSmallest()` would return the tuple with the next smallest value, etc. The algorithm now becomes just a slight modification of the merge step for sort-merge join presented in Figure 10.7:

Input: n runs, $\mathbf{r}_1, \ldots, \mathbf{r}_n$, of relation $\mathbf{r}$ sorted on the attribute `A`
m runs, $\mathbf{s}_1, \ldots, \mathbf{s}_m$, of relation $\mathbf{s}$ sorted on the attribute `B`
Output: $\mathbf{r} \bowtie_{\mathtt{A}=\mathtt{B}} \mathbf{s}$

```
Result := { }                          // initialize Result
t_r := getSmallest(A,[r_1, ..., r_n])  // get first tuple
t_s := getSmallest(B,[s_1, ..., s_m])

while !eof(r) && !eof(s) do {
  while !eof(r) && t_r.A < t_s.B do
    t_r := getSmallest(A,[r_1, ..., r_n])   // get next tuple
  while !eof(s) && t_r.A > t_s.B do
    t_s := getSmallest(B,[s_1, ..., s_m])
  if t_r.A = t_s.B = c then {           // for some const c
    Result := (σ_{A=c}(r_1 ∪ ··· ∪ r_n) × σ_{B=c}(s_1 ∪ ··· ∪ s_m)) ∪ Result;
    t_r := the next tuple t ∈ r where t[A] > c
  }
}
return Result;
```

10.11 Estimate the number of page transfers needed to compute $\mathbf{r} \bowtie_{\mathtt{A}=\mathtt{B}} \mathbf{s}$ using a sort-merge join, assuming the following:

- The size of $\mathbf{r}$ is 1000 pages, 10 tuples per page; the size of $\mathbf{s}$ is 500 pages, 20 tuples per page.
- The size of the main memory buffer for this join computation is 10 pages.
- The Cartesian product of matching tuples in $\mathbf{r}$ and $\mathbf{s}$ (see Figure 10.7) is computed using a block-nested loops join.
- `r.A` has 100 distinct values and `s.B` has 50 distinct values. These values are spread around the files more or less evenly, so the size of $\sigma_{\mathtt{A}=c}(\mathbf{r})$, where $c \in$ `r.A`, does not vary much with c.

Solution:

1. External sort of relation $\mathbf{r}$

$$2 \times 1,000(log_9\lceil 1,000\rceil) = 8,000 \text{ pages}$$

2. External sort of relation $\mathbf{s}$

$$2 \times 500(log_9\lceil 500\rceil) = 3,000 \text{ pages}$$

3. The merge step
First, clearly, we need to scan both **r** and **s**, which costs us 1,500 page transfers.

For each value of the attribute `A`, the number of pages of **r** that contain tuples that match that value is 1,000/100=10. Similarly, for each value of `B`, the number of pages of **s** that contain tuples that match that value is 500/50=10. Thus, each time we have a match, we compute a Cartesian product of two 10-page relations. Using block-nested loops with 10 buffer pages requires that we scan the **r**-part once and the **s**-part twice for a total of 30 pages. However, one scan of **r** and **s** needed to compute the produce is accounted for in the aforesaid initial scan (the one that takes 1,500 page transfers). So, the extra cost is only 10 pages needed to scan the **s**-part the second time.

The output has $10 \times 10 = 100$ pages, so for each match computing the Cartesian product takes 110 page transfers. The question is how many matches we can have. Since `A` has 100 values and `B` has 50, we can have at most 50 matches, so computing and outputting the joined tuples will thus take *at most* $50 \times 110 = 5,500$ page transfers. Together we get:

$$1,000 + 500 + 5,500 = 7,000 \text{ pages}$$

4. Thus, the total cost is

$$6,000 + 2,000 + 7,000 = 15,000 \text{ pages}$$

10.12 The methods for computing joins discussed in Section 10.5 all deal with equi-joins. Discuss their applicability to the problem of computing inequality joins, such as $\mathbf{r} \bowtie_{\mathtt{A}<\mathtt{B}} \mathbf{s}$.

Solution:
Suppose we use sort-merge join to compute inequality joins, such as $\mathbf{r} \bowtie_{\mathtt{A}<\mathtt{B}} \mathbf{s}$.

Input: `relation r sorted on attribute A;`
`relation s sorted on attribute B`
Output: $\mathbf{r} \bowtie_{\mathtt{A}<\mathtt{B}} \mathbf{s}$

```
Result := {}                          // initialize Result
t_r := getFirst(r)                    // get first tuple
t_s := getFirst(s)

while !eof(r) && !eof(s) do {
  while !eof(s) && t_r.A >= t_s.B do
    t_s := getNext(s);
  if t_r.A = c < t_s.B then {
    Result := (σ_{A=c}(r) × σ_{B>c}(s)) ∪ Result;
    t_r := the next tuple t ∈ r where t[A]>c
  }
}
return Result;
```

As to other methods, hash join is not useful for equality conditions. Nested loops and block-nested loops work with little change. Index-nested loops join also works is minor changes, if the index on B in **s** is B^+ tree. If the B^+ tree is unclustered, then index nested loops join is going to be expensive, but the algorithm still works.

If the index is a hash index, then Index-nested loops join is not useful for computing inequality joins.

10.13 Consider a relation schema, $\mathbf{R}(\mathtt{A}, \mathtt{B})$, with the following characteristics:

- Total number of tuples: 1,000,000
- 10 tuples per page
- Attribute A is a candidate key; range is 1 to 1,000,000
- Clustered B^+ tree index of depth 4 on A
- Attribute B has 100,000 distinct values
- Hash index on B

Estimate the number of page transfers needed to evaluate each of the following queries for each of the proposed methods:

- $\sigma_{\mathtt{A}<3000}$: sequential scan; index on A
- $\sigma_{\mathtt{A}>3000 \wedge \mathtt{A}<3200 \wedge \mathtt{B}=5}$: index on A; index on B
- $\sigma_{\mathtt{A}\neq 22 \wedge \mathtt{B}\neq 66}$: sequential scan; index on A; index on B

Solution:
The relation has 100,000 pages.

1. $\sigma_{\mathtt{A}<3000}$

   – Sequential scan
   Since the index is clustered, the relation is sorted, so to scan the tuples with $\mathtt{A} < 3000$ we need to fetch

   $$100,000 * 3,000/1,000,000 = 300 \text{ pages}$$

   – Index on A
   Use the index to find the last page page where $\mathtt{A} < 3000$ and then scan the relevant pages. The cost is 4 to search the index plus the cost of the scan, which is the same as above:

   $$4 + 300 = 304 \text{ pages}$$

2. $\sigma_{\mathtt{A}>3000 \wedge \mathtt{A}<3200 \wedge \mathtt{B}=5}$

   – Index on A
   Use the index to find the first page where $\mathtt{A} > 3000$ then scan the pages until you find the first page where $\mathtt{A} \geq 3200$:

   $$4 + 100,000 * 200/1,000,000 = 24 \text{ pages}$$

   – Index on B
   On the average there are 1,000,000/100,000 = 10 tuples per any given value of B. Use the hash index on B to find the tuples where B=5. Then we need one page transfer for each tuple in the buffer. In total about $1.2 + 10 = 11.2$ page transfers.

3. $\sigma_{\texttt{A}\neq 22 \wedge \texttt{B}\neq 66}$

   - Sequential scan
     Need to scan the entire relation: 100,000 page transfers.
   - Index on A
     Not helpful. For every value in the range of 1 to 1,000,000, except 22, use the index to get the page that holds that tuple: 4(depth of B^+ tree) $\times$ $999,999$(tuples) $=$ $3,999,996$ page transfers.
   - Index on B
     This index is also not of much use. For each value in the range of 1 to 100,000, except 66, find the bucket that contains pointers to the actual tuples: $1.2 \times 99,999$. For each such tuple (there are about 999,990 of them) fetch the corresponding page. The total cost is thus

$$1.2 \times 99,999 + 999,990 = 1119989 \text{ pages}$$

10.14 Design an algorithm for incremental maintenance of a join index for a multiway star join.

Solution:
With a star join of the form $\mathbf{r} \bowtie_{cond_1} \mathbf{r}_1 \bowtie_{cond_2} \mathbf{r}_2 \ldots \bowtie_{cond_n} \mathbf{r}_n$ we maintain n binary join indices, for $\mathbf{r} \bowtie_{cond_1} \mathbf{r}_1$, $\mathbf{r} \bowtie_{cond_2} \mathbf{r}_2$, etc. The maintenance algorithm for these was described in Exercise 9.24.

10.15 Design a join algorithm that uses a join index. Define the notion of a *clustered* join index (there are three possibilities in the case of a binary join!) and consider the effect of clustering on the join algorithm.

Solution:
The actual join algorithm was described in Section 9.7.2. Here we only need to consider clustering and its effect on the algorithm.

We can possibly say that a join index, $\mathcal{J}$, for $\mathbf{r} \bowtie \mathbf{s}$ is clustered in either of the three cases:

1. When $\mathcal{J}$ is sorted on the **r**-part, i.e., the **r**-side rids in $\mathcal{J}$ are organized in the same order as their corresponding tuples in **r**
2. When $\mathcal{J}$ is sorted on the **s**-part
3. When $\mathcal{J}$ is sorted simultaneously on both parts, i.e., the **r**-side rids in $\mathcal{J}$ are organized in the order of their corresponding tuples in **r** and the **s**-side rids are organized in the order of the corresponding tuples in **s**.

Note that we can always achieve options (1) and (2) by simply sorting $\mathcal{J}$ on its **r**-side or **s**-side. In fact, it would be silly not to do so. For this reason, only the third possibility can be reasonably considered to deserve the name "clustered" join index.

In the first two cases, the join algorithm was described in Section 9.7.2. In the third case, the algorithm is reminiscent of the merge stage of the sort-merge join. We simply scan the join index and retrieve the pages of **r** and **s** that correspond to the matching tuples. Once the pages are in, we output the requisite joined tuples. Since the tuples in the index are organized in the same order as the tuples of *both* **r** and **s**, the join is computed in less than one scan of these relations. It takes less than one scan because only the pages that contain the tuples to be joined need to be retrieved (we know which pages these are because the join index gives us the exact rids).

11

An Overview of Query Optimization

EXERCISES

11.1 Is there a *commutativity* transformation for the projection operator? Explain.

Solution:
No. Commutativity of projection means

$$\pi_{attr_1}(\pi_{attr_2}(R)) \;=\; \pi_{attr_2}(\pi_{attr_1}(R))$$

The left side requires $attr_1 \subseteq attr_2$ and the right side requires $attr_2 \subseteq attr_1$. This is impossible unless $attr_1 \;=\; attr_2$.

11.2 Write down the sequence of steps needed to transform $\pi_A((\mathbf{R} \bowtie_{B=C} \mathbf{S}) \bowtie_{D=E} \mathbf{T})$ into $\pi_A((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S})) \bowtie_{C=B} \mathbf{R})$. List the attributes that each of the schemas **R**, **S**, and **T** *must* have and the attributes that each (or some) of these schemas must *not* have in order for the above transformation to be correct.

Solution:

- **R** must have: B (because of the join)
- **S** must have: ACD (because of π_{ACD})
- **T** must have: E (because of π_E)
- These schemas should not have identically named attributes, because otherwise it will not be clear which of the two identically named attributes will be renamed in the joins. In particular, **T** should not have A and C, because $\pi_{ACD}(\mathbf{S})$ clearly suggests that it is expected that **S** will have the attribute A that will survive for the outermost projection π_A to make sense, and the attribute C, which should survive in order for the join with **R** to make sense.

1. Associativity of the join: $\pi_A(\mathbf{R} \bowtie_{B=C} (\mathbf{S} \bowtie_{D=E} \mathbf{T}))$
2. Commutativity of the join: $\pi_A((\mathbf{S} \bowtie_{D=E} \mathbf{T}) \bowtie_{C=B} \mathbf{R})$
3. Commutativity of the join: $\pi_A((\mathbf{T} \bowtie_{E=D} \mathbf{S}) \bowtie_{C=B} \mathbf{R})$
4. Pushing projection π_A to the first operand of the join: $\pi_A(\pi_{AC}((\mathbf{T} \bowtie_{E=D} \mathbf{S})) \bowtie_{C=B} \mathbf{R})$
5. Pushing projection to the first operand in the innermost join: $\pi_A(\pi_{AC}((\pi_E(\mathbf{T}) \bowtie_{E=D} \mathbf{S})) \bowtie_{C=B} \mathbf{R})$. This is possible if AC are the attributes of **S** and not of **T**.

6. Pushing projection to the second operand in the innermost join: $\pi_A(\pi_{AC}((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S}))) \bowtie_{C=B} \mathbf{R})$.
7. Reverse of pushing a projection: $\pi_A((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S})) \bowtie_{C=B} \mathbf{R})$

11.3 Under what conditions can the expression $\pi_A((\mathbf{R} \bowtie_{cond_1} \mathbf{S}) \bowtie_{cond_2} \mathbf{T})$ be transformed into $\pi_A(\pi_B(\mathbf{R} \bowtie_{cond_1} \pi_C(\mathbf{S})) \bowtie_{cond_2} \pi_D(\mathbf{T}))$ using the heuristic rules given in Section 11.2?

Solution:

- $cond_1$ must involve only the attribute C from the **S**-side
- $cond_2$ must involve only the attribute B from the left side of the join and only the attribute D from the right.

11.4 Consider the join PROFESSOR $\bowtie_{\texttt{Id=ProfId}}$ TEACHING used in the running example of Section 11.3. Let us change the statistics slightly and assume that the number of distinct values for TEACHING.`ProfId` is 10,000 (which translates into lower weight for this attribute).

a. What is the cardinality of the PROFESSOR relation?

Solution:
We cannot tell exactly, because some professors might not be teaching, but since `ProfId` in TEACHING must be a foreign key into PROFESSOR, the latter must have at least 10,000 tuples. We can assume that this is the correct size of this relation, because it is unlikely that many professors did not teach ever (which is the only way how there might not be a corresponding tuple in TEACHING).

b. Let there be an unclustered hash index on `ProfId` and assume that, as before, 5 PROFESSOR tuples fit in one page, 10 TEACHING tuples fit in one page, and the cardinality of TEACHING is 10,000. Estimate the cost of computing the above join using index-nested loops and block-nested loops with a 51-page buffer.

Solution:
Since in the previous problem we concluded that PROFESSOR has 10,000 tuples and there are 5 tuple/page, this relation will have 2,000 pages. For TEACHING, the statistics do not change: 10,000 tuples, 1,000 pages.

For index-nested loops join, we scan PROFESSOR and use the index on `ProfId` to fetch the matching tuples in TEACHING. Since there are 10,000 tuples in TEACHING and 10,000 values of `ProfId`, there is 1 matching tuple per value. So, the unclustered nature of the index on `ProfId` does not matter. There is going to be 1.2*10,000 page transfers in order to find all matching tuples in TEACHING plus the cost of the scan of PROFESSOR is 2,000. The total cost is 14,000 page transfers.

For the block-nested loops join, we scan the smaller relation, TEACHING, in the outer loop. Since we have 51 buffer pages, we use 49 for TEACHING, one for PROFESSOR, and one for the output buffer. Thus, we will scan PROFESSOR once for every 49-page block of TEACHING: 1000/49 = 21 scans. So, the cost is 21*2,000 + 1,000 = 43,000 page transfers.

11.5 Consider the following query:

```
SELECT DISTINCT E.Ename
```

```
FROM    EMPLOYEE E
WHERE   E.Title = 'Programmer' AND E.Dept = 'Production'
```

Assume that

- 10% of employees are programmers
- 5% of employees are programmers who work for the production department
- There are 10 departments
- The EMPLOYEE relation has 1000 pages with 10 tuples per page
- There is a 51-page buffer that can be used to process the query

Find the best query execution plan for each of the following cases:

a. The only index is on `Title`, and it is a clustered 2-level B^+ tree.

Solution:
Because the only index is a clustered index on `Title`, we can use it to find the 1000*0.1=100 pages that contain employees who are programmers. Let us assume that the index is integrated. Then we spend one I/O to get the top page of the B^+ tree and then it takes 1 disk access per page to get the right pages. We select on `E.Dept = Production` on the fly, which cuts the size down to 50 pages (since 5% of the employees are production programmers). This can fit in main memory. We can now project on `Ename` and sort in main memory to eliminate duplicates. Thus, the cost is 1 + 100 = 101 page transfers.

b. The only index is on the attribute sequence `Dept`, `Title`, `Ename`; it is clustered and has two levels.

Solution:
Because we have a clustered index on `Dept`, `Title`, `Ename`, we can use an index-only strategy. We get the top-level page of the index and from there access the index pages that contain the relevant tuples. There are 50 data pages that contain the relevant tuples, since 5% of the employees are production programmers. So, the cost of the access is at most 1 + 50 = 51 page I/Os. If the index is integrated, then this is the number of I/Os that is going to happen. However, if the index is *not* integrated, then the the index pages that contain the relevant tuples are likely to be smaller, and there will be less I/O. Note that since `Ename` is part of the key, all the production programmers will be sorted on `Ename`, and we will not need to do it even in main memory.

c. The only index is on `Dept`, `Ename`, `Title`; it is a clustered 3-level B^+ tree.

Solution:
We cannot use the index to search on `Title`, but we can use it to search on `Dept`. This will yield 1000/10=100 pages. We can also use an index-only strategy here. Search the index to find all the tuples for the production department and do projection and selection on the fly. No sorting is needed, because within each department the tuples are already sorted on `Ename`. The cost is 2 (for the upper two levels of the index) + the number of index pages that refer to the tuples in the production department. There are 100 pages of the corresponding data tuples, but the index is expected to be much smaller.

d. There is an unclustered hash index on `Dept` and a 2-level clustered tree index on `Ename`.

Solution:
Searching the hash index on `Dept` will give us the bucket of pointers to the production employees (about 10% of the total number of tuples, 10,000). But each such pointer will result in a page fetch, since the index in unclustered. So, the cost is 1.2 I/Os to find the bucket plus 1,000 to find the relevant tuples.

The above is actually slightly worse than the direct scan, which takes only 1,000 pages. The selection and projection is done on the fly and the result (50 pages) is placed in the buffer, where it is projected and then sorted to eliminate duplicates.

Using a clustered index on `Ename` would require us to take a scan anyway. But since the tuples in the result are already sorted on `Ename`, we do not need to do any in-memory sorting, so this method is slightly preferable to the other two.

11.6 Consider the following schema, where the keys are underlined:

```
EMPLOYEE(SSN, Name,Dept)
PROJECT(SSN,PID,Name,Budget)
```

The `SSN` attribute in PROJECT is the Id of the employee working on the project, and `PID` is the Id of the project. There can be several employees per project, but the functional dependency `PID` → `Name,Budget` holds (so the relation is not normalized). Consider the query

```
SELECT   P.Budget, P.Name, E.Name
FROM     EMPLOYEE E, PROJECT P
WHERE    E.SSN = P.SSN AND
         P.Budget > 99 AND
         E.Name = 'John'
ORDER BY P.Budget
```

Assume the following statistical information:

- 10,000 tuples in EMPLOYEE relation
- 20,000 tuples in PROJECT relation
- 40 tuples per page in each relation
- 10-page buffer
- 1000 different values for `E.Name`
- The domain of `Budget` consists of integers in the range of 1 to 100
- Indices
  - EMPLOYEE relation
    - On `Name`: Unclustered, hash
    - On `SSN`: Clustered, 3-level B^+ tree
  - PROJECT relation
    - On `SSN`: Unclustered, hash
    - On `Budget`: Clustered, 2-level B^+ tree

a. Draw the *fully pushed* query tree.

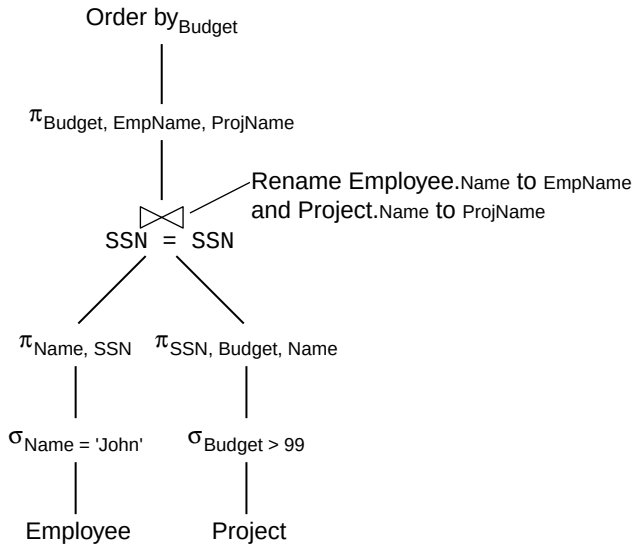


FIGURE 11.1

Solution:
See Figure 11.1.

b. Find the "best" execution plan and the second-best plan. What is the cost of each? Explain how you arrived at your costs.

Solution:
There are three reasonable plans:

– Select EMPLOYEE on `Name`. Since there are about 10000/1000=10 tuples in the result, it would take 1.2*10=12 page transfers. The result contains 10 tuples, 1 page.

  Using index-nested loops, join the 10 EMPLOYEE tuples with PROJECT. There are two projects per employee on the average, so it would require to bring in 1.2 * 10 * 2 = 24 pages, since we will be using an unclustered hash index on `SSN` in PROJECT. The result will contain 20 tuples (twice as wide as the tuples in the original relations), 1 page. We do not need to write anything out on disk, since after extracting the joined tuples we will be discarding the pages of PROJECT that we bring in for the join.

  Select the result on `Budget` on the fly, yielding 0.01 * 20 = 1 tuple (at no cost), then project. The total cost is 12 + 24 = 36 page transfers.

– Select PROJECT on `Budget`. Because we can use a clustered 2-level index, it will take 2 I/Os to find the address of the first relevant page. PROJECT has 20000/40 = 500 pages and we select 0.01 of them, i.e., 5 pages, 200 tuples, at the cost of 2 (index search) + 5 (reading the relevant pages) = 7 page transfers.

  Use index-nested loops to join the result with EMPLOYEE. The cost: at least 200 (project tuples) * 3 (to fetch an EMPLOYEE page) = 600 I/Os. The

resulting number of tuples is about 200, twice the original size. This is about 10 pages.

Apply selection and projection on the fly, yielding one page (at no cost). Total: over 607 I/Os.

– Select EMPLOYEE on `Name`, and PROJECT on `Budget`. From before, we know that it takes 12 + 7 I/Os and yields 10 and 200 tuples (1 + 5 pages), respectively.

Use sort-merge-join, then project. We can sort everything in memory (we have 10 pages in the buffer) and then merge, select and project in one pass (also in memory). Thus, the cost is 19 I/Os.

The best plan is therefore plan #3. The second best is plan #1.

11.7 Consider the following schema, where the keys are underlined (different keys are underlined differently):

```
PROFESSOR(Id, Name, Department)
COURSE(CrsCode, Department, CrsName)
TEACHING(ProfId, CrsCode, Semester)
```

Consider the following query:

```
SELECT   C.CrsName, P.Name
FROM     PROFESSOR P, TEACHING T, COURSE C
WHERE    T.Semester='F1995' AND P.Department='CS'
         AND P.Id = T.ProfId AND T.CrsCode=C.CrsCode
```

Assume the following statistical information:

- 1000 tuples with 10 tuples per page in PROFESSOR relation
- 20,000 tuples with 10 tuples per page in TEACHING relation
- 2000 tuples, 5 tuples per page, in COURSE
- 5-page buffer
- 50 different values for `Department`
- 200 different values for `Semester`
- Indices
  - PROFESSOR relation
    - On `Department`: Clustered, 2-level B^+ tree
    - On `Id`: Unclustered, hash
  - COURSE relation
    - On `CrsCode`: Sorted (no index)
    - On `CrsName`: Hash, unclustered
  - TEACHING relation
    - On `ProfId`: Clustered, 2-level B^+-tree
    - On `Semester, CrsCode`: Unclustered, 2-level B^+ tree

a. First, show the *unoptimized* relational algebra expression that corresponds to the above SQL query. Then *draw* the corresponding *fully pushed* query tree.

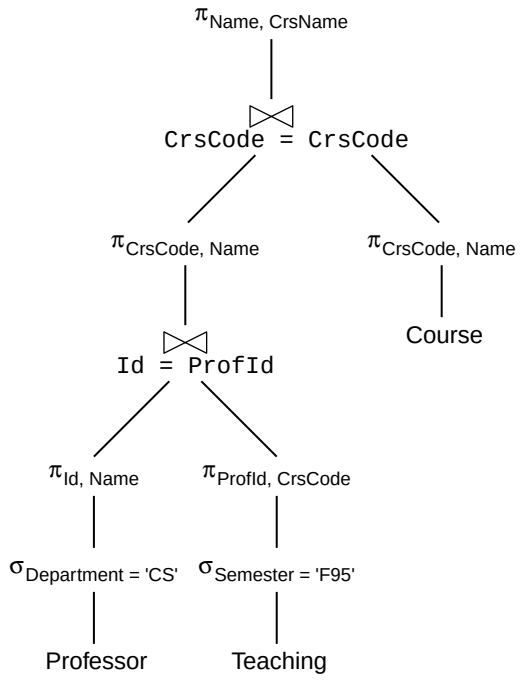


FIGURE 11.2

Solution:
The unoptimized algebraic expression is:

$$\pi_{\texttt{CrsName,Name}}(\sigma_{\texttt{Semester='F1995'}\ \text{AND}\ \textsc{Professor}.\texttt{Department='CS'}} (\textsc{Professor} \bowtie_{\texttt{Id=ProfId}} \textsc{Teaching} \bowtie_{\texttt{CrsCode=CrsCode}} \textsc{Course}))$$

One fully-pushed query tree is depicted in Figure 11.2. A slightly different tree results if we first join TEACHING with COURSE.

b. Find the best execution plan and its cost. Explain how you arrived at your costs.

Solution:
There are 100 pages in PROFESSOR, 2000 in TEACHING, and 400 pages in COURSES. With such large relations, doing straight joins is going to take many pages transfers, so we try selections first.

Selection on `Department` in the PROFESSOR relation can use the clustered 2-level index and yields $100/50 = 2$ data pages. The cost: 2 page transfers to search the index and 2 to fetch the data pages of PROFESSOR. In total, 4 page transfers.

It is tempting to select TEACHING on `Semester`, yielding $20000/200 = 100$ tuples (10 pages), but the index on `Semester` is unclustered, so it may take us 100 page transfers to find the relevant tuples, not counting the cost of the index search.

So, instead, we will use index-nested loops to join the 2 selected pages of PROFESSOR with TEACHING. Since the index on `ProfId` in TEACHING is clustered, we can fetch all tuples in TEACHING that match any given tuple in PROFESSOR in two page transfers (20000 teaching tuples per 1000 professors yields 20 tuples per professor, i.e., 2 pages). For each such fetch, we need to search the B^+ tree index on `ProfId`. We can keep the top level of that index in main memory, so there is only one index page fetch per search. Thus, the join takes 1 (to fetch the top level of index) + 20 (professors) * 3 (index search plus fetching the matching tuples) = 61.

The intermediate join has 20 * 20 = 400 tuples, each tuple twice the size of the tuples in either PROFESSOR or TEACHING. So, only 5 tuples fit in a page for a total of 80 pages. However, we can select the result of the join on `Semester` on the fly, reducing the size to just 2 tuples, 1 page. We could also apply projection on `Name` and `CrsCode`, but this will not give us any more gains. Since we are talking about just a 1-page result, there is no need to write it on disk — we can keep it in main memory.

Now, since COURSE is sorted on `CrsCode`, we can use binary search on COURSE (for each of the two tuples in the join result) to find the matches. This would take us $\lceil log_2 400 \rceil = 9$ page transfers per search, 18 page transfers in total.

Thus, the total cost is 4+61+18=83 page transfers.

Note that joining TEACHING with COURSE first would require us to scan COURSE at least once (since there is no index on `CrsCode` in either relation). So the cost would be at least 400 pages — by far higher than the 83 pages that we came up with.

11.8 Consider the following relations that represent part of a real estate database:

```
AGENT(Id, AgentName)
HOUSE(Address, OwnerId, AgentId)
AMENITY(Address, Feature)
```

The AGENT relation keeps information on real estate agents, the HOUSE relation has information on who is selling the house and the agent involved, and the AMENITY relation provides information on the features of each house. Each relation has its keys underlined. Consider the following query:

```
SELECT   H.OwnerId, A.AgentName
FROM     HOUSE H, AGENT A, AMENITY Y
WHERE    H.Address=Y.Address AND A.Id = H.AgentId
             AND Y.Feature = '5BR' AND H.AgentId = '007'
```

Assume that the buffer space available for this query has 5 pages and that the following statistics and indices are available:

- AMENITY
  - 10,000 records on 1000 houses, 5 records per page
  - Clustered 2-level B^+ tree index on `Address`
  - Unclustered hash index on `Feature`, 50 features
- AGENT

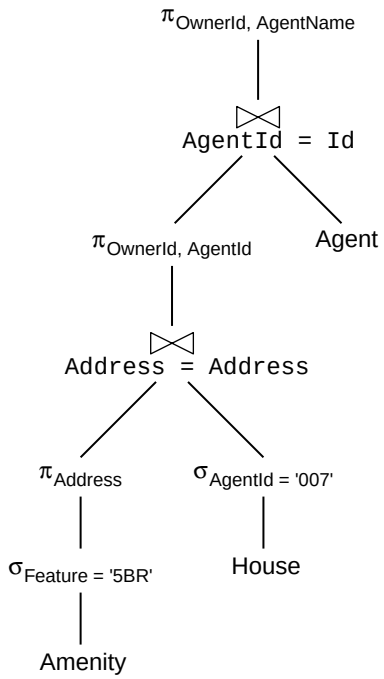


FIGURE 11.3

200 agents with 10 tuples per page
Unclustered hash index on `Id`

- HOUSE
  1000 houses with 4 records per page
  Unclustered hash index on `AgentId`
  Clustered 2-level B^+ tree index on `Address`

Answer the following questions (and explain how you arrived at your solutions).

a. Draw a fully pushed query tree corresponding to the above query.

Solution:
See Figure 11.3.

b. Find the best query plan to evaluate the above query and estimate its cost.

Solution:
We could join HOUSE with AGENT or AMENITY, but in any case it is clear that we should first select HOUSE on `AgentId`, because of the large reduction in size: There are 200 agents, 1000 houses, so agent 007 must be handling about 5 houses. At 4 records per page, this would occupy 2 pages.

Because the index on `AgentId` is unclustered, it would take 1.2 I/Os to find the bucket and 5 I/Os to fetch the relevant pages: 6.2 I/Os in total.

Next we can join with AGENT, which would take 1.2 page I/Os to search the index, since AGENT has an unclustered index on `Id`. In addition, 1 I/O is required to fetch the data page. The result will still have 5 tuples, but the tuples will be about 50% larger (AGENT has 10 tuples/page, while HOUSE only 4). However, we can project out `Id` and `AgentId`, which will bring the tuple size to about the size of the tuples in HOUSE. So, the join will still occupy a little over a page. We keep the result of the join in the main memory buffer.

Finally, we join the result with AMENITY. Since the statistics indicate that there are about 10 amenities per house, it does not make much sense to select AMENITY on `Feature`: the size will go down by the factor of 10 at a very high price (unclustered hash or scan of 2,000 blocks of the AMENITY relation), and we will loose the index on `Address` — the attribute used in the join.

So, the best way to do the join is to use index-nested loops join using the clustered index on the attribute `Address` of AMENITY. It would take 2 I/Os to search the B^+ tree index for each of the 5 tuples in the result of the previous join (i.e., 2*5; if we cache the top level of the B^+ tree then this search would take 2+4=6 I/Os). The number of tuples retrieved would be 50 (10 features per house * 5 houses), which occupies 10 pages. Therefore, the total needed to retrieve the matching tuples in AMENITY is 16 I/Os.

Note that we still have enough room in the buffer. The expected size of the join is 50 tuples (5*10), which is too large for our 5 page buffer. However, we can also select on `Feature='5BR'` on the fly, reducing the size to about 5 tuples, each about twice the size of the tuples in HOUSE. We can also project)on the fly) on `OwnerId` and `AgentName` further reducing the size. Thus, we will need 2 pages in the buffer for the result of the join of HOUSE and AGENT, one page for the input buffer that is needed for reading the matching tuples of AMENITY, and two to store the final result. This fits in the available 5 buffer pages.

In total, thus, the query will take 6.2 + 1.2 + 1 + 16 = 24.4 I/Os.

c. Find the next-best plan and estimate its cost.

Solution:
Similar, except that what is left of HOUSE is first joined with AMENITY. The number of I/Os is going to be the same, so this is also a best plan.

The next plan after that would be to do something silly, like joining HOUSE and AGENT using nested loops. Since AGENT occupies 20 blocks, this can be done in 20 I/Os. Then we could proceed to join with AMENITY.

11.9 None of the query execution plans in Figure 11.3 for queries (11.2)–(11.5) does duplicate elimination. To account for this, let us add one more relational operator, δ, which denotes the operation of duplicate elimination. Modify the plans in Figure 11.3 by adding δ in appropriate places so as to minimize the cost of the computation. Estimate the cost of each new plan.

Solution:
Duplicates arise only as a result of the projection operator. Since all plans in Figure 11.3 have projection only as the last step, there is not much room for duplicate elimination: δ should be the last operator applied after $\pi_{\texttt{Name}}$. Duplicate elimination can done at no extra cost (in terms of I/O), because the in-memory buffer is large enough to hold the result of the join (which we can sort on `Name` before passing it to the projection operator and then to the duplicate elimination operator).

Duplicate elimination does not save anything here either.

11.10 Build a database for the scenario in Exercise 11.5 using the DBMS of your choice. Use the EXPLAIN PLAN statement (or an equivalent provided by your DBMS) to compare the best plan that you found manually with the plan actually generated by the DBMS.

Solution:
This is a hands-on exercise where the students are invited to experiment with the EXPLAIN PLAN statement or the visual tuning tools provides by some DBMS vendors.

11.11 Follow Exercise 11.10, but use the scenario in Exercise 11.6.

Solution:
This is a hands-on exercise like 11.10 above.

11.12 Follow Exercise 11.10, but use the scenario in Exercise 11.7.

Solution:
This is a hands-on exercise like 11.10 above.

11.13 Consider the query execution plans in Figure 11.3. Show how each of these plans can be *enhanced* by pushing the projection operator past the join *without altering the strategies used for computing the join*.

Solution:
In plans (a) and (c), projection cannot be pushed to TEACHING, because this would obliterate the index on `ProfId` and prevent the use of the indexed join strategies.

In case (a), we must preserve `Id`, `DeptId`, and `Name` for the subsequent operations. Since these are all the attributes in PROFESSOR, there is nothing to push. In case (c), we can push $\pi_{\texttt{Id,Name}}$.

In case (b), we can push $\pi_{\texttt{Id,Name}}$ to PROFESSOR and $\pi_{\texttt{ProfId}}$ to TEACHING.

In case (d), we cannot push anything to PROFESSOR for the same reason as in case (a), but we can push $\pi_{\texttt{ProfId}}$ to TEACHING.

11.14 Using the result of Exercise 11.13, show which of the enhanced plans can be further enhanced by adding the duplicate elimination operator δ introduced in Exercise 11.9.

Solution:
Since projections applied to PROFESSOR must always preserve the attribute `Id` for the subsequent join and since this attribute forms a key, no duplicates are possible and, thus, δ would be useless after such a projection (and might actually incur costs only to find out that the operation has no effect).

In case of TEACHING, as discussed in the solution to exercise 11.13, projection ($\pi_{\texttt{ProfId}}$) can be pushed only in cases (b) and (d), because in other cases we use the index on the attribute `ProfId`. Since in a given semester a professor teaches on the average of two courses, duplicate elimination would reduce the size of the intermediate relation by the factor of two, maybe three. Let us examine whether the cost of such duplicate elimination is justified by the savings in computing the subsequent join.

In case (d), since sort-merge-join is used, the operands must be sorted anyway, so duplicate elimination can be done at no cost and would reduce the overall cost of the join.

In case (b), the result of selection on TEACHING has 250 pages. Since in this example we use a 51-page buffer, 5 runs can be produced at the cost of 500 I/Os

(read then write), and then they are merged and duplicates eliminated at the cost of another 250 I/Os (to just read the runs into the main memory). Thus, using duplicate elimination does not make sense in this case, because the total estimated cost of plan (b) in Figure 11.3 was just 512 pages.

12

Database Tuning

EXERCISES

12.1 Choose an index for each of the following SELECT statements. Specify whether your choice is clustered or unclustered and whether it is a hash index or a B^+ tree.

a.
```
SELECT   S.Name
FROM     STUDENT S
WHERE    S.Id = '111111111'
```

Solution:
Since `Id` is a primary key, an index on `Id` will automatically be created by the DBMS. Any index, clustered or unclustered, hash or B^+ tree, will support this query. Hence, no additional index is necessary.

b.
```
SELECT   S.Name
FROM     STUDENT S
WHERE    S.Status = 'Freshman'
```

Solution:
Roughly 25% of the rows in the Student table satisfy the condition and the DBMS will not automatically create an index on `Status` since it is not unique. A clustered index is needed in this case because of the number of rows that have to be retrieved. The index can be a hash or B^+ tree since all that is required is a grouping of all rows with a particular value of Status. The main index (on `Id`) will be unclustered in this case.

c.
```
SELECT   T.StudId
FROM     TRANSCRIPT T
WHERE    T.Grade = 'B' AND T.CrsCode = 'CS305'
```

Solution:
An index on `CrsCode` (rather than `Grade`) is needed since `CrsCode` is more selective. All rows satisfying the condition on `CrsCode` can be retrieved through the index and scanned, and only those satisfying the condition on `Grade` will be returned. The index should be clustered because the number of rows that will be selected is expected to be high. If the primary key (which involves the attributes `StudId`, `Semester` and `CrsCode`) is specified to be a clustered B+ tree with `CrsCode` as

the first search key attribute then it can support the query (note that a clustered hash index could not be used in this way).

d.
```
SELECT   P.Name
FROM     PROFESSOR P
WHERE    P.Salary BETWEEN 20000 AND 150000
```

Solution:
The DBMS will not automatically create an index on `Salary` since it is not unique. A clustered index is needed in this case because of the number of rows that have to be retrieved. The index must be a B^+ tree since since a range of salaries is requested. Hence the primary index (on `Id`) will be unclustered.

e.
```
SELECT   T.ProfId
FROM     TEACHING T
WHERE    T.CrsCode LIKE 'CS%' AND T.Semester = 'F2000'
```

Solution:
An index on `CrsCode` (rather than `Semester`) is needed since `CrsCode` is more selective. All rows satisfying the condition on `CrsCode` can be retrieved through the index and scanned, and only those satisfying the condition on `Semester` will be returned. The index should Be clustered B^+ tree because the number of rows that will be selected is expected to be high and due to the fact that a range of search key values has effectively been specified (note: `CS` is a prefix of the required course codes, so the range is `CS000` through `CS999`). If the primary key (which involves the attributes `Semester` and `CrsCode`) is specified to be a clustered B^+ tree and `CrsCode` is the first search key attribute in that tree then this index can support the query.

f.
```
SELECT   C.CrsName
FROM     COURSE C, TEACHING T
WHERE    C.CrsCode = T.CrsCode AND T.Semester = 'F2002'
```

Solution:
The key of COURSE is `CrsCode` and there will be an index on this. This is sufficient to support an index nested loop join with TEACHING in the outer loop. Each row of TEACHING can be scanned (no index on TEACHING used) and if it satisfies the condition on `Semester`, the unique row of COURSE that it joins with can be found using this index. Any type of index will work (clustered, unclustered, B^+ tree, hash). An improvement on this is to have a clustered index (B^+ tree or hash) on `Semester` of TEACHING since then a complete scan of TEACHING can be avoided. By declaring the key of TEACHING to be (`Semester`, `CrsCode`) the main index can be used for this purpose.

12.2 Suppose both queries (e) and (f) from the previous exercise need to be supported. What indices should be chosen for TEACHING and COURSE?

Solution:
A clustered index on `Semester` supports (f). It can also be used to support (e) (although it does not do the job as well as a clustered index on `CrsCode`). Use this solution if (f) is the more important query. If (e) is more important, use a clustered index on `CrsCode` of TEACHING to evaluate (e); the DBMS can use an index-nested

loop join (using the index on `CrsCode` in COURSE and scanning TEACHING in the outer loop) to do (f). Recall that `CrsCode` is the primary key in COURSE so COURSE will have an index on `CrsCode` to support such a join.

12.3 The table FACULTY has 60,000 rows, each row occupies 100 bytes, and the database page size is 4^k bytes. Assuming pages in the index and data files are 100% occupied, estimate the number of page transfers required for the following `SELECT` statement in each of the cases listed below.

```
SELECT   F.DeptId
FROM     FACULTY F
WHERE    F.Id = '111111111'
```

a. The table has no index.

Solution:
Since each page of the table accommodates about 40 rows, the table occupies about 1500 pages and on average half of these will have to be transferred.

b. The table has a clustered B^+ tree index on `Id`. Assume a (nonleaf) index entry has 20 characters.

Solution:
A non-leaf page in the index can accommodate 200 index entries, so a three-level index is required. Hence, 3 page transfers will be required.

c. The table has an unclustered B^+ tree index on `Id`. Assume a (nonleaf) index entry has 20 characters.

Solution:
A three-level index is still required. Hence, 4 page transfers are required (one additional transfer to get the row).

d. The table has an unclustered B^+ tree index on (`Id`, `DeptId`). Assume that an index entry now has 25 characters.

Solution:
160 index entries fit in a page and a three level index is required. An index-only strategy is now possible involving 3 page transfers

e. The table has an unclustered B^+ tree index on (`DeptId`, `Id`). Assume that an index entry now has 25 characters.

Solution:
160 index entries fit in a page and a three level index is required. An index-only strategy is now possible although a scan of the leaf level is now required. The leaf level contains about $60000/160 = 375$ pages, and about half of these will have to be transferred.

12.4 The table FACULTY has 60,000 rows, each row occupies 100 bytes, and the database page size is 4^k bytes. The table contains an attribute `City`, indicating the city in which a professor lives, there are 50 cities with names `city10` . . . `city50`, and professors are randomly distributed over the cities. Assuming that pages in the index and data files

are 100% occupied, estimate the number of page transfers required for the following SELECT statement in each of the cases listed below.

```
SELECT   F.Id
FROM     FACULTY F
WHERE    F.City > 'city10' AND F.City < 'city21'
```

a. The table has no index.

Solution:
Since each page of the table accommodates about 40 rows, the table occupies about 1500 pages and all will have to be transferred.

b. The table has a clustered B^+ tree index on `City`. Assume a (nonleaf) index entry has 25 characters.

Solution:
A three level index will be required. Instead of transferring the entire table, only pages containing cities in the range will have to be transferred: about 1/5 of the data pages, or 300 pages (ignoring the I/O required to search the index).

c. The table has an unclustered B^+ tree index on `City`. Assume an index entry has 25 characters.

Solution:
Since rows in the range are scattered throughout the data file we can expect the virtually every page will contain a row to be retrieved. Hence, it is unlikely that the query plan will use the index and a file scan involving 1500 page transfers is likely.

d. The table has an unclustered B^+ tree index on (`City`, `Id`). Assume an index entry has 40 characters.

Solution:
An index-only query plan can be used. Since 100 index entries fit in a page, the leaf level will have 600 pages and 120 will be transferred (ignoring the I/O required to search the index).

e. The table has an unclustered B^+ tree index on (`Id`, `City`). Assume an index entry has 40 characters.

Solution:
Now the entire leaf level of the index will have to be scanned, costing 600 page transfers.

12.5 Choose indices for the following SELECT statement. Specify whether your choices are clustered or unclustered, hash index or B^+ tree.

```
SELECT   C.CrsName, COUNT(*)
FROM     COURSE C, TRANSCRIPT T
WHERE    T.CrsCode = C.CrsCode AND T.Semester = :sem
GROUP BY T.CrsCode, C.CrsName
```

```
HAVING COUNT(*) ≥ 100
```

Solution:
Suppose the primary key of TRANSCRIPT is declared to be (`Semester`, `CrsCode`, `StudId`) with a corresponding B^+ tree. Then the DBMS can identify all rows of TRANSCRIPT satisfying the condition on `Semester`. These rows will already be grouped on `CrsCode` and hence the index supports the GROUP BY clause of the query. Each group can be scanned to determine which satisfies the HAVING clause and then any index on the `CrsCode` attribute of Course can be used to locate the unique row of COURSE containing the `CrsName` attribute for the group. Note that since the search key of the index on TRANSCRIPT contains both `Semester` and `CrsCode`, it supports an index-only search. Hence, the rows of TRANSCRIPT do not have to be accessed and as a result the index does not have to be clustered.

12.6 Consider the following query:

```
SELECT   T.CrsCode, T.Grade
FROM     TRANSCRIPT T, STUDENT S
WHERE    T.StudId = S.Id AND S.Name = 'Joe'
```

Assume that `Id` is the primary key of STUDENT, (`CrsCode, Semester, StudId`) is the primary key of TRANSCRIPT, and that `Name` is not unique. Set up a database containing these two tables on the DBMS available to you. Initialize the tables with a large number of rows. Write a program that measures the query execution time by reading the clock before and after submitting the query. Be sure to flush the cache between successive measurements (perhaps by executing a query that randomly reads a sufficient number of rows of a large dummy table).

a. Test your understanding by making an educated guess of what query plan will be chosen by the DBMS assuming that there are no indices other than those for the primary keys. Run the query, output the query plan, and check your guess. Measure the response time.

Solution:
A reasonable plan is one in which a selection is done on STUDENT using the condition `S.Name = :name` and then the `Id`s obtained from the resulting rows are used in a scan of TRANSCRIPT.

b. Now assume that an unclustered index on StudId on TRANSCRIPT is added. What query plan would you expect? Run the query, check your answer, and measure the response time. Try the query under two conditions: Joe has taken very few courses; Joe has taken many courses.

Solution:
A reasonable plan is one in which a selection is done on STUDENT using the condition `S.Name = :name` and then the index is used on TRANSCRIPT to locate the matching rows. If Joe has taken only a few courses the response time should be much smaller than that in (a). If Joe has taken many courses the response time will be worse. Unless your DBMS keeps statistics on the number of courses per student, it probably won't be smart enough to know that a table scan is better in this case.

c. In addition to the index added in (b), assume that an unclustered index on STUDENT on `Name` has been added and repeat the experiment.

Solution:
Now the selection on STUDENT is replaced by an index search using the new index, causing a further reduction in the response time.

12.7 Consider the table AUTHORS with attributes `Name`, `Publ`, `Title`, and `YearPub`. Assume that `Name` is the primary key (authors' names are unique) and hence one would expect that the DBMS would automatically create a clustered index on that attribute. Consider the statement

```
SELECT   A.Publ, COUNT(*)
FROM     AUTHORS A
WHERE     . . . range predicate on YearPub . . .
GROUP BY A.Publ
```

a. Assume that the statement is generally executed with a very narrow range specified in the WHERE clause (the publication year of only a few books will fall within the range). What indices would you create for the table and what query plan would you hope the query optimizer would use (include any changes you might make to the index on `Name`).

Solution:
You might choose an unclustered B^+ tree index on `YearPub` in which case the query plan would search the index using the small end of the range and then scan forward along the leaf level to locate the (few) rows satisfying the WHERE clause. Unfortunately, these rows would not be grouped together in the table, so they would have to be randomly retrieved. The plan would then sort them by `Publ` and count the size of each group. Alternatively, you might change the index on `Name` from clustered to unclustered and create a clustered B^+ tree index `YearPub`. In this case the qualifying rows would be contiguous in the table.

b. Repeat (a) assuming that a very broad range is generally specified.

Solution:
You might change the index on `Name` from clustered to unclustered and create a clustered B^+ tree index `Publ`. This would be useful if it is expected that a significant fraction of the rows in the table fit within the range and hence most of the pages will be read. In that case the query plan will scan the entire table (instead of trying to retrieve qualifying rows through an index on `YearPub`). Groups are stored contiguously in the table and the scan can simply count the qualifying rows one group at a time as the scan progresses without having to do an expensive sort.

12.8 Give the trigger that maintains the consistency of the database when a `GPA` column is added to the table STUDENT, as described in Section 12.2.2.

Solution:

12.9 In applications that cannot tolerate duplicates it may be necessary to use DISTINCT. However, the query plan needed to support DISTINCT requires a sort, which is

expensive. Therefore you should only use DISTINCT when duplicates are possible in the result set. Using the schema of Section 4.8, check the following queries to see if duplicates are possible. Explain your answer in each case.

a.
```
SELECT   S.Name
FROM     STUDENT S
WHERE    S.Id LIKE '1'
```

Solution:
Duplicates are possible since students with different Ids might have the same name

b.
```
SELECT   S.Id
FROM     STUDENT S, FACULTY F
WHERE    S.Address = F.Address
```

Solution:
Since Address is not a key of FACULTY, there may be several rows of FACULTY that match a row of STUDENT. Hence duplicates are possible.

c.
```
SELECT   C.CrsCode, COUNT(*)
FROM     TRANSCRIPT T
GROUP BY T.CrsCode
```

Solution:
No duplicate possible since all rows of TRANSCRIPT with a particular value of CrsCode are grouped into a unique row in the result set.

d.
```
SELECT   F.Name, F.DeptId, C.ClassTime, C.CrsCode,
         C.Semester, C.Year
FROM     FACULTY, CLASS C
WHERE F.Id = C.InstructorId
```

Solution:
Since a professor can teach only one class at a particular time there can be no duplicates

e.
```
SELECT   S.Name, F.Name, T.Semester, T.Year
FROM     FACULTY, CLASS C, TRANSCRIPT T, STUDENT S
WHERE F.Id = C.InstructorId AND S.Id = T.StudId AND
         C.CrsCode = T.CrsCode AND
         C.SectionNo = T.SectNo AND
         C.Year = T.Year AND C.Semester = T.Semester
```

Solution:
Duplicates are possible since a student may be taught by a professor in two different courses in the same semester.

12.10 A particular query can have several formulations, and a query optimizer may produce different query plans with different costs for each.

a. Assume that the Computer Science Department teaches only three 100-level courses: CS110, CS113, and CS114. Write an SQL statement whose result set contains the course codes of all courses that have these as prerequisites in three ways: using OR, UNION, and a nested subquery involving LIKE.

Solution:

```
SELECT   R.CrsCode
FROM     REQUIRES R
WHERE    R.PrereqCrsCode = 'CS110' OR
         R.PrereqCrsCode = 'CS113'
         R.PrereqCrsCode = 'CS114'
```

```
SELECT   R.CrsCode
FROM     REQUIRES R
WHERE    R.PrereqCrsCode = 'CS110'
UNION
SELECT   R.CrsCode
FROM     REQUIRES R
WHERE    R.PrereqCrsCode = 'CS113'
UNION
SELECT   R.CrsCode
FROM     REQUIRES R
WHERE    R.PrereqCrsCode = 'CS114'
```

```
SELECT   R.CrsCode
FROM     REQUIRES R
WHERE    R.PrereqCrsCode LIKE 'CS1%'
```

b. Write an SQL statement whose result set contains the names of all computer science courses that are prerequisites to other courses in three ways: using a join, a nested subquery involving EXISTS, and a nested subquery involving IN.

Solution:

```
SELECT   C.CrsName
FROM     COURSE C, REQUIRES R
WHERE    C.CrsCode = R.PrereqCrsCode
```

```
SELECT   C.CrsName
FROM     COURSE C
WHERE EXISTS (
              SELECT *
              FROM     REQUIRES R
```

```
                WHERE    C.CrsCode = R.PrereqCrsCode)
```

```
SELECT   C.CrsName
FROM     COURSE C
WHERE C.CrsCode IN (
              SELECT   R.PrereqCrsCode
              FROM     REQUIRES R)
```

12.11 On page 440 we discussed the choice of indexes to optimize the execution of the nested query (12.1) and pointed out that a clustered index on TEACHING with search key `CrsCode` would not be considered. It might, however, be used in optimizing the execution of the equivalent, non-nested query

```
SELECT   C.CrsName, P.Name
FROM     PROFESSOR P, TEACHING T, COURSE C
WHERE    T.Semester='S2003' AND P.Department='CS'
         AND C.DeptId = 'MAT'
         AND P.Id = T.ProfId AND T.CrsCode=C.CrsCode
```

Under what conditions might such an index be used?

Solution:
If, in addition, there was a clustered index on COURSE with search key `DeptId` the computation of the second join in (12.2) could be done very quickly. If the join resulted in a small number of rows, the remaining computation could also be done quickly.

PART FOUR

Advanced Topics in Databases

13

Relational Calculus, Visual Query Languages, and Deductive Databases

EXERCISES

13.1 a. Explain why tuple relational calculus is said to be declarative whereas relational algebra is said to be procedural.

Solution:
A query in relational algebra is an expression involving relational operators, which provide a step-by-step procedure for computing the desired answer. A query in relational calculus describes the desired answer without specifying how the answer is to be computed. It specifies the result without specifying an algorithm.

b. Explain why, even though SQL is declarative, the query optimizer in a relational DBMS translates SQL statements into relational algebra, which is procedural.

Solution:
The relational algebra is procedural and thus provides a concrete algorithm for query evaluation. Algebraic equivalences are then used to find equivalent but more efficient expressions.

13.2 Express in words the meaning of each of the following expressions (where $\textsc{Took}(s, c)$ means "student s took course c"). For example, the meaning of one of these expressions is "Every student has taken at least one course."

a. $\exists \mathtt{S} \in \textsc{Student}\ (\forall \mathtt{C} \in \textsc{Course}\ \ \textsc{Took}(\mathtt{S}, \mathtt{C}))$

Solution:
There is a student who has taken all courses.

b. $\forall \mathtt{S} \in \textsc{Student}\ (\exists \mathtt{C} \in \textsc{Course}\ \ \textsc{Took}(\mathtt{S}, \mathtt{C}))$

Solution:
Every student has taken at least one course.

c. $\exists \mathtt{C} \in \textsc{Course}\ (\forall \mathtt{S} \in \textsc{Student}\ \ \textsc{Took}(\mathtt{S}, \mathtt{C}))$

Solution:
There is a course that was taken by all students.

d. $\forall \mathtt{C} \in \textsc{Course}\ (\exists \mathtt{S} \in \textsc{Student}\ \ \textsc{Took}(\mathtt{S}, \mathtt{C}))$

Solution:
Every course was taken by at least one student.

13.3 Write the query that finds all students who took some course from Professor Joe Public; use TRC and DRC.

Solution:
TRC:

```
{ S.Name | STUDENT(S) AND TRANSCRIPT(R) AND
       TEACHING(T) AND PROFESSOR(P) AND
        S.Id = R.StudId AND R.CrsCode = T.CrsCode AND
        R.Semester = T.Semester AND T.ProfId = P.ProfId AND
        P.Name = 'Joe Public' }
```

DRC:

```
{ SN | ∃ Id,Addr,Status,CrsCode,Sem,Gr,PId,Dept (
     STUDENT(Id,SN,Addr,Status) AND
     TRANSCRIPT(Id,CrsCode,Sem,Gr) AND
     TEACHING(PId,CrsCode,Sem) AND
     PROFESSOR(PId,'Joe Public',Dept) ) }
```

13.4 Write a query that finds all students who took MAT123 from Professor Ann White but did not take MAT321 from her.

Solution:
We will use DRC. The TRC solution is similar.

```
{ SN | ∃ Id,Addr,Status,Sem,Gr,PId,Dept (
          STUDENT(Id,SN,Addr,Status) AND
          TRANSCRIPT(Id,'MAT123',Sem,Gr) AND
          TEACHING(PId,'MAT123',Sem) AND
          PROFESSOR(PId,'Ann White',Dept) AND
          NOT ∃ Sem1,Gr1 (
               TRANSCRIPT(Id,'MAT321',Sem1,Gr1) AND
               TEACHING(PId,'MAT321',Sem1) )
     )}
```

13.5 Prove that any query in tuple relational calculus has an equivalent query in domain relational calculus, and vice versa.

Solution:
Here we only show the transformation from TRC to DRC and back. A rigorous proofs of the fact that these are equivalence transformations are long and tedious because of the need to test all cases during the structural induction on the form of the queries.

TRC to DRC: This transformation is simple: Each tuple variable of degree n is replaced with n new domain variables, one for each attribute of the relation over which T ranges. Terms like $T.attr$ are replaced with the domain variable that corresponds to the *attr*'s component of T. For instance, assuming that T is a tuple variable of degree 3, S of degree 2, A is the first attribute of $\mathbf{P}$ and B is the second attribute of $\mathbf{R}$,

$$\forall T \exists S(\ldots \mathbf{P}(T) \ldots\ T.A = S.B \ldots \mathbf{R}(S))$$

becomes

$$\forall T_1 \forall T_2 \forall T_3 \exists S_1 \exists S_2(\ldots \mathbf{P}(T_1, T_2, T_3) \ldots\ T_1 = S_2 \ldots \mathbf{R}(S_1, S_2))$$

DRC to TRC: This direction is much more complex. To simplify the language, we adopt the positional notation for accessing the components of tuple variables. For instance, if T ranges over $\mathbf{R}$ with the attributes `A`, `B`, `C`, then instead of $T.\mathtt{A}$, $T.\mathtt{B}$, and $T.\mathtt{C}$ we write $T.1$, $T.2$, and $T.3$.

To see the difficulty in translating DRC to TRC, consider the following DRC query:

$$\{X \mid \forall Y \exists Z(\mathbf{P}(X, Y, Z))\}$$

We cannot simply replace $\langle X, Y, Z\rangle$ with a single tuple variable T because X is free, Y is quantified with $\forall$, and Z with $\exists$ and, thus, the proper quantification of T is not obvious. After moment's reflection one might come up with the following solution:

$$\{T_X \mid \forall T_Y \exists T_Z \exists T(\mathbf{P}(T) \text{ AND } T.1 = T_X.1 \text{ AND } T.2 = T_Y.1 \text{ AND } T.3 = T_Z.1)\}$$

where the subscript of a tuple variable indicates which domain variable is replaced by that tuple variable.

The overall proof is by induction on the number of predicates, connectives, and quantifiers in the DRC query to be converted into TRC.

Base case: The query is $\{X_1, \ldots, X_n \mid \mathbf{P}(X_1, \ldots, X_n)\}$. The translation is simple: $\{T \mid \mathbf{P}(T)\}$.

Inductive hypothesis: Suppose the claim is true for all DRC queries of length less than N. Consider a DRC formula of length $N + 1$. We have to consider a number of cases, which correspond to the basic operations through which one builds a DRC query.

Case 1: the query is

$$\{X_1, \ldots, X_m, \ldots, X_n, \ldots, X_r \mid \mathcal{P}(X_1, \ldots, X_m, \ldots, X_n) \text{ AND } \mathcal{Q}(X_m, \ldots, X_n, \ldots, X_r)\}$$

where $\mathcal{P}(X_1, \ldots, X_m, \ldots, X_n)$ is a domain calculus query with the free variables $X_1, \ldots, X_m, \ldots, X_n$ and $\mathcal{Q}(X_m, \ldots, X_n, \ldots, X_r)$ is a domain calculus query with the free variables $X_m, \ldots, X_n, \ldots, X_r$. Since the two components of the query are at most N long, we know, by the inductive hypothesis, that they translate into TRC: $\{T \mid \mathcal{P}'(T)\}$ and $\{S \mid \mathcal{Q}'(S)\}$. The original query then translates as

$$\{R \mid \exists T \exists S(\mathcal{P}'(T) \text{ AND } \mathcal{Q}'(S)$$

$R.1 = T.1$ AND . . . AND $R.n = T.n$ AND $R.m = S.m$ AND . . . AND $R.r = S.r)\}$

Cases 2 and 3: The query is formed using OR or NOT. These cases are similar to the above.

Case 4: The query is $\{X_1, \ldots, X_m | \exists X_{m+1}(\mathcal{P}(X_1, \ldots, X_m, X_{m+1}))\}$, where $\mathcal{P}(X_1, \ldots, X_m, X_{m+1})$ has $X_1, \ldots, X_m, X_{m+1}$ as free variables. Since this formula has length N, the inductive hypothesis says that it translates into TRC: $\{T|\mathcal{P}'(T)\}$.

The original query can then be translated as

$\{R|\exists T(\mathcal{P}'(T)$ AND $R.1 = T.1$ AND . . . AND$R.m = T.m)\}$.

Case 5: The query is $\{X_1, \ldots, X_m | \forall X_{m+1}(\mathcal{P}(X_1, \ldots, X_m, X_{m+1}))\}$, where $\mathcal{P}(X_1, \ldots, X_m, X_{m+1})$ has $X_1, \ldots, X_m, X_{m+1}$ as free variables. Again, by the inductive hypothesis, this translates to the TRC query $\{T|\mathcal{P}'(T)\}$.

The original query can then be translated as

$\{R|\forall T' \exists T(\mathcal{P}'(T)$ AND $R.1 = T.1$ AND . . . AND $R.m = T.m$
AND $T.(m+1) = T'.(m+1))\}$

* **13.6** Prove that relational algebra queries are domain-independent. *Hint:* Use induction on the structure of the relational algebra expression.

Solution:

The proof is by induction on the structure of algebraic expressions.

Base case: The expression is **P**, where **P** is a relation name. Clearly, the contents of the relation assigned to **P** in the database is fixed and does not depend on the domain.

Inductive hypothesis: Assume that all expressions of length less than or equal to N are domain independent. We will prove that then any expression of length $M + 1$ is domain independent.

We need to consider the various cases that correspond to each of the relational operators. In all cases, the reasoning is similar. For instance, let the query be $E_1 \cup E_2$, where E_1 and E_2 are relational expressions of smaller length. By inductive hypothesis, they are domain independent. Clearly, then, their union is domain independent as well.

13.7 Consider the relation schema corresponding to the IsA hierarchy in Figure 4.6. Assume that this schema has one relation per entity. (Consult Section 4.4 to refresh your memory about translation of IsA hierarchies into the relational model.) Write the following queries both in tuple and domain relational calculus:

a. Find the names of all sophomores in the computer science major.

Solution:

```
{Name | ∃DOB∃SSN∃Major
     (PERSON(SSN,Name,DOB) AND SOPHOMORE(SSN,Major)) }
```

b. Find the names of all students in the computer science major.

Solution:

```
{Name | ∃DOB∃SSN∃Major∃Advisor (PERSON(SSN,Name,DOB) AND
     (SOPHOMORE(SSN,Major) OR
      JUNIOR(SSN,Major) OR
      SENIOR(SSN,Major,Advisor) ) ) }
```

c. Find all departments where some technician has every specialization that any other technician (in the same or another department) has.

Solution:
One way to answer this query is to define a view with one attribute, `Name`:

```
SPECIALIZATION = {Spec | ∃SSN(TECHNICIAN(SSN, Spec))}
```

Then we can write the query as

```
{Department | ∃SSN∃Salary (EMPLOYEE(SSN,Department,Salary)
     AND ∀Spec ∈ SPECIALIZATION.Name (TECHNICIAN(SSN,Spec)) )
```

Without the use of views, the query would require the implication connective $\rightarrow$:

```
{Department | ∃SSN∃Salary (EMPLOYEE(SSN,Department,Salary)
     AND ∀Specialization∀SSN2 (TECHNICIAN(SSN2,Specialization)
               → TECHNICIAN(SSN,Specialization)) )
```

13.8 Write a domain relational calculus query that is equivalent to the following algebraic expressions:

a. $\mathbf{R} - \mathbf{S}$

Solution:

$$\{X_1, \ldots, X_n \mid \mathbf{R}(X_1, \ldots, X_n) \text{ AND } \text{NOT}(\mathbf{S}(X_1, \ldots, X_n)) \}$$

b. $\mathbf{R}/\mathbf{S}$, where relation $\mathbf{R}$ has attributes A, B and $\mathbf{S}$ has only one attribute, B.

Solution:

$$\{A \mid \forall B \in \mathbf{S} \ (\mathbf{R}(A, B)) \}$$

13.9 Express each of the following queries in tuple relational calculus, domain relational calculus, and QBE using the schema of Figure 3.4, page 38.

a. Find all courses that are taught by professors who belong to either of the departments `EE` and `MGT`. (Assume that certain courses that are listed in one department can be taught by professors from other departments.)

Solution:

– TRC:

```
{C.CrsCode, C.CrsName |
     COURSE(C) AND ∃T ∈TEACHING ∃P ∈PROFESSOR
     (P.Id = T.ProfId AND T.CrsCode = C.CrsCode
      AND (P.DeptId = 'EE' OR P.DeptId = 'MGT')) }
```

– DRC:

```
{CrsCode, CrsName |
     ∃DeptId1∃DeptId2∃Descr∃ProfId∃Semester∃Name
          (COURSE(DeptId1,CrsCode,CrsName,Descr) AND
            TEACHING(ProfId,CrsCode,Semester) AND
            PROFESSOR(ProfId,Name,DeptId2) AND
             (DeptId2 = 'EE' OR DeptId2 = 'MGT')) }
```

b. List the names of all students who took courses in spring 1997 and fall 1998.

Solution:

– TRC:

```
{S.Name | STUDENT(S) AND ∃T1 ∈TRANSCRIPT ∃T2 ∈TRANSCRIPT
          (S.Id = T1.StudId AND T1.Semester = 'S1997'
           AND S.Id = T2.StudId AND T2.Semester = 'F1998') }
```

– DRC:

```
{Name | ∃Id∃Address∃Status∃CrsCode1∃CrsCode2∃Grade1∃Grade2
          (STUDENT(Id,Name,Address,Status) AND
           TRANSCRIPT(Id,CrsCode1,'S1997',Grade1) AND
           TRANSCRIPT(Id,CrsCode2,'F1998',Grade2) ) }
```

c. List the names of all students who took courses from at least two professors in different departments.

Solution:

– TRC:

```
{S.Name | STUDENT(S) AND ∃R1 ∈ TRANSCRIPT ∃R2 ∈ TRANSCRIPT
          ∃T1 ∈ TEACHING ∃T2 ∈ TEACHING
          ∃P1 ∈ PROFESSOR ∃P2 ∈ PROFESSOR
          (R1.StudId = S.Id AND R2.StudId = S.Id
          AND R1.CrsCode = T1.CrsCode AND R2.CrsCode = T2.CrsCode
          AND T1.ProfId = P1.ProfId AND T2.ProfId = P2.ProfId
          AND P1.DeptId ≠ P2.DeptId) }
```

– DRC:

```
{Name | ∃Id∃Address∃Status
        ∃CrsCode1∃Semester1∃Grade1
        ∃CrsCode2∃Semester2∃Grade2
        ∃ProfId1∃ProfId2
        ∃PName1∃DeptId1∃PName2∃DeptId2
        (STUDENT(Id,Name,Address,Status) AND
         TRANSCRIPT(Id,CrsCode1,Semester1,Grade1) AND
         TRANSCRIPT(Id,CrsCode2,Semester2,Grade2) AND
         TEACHING(ProfId1,CrsCode1,Semester1) AND
         TEACHING(ProfId2,CrsCode2,Semester2) AND
         PROFESSOR(ProfId1,PName1,DeptId1) AND
         PROFESSOR(ProfId2,PName2,DeptId2) AND
            (DeptId1 ≠ DeptId2) }
```

d. Find all courses in department MGT that were taken by all students.

Solution:

– TRC:

```
{C.CrsCode, C.CrsName | COURSE(C) AND C.DeptId = 'MGT' AND
          ∀S ∈ STUDENT ∃R ∈ TRANSCRIPT
          (R.StudId = S.Id AND R.CrsCode = C.CrsCode) }
```

– DRC:

```
{CrsCode, CrsName | ∃Descr (COURSE('MGT',CrsCode,CrsName,Descr)) AND
          ∀Id ∈ STUDENT.Id ∃Semester∃Grade
            (TRANSCRIPT(Id,CrsCode,Semester,Grade) ) }
```

*e. Find every department that has a professor who has taught all courses ever offered by that department.

Solution:

– TRC:

```
{P.DeptId | PROFESSOR(P) AND
     ∀C ∈ COURSE(C.DeptId = P.DeptId →
          ∃T ∈ TEACHING(T.ProfId = P.Id AND T.CrsCode = C.CrsCode))}
```

– DRC:

```
{DeptId | ∃ProfId∃Name (PROFESSOR(ProfId, Name, DeptId) AND
     ∀CrsCode∀CrsName∀Descr
          (COURSE(DeptId, CrsCode, CrsName, Descr) →
           ∃Semester TEACHING(ProfId, CrsCode, Semester))) }
```

Compare the two calculi, QBE, and SQL with respect to the ease of their use for formulating the above queries.

Solution:
The DRC formulation is more complicated due to the need to quantify every domain variable. However, in cases (a), (b), and (c), all quantifiers are existential and so they can be dropped according to the convention discussed in this chapter. Dropping quantification is not feasible in (d) and (e) without introducing ambiguity, because ∃ and ∀ are mixed together (see the discussion at the end of Section 13.3).

13.10 If you have a copy of MS Access or of a similar DBMS, design the above queries using the visual languages that come with them.

Solution:
This is a hands-on exercise that involves a graphical interface to a database.

* **13.11** Write the query of Exercise 5.20 using TRC and DRC.

Solution:
Find all students (StudIds) who had taken a course from each professor in the MUS department.

One way to write this query is to first define a view, MUSPROF, which contains all tuples from PROFESSOR that correspond to Musics Department. Then we could use the quantifier $\forall P \in$ MUSPROF to express this query similarly to the use of the view CSPROF at the end of Section 13.1. The solution below uses the implication instead of the view.

- TRC:

```
{S.StudId | Student(S) AND
     ∀P ∈ PROFESSOR(P.Dept = 'MUS' →
          ∃R ∈ TRANSCRIPT ∃T ∈ TEACHING
          (S.Id = R.StudId AND P.Id = T.ProfId AND
           T.CrsCode = R.CrsCode AND T.Semester = R.Semester)) }
```

■

- DRC:

$$\{\texttt{StudId} \mid \exists \texttt{Name} \exists \texttt{Addr} \exists \texttt{Status}\ \textsc{Student}(\texttt{StudId}, \texttt{Name}, \texttt{Addr}, \texttt{Status})\ \texttt{AND}\ \forall \texttt{ProfId} \forall \texttt{ProfName}\ (\textsc{Professor}(\texttt{ProfId}, \texttt{ProfName}, \texttt{'MUS'}) \rightarrow \exists \texttt{CrsCode} \exists \texttt{Sem} \exists \texttt{Grade}\ (\textsc{Transcript}(\texttt{StudId}, \texttt{CrsCode}, \texttt{Sem}, \texttt{Grade})\ \texttt{AND}\ \textsc{Teaching}(\texttt{ProfId}, \texttt{CrsCode}, \texttt{Sem})))\ \}$$

* **13.12** Write the query of Exercise 5.22 using TRC and DRC.

Solution:
Find the names of all brokers who have made money in all accounts assigned to them.

- TRC:

$$\{\texttt{B.Name} \mid \textsc{Broker}(\texttt{B})\ \texttt{AND}\ \forall \texttt{A} \in \textsc{Account}\ (\texttt{A.BrokerId} = \texttt{B.Id} \rightarrow \texttt{A.Gain} > 0)\}$$

- DRC:

$$\{\texttt{Name} \mid \exists \texttt{BrokerId}\ (\textsc{Broker}(\texttt{BrokerId}, \texttt{Name})\ \texttt{AND}\ \forall \texttt{Acc\#} \forall \texttt{Gain}\ (\textsc{Account}(\texttt{Acc\#}, \texttt{BrokerId}, \texttt{Gain}) \rightarrow \texttt{Gain} > 0))\ \}$$

** **13.13** Consider the relation schema of Exercise 5.24. Write the following queries using TRC and DRC.

a. Find all customers who are interested in every house listed with Agent "007".

Solution:

– TRC:

$$\{\texttt{C.Id}, \texttt{C.Name} \mid \textsc{Customer}(\texttt{C})\ \texttt{AND}\ \forall \texttt{H} \in \textsc{House}\ (\texttt{H.AgentId = '007'} \rightarrow \forall \texttt{P} \in \textsc{Preference}\ \exists \texttt{A} \in \textsc{Amenity}\ (\texttt{H.Address = A.Address AND P.CustomerId = C.Id} \rightarrow \texttt{P.Feature} = \texttt{A.Feature}))\ \}$$

– DRC:

$$\{\texttt{Id}, \texttt{Name} \mid \textsc{Customer}(\texttt{Id}, \texttt{Name})\ \texttt{AND}\ \forall \texttt{Addr} \forall \texttt{OwnerId}\ (\textsc{House}(\texttt{Addr}, \texttt{OwnerId}, \texttt{'007'}) \rightarrow \forall \texttt{Feature}(\textsc{Preference}(\texttt{Id}, \texttt{Feature}) \rightarrow \textsc{Amenity}(\texttt{Addr}, \texttt{Feature})))\ \}$$

b. Using the previous query as a view, retrieve a set of tuples of the form $\langle$*feature*, *customer*$\rangle$ where each tuple in the result shows a feature and a customer who wants it such that

– Only the customers who are interested in every house listed with Agent "007" are considered; and
– The number of customers interested in "feature" is greater than 2. (If this number is 2 or less, the corresponding tuple ⟨*feature*, *customer*⟩ is not added to the result.)

This part of the query cannot be *conveniently* expressed by TRC or DRC because they lack the counting operator. However it is possible nevertheless (and is not hard).

Solution:
Let us denote the result of the above query as ClientOf007 and use it as a view.

– TRC:

```
{P.Feature,C.CustName |
     Preference(P) AND ClientOf007(C) AND P.CustId=C.CustId AND
     ∃P2 ∈ Preference ∃C2 ∈ ClientOf007
       ∃P3 ∈ Preference ∃C3 ∈ ClientOf007
          (P.Feature = P2.Feature AND P.Feature = P3.Feature AND
           P2.CustId = C2.CustId AND P3.CustId = C3.CustId
           P.CustId ≠ P2.CustId AND P.CustId ≠ P3.CustId
               AND P2.CustId ≠ P2.CustId) }
```

– DRC:

```
{Feature,CustName |
     Preference(CustId,Feature) AND ClientOf007(CustId,CustName)
     AND ∃CId2 ∈ ClientOf007.CustId ∃CId3 ∈ ClientOf007.CustId
          (Preference(CId2,Feature) AND Preference(CId3,Feature)
           AND CustId ≠ CId2 AND CustId ≠ CId3 AND CId2 ≠ CId3)
```

13.14 Write SQL query (5.10), page 150, and query (5.11), page 151, using TRC and DRC.

Solution:
Find all courses taught in fall 1995 together with the professors who taught those courses.

- TRC:

```
{C.Name, P.Name | Course(C) AND Professor(P) AND
     ∃T ∈ Teaching (T.Semester = 'F1995' AND
          T.CrsCode = C.CrsCode AND T.ProfId = P.Id)) }
```

- DRC:

```
{CName, PName | ∃CDept∃CrsCode∃PId∃PDept
     (Course(CDept,CrsCode,CName)
          AND Professor(PId,PName,PDept)
```

```
                AND TEACHING(PId,CrsCode,'F1995')) }
```

13.15 Investigate the logical relationship between the SQL operator EXISTS and the TRC quantifier ∃. For concreteness, express query (5.26), page 160, using TRC.

Solution:
The SQL EXISTS operator is more limited: it tests whether a set is empty. The ∃ quantifier in TRC is more versatile, especially when the query contains universal quantifiers and the NOT operator in addition to ∃. For instance, the query (5.26), page 160, can be expressed much more naturally in TRC using ∃ than in SQL using EXISTS:

```
{S.Id | STUDENT(S) AND
     NOT (∃T ∈ TRANSCRIPT ∃C ∈ COURSE
               (S.Id = T.StudId AND T.CrsCode = C.CrsCode
                    AND C.DeptId = 'CS') )
```

13.16 Consider the following relational schema:

```
SUPPLIER(Name,Part)
PROJECT(Name,Part)
```

A tuple, ⟨n,p⟩, in the first relation means that supplier n has part p. A tuple ⟨n,p⟩ in the second relation means that the project named n uses part p. Write the following query in tuple and domain relational calculi: *Find the names of suppliers who have a part, that is used by every project.*

Solution:

```
{ SN | ∃ Part( SUPPLIER(SN,Part) AND
               ∀ Prj ∈ PROJECT.Name PROJECT(PRJ,PART)) }

{ S.Name | SUPPLIER(S) AND
          ∃S1∈SUPPLIER(S1) ( S.Part = S1.Part
               AND ∀P∈PROJECT ∃P1∈PROJECT
                    ( P1.Name = P.Name AND P1.Part = S1.Part)) }
```

*13.17 Show that the iterative process of computing the transitive closure of PREREQ terminates after a finite number of steps. Show that this process can compute the transitive closure in polynomial time.

Solution:
A recursive definition of INDIRECTPREREQVIEW can be viewed as an operator, $T(X)$, which computes a relational algebra expression that involves X and some other, constant relations. For any X, $T(X)$ is a relation whose tuples consist of constants that appear in X and in the other constant relations that are used by T.

The computation starts by computing $T(\emptyset)$, then $T(T(\emptyset))$, etc. First, it is easy to see from the form of the definition of IndirectPrereqView that T is a monotone *operator*, i.e., if $X \subseteq Y$ then $T(X) \subseteq T(Y)$. Since $\emptyset \subseteq T(\emptyset)$, it is easy to see that

$$\emptyset \subseteq T(\emptyset) \subseteq T(T(\emptyset)) \subseteq \ldots$$

Let us denote the nth application of T by $T^n(\emptyset)$. Due to an earlier observation, the tuples in $T^n(\emptyset)$ must be composed only of the constants that appear in the constant relations that are part of T. Thus, the relation $T^n(\emptyset)$ cannot grow indefinitely and for some m we must have $T^{m+1}(\emptyset) = T^m(\emptyset)$, and the computation can stop there.

To see how this computation can be done in polynomial time, observe that $T(X_1 \cup X_2) = T(X_1) \cup T(X_2)$. Thus, the above computation can be modified as follows. Let us extend the definition of T^n with $T^1(\emptyset) = T(\emptyset)$ and $T^0(\emptyset) = \emptyset$.

$\Delta_0 = \emptyset$
$S_0 = \emptyset$
$\Delta_1 = T(\Delta_0) - S_0$
$S_1 = \Delta_1 \cup S_0$
$\Delta_2 = T(\Delta_1) - S_1$
$S_2 = \Delta_2 \cup S_1$
. . . . . .
$\Delta_n = T(\Delta_{n-1}) - S_{n-1}$
$S_n = \Delta_n \cup S_{n-1}$
. . . . . .

The computation terminates when $\Delta_k = \emptyset$ for some k. The number of steps in the above computation is bounded by the maximal number of tuples in $T^n(\emptyset)$ (because at each step we must add at least one tuple), and is polynomial (it is bounded by the number of rows in the Cartesian product of all columns of all constant relations in T. Each step in the above computation is also polynomial, because it consists of a join followed by a set-difference operator. Thus, the total time complexity of this computation is polynomial.

13.18 Show that in SQL-92 the use of negation in every query is stratified.

Solution:
Since SQL-92 has no recursion, whenever a view, **V**, depends on another relation **R** (view or base) through negation (`NOT` or `EXCEPT`) then the definition of **R** does not depend on **V**. This implies stratification.

13.19 Consider a relation DirectFlight`(StartCity, DestinationCity)` that lists all direct flights among cities. Use the recursion facility of SQL:1999 to write a query that finds all pairs $\langle city_1, city_2 \rangle$ such that there is an *indirect* flight from $city_1$ to $city_2$ with at least two stops in between.

Solution:
The simplest way to do this is to compute IndirectFlight similarly to IndirectPrereqView:

```
CREATE RECURSIVE VIEW IndirectFlight(From,To) AS
    SELECT * FROM DirectFlight
    UNION
    SELECT D.StartCity, I.To
    FROM DirectFlight D, IndirectFlight I
    WHERE D.DestinationCity = I.From
```

Then we can compute all flights with just one stop — Flight1 — and then subtract DirectFlight and Flight1 from IndirectFlight.

One might be tempted to first create a recursive view IndirectFlight2(From,TO,NumOfStops) and then select the flights with NumOfStops > 1.

```
CREATE RECURSIVE VIEW IndirectFlight2(From,To,Stops) AS
    SELECT D.StartCity, D.DestinationCity, 0
    FROM DirectFlight D
    UNION
    SELECT D.StartCity, I.To, I.Stops+1
    FROM DirectFlight D, IndirectFlight2 I
    WHERE D.DestinationCity = I.From
```

However, this recursive definition has a problem: because we keep incrementing the number of stops with each iteration, the evaluation process will not terminate. (Check that the termination condition will never be true!)

13.20 Use the recursion facility of SQL:1999 to express a so-called "same generation" query: *Given a* Parent *relation, find all pairs of people who have the same ancestor and are removed from her by equal number of generations*. (For example, a child is removed from her parent by one generation and from grandparent by two.)

Solution:

```
CREATE RECURSIVE VIEW SameGeneration(Cousin1, Cousin2) AS
    SELECT DISTINCT P.Parent, P.Parent
    FROM Parent P
    UNION
    SELECT DISTINCT P.Child, P.Child
    FROM Parent P
    UNION
    SELECT P1.Child, P2.Child
    FROM Parent P1, Parent P2, SameGeneration S
    WHERE P1.Parent = S.Cousin1 AND S.Cousin2 = P2.Parent
```

The first two (non-recursive) queries create the initial pairs of the form $\langle p, p \rangle$ (certainly, two occurrences of the same person is removed from a common ancestor by the same number of generations). The last, recursive, subquery computes the rest.

⋆**13.21** Consider the following bill of materials problem: the database has a relation SUBPART(**Part**, **Subpart**, **Quantity**), which tells which direct subparts are needed for each part and in what quantity. For instance, SUBPART(**mounting_assembly**, **screw**, 4) means that the mounting assembly includes four screws. For simplicity, let us assume that parts that do not have subparts (the *atomic* parts) are represented as having NULL as the only subpart (for instance, SUBPART(**screw**, **NULL**,0)). Write a recursive query to produce a list of all parts and for each part the number of primitive subparts it has.

Solution:

```
CREATE RECURSIVE VIEW PARTLISTCOUNT(Part, PrimitiveCount) AS
     SELECT S.Part, 1
     FROM SUBPART S
     WHERE S.Subpart IS NULL
     UNION
     SELECT S.Part, SUM(S.Quantity * L.PrimitiveCount)
     FROM SUBPART S, PARTLISTCOUNT L
     WHERE S.Subpart = L.Part
     GROUP BY S.Part
```

The first, non-recursive subquery starts the computation by producing the list of primitive parts and assigning 1 as the number of primitive components of each such part (the check IS NULL is intended to ensure that the part is primitive). The recursive subquery then computes the total number of primitive components for parts that are composed of several subparts.

13.22 Consider the following relational schema:

```
DIRECTFLIGHT(From,To,Distance)
```

Write a recursive SQL query that returns tuples of the form ⟨**From**,**To**,**Distance**⟩, which represent direct or indirect flights (i.e., flights composed of one or more segments) and the aggregate distance over all segments for each such flight. (If there are several ways to reach **B** from **A** and the total distance is different in each case, then the output would have several tuples such as ⟨**A**,**B**,1000⟩, ⟨**A**,**B**,1500⟩, etc.) The exact query is: *Find all tuples of the above form provided that the distance is less than 10,000*.

Solution:

```
WITH RECURSIVE INDIRECTFLIGHT(From,To,Distance) AS
   ( (SELECT  *  FROM  DIRECTFLIGHT D
      WHERE D.Distance < 10000)
      UNION
     (SELECT  From, To, D.Distance+I.Distance
```

```
      FROM  DirectFlight D, IndirectFlight I
      WHERE D.To = I.From AND D.Distance+I.Distance < 10000) )
  SELECT * FROM IndirectFlight
```

The iterative evaluation procedure for this query terminates, because the values for the `Distance` attribute cannot grow indefinitely due to the upper bound on distances. Note that there is another way to write this query, which might or might not terminate depending on whether the query optimizer is smart enough:

```
WITH RECURSIVE IndirectFlight1(From,To,Distance) AS
   ( (SELECT  *  FROM  DirectFlight)
      UNION
     (SELECT  From, To, D.Distance+I.Distance
      FROM  DirectFlight D, IndirectFlight1 I
      WHERE D.To = I.From) )
  SELECT *
  FROM IndirectFlight1 I
  WHERE  I.Distance < 10000
```

The difference is that now the selection is done in the query rather than in the recursive definition. By itself, the recursively defined relation IndirectFlight1 is infinite and the iterative procedure will never end. However, the optimizer might notice the selection condition in the outer query, `I.Distance < 10000`, and push it to the recursive part during the evaluation, thereby essentially reducing the problem to the first solution above. Algorithms for doing this kind of optimization for recursive queries are well known, but vendors might choose to not implement them.

13.23 Using Datalog, write the query that finds all students who took some course from Professor Joe Public.

Solution:

```
Answer(?SN) :- Student(?Id,?SN,?Addr,?Status),
               Transcript(?Id,?Crs,?Semester,?Grade),
               Teaching(?PId,?Crs,?Semester),
               Professor(?PId,'Joe Public',?Dept).
```

13.24 Use Datalog to find all students who took all classes ever offered by Professor Joe Public.

Solution:

```
Answer(?SN) :- Student(?Id,?SN,?Addr,?Status),
               not StudWhoDidNotTakeSomeClass(?Id).
StudWhoDidNotTakeSomeClass(?Id) :-
```

```
                        Student(?Id,?SN,?Addr,?Status),
                        Teaching(?PId,?Crs,?Sem),
                        Professor(?PId,'Joe Public',?Dept),
                        not TookClass(?Id,?Crs,?Sem).
TookClass(?Id,?Crs,?Sem) :- Transcript(?Id,?Crs,?Sem,?Gr).
```

13.25 Show how to express the relational operators of projection, selection, Cartesian product, natural join, union, and set-difference using Datalog.

Solution:
Projection on the first and third arguments of relation Rel with four attributes:

```
Projection(?A,?C) :- Rel(?A,?B,?C,?D).
```

Selection on the second attribute of the form `?B=1` and `?B > 12`:

```
Selection1(?A,1,?C,?D) :- Rel(?A,1,?C,?D).
Selection2(?A,?B,?C,?D) :- Rel(?A,?B,?C,?D), ?B > 12.
```

Cartesian product and natural join (for join, we assume that the second attribute of Rel1 has the same name as the first attribute of Rel2:

```
Product(?A,?B,?X,?Y,?Z) :- Rel1(?A,?B), Rel2(?X,?Y,?Z).
Join(?A,?B,?Y,?Z) :- Rel1(?A,?B), Rel2(?B,?Y,?Z).
```

Union and set-difference:

```
Union(?A,?B) :- Rel1(?A,?B) ; Rel2(?A,?B).
Difference(?A,?B) :- Rel1(?A,?B), not Rel2(?A,?B).
```

13.26 Express the query of Exercise 13.19 in Datalog.

Solution:
First, we define the transitive closure of DirectFlight:

```
Flight(?F,?T) :- DirectFlight(?F,?T).
Flight(?F,?T) :- DirectFlight(?F,?I), Flight(?I,?T).
```

Then we define the required query:

```
FlightsWithTwoOrMoreStops(?F,?T) :-
                        DirectFlight(?F,?S1),
                        DirectFlight(?S1,?S2),
```

```
                                    Flight(?S2,?T).
```

13.27 Using Datalog, express the "bill of materials" query described in Exercise 13.21.

Solution:
Here the student needs to explore the arithmetic operators of Datalog. We use the usual operator *is*, which evaluates arithmetic expressions.

```
PrimitiveSubpartCounts(?P,?Sub,?Qty) :-
                    PartList(?P,?Sub,?Qty), Primitive(?Sub).
PartList(?P,?Sub,?Qty) :- Subpart(?P,?Sub,?Qty).
PartList(?P,?Sub,?Qty) :-
                      Subpart(?P,?Sub1,?Qty1),
                      PartList(?Sub1,?Sub,?Qty2),
                      ?Qty is ?Qty1 * ?Qty2.
Primitive(?P) :- Subpart(?P,NULL,?Q).
```

There is no standard set of aggregate operators for Datalog. Some implementations have very powerful sets of such operators and some have none. Stronger students might explore Prolog's `fundall` operator here. For others, the instructor can just ask to use something like this: `sum(AggrVar,GroupByVar,Query)`. So, the required view predicate will then look like this:

```
PartListCount(?P,?Qty) :-
          ?Qty = sum(?Q,?P,PrimitiveSubpartCounts(?P,?Sub,?Q)).
```

13.28 Express the query of Exercise 13.22 in Datalog.

Solution:
Here the student needs to explore the arithmetic operators of Datalog. We use the usual operator *is*, which evaluates arithmetic expressions.

First, we define the transitive closure of DirectFlight:

```
Flight(?F,?T,?Dist) :- DirectFlight(?F,?T,?Dist).
Flight(?F,?T,?D) :-
             DirectFlight(?F,?I,?D1), Flight(?I,?T,?D2),
             ?D is ?D1+?D2.
```

The required query is now

```
?- Flight(?F,?T,?Dist), ?Dist < 10000.
```

14

Object Databases

EXERCISES

14.1 Give examples from the Student Registration System where

a. It would be convenient to use a set-valued attribute.

Solution:
The information about the semesters in which a particular course is offered (which is represented by the attribute `SemestersOffered` in Figure 4.33, page 114).

```
CREATE TYPE COURSE AS (
    ...
    SemestersOffered CHAR(6) MULTISET,
    ...
)
```

or

```
class COURSE {
    ...
    attribute Set<String> SemestersOffered;
    ...
}
```

b. It would be convenient to express a relationship (in the ODMG style) between two objects.

Solution:
The relationship REQUIRES in Figure 4.33, page 114:

```
class COURSE {
    ...
    relationship Set<COURSE> Requires
    ...
```

```
}
```

c. It would be convenient to use inheritance.

Solution:
The class CLASS inherits from COURSE:

```
CREATE TYPE CLASS UNDER COURSE AS (
    ...
)
```

or

```
class CLASS extends COURSE {
    ...
}
```

14.2 Specify an appropriate set of classes for the Student Registration System. Do this using first UDTs of SQL:2003 and then the ODL language of ODMG.

Solution:
We modify the tables defined in Section 4.8 in two ways: Using references to types instead of foreign key constraints and by making use of set-valued attributes.

```
CREATE TYPE COURSE AS (
   CrsCode CHAR(6),
   DeptId CHAR(4) NOT NULL,
   CrsName CHAR(20) NOT NULL,
   creditHours INTEGER NOT NULL,
   SemestersOffered CHAR(6) MULTISET,
   Requires REF(COURSE) MULTISET,
   PRIMARY KEY (CrsCode)
)

CREATE TYPE CLASS UNDER COURSE AS (
   CrsCode CHAR(6),
   SectionNo INTEGER,
   Semester CHAR(6),
   Year INTEGER,
   ClassTime CHAR(6),
   Enrollment INTEGER,
   MaxEnrollment INTEGER,
   Textbook CHAR(50),
   TaughtIn REF(ROOM),
   Instructor REF(FACULTY),
   PRIMARY KEY (CrsCode, SectionNo, Semester, Year)
```

```
)

CREATE TYPE Room AS (
   ClassroomId INTEGER,
   Seats INTEGER,
   PRIMARY KEY(ClassroomId)
)

CREATE TYPE Student AS (
   Id CHAR(9),
   Address CHAR(50),
   Name CHAR(20),
   Password CHAR(10),
   Transcript REF(TranscriptRecord) MULTISET,
   PRIMARY KEY(Id)
)

CREATE TYPE TranscriptRecord AS (
   Grade  CHAR(1),
   Class  REF(Class)
)

CREATE TYPE Faculty AS (
   Id CHAR(9),
   Name CHAR(20),
   Address CHAR(50),
   DeptId CHAR(4),
   Password CHAR(10),
   PRIMARY KEY(Id)
)
```

The ODMG version of the Student Registration Schema is as follows:

```
class Course
   (extent CourseExt
    keys   CrsCode)
{
   attribute String CrsCode;
   attribute String DeptId;
   attribute String CrsName;
   attribute Integer creditHours;
   attribute Set<String> SemestersOffered;
   relationship Set<Course> Requires;
}
```

```
class Class extends Course
    (extent ClassExt
     keys   (CrsCode, SectionNo, Semester, Year))
{
    attribute String  CrsCode;
    attribute Integer SectionNo;
    attribute String  Semester;
    attribute Integer Year;
    attribute String  ClassTime;
    attribute Integer Enrollment;
    attribute Integer MaxEnrollment;
    attribute String  Textbook;
    relationship Room    TaughtIn;
    relationship Faculty Instructor;
}

class Room
    (extent RoomExt
     keys   ClassroomId)
{
    attribute Integer ClassroomId;
    attribute Integer Seats;
}

class Student
    (extent StudentExt
     keys   Id)
{
    attribute String Id;
    attribute String Address;
    attribute String Name;
    attribute String Password;
    relationship Set<TranscriptRecord> Transcript;
}

class TranscriptRecord
    (extent TranscriptRecordExt)
{
    attribute    String  Grade;
    relationship Class   Class;
}

class Faculty
    (extent ProfessorExt
     keys   Id)
```

```
{
    attribute String Id;
    attribute String Name;
    attribute String Address;
    attribute String DeptId;
    attribute String Password;
}
```

14.3 Explain the difference between the object id in an object database and the primary key of a relation in a relational database.

Solution:
In an object database, each object has a unique and immutable identity, called its object id (oid). It which is assigned by the system when the object is created and does not change during the life of that object.

A primary key might change (a person might change her social security number), and it is typically assigned by the programmer, not DBMS.

14.4 Explain the different senses in which the objects in an object database can be considered equal.

Solution:

1. Two objects can have the same oid, in which case they are the same object.
2. They can have different oids, but both have the same attributes and relationships and the values of these attributes/relationships can be the same for both objects. In this case, we can have two further possibilities when we look deeper at what it means for the results returned by two relationships to be equal. Since these results are objects, they can be equal in the sense (1) or (2), which gives rise to two different notions.

14.5 A relational database might have a table called ACCOUNTS with tuples for each account and might support stored procedures `deposit()` and `withdraw()`. An object database might have a class (a UDT) called ACCOUNTS with an object for each account and methods `deposit()` and `withdraw()`. Explain the advantages and disadvantages of each approach.

Solution:
In an object-oriented database, methods would be stored together with the object. Although in both cases the method is likely to be executed on the server, an object DBMS might execute the method on the client side, if it supports the feature of code shipment between machines.

Also, in an ODMS we can use path expressions, such as `P.deposit()`, to invoke methods, which is slightly more convenient.

14.6 Consider the following type in CODM: `[Name: STRING, Members: {PERSON}, Address: [Building: INTEGER, Room: INTEGER]]`. Give three examples of subtype for this type. In one example, add more structure to the attribute `Members`; in another, add structure to the attribute `Address`; and in the third, add structure by introducing new attributes.

Solution:
In the first example, replace PERSON with EMPLOYEE. In the second, add the attribute `Floor` of type `INTEGER` to the type of `Address`. In the last example, add the attribute `Manager` of type PERSON to the top-level type.

14.7 Give an example of an object that belongs to the domain of the type `[Name: STRING, Children: {PERSON}, Cars: {[Make: STRING, Model: STRING, Year: STRING]}]`. Consider a supertype `[Name: STRING, Cars: {[Make: STRING, Model: STRING]}]` of that type. Show how one can obtain an object of the second type from the object of the first type, which you constructed earlier.

Solution:
An object of the first type can be

```
(#23, [John, {Bill, Bob},
             {[Dodge, Spirit, 2000], [Honda, Accord, 2004]}])
```

The corresponding object in the supertype is

```
(#23, [John, {[Dodge, Spirit], [Honda, Accord]}])
```

14.8 Consider the following type, which describes projects: `[Name: STRING, Members: {PERSON}, Address: [Building: INTEGER, Room: INTEGER]]`. Use SQL:1999/2003 to specify the UDT corresponding to this type.

Solution:
Let us assume that the type is called PROJECTTYPE.

```
CREATE TYPE ProjectType AS (
     Name CHAR(20),
     Members REF(Person) MULTISET,
     Address ROW(Building INTEGER, Room INTEGER) );
```

14.9 Use the UDT constructed in Exercise 14.8 to answer the following query: *List the names of all projects that have more than five members*.

Solution:
Assume that we have a table, PROJECT, of type PROJECTTYPE.

```
SELECT P.Name
FROM   Project P
WHERE  count(P->Id) > 5
```

14.10 Suppose that the ACCOUNTS class in the object database of the previous example has child classes SAVINGSACCOUNTS and CHECKINGACCOUNTS and that CHECKINGACCOUNTS has a child class ECONOMYCHECKINGACCOUNTS. Explain how the semantics of inheritance affects the retrieval of objects in each class. (For example, what classes

need to be accessed to retrieve all checking account objects that satisfy a particular predicate?)

Solution:
Different accounts can be in different classes. A query that potentially might need to access every account would have to access the classes ACCOUNTS, SAVINGSACCOUNTS, CHECKINGACCOUNTS, ECONOMYCHECKINGACCOUNTS.

14.11 Use SQL:2003 (with the MULTISET construct, if necessary) to complete the schema partially defined in Section 14.4. Include UDTs for the following tables: PERSON, STUDENT, COURSE, PROFESSOR, TEACHING, and TRANSCRIPT. Your solution should follow the object-oriented design methodology. Repeating the statements from Chapters 3 and 4, which use SQL-92, is *not* acceptable.

Solution:
The objective of this exercise is to have the students use an object-oriented design and not merely repeat the SQL-92 statements from Chapters 3 and 4.

```
CREATE TYPE PERSONTYPE AS (
     Name CHAR(20),
     Address ROW(Number INTEGER, Street CHAR(20), ZIP CHAR(5)) );
```

```
-- We make transcripts accessible from STUDENT, only
-- because this is how transcript records are usually accessed
-- Note: we are not using references for TRANSCRIPTRECORDTYPE,
-- because we are not going to gain much by sharing these objects
CREATE TYPE STUDENTTYPE UNDER PERSONTYPE AS (
     Id INTEGER,
     Status CHAR(2),
     Transcript TRANSCRIPTRECORDTYPE MULTISET );
CREATE TABLE STUDENT OF STUDENTTYPE;
```

```
CREATE TYPE PROFESSORTYPE UNDER PERSONTYPE AS (
     Id INTEGER,
     DeptId CHAR(4) );
-- Note: we define an extra oid attribute, prof_oid, using REF IS
-- because TEACHING references professor objects. Without the
-- explicit oid attribute we will not be able to populate TEACHING
-- with tuples that have correct references to PROFESSOR objects
CREATE TABLE PROFESSOR OF PROFESSORTYPE
REF IS prof_oid;
```

```
CREATE TYPE COURSETYPE AS (
```

```
    CrsCode CHAR(6),
    DeptId CHAR(4),
    CrsName CHAR(20),
    Description CHAR(50) );
-- Note: we define an extra oid attribute, course_oid, using REF IS
-- because TEACHING and TRANSCRIPTRECORDTYPE references course
-- objects. Without the explicit oid attribute we will not be able to populate
-- TEACHING and STUDENT (which uses TRANSCRIPTRECORDTYPE)
-- with tuples that have correct references to COURSE objects
CREATE TABLE COURSE OF COURSETYPE
REF IS course_oid;
```

```
CREATE TYPE TRANSCRIPTRECORDTYPE AS (
    Course REF(COURSETYPE) SCOPE COURSE,
    Semester CHAR(6),
    Grade CHAR(2) );
```

```
-- We could make teaching records accessible from PROFESSOR
-- but they are more often used to print catalogs, which justifies
-- having them as a separate table
CREATE TABLE TEACHING (
    Professor REF(PROFESSORTYPE) SCOPE PROFESSOR,
    Course REF(COURSETYPE) SCOPE COURSE,
    Semester CHAR(6),
    Grade CHAR(1) );
```

14.12 Use the schema defined for the previous problem to answer the following queries:

a. Find all students who have taken more than five classes in the Mathematics Department.

Solution:

```
SELECT S.Name
FROM STUDENT S
WHERE  5 < (SELECT count(S1.Transcript->Course)
            FROM   STUDENT S1
            WHERE  S1.Transcript->Course.DeptId = 'MAT'
                   AND  S1.Id = S.Id)
```

b. Represent grades as a UDT, called GRADETYPE, with a method, `value()`, that returns the grade's numeric value.

Solution:

```
CREATE TYPE GradeType AS (
    LetterValue CHAR(2) )
METHOD value() RETURNS DECIMAL(3);

CREATE METHOD value() FOR GradeType
RETURNS DECIMAL(3)
LANGUAGE SQL
   BEGIN
      IF LetterValue = 'A' THEN RETURN 4;
      IF LetterValue = 'A-' THEN RETURN 3.66;
      ... ... ...
   END
```

c. Write a method that, for each student, computes the average grade. This method requires the `value()` method that you constructed for the previous problem.

Solution:

```
SELECT S.Name, avg(S.Transcript.Grade.value())
FROM   Student S
```

14.13 Use SQL:2003 and its MULTISET construct to represent a bank database with UDTs for accounts, customers, and transactions.

Solution:

```
CREATE TYPE AcctType AS (
   Number  INTEGER,
   Type    CHAR(1),    -- e.g., 's' for savings
   Balance DECIMAL(10),
   Owners  REF(Customer MULTISET~),
   -- Note: we embed transactions directly in the data type for accounts
   Transactions TransactionType MULTISET );
CREATE TABLE Accounts OF AcctType;

CREATE TYPE CustomerType AS (
   Id       INTEGER,
   Name     CHAR(20),
   Address  ROW(Number INTEGER, Street CHAR(40),
                Town CHAR(30), Zip CHAR(5)),
   Accounts REF(AcctType MULTISET ) );
```

```
CREATE TABLE Customers OF CustomerType;

CREATE TYPE TransactionType AS (
   TrId     INTEGER,
   Date     DATE,
   Type     CHAR(1),     -- E.g., 'w', 'd'
   Amount   INTEGER );
```

Alternatively, we could have used the row type

```
ROW(TrId INTEGER, Date DATE, Type CHAR(1), Amount INTEGER)
```

directly in the declaration of the attribute `Transactions` in the UDT AcctType.

14.14 Use SQL:1999/2003 and the schema constructed for the previous exercise to answer the following queries:

a. Find all accounts of customers living at the postal ZIP code 12345.

Solution:

```
SELECT  A.Number
FROM    Accounts A
WHERE   A.Owners.Address.Zip = '12345'
```

b. Find the set of all customers satisfying the property that for each the total value of his or her accounts is at least $1,000,000.

Solution:

```
SELECT  C.Name
FROM    Customers C
WHERE   sum(C.Accounts->Balance) > 1000000
```

14.15 Explain the difference between a set object and a set of objects.

Solution:
A set-object is a single object, which represents a set. The elements of that set are other objects. A set of objects is just what it says: a set of distinct objects — it is not an object in its own right.

14.16 a. Explain the difference between ODMG attributes and relationships.

Solution:
ODMG distinguishes between *attributes* and *relationships*. An **attribute** refers to some value (not an object) that is stored within the object being specified. A **relationship** refers to some other object stored in the database and specifies how that

object is related to the object being specified. For instance, the PERSON definition includes two relationships. The `Spouse` relationship refers to another PERSON-object, (stored separately) that correspond to its spouse. The `Child` relationship refers to the set of PERSON objects (also stored separately) that correspond to its children.

b. Explain the difference between ODMG relationships and E-R relationships.

Solution:
The ODMG term "relationship" clashes with the use of this term in the E-R model. ODMG relationships and attributes are quite similar, except that relationships point to objects instead of values. In contrast, E-R relationships are similar to objects. They have their own structure, which is represented using attributes and roles. In fact, the notion of a role in E-R corresponds to the ODMG notion of relationship, because roles are essentially attributes that point to entities (instead of primitive types like integers and strings).

14.17 Explain the concept of type consistency of a path expression.

Solution:
Type consistency means that the type of any prefix in a path expression must be consistent with the use of the next attribute. For example, earlier we used the path expression

```
S.Address.StName
```

to find street names where people named `Smith` live. Type consistency here means that the type of the variable `S` is consistent with the use of the attribute `Address` and the type of the prefix `S.Address` is consistent with the use of the attribute `StName`. Type consistency of this expression follows from the following facts:

1. the variable `S` refers to an object of type PERSON;
2. the type PERSON has an `Address` attribute;
3. the subexpression `S.Address` returns a value of type ADDRESS;
4. the type ADDRESS has an `StName` attribute.

14.18 Add the appropriate inverse to the `Spouse` relationship in the PERSON definition given in Section 14.5.1 on page 546.

Solution:

```
class PERSON : OBJECT
(     extent PERSONEXT
      keys SSN, (Name, PhoneN) ): persistent
// properties of the instances of the type
{     attribute String Name;
      attribute String SSN;
      attribute enum Sex {m,f};
      attribute ADDRESS Address;
      attribute Set<String> PhoneN;
// relationships of instances of the type
```

```
        relationship PERSON Spouse
              inverse PERSON::Spouse;
        relationship Set <PERSON> Child;
    // methods of instances of the type
        void add_phone_number(in String Phone);
    }
```

14.19 Section 14.5.1 on page 546 has an ODL description of a PERSON object with the relationship `Spouse`.

a. How would you express in ODL that a person has a spouse?

Solution:

```
class PERSON {
    attribute   Integer Id;
    attribute   String  Name;
    relationship PERSON Spouse;
    ...
}
```

b. Give an OQL query that returns the name of a particular person's spouse.

Solution:

```
SELECT P.Spouse.Name
FROM   PERSONEXT P
WHERE P.Id = 123456789
```

14.20 Consider an ACCOUNT class and a TRANSACTIONACTIVITY class in a banking system.

a. Posit ODMG ODL class definitions for them. The ACCOUNT class must include a relationship to the set of objects in the TRANSACTIONACTIVITY class corresponding to the deposit and withdraw transactions executed against that account.

Solution:

```
class ACCOUNT {
    attribute Integer AcctId;
    relationship Set<PERSON> Owner;
    relationship Set<TRANSACTIONACTIVITY> Transactions
                 inverse TRANSACTIONACTIVITY::ActivityAccount;
}

class TRANSACTIONACTIVITY {
```

```
    attribute enum TransType {deposit,withdraw}  Type;
    attribute Float    Amount;
    attribute Date     ActivityDate;
    relationship ACCOUNT ActivityAccount
                 inverse ACCOUNT::Transactions;
}
```

Note that `Transactions` and `ActivityAccount` are declared to be inverses of each other. This means that if *ta* is a transaction activity and *ac* is an account such that $ta \in ac$.`Transactions` then $ac = ta$.`ActivityAccount`, and vice versa, if $ac = ta$.`ActivityAccount` then $ta \in ac$.`Transactions`.

b. Give an example of a database instance satisfying that description.

Solution:
An account:

```
(#123,   [12345,
           {#p4345, #p0987},
           {#t435, #t8132}
         ])
 . . .
```

Some transactions:

```
(#t435, [withdraw, 58.34, 2001-3-4, #123])
(#t8132, [deposit, 231.99, 2001-4-5, #123])
```

Some people (account holders):

```
(#p4345, [123456789, "John Doe"]),
(#p0987, [654345643, "Mary Doe"])  . . .
```

c. Write an OQL query against that database that will return the account numbers of all accounts for which there was at least one withdrawal of more than $10,000.

Solution:

```
SELECT A.AcctId
FROM  ACCOUNTEXT A, A.Transactions T
WHERE T.Type = withdraw AND T.Amount > 10000
```

14.21 Give an OQL query that returns the names of all spouses of all grandchildren of the person with SSN `123-45-6789` in the PERSON definition given in Section 14.5.1 on page 546.

Solution:

```
SELECT   distinct S.Child.Child.Spouse.Name
FROM     PersonExt S
WHERE    S.SSN = '123-45-6789'
```

14.22 Consider the class Person with an additional attribute, age. Write an OQL query that, for each age, produces a count of people of this age. Use two methods: with the GROUP BY clause and without it (using a nested query in SELECT). Describe a plausible query evaluation strategy in each case and explain which query will run faster.

Solution:

```
SELECT DISTINCT age: P.age,
                count: count(SELECT P2.Id
                             FROM PersonExt P2
                             WHERE P2.age = P.age)

FROM PersonExt P


SELECT DISTINCT age: P.age, count: count(P2.Id)
FROM PersonExt P, PersonExt P2
WHERE P.age = P2.age
GROUP BY P.age
```

In the first case, the DBMS is unlikely to figure out that it needs to count the Ids, so it would probably sort (or hash) the extent of the class Person on age and then for each value of this attribute it would execute the subquery. This would result in another access to the extent of the class Person.

In the second case, the ODBMS will know that it has to sort the class Person on age and then it will compute the result in one scan of the extent of that class.

14.23 Write an OQL query that, for each major, computes the number of students who have that major. Use the Student class defined in (14.14) on page 552.

Solution:

```
SELECT Major: M,  count: count(S2.Id)
FROM StudentExt S, StudentExt S2, S.Major M
WHERE  M IN S2.Major
GROUP BY M
```

We could also write this query using nested queries as follows:

```
SELECT DISTINCT Major: M,
                count: count(SELECT S2.Id
                             FROM StudentExt S2
                             WHERE M IN S2.Major)
FROM StudentExt S, S.Major M
```

14.24 E-R diagrams can be used for designing class definitions for object databases. Design ODMG class definitions for the E-R diagrams in Figure 4.1, Figure 4.6, and Figure 4.36.

Solution:

```
// For Figure 4.1
class Person : Object
    (extent PersonExt
     keys   SSN)
{
     attribute Integer SSN;
     attribute String  Name;
     attribute String  Address;
     attribute Set<String> Hobbies
}

// For Figure 4.6
class Person : Object
    (extent PersonExt
     keys   SSN)
{
     attribute Integer SSN;
     attribute String  Name;
     attribute Date    DOB;
}
class Employee extends Person
    (extent EmployeeExt)
{
     attribute String Department;
     attribute Float  Salary;
}
class Student extends Person
    (extent StudentExt)
{
     attribute Date  StartDate;
     attribute Float GPA;
```

```
}
class SECRETARY extends EMPLOYEE
    (extent SECRETARYEXT)
{
}
class TECHNICIAN extends EMPLOYEE
    (extent TECHNICIANEXT)
{
     attribute String  Specialization;
}
class FRESHMAN extends STUDENT
    (extent FRESHMANEXT)
{
}
class SOPHOMORE extends STUDENT
    (extent SOPHOMOREEXT)
{
     attribute String  Major;
}
class JUNIOR extends STUDENT
    (extent JUNIOREXT)
{
     attribute String  Major;
}
class SENIOR extends STUDENT
    (extent SENIOREXT)
{
     attribute    String    Major;
     relationship EMPLOYEE  Advisor;
}

// For Figure 4.36
class SEMESTER : OBJECT
    (extent SEMESTEREXT)
{
     attribute    Integer          Enrollment;
     relationship Set<Date> Holidays;
}
class COURSE : OBJECT
    (extent COURSEEXT)
{
     ... ... ...
}
struct TRANSCRIPTRECORD
{
```

```
        String       Grade;
        SEMESTER Semester;
        COURSE   Course;
}
class STUDENT extends PERSON
   (extent STUDENTEXT)
{
    attribute  Set<TRANSCRIPTRECORD>  Enrolled;
}
```

14.25 Consider the following ODL definitions:

```
class PERSON : OBJECT
      ( extent PERSONEXT ) : PERSISTENT;
{
  attribute String Name;
  ... ... ...
}
class STUDENT extends PERSON
      ( extent STUDENTEXT ) : PERSISTENT;
{
  attribute Integer Id;
  attribute Set<TRANSCRIPTRECORD> Transcript;
  ... ... ...
}
struct TRANSCRIPTRECORD {
   String CrsCode;
   float  Grade;
   String Semester;
}
```

Write the following query with and without the GROUP BY clause: *List all students with their corresponding average grade.*

Solution:

```
SELECT S.Name, avg(flatten(S.Transcript).Grade)
FROM   STUDENTEXT  S

SELECT S.Name, avg(T.Grade)
FROM   STUDENTEXT S, S.Transcript T
GROUP BY  S
```

The second query just complicates things — all for the sake of being able to use GROUP BY.

15

XML and Web Data

EXERCISES

15.1 Use XML to represent the contents of the STUDENT relation in Figure 3.2, page 36. Specify a DTD appropriate for this document. Do *not* use the representation proposed by the SQL/XML specification discussed in Section 15.4.4.

Solution:
Two representations are reasonable. One uses elements only:

```
<?xml version="1.0" ?>
<Students>
   <Student>
      <Id>s111111111</Id><Name>John Doe</Name>
      <Address>123 Main St.</Address><Status>Freshman</Status>
   </Student>
   .... and similarly for other students ...
</Students>
```

and the other uses attributes:

```
<?xml version="1.0" ?>
<Students>
   <Student Id="s111111111" Name="John Doe"
            Address="123 Main St." Status="Freshman"/>
   ..........
</Students>
```

The DTD for the first representation is:

```
<!DOCTYPE Students [
   <!ELEMENT Students (Student*)>
   <!ELEMENT Student (Id,Name,Address,Status)>
```

```
   <!ELEMENT Id (#PCDATA)>
   <!ELEMENT Name (#PCDATA)>
   <!ELEMENT Address (#PCDATA)>
   <!ELEMENT Status (#PCDATA)>
]>
```

For the second representation, the DTD can be:

```
<!DOCTYPE Students [
   <!ELEMENT Students (Student*)>
   <!ELEMENT Student EMPTY>
   <!ATTLIST Student
               Id ID #REQUIRED
               Name CDATA #REQUIRED
               Address CDATA #IMPLIED
               Status CDATA #IMPLIED>
]>
```

15.2 Specify a DTD appropriate for a document that contains data from both the COURSE table in Figure 4.34, page 116, and the REQUIRES table in Figure 4.35, page 117. Try to reflect as many constraints as the DTDs allow. Give an example of a document that conforms to your DTD.

Solution:
One good example of such a document is shown below. Note that it does not try to mimic the relational representation, but instead courses are modeled using the object-oriented approach.

```
<Courses>
   <Course CsrCode="CS315" DeptId="CS"
           CrsName="Transaction Processing" CreditHours="3">
      <Prerequisite CrsCode="CS305" EnforcedSince="2000/08/01"/>
      <Prerequisite CrsCode="CS219" EnforcedSince="2001/01/01"/>
   </Course>
   <Course>
      .....
   </Course>
   .....
</Courses>
```

An appropriate DTD would be:

```
<!DOCTYPE Courses [
   <!ELEMENT Courses (Course*)>
   <!ELEMENT Course (Prerequisite*)>
```

```
   <!ATTLIST Course
                CrsCode ID #REQUIRED
                DeptId  CDATA #IMPLIED
                CrsName CDATA #REQUIRED
                CreditHours CDATA #REQUIRED >
   <!ATTLIST Prerequisite
                CrsCode IDREF #REQUIRED
                EnforcedSince CDATA #REQUIRED>
]>
```

15.3 Restructure the document in Figure 15.4, page 591, so as to completely replace the elements `Name`, `Status`, `CrsCode`, `Semester`, and `CrsName` with attributes in the appropriate tags. Provide a DTD suitable for this document. Specify all applicable `ID` and `IDREF` constraints.

Solution:
The `Student` elements would now look as follows:

```
<Student StudId="s111111111" Name="John Doe" Status="U2">
   <CrsTaken CrsCode="CS308" Semester="F1997"/>
   <CrsTaken CrsCode="MAT123" Semester="F1997"/>
</Student>
```

The `Class` elements will now have the following form:

```
<Class CrsCode="CS308" Semester="F1994">
   <ClassRoster Members="s666666666 s987654321"/>
</Class>
```

The `Course` elements would now look as follows:

```
<Course CrsCode="CS308" CrsName="Software Engineering"/>
```

The corresponding DTDs will therefore be:

```
<!DOCTYPE Report [
    <!ELEMENT Report (Students,Classes,Courses)>
    <!ELEMENT Students (Student*)>
    <!ELEMENT Classes (Class*)>
    <!ELEMENT Courses (Course*)>
    <!ELEMENT Student (CrsTaken*)>
    <!ELEMENT CrsTaken EMPTY>
    <!ELEMENT Class (ClassRoster)>
    <!ELEMENT Course EMPTY>
    <!ELEMENT ClassRoster EMPTY>
```

```
    <!ATTLIST Report Date #IMPLIED>
    <!ATTLIST Student StudId ID #REQUIRED
                     Name CDATA #REQUIRED
                     Status CDATA #REQUIRED>
    <!ATTLIST Class CrsCode ID #REQUIRED
                    Semester CDATA #REQUIRED>
    <!ATTLIST Course CrsCode ID #REQUIRED
                     CrsName CDATA #REQUIRED>
    <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED
                       Semester CDATA #REQUIRED>
    <!ATTLIST ClassRoster Members IDREFS #IMPLIED>
]>
```

15.4 Define the following simple types:

a. A type whose domain consists of lists of strings, where each list consists of seven elements

Solution:
Assume that the target namespace is denoted by the prefix `my`.

```
<simpleType name="ListsOfStrings">
   <list itemType="string" />
</simpleType>
<simpleType name="ListsOfLength7">
   <restriction base="my:ListsOfStrings">
        <length value="7"/>
   </restriction>
</simpleType>
```

b. A type whose domain consists of lists of strings, where each string is of length seven

Solution:
Assume that the target namespace is denoted by the prefix `my`.

```
<simpleType name="StringsOfLength7">
   <restriction base="string">
      <length value="7"/>
   </restriction>
</simpleType>
<simpleType name="ListsOfStringsOfLength7">
   <list itemType="my:StringsOfLength7" />
</simpleType>
```

c. A type whose domain is a set of lists of strings, where each string has between seven and ten characters and each list has between seven and ten elements

Solution:
Assume that the target namespace is denoted by the prefix `my`.

```
<simpleType name="StringsOfLength7-10">
   <restriction base="string">
      <minLength value="7"/>
      <maxLength value="10"/>
   </restriction>
</simpleType>

<simpleType name="ListsOfStringsOfLength7-10">
   <list itemType="my:StringsOfLength7-10" />
</simpleType>

<simpleType name="ListsOfLength7-10 of StringsOfLength7-10">
   <restriction base="my:ListsOfStringsOfLength7-10">
        <minLength value="7"/>
        <maxLength value="10"/>
   </restriction>
</simpleType>
```

d. A type appropriate for the letter grades that students receive on completion of a course—A, A−, B+, B, B−, C+, C, C−, D, and F. Express this type in two different ways: as an enumeration and using the `pattern` tag of XML Schema.

Solution:

```
<simpleType name="gradesAsEnum">
   <restriction base="string">
      <enumeration value="A"/>
      <enumeration value="A-"/>
      <enumeration value="B+"/>
      ..........
   </restriction>
</simpleType>
<simpleType name="gradesAsPattern">
   <restriction base="string">
      <pattern value="(A-?|[BC][+-]?|[DF])"/>
   </restriction>
</simpleType>
```

In the `gradesAsPattern` representation, `(...)` represent complex alternatives of patterns separated by | (W3C has adopted the syntax of regular expressions used in

Perl), [...] represent simple alternatives (of characters), and ? represents zero or one occurrence, as usual in regular expressions.

15.5 Use the `key` statement of XML Schema to define the following key constraints for the document in Figure 15.4 on page 591:

a. The key for the collection of all `Student` elements

Solution:

```
<element name="Students" type="adm:studentList">
  <key name="StudentKey">
     <selector xpath="Student"/>
     <field xpath="@StudId"/>
  </key>
</element>
```

b. The key for the collection of all `Course` elements

Solution:

```
<element name="Courses" type="adm:courseCatalog">
  <key name="CourseKey">
     <selector xpath="Course"/>
     <field xpath="@CrsCode"/>
  </key>
</element>
```

c. The key for the collection of all `Class` elements

Solution:

```
<element name="Classes" type="adm:classOfferings">
  <key name="ClassKey">
     <selector xpath="Class"/>
     <field xpath="CrsCode"/>
     <field xpath="Semester"/>
  </key>
</element>
```

15.6 Assume that any student in the document of Figure 15.4 is uniquely identified by the last name and the status. Define this key constraint.

Solution:

```
<element name="Students" type="adm:studentList">
  <unique name="SecondaryStudKey">
     <selector xpath="Student"/>
     <field xpath="Name/Last"/>
     <field xpath="Status"/>
  </unique>
</element>
```

15.7 Use the `keyref` statement of XML Schema to define the following referential integrity for the document in Figure 15.4:

a. Every course code in a `CourseTaken` element must refer to a valid course.

Solution:

```
<element name="Students" type="adm:studentList">
  <keyref name="NoBogusCrsCodesInCrsTaken" refer="adm:CourseKey">
      <selector xpath="Student/CrsTaken"/>
      <field xpath="@CrsCode"/>
  </keyref>
</element>
```

where `adm` is the namespace prefix used for the schema of the document in Figure 15.4 and `CourseKey` is defined as the primary key for `Course` elements:

```
<element name="Courses" type="adm:courseCatalog">
  <key name="CourseKey">
     <selector xpath="Course"/>
     <field xpath="@CrsCode"/>
  </key>
</element>
```

b. Every course code in a `Class` element must refer to a valid course.

Solution:

```
<element name="Classes" type="adm:studentList">
   <keyref name="NoBogusCrsCodesInClasses"
                       refer="adm:CourseKey">
        <selector xpath="Class/CrsCode"/>
        <field xpath="."/>
   </keyref>
</element>
```

where `CourseKey` was defined above.

15.8 Express the following constraint on the document of Figure 15.4: no pair of `CourseTaken` elements within the same `Student` element can have identical values of the `CrsCode` attribute.

Solution:
XML Schema does not seem to preclude references to parent nodes from the nodes in a collection specified using the `selector` tag, so the following is allowed:

```
<element name="Students" type="adm:studentList">
  <unique name="Cannot take course twice">
    <selector xpath="Student/CourseTaken"/>
    <field xpath="../@StudId">
    <field xpath="@CrsCode">
  </unique>
</element>
```

15.9 Rearrange the structure of the `Class` element in Figure 15.4 so that it becomes possible to define the following referential integrity: every student Id mentioned in a `Class` element references a student from the same document.

Solution:
One way to rearrange this element is to represent each student Id in the class using an attribute:

```
<Class>
   <CrsCode>CS308</CrsCode><Semester>F1997</Semester>
   <Taker Id="s666666666"/> <Taker Id="s987654321"/>
</Class>
```

If we were to use DTDs, then we could declare:

```
          <!ATTLIST Taker Id IDREF #REQUIRED>
```

and ensure that the attribute `Id` here refers to an existing value in the same document. Unfortunately, there is no way for a DTD to say that this value must be a student Id. In XML schema we can use the `keyref` element to enforce such a constraint:

```
<element name="Classes" type="adm:classOfferings">
    <keyref name="NoBogusTakers" refer="adm:StudentKey">
        <selector xpath="Class/Taker"/>
        <field xpath="@Id"/>
    </keyref>
</element>
```

where `adm` is the namespace prefix used for the document in Figure 15.4 and `StudentKey` is the primary key defined for `Student` elements:

```
<element name="Students" type="adm:studentList">
   <key name="StudentKey">
       <selector xpath="Student"/>
       <field xpath="@StudId"/>
   </key>
</element>
```

15.10 Write a unified XML schema that covers both documents in Figures 15.19 and 15.21. Provide the appropriate key and foreign-key constraints.

Solution:
To improve the readability, we will be using a combination of named and anonymous types. In this way we strike a balance between the level of indirection in the schema definition and the level of nesting.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tnc="http://xyz.edu/transcripts-N-courses"
        targetNameSpace="http://xyz.edu/transcripts-N-courses">
   <element name="TranscriptsAndCourses">
      <complexType>
         <sequence>
            <element name="Transcripts">
               <complexType>
                  <sequence>
                     <element name="Transcript"
                              type="tnc:transcriptType"
                              minOccurs="0"
                              maxOccurs="unbounded"/>
                  </sequence>
               </complexType>
            </element>
            <element name="Classes" type="tnc:classListType">
               <complexType>
                  <sequence>
                     <element name="Class" type="tnc:classType"
                              minOccurs="0"
                              maxOccurs="unbounded"/>
                  </sequence>
               </complexType>
            </element>
         </sequence>
      </complexType>
```

```
   </element>

   <!-- Now define transcriptType and classType -->
   <complexType name="transcriptType">
      <sequence>
         <element name="Student">
            <complexType>
               <attribute name="StudId" type="integer"/>
               <attribute name="Name" type="string"/>
            </complexType>
         </element>
         <element name="CrsTaken"
                  minOccurs="0" maxOccurs="unbounded">
               <attribute name="CrsCode" type="string"/>
               <attribute name="Semester" type="string"/>
               <attribute name="Grade" type="string"/>
         </element>
      </sequence>
   </complexType>
   <complexType name="classType">
      <sequence>
         <element name="CrsName" type="string"/>
         <element name="Instructor" type="string"/>
      </sequence>
      <attribute name="CrsCode" type="string"/>
      <attribute name="Semester" type="string"/>
   </complexType>
</schema>
```

15.11 Use XML Schema to represent the fragment of the relational schema in Figure 3.6, page 43. Include all key and foreign-key constraints.

Solution:
We use a combination of named and anonymous types to define the schema.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:stureg="http://somewhere.edu/stureg-fragment"
        targetNameSpace="http://somewhere.edu/stureg-fragment">
   <element name="Everything">
      <complexType>
        <all>
          <element name="Students" type="stureg:stuList"/>
          <element name="Professors" type="stureg:profList"/>
          <element name="Courses" type="stureg:crsList"/>
          <element name="Transcripts" type="stureg:trnList"/>
```

```
        <element name="Teachings" type="stureg:tchList"/>
      </all>
    </complexType>
    <!-- various primary and candidate keys -->
    <key name="PrimaryStuKey">
       <selector xpath="Students">
       <field xpath="Student/@Id"/>
    </key>
    <key name="PrimaryProfKey">
       <selector xpath="Professors">
       <field xpath="Professor/@Id"/>
    </key>
    <key name="PrimaryCrsKey">
       <selector xpath="Courses">
       <field xpath="Course/@CrsCode"/>
    </key>
    <unique name="SecondaryCrsKey">
       <selector xpath="Courses">
       <field xpath="Course/@DeptId"/>
       <field xpath="Course/@CrsName"/>
    </key>
    <key name="PrimaryTrnKey">
       <selector xpath="Transcripts">
       <field xpath="Transcript/@StudId"/>
       <field xpath="Transcript/@CrsCode"/>
       <field xpath="Transcript/@Semester"/>
    </key>
    <key name="PrimaryTchKey">
       <selector xpath="Teachings">
       <field xpath="Teaching/@CrsCode"/>
       <field xpath="Teaching/@Semester"/>
    </key>
    <!-- Foreign keys for Transcript -->
    <keyref name="TranscriptFK1" refer="stureg:PrimaryStuKey">
       <selector xpath="Transcripts">
       <field xpath="Transcript/@StudId">
    </keyref>
    <keyref name="TranscriptFK2" refer="stureg:PrimaryCrsKey">
       <selector xpath="Courses">
       <field xpath="Course/@CrsCode">
    </keyref>
    <!-- Foreign keys for Teaching are similar -->
</element>

<complexType name="stuList">
```

```
      <sequence>
         <element name="Student"
                  minOccurs="0" maxOccurs="unbounded">
            <complexType>
               <attribute name="Id" type="integer"/>
               <attribute name="Name" type="string"/>
               <attribute name="Address" type="string"/>
               <attribute name="Status" type="string"/>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <complexType name="profList">
      <sequence>
         <element name="Professor"
                  minOccurs="0" maxOccurs="unbounded">
            <complexType>
               <attribute name="Id" type="integer"/>
               <attribute name="Name" type="string"/>
               <attribute name="DeptId" type="DeptIdType"/>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <complexType name="crsList">
      <sequence>
         <element name="Course"
                  minOccurs="0" maxOccurs="unbounded">
            <complexType>
               <attribute name="CrsCode" type="CrsCodeType"/>
               <attribute name="DeptId" type="DeptIdType"/>
               <attribute name="CrsName" type="string"/>
               <attribute name="Descr" type="string"/>
            </complexType>
         </element>
      </sequence>
   </complexType>
   <complexType name="trnList">
      <sequence>
         <element name="Transcript"
                  minOccurs="0" maxOccurs="unbounded">
            <complexType>
               <attribute name="StudId" type="integer"/>
               <attribute name="CrsCode" type="CrsCodeType"/>
               <attribute name="Semester" type="SemesterType"/>
```

```
                <attribute name="Grade" type="GradeType"/>
             </complexType>
          </element>
       </sequence>
    </complexType>
    <complexType name="tchList">
       <sequence>
          <element name="Teaching"
                   minOccurs="0" maxOccurs="unbounded">
             <complexType>
                <attribute name="ProfId" type="integer"/>
                <attribute name="CrsCode" type="CrsCodeType"/>
                <attribute name="Semester" type="SemesterType"/>
             </complexType>
          </element>
       </sequence>
    </complexType>

    <simpleType name="crsCodeType">
       <restriction base="string">
         <pattern value="[A-Z]{3}[0-9]{3}"/>
       </restriction>
    </simpleType>
    <simpleType name="SemesterType">
       <restriction base="string">
         <pattern value="(Fall|Spring|Summer)[0-9]{4}"/>
       </restriction>
    </simpleType>
    <simpleType name="GradeType">
       <restriction base="string">
           <enumeration value="A"/>
           <enumeration value="A-"/>
           <enumeration value="B+"/>
           ... ...
       </restriction>
    </simpleType>
    <simpleType name="DeptIdType">
       <restriction base="string">
         <pattern value="[A-Z]{3}"/>
       </restriction>
    </simpleType>
 </schema>
```

15.12 Write an XML Schema specification for a simple document that lists stockbrokers with the accounts that they handle and lists client accounts separately. The information about each broker includes the broker Id, name, and a list of accounts. The information about each account includes the account `Id`, the owner's name, and the account positions (i.e., stocks held in that account). To simplify matters, it suffices to list the stock symbol and quantity for each account position. Use `ID`, `IDREF`, and `IDREFS` to specify referential integrity.

Solution:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:brk="http://somebrokerage.com/documents"
        targetNameSpace="http://somebrokerage.com/documents">
  <element name="Brokerage">
    <complexType>
      <sequence>
        <element name="Broker" type="brk:brokerType"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element name="Account" type="brk:accountType"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
  <complexType name="brokerType">
    <attribute name="Id" type="ID" />
    <attribute name="Name" type="string" />
    <attribute name="Accounts" type="IDREFS" />
  </complexType>
  <complexType name="accountType">
    <element name="Positions" />
      <complexType>
        <sequence>
          <element name="Position"
                   minOccurs="0" maxOccurs="unbounded">
            <complexType>
              <attribute name="stockSym" type="string" />
              <attribute name="qty" type="integer" />
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
    <attribute name="Id" type="ID" />
    <attribute name="Owner" type="string" />
  </complexType>
```

```
</schema>
```

15.13 Write a sample XML document, which contains

- A list of parts (part name and Id)
- A list of suppliers (supplier name and Id)
- A list of projects; for each project element, a nested list of subelements that represent the parts used in that project. Include the information on who supplies that part and in what quantity.

Write a DTD for this document and an XML schema. Express all key and referential constraints. Choose your representation in such a way as to maximize the number of possible key and referential constraints representable using DTDs.

Solution:
Note that to satisfy the requirement that the key and referential constraints must be representable using DTDs, we *must* use attributes for part and supplier Ids.

```
<MyDocument>
   <Part Name="screw" Id="1234" />
   <Part Name="paint" Id="4321" />
   <Part Name="hanger" Id="9876" />
   <Part Name="brush" Id="30900" />
   <Supplier Name="Acme" Id="AC452" />
   <Supplier Name="ACE" Id="5432345" />
   <Project Name="Remodeling">
      <Uses Part="1234" Supplier="5432345" Qty="2" />
      <Uses Part="9876" Supplier="5432345" Qty="5" />
      <Uses Part="1234" Supplier="AC452" Qty="3" />
   </Project>
   <Project Name="Painting">
      <Uses Part="4321" Supplier="AC452" Qty="1" />
      <Uses Part="30900" Supplier="AC452" Qty="3" />
   </Project>
</MyDocument>

<!DOCTYPE MyDocument [
   <!ELEMENT MyDocument (Part*,Supplier*,Project*)>
   <!ELEMENT Part EMPTY>
   <!ELEMENT Supplier EMPTY>
   <!ELEMENT Project (Uses*)>
   <!ELEMENT Uses EMPTY>
   <!ATTLIST Part
                Name CDATA #REQUIRED
                Id ID #REQUIRED>
   <!ATTLIST Supplier
                Name CDATA #REQUIRED
```

```
                Id ID #REQUIRED>
   <!ATTLIST Project Name CDATA #REQUIRED>
   <!ATTLIST Uses
                Part IDREF #REQUIRED
                Supplier IDREF #REQUIRED
                Qty CDATA #REQUIRED>
]>


<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:doc="http://mydocuments.com/"
        targetNameSpace="http://mydocuments.com/">
    <element name="MyDocument">
       <complexType>
         <sequence>
            <element name="Part"
                     minOccurs="0" maxOccurs="unbounded">
               <complexType>
                  <attribute name="Name" type="string"/>
                  <attribute name="Id" type="string"/>
               </complexType>
            </element>
            <element name="Supplier"
                     minOccurs="0" maxOccurs="unbounded">
               <complexType>
                  <attribute name="Name" type="string"/>
                  <attribute name="Id" type="string"/>
               </complexType>
            </element>
            <element name="Project" type="doc:projType"
                              minOccurs="0" maxOccurs="unbounded"/>
         </sequence>
       </complexType>
       <key name="PartKey">
          <selector xpath="Part" />
          <field xpath="@Id" />
       </key>
       <key name="SupplierKey">
          <selector xpath="Supplier" />
          <field xpath="@Id" />
       </key>
       <keyref name="RefToPart" refer="doc:PartKey">
          <selector xpath="Project"/>
          <field xpath="@Part"/>
       </keyref>
```

```
        <keyref name="RefToSupplier" refer="doc:SupplierKey">
           <selector xpath="Project"/>
           <field xpath="@Supplier"/>
        </keyref>
      </element>
      <complexType name="projType">
        <sequence>
          <element name="Uses" minOccurs="0" maxOccurs="unbounded"/>
             <complexType>
                <attribute name="Part" type="string"/>
                <attribute name="Supplier" type="string"/>
                <attribute name="Qty" type="string"/>
             </complexType>
          </element>
        </sequence>
        <attribute name="Name" type="string"/>
      </complexType>
  </schema>
```

15.14 Use XPath to express the following queries to the document in Figure 15.19:

a. Find all `Student` elements whose Ids end with `987` and who took `MAT123`.

Solution:

```
//Student[../CrsTaken/@CrsCode="MAT123"
          and ends-with(@StudId,"987")]
```

b. Find all `Student` elements whose first names are Joe and who took fewer than three courses.

Solution:

```
//Student[starts-with(@Name,"Joe") and count(../CrsTaken) &lt 3]
```

c. Find all `CrsTaken` elements that correspond to semester `S1996` and that belong to students whose names begin with `P`.

Solution:

```
//CrsTaken[@Semester="S1996" and starts-with(../Student/@Name,"P")]
```

15.15 Formulate the following XPath queries for the document in Figure 15.21:

a. Find the names of all courses taught by Mary Doe in fall 1995.

Solution:

```
//CrsName[../Instructor="Mary Doe"
           and  ../@Semester="F1995"]/text()
```

Note that we use `text()` at the end to get the text (course names themselves) as opposed to `CrsCode` element nodes.

b. Find the set of all document nodes that correspond to the course names taught in fall 1996 or all instructors who taught `MAT123`.

Solution:

```
//CrsName[../@Semester="F1996"]
      |  //Instructor[../@CrsCode="MAT123"]
```

c. Find the set of all course codes taught by John Smyth in spring 1997.

Solution:

```
//Class[Instructor="John Smyth" and @Semester="S1997"]/@CrsCode
```

15.16 Use XSLT to transform the document in Figure 15.19 into a well-formed XML document that contains the list of `Student` elements such that each student in the list took a course in spring 1996.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
     <StudentListS1996>
        <xsl:apply-templates/>
     </StudentListS1996>
  </xsl:template>
  <xsl:template match="//Student">
     <xsl:if test="../CrsTaken/@Semester='S1996'">
         <xsl:copy-of select="."/>
     </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

15.17 Use XSLT to transform the document in Figure 15.21 into a well-formed XML document that contains the list of courses that were taught by Ann White in fall 1997. *Do not* include the `Instructor` child element in the output, and discard the `Semester` attribute.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
     <AnnWhiteS1997Courses>
        <xsl:apply-templates/>
     </AnnWhiteS1997Courses>
  </xsl:template>
  <xsl:template match="//Class">
     <xsl:if test="@Semester='F1997' and Instructor='Ann White'">
         <Course CrsCode="@CrsCode"/>
     </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

15.18 Write an XSLT stylesheet that traverses the document tree and, ignoring attributes, copies the elements and the text nodes. For instance, `<foo a="1"> the<best quality="S"/>bar</foo>` would be converted into `<foo>the <best/>bar</foo>`.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/|node()">
     <xsl:copy>
        <xsl:apply-templates/>
     </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Note that `<xsl:apply-templates/>` works on element and text nodes and ignores the attributes, which is precisely what we want.

15.19 Write an XSLT stylesheet that traverses the document tree and, ignoring attributes, copies the elements and *doubles* the text nodes. For instance, `<foo a="1">the<best/>bar</foo>` would be converted into `<foo>thethe<best/> barbar</foo>`.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/|*">
     <xsl:copy>
        <xsl:apply-templates/>
     </xsl:copy>
  </xsl:template>
  <xsl:template match="text()">
     <xsl:value-of select=".">
     <xsl:value-of select=".">
  </xsl:template>
</xsl:stylesheet>
```

15.20 Write an XSLT stylesheet that traverses the document and preserves all elements, attributes, and text nodes, but discards the `CrsTaken` elements (with all their children). The stylesheet must not depend on the knowledge of where the `CrsTaken` elements reside in the source document.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="@*|text()">
     <xsl:copy>
     </xsl:copy>
  </xsl:template>
  <xsl:template match="/|*">
     <xsl:if test="name(current()) != 'CrsTaken'">
         <xsl:copy>
            <xsl:apply-templates/>
         </xsl:copy>
     </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

15.21 Write a stylesheet that traverses the document tree and, while ignoring the attributes, preserves the other aspects of the tree (the parent–child relationships among the elements and the text nodes). However, when the stylesheet hits the `foobar` element,

its attributes and the entire structure beneath are preserved and the element itself is repeated twice.

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="text()">
     <xsl:value-of select="."/>
  </xsl:template>
  <xsl:template match="/|*">
     <xsl:if test="name(current()) != 'foobar'">
         <xsl:copy>
             <xsl:apply-templates/>
         </xsl:copy>
     </xsl:if>
     <xsl:if test="name(current()) = 'foobar'">
         <xsl:copy>
             <xsl:apply-templates/>
             <xsl:apply-templates select="@*"/>
         </xsl:copy>
         <xsl:copy>
             <xsl:apply-templates/>
             <xsl:apply-templates select="@*"/>
         </xsl:copy>
     </xsl:if>
  </xsl:template>
  <xsl:template match="@*">
      <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

Actually, a more elegant solution is to not have `xsl:if`. Instead, we can have one template that matches `"/|*` and another that matches `"//foobar"`. We use this technique in the solution to the next problem.

15.22 Write a stylesheet that traverses the document tree and preserves everything. However, when it hits the element `foo` it converts every t-child of `foo` into an element with the tag name `text`. For instance,

```
<foo a="1">the<best>bar<foo>in the</foo></best>world</foo>
```

would become

```
<foo a="1"><text>the</text><best>bar<foo> <text>in the</text>
</foo></best><text>world</text></foo>
```

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/|*">
     <xsl:copy>
         <xsl:apply-templates/>
         <xsl:apply-templates select="@*"/>
         <xsl:apply-templates select="text()"/>
     </xsl:copy>
  </xsl:template>
  <xsl:template match="//foo//text()">
     <text>
         <xsl:value-of select="."/>
     </text>
  </xsl:template>
  <xsl:template match="text()">
     <xsl:value-of select="."/>
  </xsl:template>
  <xsl:template match="@*">
      <xsl:value-of select="."/>
  </xsl:template>
</xsl:stylesheet>
```

The trick here is that we use two templates for text nodes. When a text node is a descendant of an element node `foo`, the first text template is used, because it is more specific. Otherwise, the second template is used.

15.23 Consider the relational schema in Figure 3.6, page 43. Assume that the Web server delivers the contents of these relations using the following XML format: the name of the relation is the top-level element, each tuple is represented as a `tuple` element, and each relation attribute is represented as an empty element that has a `value` attribute. For instance, the STUDENT relation would be represented as follows:

```
<Student>
     <tuple>
         <Id value="s111111111"/> <Name value="John Doe"/>
         <Address value="123 Main St."/> <Status value="U1"/>
     </tuple>
```

```
      .
      .
      .
  </Student>
```

Formulate the following queries using XQuery:

a. Produce the list of all students who live on Main Street.

Solution:

```
<StudentList>
{
  FOR  $s IN doc("http://xyz.edu/student.xml")//tuple
  WHERE  contains($s/Address/@value,"Main St")
  RETURN $s
}
</StudentList>
```

b. Find every course whose `CrsCode` value does not match the Id of the department that offers the course. (For instance, the course `IS315` in information systems might be offered by the Computer Science (`CS`) Department.)

Solution:

```
<CrsList>
{
  FOR $c IN doc("http://xyz.edu/course.xml")//tuple
  WHERE  NOT contains($c/CrsCode/@value,$c/DeptId/@value)
  RETURN $c
}
</CrsList>
```

c. Create a list of all records from the TEACHING relation that correspond to courses taught in the fall semester.

Solution:

```
<Teaching>
{
  FOR $t IN doc("http://xyz.edu/teaching.xml")//tuple
  WHERE  starts-with($t/Semester/@value,"F")
  RETURN $t
}
```

```
</Teaching>
```

15.24 Using the document structure described in Exercise 15.23, formulate the following queries in XQuery:

a. Create the list of all professors who ever taught MAT123. The information must include all attributes available from the PROFESSOR relation.

Solution:

```
<ProfList>
{
  FOR $p IN doc("http://xyz.edu/professor.xml")//tuple
  WHERE  doc("http://xyz.edu/teaching.xml")
                        //tuple[ProfId/@value=$p/Id/@value
                                and CrsCode/@value="MAT123"]
  RETURN $p
}
</ProfList>
```

b. Create the list of all students (include student Id and name) who took a course from John Smyth and received an A.

Solution:

```
<StudList>
{
  LET  $pset := doc("http://xyz.edu/professor.xml")
                          //tuple[Name/@value="John Smyth"]
  LET  $tset := doc("http://xyz.edu/teaching.xml")
                          //tuple[ProfId/@value=$pset/Id/@value]
  FOR  $s IN doc("http://xyz.edu/student.xml")//tuple,
       $p IN $pset,
       $t IN $tset
  WHERE  doc("http://xyz.edu/transcript.xml")
                    //tuple[StudId/@value=$s/Id/@value
                            and CrsCode/@value=$t/CrsCode/@value
                            and Semester/@value=$t/Semester/@value
                            and Grade/@value="A"]
  RETURN
      <tuple>
         $s/Id,
         $s/Name
      </tuple>
}
```

```
</StudList>
```

Note that the problem statement requires that we return only the name and the Id, so we cannot just return `$s`.

c. Create the list of all courses in which Joe Public received an A.

Solution:

```
<CrsList>
{
  LET $sset := doc("http://xyz.edu/student.xml")
                              //tuple[Name/@value="Joe Public"]
  FOR $c IN doc("http://xyz.edu/course.xml")//tuple,
      $s IN $sset
  WHERE  doc("http://xyz.edu/transcript.xml")
                             //tuple[StudId/@value=$s/Id/@value
                                    and Grade/@value="A"]
  RETURN $c
}
</CrsList>
```

15.25 Use the document structure described in Exercise 15.23 to formulate the following queries in XQuery:

a. Create an XML document that lists all students by name and, for each student, lists the student's transcript. The transcript records must include course code, semester, and grade.

Solution:

```
<StudList>
{
  FOR $s IN doc("http://xyz.edu/student.xml")//tuple
  LET $tr := doc("http://xyz.edu/transcript.xml")
                              //tuple[StudId/@value=$s/Id/@value]
  RETURN
        <Student>
            $s/Name
            {
                FOR $t IN $tr
                RETURN
                   <Transcript>
                     $t//CrsCode,
                     $t//Semester,
```

```
                    $t//Grade
                  </Transcript>
            }
        </Student>
}
</StudList>
```

b. Create a document that for each professor lists the name and classes taught. (A class is identified by a course code and semester.)

Solution:

```
<ProfList>
{
  FOR $p IN doc("http://xyz.edu/professor.xml")//tuple
  LET $tch := doc("http://xyz.edu/teaching.xml")
                          //tuple[ProfId/@value=$p/Id/@value]
  RETURN
        <Professor>
            $p/Name
            {
                FOR $t IN $tch
                RETURN
                   <Class>
                       $t//CrsCode,
                       $t//Semester
                   </Class>
            }
        </Professor>
}
</ProfList>
```

c. Create a document that lists every course and the professors who taught it. The course information must include the course name, and the professor information should include the professor's name.

Solution:

```
<Courses>
{
  FOR $c IN doc("http://xyz.edu/courses.xml")//tuple
  LET $tch := doc("http://xyz.edu/teaching.xml")
                        //tuple[CrsCode/@value=$c/CrsCode/@value]
  LET $profs := {
```

```
        FOR  $t IN $tch,
             $p IN doc("http://xyz.edu/professor.xml")
                                  //tuple[Id/@value=$t/ProfId/@value]
                }
  RETURN
       <Course>
           $c/CrsName
           {
               FOR  $p IN distinct-values($profs)
               RETURN <Prof> $p/Name </Prof>
           }
       </Course>
}
</Courses>
```

15.26 Use the document structure described in Exercise 15.23 to formulate the following queries in XQuery:

a. List all students who took more than three courses.

Solution:

```
<StudList>
{
  FOR $s IN doc("http://xyz.edu/student.xml")//tuple
  LET $tr := doc("http://xyz.edu/transcript.xml")
                              //tuple[StudId/@value=$s/Id/@value]
  WHERE  count($tr) > 3
  RETURN $s
}
</StudList>
```

b. List all students who received an A for more than three courses.

Solution:

```
<StudList>
{
  FOR $s IN doc("http://xyz.edu/student.xml")//tuple
  LET $tr := doc("http://xyz.edu/transcript.xml")
                              //tuple[StudId/@value=$s/Id/@value
                                      and Grade/@value="A"]
  WHERE  count($tr) > 3
  RETURN $s
}
```

```
</StudList>
```

c. List all professors who gave an A to more than three students.

Solution:

```
<ProfList>
{
  FOR $p IN doc("http://xyz.edu/professor.xml")//tuple
  LET $studs := {
        FOR
           $s IN doc("http://xyz.edu/student.xml")//tuple,
           $tch IN doc("http://xyz.edu/teaching.xml")
                                  //tuple[ProfId/@value=$p/Id/@value]
        WHERE
           doc("http://xyz.edu/transcript.xml")
                   //tuple[StudId/@value=$s/Id/@value
                           and Grade/@value="A"
                           and CrsCode/@value=$tch/CrsCode/@value
                           and Semester/@value=$tch/Semester/@value]
                   }
  WHERE  count(distinct-values($studs)) > 3
  RETURN $p
}
</ProfList>
```

d. List all classes (identified by a course code and semester) where average grade is higher than B.

Solution:
As in Section 15.4.3, assume that there is a function, `numericGrade()`, which returns the numeric value of a grade. For instance, in the U.S. `numericGrade("A")` = 4, `numericGrade("A-")` = 3.66, etc.

```
DECLARE FUNCTION  classesFromTranscripts($elts AS element()*) AS element()*
{
    FOR  $e  IN  $elts
    RETURN
        <tuple>
        { $e//CrsCode, $e//Semester }
        </tuple>
}

<EasyClasses>
{
```

```
  LET $transcripts := doc("http://xyz.edu/transcript.xml")
  FOR $c IN
      distinct-values(classesFromTranscripts($transcripts//tuple))
  LET $grades :=
      $transcripts//tuple[CrsCode/@value=$c/CrsCode/@value
                          and Semester/@value=$c/Semester/@value]
                     /Grade/@value
  WHERE  avg(numericGrade($grades)) > numericGrade("B")
  RETURN $c
}
</EasyClasses>
```

e. List all professors whose given grade (the average among all grades the professor ever assigned to any student) is B or higher. For this problem, you must write an XQuery function that computes numeric values of letter grades.

Solution:
Assume that `http://xyz.edu/auxtypes.xsd` contains a definition of the type `Grade` (a suitable restriction of `xs:string`).

```
DECLARE NAMESPACE  xs = "http://www.w3.org/2001/XMLSchema"
DECLARE NAMESPACE  xyzaux = "http://xyz.edu/auxtypes.xsd"
DECLARE FUNCTION numericGrade($g AS xyzaux:Grade) AS xs:float
   {
       IF $g = "A" THEN RETURN 4
       ELSE IF $g = "A-" THEN RETURN 3.66
       ELSE IF $g = "B+" THEN RETURN 3.33
       ELSE IF $g = "B" THEN RETURN 3
       ELSE IF $g = "B-" THEN RETURN 2.66
       ELSE IF $g = "C+" THEN RETURN 2.33
       ELSE IF $g = "C" THEN RETURN 2
       ELSE IF $g = "C-" THEN RETURN 1.66
       ELSE IF $g = "D" THEN RETURN 1
       ELSE IF $g = "F" THEN RETURN 0
   }
<EasyProfessors>
{
  FOR $p IN doc("http://xyz.edu/professor.xml")//tuple
  LET $transcripts := {
        FOR $tch IN doc("http://xyz.edu/teaching.xml")
                                 //tuple[ProfId/@value=$p/Id/@value],
            $tr IN doc("http://xyz.edu/transcript.xml")
                           //tuple[CrsCode/@value=$tch/CrsCode/@value
                                   and Semester/@value=$tch/Semester/@value]
        RETURN $tr
```

```
                        }
  WHERE
        avg(numericGrade($transcripts/Grade/@value)) >= numericGrade("B")
  RETURN $c
}
</EasyProfessors>
```

f. List all classes (identified by a course code and semester) where the average grade is less than the professor's given grade.

Solution:
First, create a document similar to the one computed in the previous sub-exercise, where the structure is

```
<GivenGrades>
   <tuple><ProfId value="..."/><Grade value="..."/></tuple>
   ... ... ...
</GivenGrades>
```

The `Grade` element gives the value of the given grade of the professor identified by the `ProfId` element. Assume that this document is in `file:///tmp/givengrades.xml`

```
<ClassList>
{
  FOR $tch IN doc("http://xyz.edu/teaching.xml")//tuple,
      $gr IN doc("file:///tmp/givengrades.xml)
                          //tuple[ProfId/@value=$tch/ProfId/@value]
                               /Grade/@value
  LET $grades :=
            doc("http://xyz.edu/transcript.xml")
                  //tuple[CrsCode/@value=$tch/CrsCode/@value
                           and Semester/@value=$tch/Semester/@value]
                        /Grade/@value
  WHERE  avg(numericGrade($grades)) < $gr
  RETURN
        <tuple>
            { $tch/CrsCode, $tch/Semester }
        </tuple>
}
</ClassList>
```

15.27 Use the document structure described in Exercise 15.23 to formulate the following queries in XQuery:

a. List all classes (identified by a course code and semester) where every student received a B or higher.

Solution:

```
DECLARE FUNCTION  classesFromTranscripts($elts AS element()*) AS element()*
{
    FOR  $e  IN  $elts
    RETURN
        <tuple>
        { $e//CrsCode, $e//Semester }
        </tuple>
}

<EasyClasses>
{
  LET $transcripts := doc("http://xyz.edu/transcript.xml")
  FOR $c IN
      distinct-values(classesFromTranscripts($transcripts//tuple))
  LET $grades :=
            doc("http://xyz.edu/transcript.xml")
                  //tuple[CrsCode/@value=$c/CrsCode/@value
                          and Semester/@value=$c/Semester/@value]
                       /Grade/@value
  WHERE  EVERY $g IN $grades
             SATISFIES numericGrade($g) >= numericGrade("B")
  RETURN $c
}
</EasyClasses>
```

b. List all students who never received less than a B.

Solution:

```
DECLARE FUNCTION  classesFromTranscripts($elts AS element()*) AS element()*
{
    FOR  $e  IN  $elts
    RETURN
        <tuple>
        { $e//CrsCode, $e//Semester }
        </tuple>
}

<GoodStudents>
{
  LET $transcripts := doc("http://xyz.edu/transcript.xml")
```

```
    FOR $sid IN
        distinct-values(classesFromTranscripts($transcripts//tuple)),
        $s IN doc("http://xyz.edu/student.xml")
                          //tuple[Id/@value=$sid/StudId/@value]
    LET $grades :=
        $transcripts//tuple[StudId/@value=$sid/StudId/@value]/Grade/@value
    WHERE  EVERY $g IN $grades
              SATISFIES numericGrade($g) >= numericGrade("B")
    RETURN $s
}
</GoodStudents>
```

15.28 Write an XQuery function that traverses a document and computes the maximum branching factor of the document tree, that is, the maximal number of children (text or element nodes) of any element in the document.

Solution:

```
DECLARE NAMESPACE  xs = "http://www.w3.org/2001/XMLSchema"
DECLARE FUNCTION maxBranching($n AS node()) AS xs:integer
   {
        LET $children := $n/node()
        LET $maxChildBranching :=
                        max{FOR $m IN $children RETURN maxBranching($m)}
        LET $childCount := count($children)
        RETURN
              IF $childCount >= $maxChildBranching
              THEN $childCount
              ELSE $maxChildBranching
    }
```

Recall that the XPath function `node()` matches every e- or t-node.

15.29 Write an XQuery function that traverses a document and strips element tags. For instance,

```
<the>best<foo>bar</foo>in the world</the>
```

would become

```
<result>bestbarin the world</result>
```

Solution:
Note that in this problem we need to distinguish between text nodes and element nodes. Since the exact notation for such tests has not been finalized by the XQuery

working group, we use a roundabout method, by counting the number of *e*- and *t*-children of a node. For text nodes and empty element nodes this count equals 0.

```
DECLARE NAMESPACE  xs = "http://www.w3.org/2001/XMLSchema"
DECLARE FUNCTION isTextNode($n AS node()) AS xs:boolean
   {
       LET $children := ./node()
       IF count($children) = 0
       THEN  -- it is a text node or an empty element
       RETURN
          {
            -- if descendant-or-self node set has a text node,
            -- then the current node must be a text node;
            -- else, it is an empty element
            IF count(.//text()) > 0 THEN  true ELSE  false
          {
       ELSE  -- it is a nonempty element
          RETURN false

   }
DECLARE FUNCTION stripTags($n AS node()) AS xs:integer
   {
        LET $children := ./node()
        RETURN
            IF isTextNode(.)
            THEN data(.)
            ELSE stripTags($children)
   }
```

15.30 Consider a document that contains a list of professors (name, Id, department Id) and a separate list of classes taught (professor Id, course code, semester). Use aggregate functions of XQuery to produce the following documents:

a. A list of professors (name, Id) with the number of different courses taught by each professor.

Solution:
Assume that the document consists of two elements, `Professors` and `Classes`, which in turn are lists of elements `Professor` and `Class`, respectively.

```
<ProfCourseCounts>
{
  FOR $p IN
      doc("http://xyz.edu/profs-courses.xml")//Professor
  LET $courses :=   -- All courses ever taught by $p
         distinct-values(
```

```
                doc("http://xyz.edu/profs-courses.xml")
                                    //Class[ProfId=$p/Id]/CrsCode
                    )
  RETURN $p <CourseCount>count($courses)</CourseCount>
}
</ProfCourseCounts>
```

b. A list of departments (department Id) along with the number of different courses ever taught by the professors in them. (The same course taught in different semesters or a course taught in the same semester by different professors is counted as one.)

Solution:

```
<DeptCourseCounts>
{
  LET $depts :=
     distinct-values(doc("http://xyz.edu/profs-courses.xml")//DeptId)
  FOR $d IN $depts
  LET $courses := {   -- All courses ever taught by $d
      FOR $p IN doc("http://xyz.edu/profs-courses.xml")
                                    //Professor[DeptId=$d/DeptId],
          $c IN doc("http://xyz.edu/profs-courses.xml")
                                    //Class[ProfId=$p/Id]/CrsCode
      RETURN  $c
                  }
  RETURN <DeptCrsCount>$d count(distinct-values($courses))</DeptCrsCount>
}
</DeptCourseCounts>
```

15.31 Consider the following query:

```
FOR      $c IN doc("http://xyz.edu/classes.xml")//Class,
         $t IN doc("http://xyz.edu/transcripts.xml")
              //CrsTaken
WHERE    $c/@CrsCode = $t/@CrsCode
AND      $c/@Semester = $t/@Semester
RETURN   $c/CrsName
```

Explain why every course name is likely to be output more than once. Can the use of the `distinct-values()` function (without any additional features) solve this problem with duplicates? Explain your answer.

Solution:
If a class has more than one student, there will be multiple transcript records for such

a class in the document `transcripts.xml`. Therefore, the RETURN clause will be executed for every such record separately.

The `distinct-values()` function applies to collections produced by XQuery expressions and eliminates duplicates from them. In our case, the only such collections occur in the FOR clause. Using

```
distinct-values(doc("http://xyz.edu/transcripts.xml")//CrsTaken)
```

as the range for `$t` will not completely eliminate duplicate transcript records corresponding to the same class: if the same class has students who received different grades, then these `CrsTaken` elements will be still considered to be distinct.

So, by itself, `distinct-values()` cannot always help solve the problem of duplicates.

15.32 Consider the relational schema in Figure 3.6, page 43. Assume that the contents of these relations are stored in a single XML column of a relation using the following XML format: the name of the relation is the top-level element, each tuple is represented as a `tuple` element, and each relation attribute is represented as an empty element that has a `value` attribute. For instance, the STUDENT relation would be represented as follows:

```
<Student>
    <tuple>
        <Id value="s111111111"/> <Name value="John Doe"/>
        <Address value="123 Main St."/> <Status value="U1"/>
    </tuple>
    .
    .
    .
</Student>
```

Formulate the following queries in SQL/XML:

a. Create the list of all professors who ever taught `MAT123`. The information must include all attributes available from the PROFESSOR relation.

Solution:

b. Create the list of all courses in which Joe Public received an A.

Solution:

c. Create the list of all students (include student Id and name) who took a course from John Smyth and received an A.

Solution:

15.33 Consider an SQL/XML database schema that consists of two relations:

- The SUPPLIER relation has the following attributes:
  - `Id`, an integer
  - `Name`, a string
  - `Address`, an XML type appropriate for addresses
  - `Parts`, an XML type that represents the parts supplied by the supplier. Each part has `Id`, `Name`, and `Price`.
- The PROJECT relation has the following attributes:
  - Project `Name`, a string
  - Project `Members`, an appropriate XML type
  - Project `Parts`, an XML type. This attribute represents the list of parts used by the project and supplied by a supplier. Each part has an `Id`, a `Name`, and a `SupplierId`.

Use SQL/XML to define the database schema. Make sure that a CHECK constraint validates the XML documents inserted into the database using an appropriate XML Schema document. Then answer the following queries:

a. Find all projects that are supplied by Acme Inc. and that have Joe Public as a member.

Solution:

b. Find all projects that are *not* supplied by Acme Inc., that is, none of the parts used by the project come from Acme Inc.

Solution:

c. Find all project members who participate in every project.

Solution:

d. Find the projects with the highest number of members.

Solution:

15.34 Consider the database depicted in Figure 3.5 on page 39. Use SQL/XML to answer the following queries:

a. Based on the relation TRANSCRIPT, output the same information in a different format. The output should have two attributes: `StudId` and `Courses`, where `Courses` should be of type XML; it is similar to the type used throughout this chapter for the `CourseTaken` element (e.g., as in Figure 15.4).

Solution:

b. Repeat the previous query, but use TEACHING instead. The output should have the attributes `ProfId` and `Courses`. The latter should be of type XML; it should describe the courses taught by the corresponding professor with the given `ProfId`.

Solution:

c. Produce a list of professors (`Id` and `Name`) along with the list of courses that that professor teaches in spring 2004. The list of courses should have the XML type and should include `CrsCode` and `DeptId` as attributes of an element and `CrsName` as a text node.

Solution:

d. For each course, produce a list of the professors who have ever taught it. The professor list should have the type XML. Choose your own schema.

Solution:

e. Repeat the previous query, but output only the courses that have been taught by the largest number of professors.

Solution:

16

Distributed Databases

EXERCISES

16.1 Discuss the advantages to the application designer of designing an application as a homogeneous system in which all databases are supplied by the same vendor.

Solution:
The vendor of a homogeneous supplies all the software and hardware to allow the components of the system to communicate and be integrated into a working system. The vendor will take the responsibility for fixing any problems, since it is clear that his components are causing the problem.

16.2 Explain why a table might be partitioned in the schema of a centralized system.

Solution:
If different applications or different transactions in the same application need to access different parts of a table, the table might be partitioned so that these parts are in different partitions. This might have the effect of speeding up the applications, since a transaction need not obtain any locks on the partitions it does not access.

16.3 Explain whether or not the following statement is true: the join of two tables obtained by a (vertical or horizontal) partitioning of a table, **T**, can never contain more tuples than are contained in **T**.

Solution:
By *join* in this exercise, we mean *natural join*.

1. For a vertically partitioned table, the natural join of all the partitions is exactly the original table. The natural join of some subset of the partitions yields some subset of the table.
2. For a horizonally partitioned table, the natural join of all or any subset of the partitions is an empty table, because each row of the original table is in exactly one partition.

16.4 Consider the two examples of query design in a multidatabase system given in Section 16.3.2. Write programs in Java and JDBC that implement both.

16.5 Give an example of a program at site A that requires the join of two tables, one at site B and one at site C. State the assumptions needed to justify the result that, as far

as communication costs are concerned, the best implementation is to ship the table at site B to site C, do the join there, and then send the result to site A.

Solution:
The three alternatives are

1. Send the table at site B to site C, do the join there, and then send the result to site A. The total communication cost is the number of bytes in the table in site B, plus the number of bytes in the join. ($bytes_B + bytes_J$)
2. Send the table at site C to site B, do the join there, and then send the result to site A. The total communication cost is the number of bytes in the table in site C, plus the number of bytes in the join. ($bytes_C + bytes_J$)
3. Send both tables to site A and do the join there. The total communication cost is the number of bytes in the table in site B, plus the number of bytes in the table in site C. ($bytes_B + bytes_C$)

The first alternative is better than the second if

$$bytes_B + bytes_J \leq bytes_C + bytes_J$$

and better than the third if

$$bytes_B + bytes_J \leq bytes_B + bytes_C$$

Simplifying the math

$$bytes_B \leq bytes_C$$

and

$$bytes_J \leq bytes_C$$

16.6 You are considering the possibility of horizontally partitioning the relation

EMPLOYEE (SSN, Name, Salary, Title, Location)

by location and storing each partition in the database at that location, with the possibility of replicating some partitions at different sites. Discuss the types of queries and updates (and their frequencies) that might influence your decision.

Solution:
If most of the queries and updates of that table are done at that location, partitioning the table might be appropriate. If there are queries at other locations, it might be appropriate to replicating the partition at other locations. However, if there are many updates, replication might not be appropriate because all replicas would have to be updated and that might increase communication costs.

16.7 Suppose that we have a relation

EMPLOYEE2 (SSnum, Name, Salary, Age, Title, Location)

which is partitioned as

EMP21 (SSnum, Name, Salary)
EMP22 (SSnum, Title, Age, Location)

where EMP21 is stored at site B and EMP22 is stored at site C. A query at site A wants the names of all managers in the accounting department whose salary is greater than their age. Design a plan for this query, using the assumptions on page 704 for table and attribute sizes. Assume that the items in the Age column are two bytes long.

Solution:

1. At site C performs a SELECT statement that returns the SSnum and Age of all employees for which Title = 'Manager'. Call this table T_1
2. Send T_1 to site B. Since there are 50 managers, the total amount of information that must be sent is $50 * (9 + 6)$ or 750 bytes
3. At site B, perform the join of T_1 with EMP21, select those names whose salary is greater than their age, and send the result to site A.

Since we do not know how many managers have a salary greater than their age we cannot estimate the communication costs for the third step.

16.8 Design a multidatabase query plan and a set of SQL statements that implement the query of the previous exercise.

Solution:

1. At site A, perform a SELECT statement on the database at site C that returns the SSnum and Age of all employees for which Title = 'Manager'. Since there are 50 managers, the total amount of information returned is $50 * (9 + 6)$ or 750 bytes
2. At site A, perform a SELECT on the database at site B that returns the entire EMP21 database. The total amount of information returned is $10000 * 30$ or 3 megabytes.
3. At site A, perform the join of the two tables from steps 1 and 2, and select those names whose salary is greater than their age.

16.9 Show that step 2 of the method used in Section 16.3.1 to perform a join using a semi-join does in fact generate the semi-join. For simplicity, assume that the join we are attempting is a natural join. That is, prove that

$$\pi_{attributes(\mathbf{T}_1)}\,(\mathbf{T}_1 \bowtie \mathbf{T}_2) = \pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \pi_{attributes(join\text{-}condition)}\,(\mathbf{T}_2))$$

Solution:
This a special case of the rule for pushing projections through joins in Section 11.2.

For every tuple $t \in \mathbf{T}_1 \bowtie \mathbf{T}_2$ there is a tuple $t' \in \mathbf{T}_1 \bowtie \pi_{attributes(join\text{-}condition)}\,(\mathbf{T}_2)$ that agrees with t on all attributes of $\mathbf{T}_1$ and vice versa. Thus the above two expressions are equivalent.

16.10 Show that step 3 of the method used in Section 16.3.1 to perform a join using a semi-join does in fact generate the join. For simplicity, assume that the join we are attempting is a natural join. In other words, show that

$$(\mathbf{T}_1 \ltimes \mathbf{T}_2) \bowtie \mathbf{T}_2 = (\mathbf{T}_1 \bowtie \mathbf{T}_2)$$

Solution:
Clearly, $\pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \mathbf{T}_2) \subseteq \mathbf{T}_1$, so $\pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \mathbf{T}_2) \bowtie \mathbf{T}_2 \subseteq (\mathbf{T}_1 \bowtie \mathbf{T}_2)$.

On the other hand, if a tuple, t, is in $\mathbf{T}_1 \bowtie \mathbf{T}_2$, it must be a join of a tuple $t_1 \in \mathbf{T}_1$ and $t_2 \in \mathbf{T}_2$. But then $t_1 \in \pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \mathbf{T}_2)$ and so t must be in $\pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \mathbf{T}_2) \bowtie \mathbf{T}_2$. Hence, the opposite inclusion also holds: $\pi_{attributes(\mathbf{T}_1)}(\mathbf{T}_1 \bowtie \mathbf{T}_2) \bowtie \mathbf{T}_2 \supseteq (\mathbf{T}_1 \bowtie \mathbf{T}_2)$.

16.11 Design a query plan for the join example in Section 16.3.1, assuming the same table sizes as in that section, but with the following differences:

a. An application at site B requested the join.

Solution:

1. If we send STUDENT to site C, compute the join there, and then send the result to site B, we have to send 540,000 bytes ($= 15{,}000 * 12 + 20{,}000 * 18$).

2. If we send TRANSCRIPT to site B and compute the join there, we have to send 300,000 bytes ($= 20{,}000 * 15$).

Thus alternative 2 is better. If we also allow the alternative of the semi-join described in that section, except that after the join is performed in step 3 of that protocol, the result is sent to site B, we have to send 465,000 bytes ($= 45{,}000 + 60{,}000 + 360{,}000$), as described in that section Thus alternative 2 is still the best.

b. An application at site C requested the join.

Solution:

1. If we send STUDENT to site C and compute the join there, we have to send 180,000 bytes ($= 15{,}000 * 12$).

2. If we send TRANSCRIPT to site B, compute the join there, and then send the result to site C, we have to send 660,000 bytes ($= 20{,}000 * 15 + 20{,}000 * 18$).

Thus alternative 1 is better. If we also allow the alternative of the semi-join described in that section, except that after the join is performed in step 3 of that protocol, the result is just kept at site C, we have to send just 105,000 bytes ($= 45{,}000 + 60{,}000$). The semi-join is thus the best alternative.

16.12 Show that the method of designing horizontal partitions described in Section 16.2.1 works as advertised.

Solution:
Since in that method every tuple of $\mathbf{T}$ satisfies some C_i, it follows that $\mathbf{T} \subseteq \cup \mathbf{T}_i$. Since, clearly, every $\mathbf{T}_i$ is a subset of $\mathbf{T}$, the other inclusion holds as well.

16.13 Show that the semi-join operation is not commutative, that is, $\mathbf{T}_1$ semi-joined with $\mathbf{T}_2$ is not the same as $\mathbf{T}_2$ semi-joined with $\mathbf{T}_1$.

Solution:
The semi-join is the projection over the columns of $\mathbf{T}_1$ of the join of $\mathbf{T}_1$ and $\mathbf{T}_2$:

$$\pi_{attributes(\mathbf{T}_1)} (\mathbf{T}_1 \bowtie_{join\text{-}condition} \mathbf{T}_2)$$

The projection over the attributes of $\mathbf{T}_1$ is different than the projection over the attributes of $\mathbf{T}_2$, unless $\mathbf{T}_1$ and $\mathbf{T}_2$ happen to share exactly the same attribute set.

16.14 Use the example schema (16.4) on page 703 to design a query for finding the names of all managers (employees with `Title = 'manager'`) whose salary is more than $20,000, but assume that there are three warehouses. Also assume that the total number of employees is 100,000, that 5000 of them make over $20,000, that the total number of managers is 50, and that 90% of the managers make more than $20,000.

Solution:
The best alternative is the one given in the text.

1. At each of the warehouse sites, select all tuples for which the title is manager.

2. Send these tables to the headquarters site. Do the join there, Then send the sames to site A

The number of bytes to be sent is exactly the same as in the text — 1,300 bytes from all the warehouse sites to the headquarters, plus 675 bytes from the headquarters site to site A, for a total of 1,975 bytes.

16.15 In Section 16.3.2 we pointed out that in a multidatabase system, data could not be communicated directly from one database site to another and that even indirect communication was difficult since a query site cannot send data to a database site. However, the following "end run" might be considered for computing a semi-join. The query site, A, executes a `SELECT` statement at the first database site, B, which returns the projection, **P**, used in computing the semi-join. Site A then uses the projection to dynamically construct another `SELECT` statement whose result set is the semi-join. Using Example 16.3.1 on page 699 of Section 16.3.1 involving tables STUDENT and TRANSCRIPT, give the two queries. Under what circumstances could such an approach be considered?

Solution:
The projection of TRANSCRIPT is returned by query

```
SELECT   T.StudId
FROM     TRANSCRIPT, T
```

If the result of this query is the set of values a, b, $\cdots$ z, then the following query submitted by A to C returns the semi-join

```
SELECT   S.Id, S.Major
```

```
FROM    STUDENT, S
WHERE   S.Id = 'a' OR S.Id = 'b' OR ··· OR S.Id = 'z'
```

Such an approach would only make sense if the first query produced a very small result set.

17

OLAP and Data Mining

EXERCISES

17.1 Is a typical fact table in BCNF? Explain.

Solution:
A fact table is usually in BCNF. Recall that the definition of BCNF is that the only non-trivial functional dependencies are key constraints. In a fact table, the key is the set of all dimensions, and the only FD is that the "fact" depends on all the dimensions.

17.2 Explain why, in an E-R model of a star schema, the fact table is a relationship and the dimension tables are entities.

Solution:
In the ER-Model, a relationship connects several entities together to reflect how they are related, which is just the purpose of the fact table.

17.3 Design another fact table and related dimension tables that a supermarket might want to use for an OLAP application.

Solution:
The fact table called RETURNED_ITEMS with attributes PRODUCT_ID, TIME_ID, MARKET_ID, and RETURNED_AMT. Dimension tables PRODUCT, TIME, and MARKET as in the text

17.4 Explain why it is not appropriate to model the database for the Student Registration System as a star schema.

Solution:
It would not efficiently support the OLTP transactions that are the main purpose of the system.

17.5 Design SQL queries for the supermarket example that will return the information needed to make a table similar to that of Figure 17.10, except that markets are aggregated by state, and time is aggregated by months.

a. Use CUBE or ROLLUP operators.

Solution:

```
SELECT   M.Region, T.month, SUM(S.Sales_Amt)
FROM     SALES S, MARKET M, TIME T
WHERE    S.Market_Id = M.Market_Id AND S.Time_Id = T.Time_Id
GROUP BY CUBE (M.Region, T.month)
```

b. Do not use CUBE or ROLLUP operators.

Solution:

```
SELECT   M.Region, T.month, SUM(S.Sales_Amt)
FROM     SALES S, MARKET M, TIME T
WHERE    S.Market_Id = M.Market_Id AND S.Time_Id = T.Time_Id
GROUP BY M.Region, T.month
```

```
SELECT   M.Region, SUM(S.Sales_Amt)
FROM     SALES S, MARKET M
WHERE    S.Market_Id = M.Market_Id
GROUP BY M.Region
```

```
SELECT   T.month, SUM(S.Sales_Amt)
FROM     SALES S, TIME T
WHERE    S.Time_Id = T.Time_Id
GROUP BY T.month
```

```
SELECT   SUM(S.Sales_Amt)
FROM     SALES S
```

c. Compute the result table.

Solution:

SUM(Sales_Amt)		Months			
		January	June	December	Total
Region	East	11633	11641	8647	31921
	West	16310	16314	16318	48942
	Total	27943	27955	24965	80863

17.6 a. Design a query for the supermarket example that will return the total sales (over time) for each supermarket.

Solution:

```
SELECT   S.Market_Id, SUM(S.Sales_Amt)
FROM     SALES S
GROUP BY S.Market_Id
```

b. Compute the result table.

Solution:

SUM(Sales_Amt)		Sales
Market_Id	M1	21012
	M2	10908
	M3	48942

17.7 Suppose that an application has four dimension tables, each of which contains 100 rows.

a. Determine the maximum number of rows in the fact table.

Solution:
The maximum number or rows in the fact table is $100 * 100 * 100 * 100 = 10^8$

b. Suppose that a one-dimension table has an attribute that can take on 10 values. Determine the size in bytes of a bit index on that attribute.

Solution:
The size of a bit index on that attribute is $100 * 10/8 = 125$ bytes.

c. Determine the maximum number of tuples in a join index for a join between one of the dimension tables and the fact table.

Solution:
The maximum size of a join index for a join between one of the dimension tables and the fact table is 10^8 rows. The reason for this is the following. Let DIMENSIONXYZ(`DimensionId,...`) be a dimension table and FACT(`DimXYZ,...`) be a fact table. Then `DimXYZ` (which represents the values of dimension XYZ) must be a foreign key referencing `DimensionId`, which is a key in DIMENSIONXYZ. Therefore, the number of tuples in the join (and therefore in the join index) is the same as in the fact table.

d. Suppose that we use the CUBE operator on this fact table to perform aggregations on all four dimensions. Determine the number of rows in the resulting table.

Solution:
The number of rows in the resulting table is

$$(100 * 100 * 100 * 100) + (4 * 100 * 100 * 100) + (4 * 3 * 100 * 100/2) +$$

$$(4 * 3 * 2 * 100/3 * 2) + 1 =$$
$$10^8 + 4 * 10^6 + 6 * 10^4 + 4 * 10^2 + 1$$

e. Suppose that we use the ROLLUP operator on this fact table. Determine the number of rows in the resulting table.

Solution:
The number of rows in the resulting table is

$$(100 * 100 * 100 * 100) + (100 * 100 * 100) + (100 * 100) + 100 + 1 =$$
$$10^8 + 10^6 + 10^4 + 10^2 + 1$$

17.8 Design a query evaluation algorithm for the ROLLUP operator. The objective of such an algorithm should be that the results of the previously computed aggregations are *reused* in subsequent aggregations and *not* recomputed from scratch.

Solution:

The idea is based on the fact that to compute

```
SELECT   A1, . . . , An aggr(B)
FROM     SomeRelation
WHERE     . . .
GROUP BY A1, . . . , An
```

where `aggr` is min, max, sum, or count, we can first compute the relation $\text{SomeRelation}_{1,\ldots,n,n+1}$

```
SELECT   A1, . . . , An, An+1, aggr(B)
FROM     SomeRelation
WHERE     . . .
GROUP BY A1, . . . , An, An+1
```

and then compute

```
SELECT   A1, . . . , An, aggr(B)
FROM     SomeRelation1,...,n,n+1
WHERE     . . .
GROUP BY A1, . . . , An
```

In this way, we are reusing the previous aggregation of `aggr(B)` over the attributes of SomeRelation minus $\mathtt{A}_1, \ldots, \mathtt{A}_n, \mathtt{A}_{n+1}, \mathtt{B}$ and now only need to aggregate over the attribute $\mathtt{A}_{n+1}$.

If `aggr` is avg, we need to compute sum and count using the above optimization and then divide one by the other.

We can thus start computing

```
GROUP BY ROLLUP (A1, . . . , An)
```

with an aggregation that groups by all the attributes, $\mathtt{A}_1, \ldots, \mathtt{A}_n$, then all but the last attribute, etc. Thus, the following aggregates will be computed, where each successive aggregation utilizes the result of the previous one:

```
GROUP BY A1, . . . , An
GROUP BY A1, . . . , An−1
 . . .  . . .  . . .
GROUP BY A1
```

The first GROUP BY above computes the relation $\textsc{SomeRelation}_{1,\ldots,n}$, the second GROUP BY uses $\textsc{SomeRelation}_{1,\ldots,n}$ to aggregate the values of B over the attribute $\mathtt{A}_n$, which produces the relation $\textsc{SomeRelation}_{1,\ldots,n-1}$. The third GROUP BY takes $\textsc{SomeRelation}_{1,\ldots,n-1}$ and aggregates the values of B over the attribute A_{n-1}, etc.

17.9 Design a query evaluation algorithm for the CUBE operator that uses the results of the previously computed aggregations to compute new aggregations. (*Hint*: Organize the GROUP BY clauses used in the computation of a data cube into a lattice, that is, a partial order with the least upper bound and the greatest lower bound for each pair of elements. Here is an example of such a partial order: GROUP BY A > GROUP BY A,B > GROUP BY A,B,C and GROUP BY B > GROUP BY B,C > GROUP BY A,B,C. Describe how aggregates computed for the lower parts of the lattice can be used in the computation of the upper parts.)

Solution:

The idea is basically the same as in the case of ROLLUP: to compute

```
SELECT   A1, . . . , An aggr(B)
FROM     SomeRelation
WHERE     . . .
GROUP BY A1, . . . , An
```

where aggr is min, max, sum, or count, we can first compute the relation $\textsc{SomeRelation}_{1,\ldots,n,n+1}$

```
SELECT   A1, . . . , An, An+1, aggr(B)
FROM     SomeRelation
WHERE     . . .
GROUP BY A1, . . . , An, An+1
```

and then compute

```
SELECT   A1, . . . , An, aggr(B)
```

```
FROM      SomeRelation_{1,...,n,n+1}
WHERE      . . .
GROUP BY A_1, . . . , A_n
```

In this way, we are reusing the previous aggregation of `aggr(B)` over the attributes of SomeRelation minus $A_1, \ldots, A_n, A_{n+1}$, B and now only need to aggregate the values of B in $\text{SomeRelation}_{1,\ldots,n,n+1}$ over the attribute A_{n+1}.

If `aggr` is avg, we need to compute sum and count using the above optimization and then divide one by the other.

We can thus start computing

```
GROUP BY CUBE (A_1, . . . , A_n)
```

with an aggregation that groups by all the attributes, $A_1, \ldots, A_n$, then move to the next layer in the lattice, where we aggregate over all attributes but one etc. For instance, the second level of aggregation will perform the following groupings:

```
GROUP BY A_2, A_3,  . . . , A_n
GROUP BY A_1, A_3, . . . , A_n
 . . .  . . .  . . .
GROUP BY A_1, A_2,  . . . , A_{n-1}
```

Thus, the second layer will have N aggregations. The first GROUP BY above aggregates B in $\text{SomeRelation}_{1,\ldots,n}$ over the attribute A_1, the second aggregates it over A_2, etc. The third layer will have $N(N-1)/2!$ aggregations, etc. The important point is that at layer 2 aggregation is done over the relation computed at layer 1, $\text{SomeRelation}_{1,\ldots,n}$, and at layer 3 we aggregate over the appropriate relations computed at layer 2. We continue in this way until we reach the top layer where we have N aggregations, each over a single attribute.

17.10 Suppose that the fact table of Figure 17.1 has been cubed and the result has been stored as a view, Sales_v1. Design queries against Sales_v1 that will return the tables of Figure 17.10 and Figure 17.7.

Solution:

```
SELECT   S.Market_Id, S.Product_Id, S.Sales_Amt
FROM     Sales_v1 S
WHERE    S.Time_Id = NULL
```

17.11 We are interested in building an OLAP application with which we can analyze the grading at our university, where grades are represented as integers from 0 to 4 (4 representing an A). We want to ask questions about average grades for different courses, professors, and departments during different semesters and years. Design a star schema for this application.

Solution:
The center of the star (the relationship in the E-R diagram) is `Grades`. The dimension tables might be `Course`, `Professor`, `Department`, and `Time`.

17.12 Discuss the difference in storage requirements for a data cube implemented as a multidimensional array and a fact table.

Solution:
If the data cube has m dimensions, and the attribute in the j^{th} dimension can take on n_j values (e.g., there are 100 markets in the market dimension), the complete description of that cube in a multidimensional array or a fact table would take $n_1 * n_2 * \ldots * n_m$ storage locations. However, if some items are missing (e.g. Market 10 had no sales of Product 4 in the time period T_4), that row can be omitted from the fact table, but the space has already been allocated in the array. Thus in practice the fact table is considerably smaller than the multidimensional array.

17.13 Give examples, different from those in the text, of syntactic and semantic transformations that might have to be made while loading data into a data warehouse.

Solution:
Syntactic transformation: one database schema use one string to representing a student's name, while another might use two strings (last name and first name).

Semantic transformation: One schema might store grades as A, B, C, D, and F, while another might allow + and - grades, such as B+ or C-.

17.14 Perform the a priori algorithm on the table of Figure 17.15 to determine all reasonable two-item associations.

Solution:
All reasonable two-item associations are:

$$Purchase\_diapers \rightarrow Purchase\_beer$$

$$Purchase\_beer \rightarrow Purchase\_diapers$$

17.15 Show that, when evaluating possible associations, the confidence is always larger than the support.

Solution:
Let the association be

$$a \rightarrow b$$

Let n be the total number of transactions, n_a be the number that contain a, and n_{ab} the number that contains both a and b. Then the support is n_{ab}/n and the confidence is n_{ab}/n_a. Since n is larger than n_a, the confidence is larger than the support

17.16 Apply the gain ratio measure to the table in Figure 17.16 to design a decision tree.

Solution:
The final tree is the same.

17.17 Apply the Gini measure to the table in Figure 17.16 to design a decision tree.

Solution:
The final tree is the same.

17.18 Show how the K-mean clustering algorithm would have worked on the table of Figure 17.28 if the original choice for cluster centers had been 17 and 18.

Solution:
The final cluster would have been the same.

17.19 a. Give an example of a set of three items on a straight line for which the K-mean algorithm, with $k = 2$, would give a different answer for the two clusters depending on the choice of items for the initial clusters.

Solution:
One choice for the three items is at distances of 1, 1.9, and 3. If the initial choice of items is 1 and 1.9, the final clusters are

1 1.9,3

and if the initial choice is 1 and 3, the final clusters are

1,1.9 3

b. Which of these final clusters would have been obtained by the hierarchical algorithm?

Solution:

1,1.9 3

17.20 Suppose the table of Figure 17.16 is stored in a relational database. Use SQL to compute the probabilities needed to compute the information gain when using the `PrevDefault` attribute as the topmost attribute of a decision tree based on that table.

Solution:

```
SELECT   C.PreviousDefault,
         COUNT(DefYes.Count)/COUNT(*),
         COUNT(DefNo.Count)/COUNT(*)
FROM     CUSTOMER C,
         (SELECT DISTINCT C1.PrevDefault COUNT(*) AS Count
          FROM CUSTOMER C1
          WHERE C1.Default = 'yes'
          GROUP BY C1.PrevDefault) AS DefYes
         (SELECT DISTINCT C1.PrevDefault COUNT(*) AS Count
          FROM CUSTOMER C1
```

```
          WHERE C1.Default = 'no'
          GROUP BY C1.PrevDefault) AS DefNo
WHERE    C.PrevDefault = DefYes.PrevDefault
         AND  C.PrevDefault = DefNo.PrevDefault
GROUP BY C.PrevDefault
```

PART FIVE

Transaction Processing

18

ACID Properties of Transactions

EXERCISES

18.1 Some distributed transaction processing systems replicate data at two or more sites separated geographically. A common technique for organizing replicated systems is one in which transactions that update a data item must change all replicas. Hence, one possible integrity constraint in a replicated system is that the values of all of the replicas of an item be the same. Explain how transactions running on such a system might violate

a. Atomicity

Solution:
A transaction updates one replica and then crashes before it can update the others, but the update to the first replica is not rolled back.

b. Consistency

Solution:
A transaction updates one replica, but not others, thus violating the integrity constraint that all replicas have the same value.

c. Isolation

Solution:
One transaction updates two different items, each of which is replicated; Another transaction sees one replica before change, the other after.

d. Durability

Solution:
Two replicas are updated; one site crashes and forgets the update

18.2 Consider the replicated system described in the previous problem.

a. What is the impact of replication on the performance of a read-only transaction?

Solution:
Read-only transactions execute faster since they can read the closest replica.

b. What is the impact of replication on the performance of a transaction that both reads and writes the data?

Solution:
Read/write transactions execute slower since they have to update replicas at several sites.

c. What is the impact of replication on the communication system?

Solution:
Replication implies an increase in communication to update replicas, but a decrease in communication to read the value of a replica.

18.3 Give three examples of transactions, other than an ATM bank withdrawal, in which a real-world event occurs if and only if the transaction commits.

Solution:

1. Students register for courses online and a confirmation of the registration is printed if and only if the registration transaction commits
2. A pay per view movie is ordered for a TV, and can be seen if and only if the ordering transaction commits.
3. Software is ordered over the Internet and is downloaded if and only if the purchasing transaction commits.

18.4 The schema of the Student Registration System includes the number of current registrants and a list of registered students in each course.

a. What integrity constraint relates this data?

Solution:
The number of students on the list equals the number of current registrants

b. How might a registration transaction that is not atomic violate this constraint?

Solution:
It might increment the number but crash before it appends to the list, and the increment might not be rolled back

c. Suppose the system also implements a transaction that displays the current information about a course. Give a nonisolated schedule in which the transaction displays inconsistent information.

Solution:
The display transaction takes a snapshot between the increment and the append

18.5 Give three examples of applications in which certain transactions need not be totally isolated and, as a result, might return data which, although not the result of a serializable schedule, is adequate for the needs of the applications.

Solution:

1. Deciding when to reorder merchandise in a store. A snapshot of the exact number of each item in the current inventory might not be needed.
2. Printing out a mailing list of customers. The list printed out need not be a snapshot of the customers at some particular instant, but might include some people who are no longer customers or might not include some people who just became customers.

3. When ordering tickets to a concert, the system might use two transactions: one to display the available seats, and a second, executed later, to purchase tickets and reserve seats. Each transaction is atomic, but the effect of executing both transactions is not atomic — a seat might be reserved by a concurrent transaction between the execution of the display and purchase transactions.

18.6 Describe a situation in which the execution of a program under your local operating system is not

a. Atomic

Solution:
The system crashes while executing a program. The partial changes made by the program are not rolled back.

b. Isolated

Solution:
Two concurrently executing programs write to the same file

c. Durable

Solution:
The disk crashes in a thunderstorm and all the data stored on it is lost. The last disk backup was made three days ago.

18.7 At exactly noon on a particular day, 100 people at 100 different ATM terminals attempt to withdraw cash from their bank accounts at the same bank. Suppose their transactions are run sequentially and each transaction takes .25 seconds of compute and I/O time. Estimate how long it takes to execute all 100 transactions and what the average response time is for all 100 customers.

Solution:
The average response time
$= (0.25 + 0.50 + 0.75 + \cdots + 25.00)/100$
$= 12.825$ (seconds)

18.8 You have the choice of running a single transaction to transfer $300 from one of your savings accounts to another savings account at the same bank or of running two transactions, one to withdraw $300 from one account and a second to deposit $300 in the other. In the first choice the transfer is made atomically; in the second it is not. Describe a scenario in which after the transfer, the sum of the balances in the two accounts is different (from what it was when both transactions started) at the instant the transfer is complete. *Hint:* Other transactions might be executing at the same time that you are doing the funds transfer.

Solution:
In the second choice an interest crediting transaction might execute between the two transactions, and no interest would be credited on the money being transferred. In the first choice, the interest crediting transaction would execute either before or after the transfer, and interest would be credited on the money transferred.

18.9 A distributed transaction consists of subtransactions that execute at different sites and access local DBMSs at those sites. For example, a distributed transaction that

transfers money from a bank account at site A to a bank account at site B executes a subtransaction at A that does the withdrawal and then a subtransaction at B that does the deposit.

Such a distributed transaction must satisfy the ACID properties in a global sense: it must be globally atomic, isolated, consistent, and durable. The issue is, if the subtransactions at each site individually satisfy the ACID properties, does the distributed transaction necessarily satisfy the ACID properties? Certainly if the subtransactions at each site are individually durable, the distributed transaction is durable. Give examples of situations in which

a. The subtransactions at each site are atomic, but the distributed transaction is not atomic.

Solution:
The subtransaction at one site commits and the subtransaction at a different site aborts. In the above example, the subtransaction to withdraw money commits and the one to deposit the money aborts—the money is lost

b. The subtransactions at each site are consistent, but the distributed transaction is not consistent.

Solution:
A distributed database might have a **global integrity constraint** that two items in databases at different sites have the same value (replicated data). The subtransactions of a distributed transaction might maintain all the local integrity constraints at their local database, but not maintain the global constraint that the values of these two items are equal.

c. The subtransactions at each site are isolated, but the distributed transaction is not isolated.

Solution:
The DBMS at one site serializes its subtransactions in one order, and the DBMS at a differents sites serializes its subtransactions in a different order. (The two sites cannot agree on a serialization order.) In the above example of transferring funds betweens accounts, a concurrent transaction to add the monthly interest to all accounts might have a subtransaction that serializes after the withdraw subtransaction at its site and a subtransaction that serializes before the deposit subtransaction at its site—those funds do not get interest that month.

18.10 Isolation is a sufficient but not necessary condition to achieve correctness. Consider an application that reserves seats for a concert. Each reservation transaction (1) reads the list of seats that have not yet been reserved, (2) presents them to a potential customer who selects one of them, and (3) marks the selected seat as reserved. An integrity constraint asserts that the same seat cannot be reserved by two different customers. Describe a situation in which two such transactions that reserve two different seats execute in a nonisolated fashion but are nevertheless correct.

Solution:
Assume there are two unreserved seats, S_a and S_b. T_1 reads that both are unreserved and while its customer is thinking which seat she wants to reserve, T_2 also reads that S_a and S_b are unreserved. Then the customer at T_1 reserves S_a and the customer at T_2 reserves S_b. That is a perfectly correct execution, but T_1 and T_2 cannot be serialized

in either order because neither saw the results of the other's execution. (The system must guarantee that if both customers had tried to reserve the same seat, only one would have been allowed to.)

18.11 Assume a schedule consists of consistent transactions that execute in an isolated fashion except for one transaction that performs a single update that is lost (as in Figure 18.2). Show that the final state of the database satisfies all integrity constraints but nevertheless is incorrect.

Solution:
The values in the final database are the same as if T had never executed. Since all the other transactions are consistent and executed in an isolated fashion, the final database satisfies all integrity constraints. The database might nevertheless be incorrect because the update that was lost might be important for the application, for example as in the schedule of Figure 18.2.

19

Models of Transactions

EXERCISES

19.1 The banking system described at the end of Section 22.1 can be structured either as a single transaction (with a savepoint after posting interest to each group of 1000 accounts) or as a chained transaction (with a commit point after posting interest to each group of 1000 accounts). Explain the differences in semantics between the two implementations. State when the printing of account statements takes place in each.

Solution:
For the savepoint implementation, printing takes place at the end. Either all account statements are printed or none. For the chain implementation, printing can take place at the end of each subtransaction in the chain.

If system crashes during execution: in the savepoint implementation, all work will be lost. In the chain implementation some account statements will have been printed and the system can continue where it left off.

19.2 Explain the difference between each of the transaction models with respect to abort, commit, rollback, and isolation in the following cases:

a. A sequence of savepoints in a transaction and a chained transaction

Solution:
The transaction is durable at end of the chain and not at a savepoint.

b. A sequence of savepoints in a transaction and a nested transaction consisting of subtransactions that are executed serially

Solution:
Rolling back to a savepoint is similar to a subtransaction of a nested transaction aborting, but if the savepoint transaction aborts, the entire transaction aborts.

c. A sequence of savepoints in a transaction and a sequence of transactions linked by a recoverable queue

Solution:
The sequence of recoverable queue transactions is not atomic, the savepoint transaction is atomic.

d. A sequence of chained transactions and a nested transaction consisting of subtransactions that are scheduled serially

Solution:
The chained transactions are durable at chain points, but individual nested subtransactions are not durable.

e. A sequence of chained transactions and a sequence of transactions linked by a recoverable queue

Solution:
They are very similar, except that the recoverable queue transactions might take place at different times.

f. A sequence of transactions linked by a recoverable queue and a nested transaction consisting of subtransactions that are executed serially

Solution:
The recoverable queue transactions are not serializable and are durable at the end of each transactions. The nested transactions are serializable and not durable until the the top-level transaction commits.

g. A nested transaction consisting of a set of concurrently executing siblings and a set of concurrently executing peer-related subtransactions of a distributed transaction

Solution:
In the nested transactions, a sibling can abort without the entire transaction aborting. In the peer related transactions, individual transactions can communicate with each other and any individual transaction can start the commit procedure.

h. A nested transaction consisting of subtransactions that execute serially and a multilevel transaction

Solution:
The ACID properties are the same for both, but the multilevel transaction executes more efficiently because it can give up its locks earlier. Also a subtransaction of a nexted transaction can abort without aborting the entire transaction, while a multilevel transaction does not have this property.

i. A nested transaction consisting of subtransactions that execute serially and a transaction using declarative demarcation consisting of modules specified with *RequiresNew*

Solution:
In the nested transaction, the subtransactions can separately abort, but the overall transaction is ACID. In the declarative demarcation example, each module is separately ACID and the top-most transaction (with the modules deleted) is ACID.

19.3 Decompose the registration transaction in the Student Registration System design (given in Section C.7) into a concurrent nested transaction.

Solution:
Checking each the of the conditions and the actual registration could be subtransactions

19.4 Redesign the registration transaction in the Student Registration System design (given in Section C.7) as a multilevel transaction.

Solution:
The levels could be the original transaction; each of the checks and the final registration; the SQL statements within the checks and the registration; and the page reads and writes.

19.5 Show how the withdraw transaction in an ATM system can be structured as a nested transaction with concurrent subtransactions.

Solution:
Checking the pin number and the actual withdraw are subtransactions.

19.6 a. Explain the difference in semantics between the two versions of chaining discussed in the text.

Solution:
In the first version, locks are given up after each chained transaction commits. In the second version, locks are retained. Therefore, in the first version each subtransaction in the chain might see a different database state than the one that existed when the subtransaction before it committed.

b. Which of these versions is implemented within SQL?

Solution:
The first version.

c. Give an example of an application where the second version would be preferable.

Solution:
The first transaction checks whether there are seats on the plane, and the second reserves a seat. If the first version were used, someone might reserve the seat between the two transactions.

19.7 Explain how the Student Registration System could interface with a student billing system using a recoverable queue.

Solution:
At the end of the registration transaction the billing transaction would be queued.

19.8 Give three examples of applications in which isolation is not required and a recoverable queue could be used.

Solution:
Registration and Billing; Telephone Ordering and Shipping the Order; Airplane Seat Assignment and Meal Ordering

19.9 Explain the difficulties in implementing a print operation in a transaction without the use of a recoverable queue. Assume that the transaction does not mind waiting until printing is complete before committing.

Solution:
If the program first printed and then committed and the system crashed between the printing and the committing, the database would be rolled back and the printing would

not reflect the state of the database. If the program first committed and then printed and the system crashed between the committing and printing, the printing would not occur.

19.10 Consider the real-world transaction for dispensing cash discussed in Section 19.3.5. For each of the critical times—before, during, and after the execution of the real-world transaction—in which the system might crash, describe how the system (after it recovers from the crash) determines whether or not the cash has been dispensed. Discuss some ways in which the cash-dispensing mechanism itself might fail in such a way that the system cannot tell whether or not the cash has been dispensed.

Solution:
If the system crashes before the real-world transaction (on the queue) commits, and before the cash is dispensed, the transaction is restarted and checks the value stored in the database and in the physical counter. They will be the same and hence the transaction is reexecuted and the cash dispensed.

If the system crashes before the real-world transaction (on the queue) commits, and after the cash is dispensed, the transaction is restarted and checks the value stored in the database and in the physical counter. They will be different, so the transaction does not dispense the cash.

The protocol will not work correctly if the counter stops working or works incorrectly.

19.11 Explain in what ways the execution of each individual SQL statement in a transaction is like a nested subtransaction.

Solution:
It is atomic, it inherits locks from its parent, it can independently abort.

19.12 Show how the credit card validation transaction described in the first paragraph of Chapter 1 can be structured as a distributed transaction.

Solution:
The credit card company and bank databases could be at different sites. The validation transaction initiates subtransactions at each site

.

19.13 The Student Registration System is to be integrated with an existing student billing system (which also bills for meal plans, dorm rooms, etc.). The databases for the two systems reside on different servers, so the integrated system is distributed. Using your imagination,

a. Give examples of two global integrity constraints that might exist for the global database of this system.

Solution:
IC1: The amount the student is billed for corresponds to the courses she is registered for.

IC2: A student is allowed in the dorm only if she is registered for at least 1 course.

b. Give examples of two transactions that access both databases.

Solution:
The dorm registration transaction (to check constraint); the billing transaction.

19.14 Describe the process that the admissions office of your university uses to admit new students as a workflow. Decompose the process into tasks, and describe the task interaction using a diagram similar to Figure 19.9.

Solution:
Advertise university; send applications; receive applications; evaluate applications, read SAT; send admission letters; receive acceptance letters; enter accepted students into student database.

19.15 Consider a transaction that transfers funds between two bank accounts. It can be structured into two subtransactions: one to debit the first account and the second to credit the second account. Describe how this can be done in (a) the hierarchical model and (b) the peer model.

Solution:
In the hierarchical model the parent initiates a debit subtransaction and if it completes successfully, initiates a credit subtransaction. When the latter completes the parent calls commit. In the peer model the two subtransactions can be initiated concurrently. If they both complete successfully, the parent commits causing the subtransactions to commit. If not, the parent aborts, causing the subtransactions to abort. Alternatively, the debit subtransaction could communicate directly to the credit subtransaction, informing it of whether or not it was successful, and the credit transaction could could make the deposit appropriately.

19.16 Explain how nested transactions can be implemented using savepoints and procedure calls. Assume that the children of each subtransaction do not run concurrently.

Solution:
A savepoint is declared immediately prior to calling a procedure, and its Id is passed as an argument to the procedure. To abort, the procedure rolls back to that savepoint. The only rollbacks to that savepoint are in the procedure's body. After executing the rollback the procedure returns, with a failure indication as an out argument.

19.17 In an application that uses declarative demarcation, one procedure that is called frequently from other procedures is a *logging* procedure that writes appropriate information about the state of the system into a log. What transaction attribute should be used for that procedure?

Solution:
RequiresNew because we want the transaction corresponding to that procedure to commit even if the transaction that called it should abort.

Another possible answer is *NotSupported*, because we want the logging to take place even if the logging procedure is called from a transaction that aborts. However, in this case the logging would not itself be transactional, which might be desirable for many applications.

20

Implementing Isolation

EXERCISES

20.1 State which of the following schedules are serializable.

a. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_2(x)\ r_1(y)$

Solution:
Yes. all reads

b. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_2(x)\ r_1(y)$

Solution:
No. T_1 before T_2 on x and after T_2 on y

c. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ r_2(y)$

Solution:
Yes

d. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ w_2(y)$

Solution:
Yes

e. $r_1(x)\ r_2(y)\ w_2(x)\ w_3(x)\ w_3(y)\ r_1(y)$

Solution:
No. T_1 before T_2 and T_2 before T_3 on x; T_2 before T_3 and T_3 before T_1 on y.

f. $w_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_1(x)\ w_2(y)$

Solution:
Yes

g. $r_1(z)\ w_2(x)\ r_2(z)\ r_2(y)\ w_1(x)\ w_3(z)\ w_1(y)\ r_3(x)$

Solution:
Yes

20.2 Give all possible conflict-equivalent serial orderings corresponding to the serialization graph in Figure 20.5.

Solution:
For the graph of Figure 20.5 (a)

$T_1\ T_2\ T_3\ T_4\ T_5\ T_6\ T_7$

$T_1\ T_2\ T_3\ T_5\ T_4\ T_6\ T_7$

$T_1\ T_2\ T_3\ T_5\ T_6\ T_4\ T_7$

$T_1\ T_2\ T_3\ T_5\ T_6\ T_7\ T_4$

$T_1\ T_3\ T_2\ T_4\ T_5\ T_6\ T_7$

$T_1\ T_3\ T_2\ T_5\ T_4\ T_6\ T_7$

$T_1\ T_3\ T_2\ T_5\ T_6\ T_4\ T_7$

$T_1\ T_3\ T_2\ T_5\ T_6\ T_7\ T_4$

$T_1\ T_3\ T_5\ T_2\ T_4\ T_6\ T_7$

$T_1\ T_3\ T_5\ T_2\ T_6\ T_4\ T_7$

$T_1\ T_3\ T_5\ T_2\ T_6\ T_7\ T_4$

20.3 Use a serialization graph to demonstrate that the schedule shown in Figure 20.4 is not conflict serializable.

Solution:
The serialization graph has an edge

$$T_1 \rightarrow T_2$$

because T_1 wrote x before T_2 wrote it, and it has an edge

$$T_2 \rightarrow T_1$$

because T_2 read y before T_1 wrote it. Thus the graph has a cycle and the schedule is not conflict serializable.

20.4 Suppose that we declare all of the database integrity constraints in the database schema so that the DBMS will not allow any transaction to commit if its updates violate any of the integrity constraints. Then, even if we do not use any concurrency control, the database always remains consistent. Explain why we must nevertheless use a concurrency control.

Solution:
We must use a concurrency control to maintain correctness. For example, the following schedule of two transactions preserves database consistency (the database satisfies its

integrity constraints), but nevertheless the final database does not reflect the effect of both transactions. Two interleaved deposit transactions of $10:

$$r_1(bal)\ r_2(bal)\ w_1(bal = bal + \$10)\ w_2(bal = bal + \$10)$$

The integrity constraint is $bal \geq 0$. When the schedule completes, the integrity constraint is satisfied, but the database is incorrect – T_1's update has been lost.

Alternatively, a schedule in which read/write transactions are serializable (and hence all integrity constraints would be preserved), but read-only transactions are not serializable with respect to read/write transactions would be allowed.

20.5 Give an example of a schedule of two transactions that preserves database consistency (the database satisfies its integrity constraints), but nevertheless yields a final database that does not reflect the effect of both transactions.

Solution:
Two interleaved deposit transactions of $10:

$$r_1(bal)\ r_2(bal)\ w_1(bal = bal + \$10)\ w_2(bal = bal + \$10)$$

The integrity constraint is $bal \geq 0$, and the update made by transaction T_1 is lost.

20.6 Prove that conflict equivalence implies view equivalence.

Solution:
Since conflicting operations are ordered the same in both schedules, corresponding reads will return the same values. Since the reads are the same, the writes will be the same, and since the writes are in the same order, the final database state will be the same

20.7 Give an example of a schedule in which transactions of the Student Registration System deadlock.

Solution:
Consider the concurrent execution of a registration transaction, T_1, and a deregistration transaction, T_2, for the same course, and assume the DBMS uses table locking (it locks the table that is being accessed):

T_2 first deletes the row in TRANSCRIPT describing the student registered in that course and then decrements `Enrollment` in CLASS. The registration transaction, T_1, first updates CLASS, and then inserts a tuple in TRANSCRIPT.

The schedule is:

$$w_2(x)\ w_1(y)\ w_2(y)\ w_1(x)$$

A deadlock results since T_1 and T_2 obtain their locks in the opposite order.

20.8 Prove that with a two-phase locking protocol, one possible equivalent serial order is the order in which the transactions perform their first unlock operation.

Solution:
If two transactions do not acquire conflicting locks in some schedule S, all their operations commute and they can be ordered arbitrarily in an equivalent serial schedule.

Suppose they acquire conflicting locks on item x, and T_1 accesses x before T_2. Then T_1's first unlock in S must precede T_2's lock on x, which must precede T_2's first unlock. If they also acquire conflicting locks on y, their accesses to y must be in the same order as their accesses to x since otherwise we could conclude that T_2's first unlock in S must precede T_1's first unlock – a contradiction. Hence, all conflicting operations in T_1 and T_2 are ordered in the same way as their first unlock events and the two transactions can be serialized in that order.

20.9 Give an example of a transaction processing system (other than a banking system) that you have interacted with, for which you had an intuitive expectation that the serial order was the commit order.

Solution:
Airlines reservation system; credit card system. You make a reservation and check a few minute later and the system says you do now have a reservation.

20.10 Give an example of a schedule that is serializable but not strict.

Solution:

$$w_1(x)\ r_2(x)\ w_2(z)\ r_1(y)$$

T_1 is serializable before T_2, but T_2 read what T_1 wrote before T_1 committed. If T_2 were to commit and then T_1 were to abort, the schedule would not be recoverable.

20.11 Give an example of a schedule that is strict but not serializable.

Solution:

$$r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)$$

or

$$r_1(x)\ w_2(x)\ w_2(y)\ commit_2\ r_1(y)$$

20.12 Give an example of a schedule produced by a nonstrict two-phase locking concurrency control that is not recoverable.

Solution:

$$lock_1(x) w_1(x) unlock_1(x)\ lock_2(x) r_2(x)\ lock_2(y) w_2(y) unlock_2(x, y)\ commit_2 abort_1$$

20.13 Give an example of a schedule produced by a recoverable but nonstrict concurrency control involving three transactions in which a deadlock occurs, causing a cascaded abort of all three.

Solution:

$$w_1(x)\ r_2(x)\ w_2(y)\ w_2(x)\ w_3(z)\ r_3(y)\ w_3(y)\ w_1(z)\ abort_1$$

Transaction 2 is waiting for transaction 1 (because of x); 3 is waiting for 2 (because of y), and 1 is waiting for 3 (because of z). So 1 is aborted (because for example, 1

wrote x and 2 requested to write x). But, 2 has read what 1 wrote and hence must be aborted when 1 is aborted. Transaction 3 has read what 2 wrote and hence must be aborted when 2 is aborted.

20.14 Give an example of a schedule produced by a nonstrict two-phase locking concurrency control that is serializable but not in commit order.

Solution:

$$lock_1(x)\ w_1(x)\ unlock_1(x)\ lock_2(x)\ r_2(x)\ unlock_2(x)\ commit_2\ commit_1$$

Transaction T_2 is after transaction T_1.

20.15 Suppose that the `account` object, the conflict table for which is described in Figure 20.18, has an additional operation, `balance`, which returns the balance in the account. Design a new conflict table for this object, including the new operation.

Solution:

	Granted Mode			
Requested Mode	`deposit()`	`withdrawOK()`	`withdrawNO()`	`balance`
`deposit()`			X	X
`withdrawOK()`	X			X
`withdrawNO()`		X		
`balance`	X	X		

20.16 Consider the following schedule of three transactions:

$$r_2(y)\ r_1(x)\ r_3(y)\ r_2(x)\ w_2(y)\ w_1(x)\ r_3(x)$$

a. Define a serialization between T_1 and T_2.

Solution:

$T_2,\ T_1$

b. In what apparent order does T_3 see the database?

Solution:

$T_1,\ T_2$

c. Is the schedule serializable?

Solution:
No

d. Assuming that each transaction is consistent, does the final database state satisfy all integrity constraints?

Solution:
Yes

e. Does the database state seen by T_3 satisfy all integrity constraints? Explain.

Solution:
Yes. T_3 is a read-only transaction. The read/write transactions, T_1 and T_2, are serializable and hence the final database satisfies all integrity constraints.

20.17

a. Assume that, in addition to the operations `read(x)` and `write(x)`, a database has the operation `copy(x,y)`, which (atomically) copies the value stored in record x into record y. Design a conflict table for these operations for use in an immediate-update pessimistic concurrency control.

Solution:
The conflict table is for operations on a single item. Therefore we must distinguish the cases where the copy is *from* the item described by the table and when the copy is *to* that item. Copy-from is like a read, and copy-to is like a write.

	Granted Mode			
Requested Mode	`read`	`write`	`copy-from`	`copy-to`
`read`		X		X
`write`	X	X	X	X
`copy-from`	X	X		X
`copy-to`	X	X	X	X

b. Assume that, in addition to the operations `read(x)` and `write(x)`, a database has the operation `increment(x,C)`, which (atomically) increments the value stored in record x by the constant C, which might be positive or negative. Design a conflict table for these operations for use in an immediate-update pessimistic concurrency control.

Solution:

	Granted Mode		
Requested Mode	`read`	`write`	`increment`
`read`		X	X
`write`	X	X	X
`increment`	X	X	

20.18 Suppose that we have a `queue` object that implements an FCFS (first-come-first-served) discipline, with operations `enqueue` and `dequeue`. `enqueue` always succeeds and `dequeue` returns `NO` if the queue is empty and `OK` otherwise. Design a conflict table for this object using partial operations and backward commutativity.

Solution:

	Granted Mode		
Requested Mode	`enqueue()`	`dequeueOK()`	`dequeueNO()`
`enqueue()`	X		X

`dequeueOK()`	X	X	X
`dequeueNO()`	X	X	

20.19 A pair of (partial) database operations, p and q, are said to **forward-commute** if, in every database state in which both p and q are defined, the sequences p, q and q, p are both defined and, in both sequences, p and q return the same values and the final database state is the same.

a. Describe how forward commutativity can be used in the design of a deferred-update pessimistic concurrency control.

Solution:
Does-not-forward-commute-with is used as the conflict relation in the conflict table. Because the transactions might commit in any order, each transaction's operations must be defined in the database state that exists when that transaction starts. Then, again since the transactions might commit in any order, the operations must commute.

b. Give a conflict table for such a control for the `account` object described in Figure 20.18.

Solution:

	Granted Mode		
Requested Mode	`deposit()`	`withdrawOK()`	`withdrawNO()`
`deposit()`			
`withdrawOK()`		X	
`withdrawNO()`			

Note that the only conflict that occurs is for two `withdrawOK()` transactions because there might not be enough money in the account to satisfy both withdrawals.

20.20 In Section 20.7 we described a policy for a concurrency control that dealt with both forward and compensating operations and guaranteed that schedules were reducible. Generalize this policy to include undo operations using the conditions on the commutativity described in that section, and show that your generalization preserves reducibility.

Solution:
to be added

20.21 Design a deferred-update pessimistic concurrency control.

Solution:

	Granted Mode	
Requested Mode	`read`	`write`
`read`		X
`write`	X	

Even though writes are deferred, a transaction must obtain a write lock when a write statement is executed (before the new value is written to the intentions list). There is no conflict between two write operations because the new values will be stored in the database at commit time in commit order.

20.22 Consider an implementation of chained transactions in which a subtransaction, when it commits, only releases locks on items that will not be accessed by subsequent subtransactions. Explain how this can affect the ACID properties of the overall chained transaction.

Solution:
The transaction might not be isolated. Some other transaction might see the intermediate results of the transaction.

20.23 Give the conflict table for each level of the multilevel control described in Section 20.8.5. Each table indicates the conflicts for the operations used to implement the transaction `Move(sec1, sec2)`. Assume that the operation `TestInc` is viewed as two partial operations `TestIncOK` and `TestIncNO`.

Solution:
At L_0:

	Granted Mode	
Requested Mode	`read`	`write`
`read`		X
`write`	X	X

At L_1:

	Granted Mode	
Requested Mode	`Upd`	`Sel`
`Upd`	X	X
`Sel`	X	

At L_2:

	Granted Mode		
Requested Mode	`TestIncOK`	`TestIncNO`	`Dec`
`TestIncOK`		X	X
`TestIncNO`	X		X
`Dec`		X	

20.24 Give an example of a schedule in which a pessimistic concurrency control makes a transaction wait but later allows it to commit, while an optimistic concurrency control restarts the transaction.

Solution:

$r_2(y)\ w_1(x)\ r_2(x)\ commit_1\ request\_to\_commit_2$

A pessimistic control would force T_2 to wait when it requested to read x but would then allow the operation after T_1 commits and subsequently allow T_2 to commit. An optimistic control would abort T_2 when it requested to commit.

20.25 Give an example of a schedule in which a pessimistic concurrency control makes a transaction wait but then allows it to commit, while an optimistic concurrency control allows the transaction to commit without waiting.

Solution:
The following schedule would be accepted by an optimistic control.

$$r_1(x)\ r_2(y)\ w_1(z)\ w_2(z)\ commit_1\ commit_2$$

A pessimistic control would force the operation $w_2(z)$ to wait.

20.26 Give an example of a schedule that is acceptable (without any delays caused by locks) by an immediate-update pessimistic strict two-phase locking concurrency control, while an optimistic concurrency control restarts one of the transactions.

Solution:

$$r_1(x)\ r_2(x)\ r_2(x)\ w_2(y)\ commit_2\ r_1(y)\ w_1(z) request\_to\_commit_1$$

20.27 Can a deadlock occur in the timestamp-ordered control described in the text?

Solution:
Yes. Assume the read and write timestamps of x and y are initially smaller than $TS(T_1)$ and that $TS(T_1) < TS(T_2)$. The schedule is:

$$w_1(y),\ w_2(x),\ r_2(y),\ w_1(x)$$

$r_2(y)$ waits since y is dirty, and $w_1(x)$ waits since it cannot overwrite x until T_2 completes

20.28 Give an example of a schedule produced by a timestamp-ordered concurrency control in which the serialization order is not the commit order.

Solution:
Assume $TS(T_1) < TS(T_2)$

$$r_1(x)\ w_2(x)\ commit_2\ w_1(y)\ commit_1$$

Transaction T_1 is before the transaction T_2 but commits later.

20.29 Give an example of a schedule that is strict and serializable but not in commit order and that could have been produced by either a timestamp-ordered concurrency control or a two-phase locking concurrency control.

Solution:

$$r_1(x)\ r_1(y)\ w_2(x)\ commit_2\ commit_1$$

20.30 Give an example of a schedule that would be accepted by a two-phase locking concurrency control but not by a timestamp-ordered concurrency control.

Solution:

$r_1(x)\ r_2(x)\ w_2(y)\ commit_2\ w_1(x)\ commit_1$

This schedule would be accepted by a two-phase locking concurrency control with T_2 before T_1 in the serial order. However it would not be accepted by a timestamp-ordered concurrency control because, with such a control, T_1 must be before T_2 because it started before T_2, and thus T_2 should have read the value of x that T_1 wrote.

20.31 Show that the following proposed protocol for a timestamp-ordered concurrency control is not recoverable.

> Store with each data item the maximum timestamp of any (not necessarily committed) transaction that has read that item and the maximum timestamp of any (not necessarily committed) transaction that has written that item.
>
> When a transaction makes a request to read (write) a data item, if the timestamp of the requesting transaction is smaller than the write (read) timestamp in the item, restart the transaction; otherwise, grant the request.

Solution:
If a transaction reads a value written by an uncommitted transaction and then commits.

20.32 The *kill-wait* concurrency control combines the concepts of the immediate update concurrency control and the timestamp-ordered control. As in the timestamp-ordered system, when a transaction, T_1 is initiated, it is assigned a timestamp, $TS(T_1)$. However, the system uses the same conflict table as the immediate-update pessimistic control does and resolves conflicts using the rule

> If transaction T_1 makes a request that conflicts with an operation of active transaction, T_2
>
> **if** $TS(T_1) < TS(T_2)$, **then** *abort* T_2, **else** *make* T_1 *wait until* T_2 *terminates*.

where *abort* T_2 is referred to as a *kill* because T_1 kills T_2.

a. Show that the kill-wait control serializes in commit order.

Solution:
It serializes in commit order because transactions that are not aborted are granted requests or wait using the same locking protocol used in an immediate-update pessimistic control.

b. Give a schedule produced by a kill-wait control that is not serializable in timestamp order.

Solution:

$w_1(x)\ w_2(y)\ commit_2\ r_1(y)\ w_1(z)\ commit_1$

T_2 started after T_1 but is serialized before T_1.

c. Explain why deadlock does not occur in a kill-wait control.

Solution:
Since a transaction waits only for an older transaction, there can be no cycle of waits.

20.33 The *wait-die* concurrency control is another control that combines the concepts of the immediate-update concurrency control and the timestamp-ordered control.

> If transaction T_1 makes a request that conflicts with an operation of active transaction T_2
> **if** $TS(T_1) < TS(T_2)$, **then** *make* T_1 *wait until* T_2 *terminates*, **else** *abort* T_1.

where *abort* T_1 is referred to as a *die* because T_1 kills itself.

a. Show that the wait-die control serializes in commit order and prevents deadlocks.

Solution:
It serializes in commit order because it uses the same locking protocol as the immediate-update pessimistic control. It prevents deadlocks because a transaction can only wait for a younger transaction.

b. Compare the fairness of the execution of the kill-wait and wait-die controls.

Solution:
In the Kill-Wait system the oldest transaction has absolute priority and never waits, but it might kill a transaction that has already been granted permission to perform an operation. In the Wait-Die system, only the requesting transaction is affected by the request.

20.34 Give a complete description of an algorithm for a parallel validation, optimistic concurrency control.

Solution:
When T_1 requests to commit:

1. If the read phase of T_1 overlaps the validation/write phase of T_2, then if the set of items read by T_1 is not disjoint from the set written by T_2, abort T_1.
2. If T_1 enters its validation/write phase after T_2 and the execution of those phases overlap, then if the set of items written by T_1 is not disjoint from the set written by T_2, abort T_1.
3. If T_1 has not been aborted because of (1) or (2), commit T_1.

20.35 Describe a serial validation, optimistic concurrency control that uses backward validation and in addition uses timestamps to distinguish the order in which conflicting read and write operations occur. The validation condition in that case need not be conservative, and only conflicts that would violate commit order cause aborts.

Solution:
The concurrency control assigns a unique timestamp to each transaction, T, when it enters its validation phase. Then when T writes an item during its write phase, the value of T's timestamp is stored with that item in the database and also stored with the information the concurrency control maintains about T's write set to perform validation of other transactions. The validation is then performed as follows: When a transaction, T_1, reads an item during its read phase, the control remembers the value

of that item's timestamp. Later when T_1 enters its validation phase, the control checks if any other transaction, T_2, that committed during T_1's read phase wrote any item that T_1 read. If so, it checks the timestamp of that item to see if T_2 changed the value of that item after T_1 read it. If the item's timestamp had been changed, T_1 is aborted. If there is no such item for which the timestamp has been changed, T_1 is allowed to commit.

20.36 Subroutines can be used in a flat transaction in an attempt to mimic the behavior of a subtransaction in the nested model: whenever a subtransaction aborts, the corresponding subroutine manually undoes any database updates it has performed and returns with status indicating failure. Explain why the nested model allows higher transaction throughput.

Solution:
When a subtransaction aborts, any new locks it has obtained are released. New locks obtained by the subroutine, however, are not released.

20.37 Suppose that transactions T_1 and T_2 can be decomposed into the subtransactions

$$T_1 : \ T_{1,1}, \ T_{1,2}$$

and

$$T_2 : \ T_{2,1}, \ T_{2,2}$$

such that the database items accessed by $T_{1,1}$ and $T_{2,1}$ are disjoint from the items accessed by $T_{1,2}$ and $T_{2,2}$. Instead of guaranteeing that all schedules involving T_1 and T_2 are serializable, suppose that a concurrency control guarantees that $T_{1,1}$ is always executed serializably with $T_{2,1}$ and that $T_{1,2}$ is always executed serializably with $T_{2,2}$.

a. Will T_1 always be serializable with T_2? Explain.

Solution:
No. $T_{1,1}$ might be serialized before $T_{2,1}$, and $T_{1,2}$ might be serialized after $T_{2,2}$. Thus there is no equivalent serial order.

b. What minimal additional condition *on the subtransactions* guarantees that the effect of executing T_1 concurrently with T_2 is the same as a serial schedule?

Solution:
One condition is that $T_{1,1}$ commutes with $T_{2,1}$ and thus they can be serialized in either order.

c. Assuming that the condition of (b) holds, what advantage does the new concurrency control have over a concurrency control that guarantees serializability?

Solution:
Locks acquired by $T_{1,1}$ or $T_{2,1}$ are released early.

20.38 Suppose that transactions T_1 and T_2 can be decomposed into the subtransactions

$$T_1 : \ T_{1,1}, \ T_{1,2}$$

and

$$T_2 : \ T_{2,1}, \ T_{2,2}$$

such that each subtransaction individually maintains the consistency constraints of the database. Instead of guaranteeing that all schedules involving T_1 and T_2 are serializable, suppose that a concurrency control guarantees that all subtransactions are always executed serializably.

a. Will T_1 always be serializable with T_2? Explain.

Solution:
No. See previous exercise.

b. Will integrity constraints be maintained by all possible schedules?

Solution:
Yes, because each subtransaction starts with a consistent database and hence produces a consistent database.

c. What possible problems might arise if the concurrency control schedules transactions in this way?

Solution:
The database states produced might not correspond to any serial schedule and hence might not be a correct model of the state of the world. However, this situation might be acceptable to some applications. A similar problem might arise with respect to the information returned to the transaction.

20.39 A **blind write** occurs when a transaction writes a database item it has not read. For example, in the Student Registration System a transaction might compute a student's GPA by reading her course grades, computing the average, and then (blindly) writing the result in the appropriate database item without first reading that item. Some applications have the property that no transactions perform blind writes. Show that for such applications

a. View equivalence is *equivalent* to conflict equivalence.

Solution:
Assume we are given a schedule, S, without blind writes that is view equivalent to another schedule S'. We first prove by induction that the order of the writes of each item, x, in S is the same as in S'. (This statement is not true in general for view equivalent schedules that have blind writes as shown by the example in Figure 20.4.) Consider the sequence of transactions in S that wrote x, $T_{w(x)1}$, $T_{w(x)2}, \ldots, T_{w(x)n}$, ordered by their occurrence in S'.

To start the induction, we show that the first write of x in S was made by $T_{w(x)1}$ (the first transaction that wrote x in S'). No later transaction in S', $T_{w(x)i}$, could have made the first write of x because, since there are no blind writes, $T_{w(x)i}$ would have had to read some version of x and since S is view equivalent to S', that version would have had to be the one written by $T_{w(x)i-1}$, but $T_{w(x)(i-1)}$ has not yet written x (no transaction has).

Now assume the statement is true for the the first $j-1$ transactions in the above list. We show that the next write of x in S was made by $T_{w(x)j}$ (the next transaction that wrote x in S'). It cannot have been made by any earlier transaction in the list because of the induction hypothesis. It cannot have been made by any later transaction using the same reasoning as in the previous paragraph.

Thus the order of the writes of each item in S is the same as the order of the writes of that item in S', and hence the writes are in the same conflict order in both schedules. Also, because S is view equivalent to S', the reads in S read the same values as the reads in S', and hence the reads are in the same conflict order with the writes in both schedules. Thus S is conflict equivalent to S'.

b. The timestamp-ordered concurrency control described in Section 20.9.1 never uses the Thomas Write Rule.

Solution:
As described in the text, the Thomas Write Rule is used when a transaction, T, requests to write an item x and $rt(x) < TS(T) < wt(x)$. But since every transaction that writes x must have previously read it, the last transaction T' that wrote x and set its write timestamp to $wt(x) = TS(T')$ must have previously read x, and hence x's read timestamp must be at least $TS(T')$ (if T' has committed, another transaction with a later timestamp might also have read it). Thus for such applications, it is always true that $rt(x) \geq wt(x)$ and hence the condition for the Thomas Write Rule is never satisfied.

c. In the timestamp-ordered concurrency control described in the text, for each item, x, that a transaction, T, writes, when T commits, $rt(x) = wt(x) = TS(T)$.

Solution:
When a transaction, T, requests to write an item, x, since T has already read x, the read timestamp, $rt(x)$ must satisfy $rt(x) \geq TS(T)$. If $rt(x) > TS(T)$, then (as described in the text) T is too old to write x and so will be aborted by the concurrency control. Hence if T is allowed to write x, it must be true that at that instant $rt(x) = wt(x) = TS(T)$. If later, but before T commits, some younger transaction requests to read x, that transaction will be made to wait until T commits (and hence cannot write either). Therefore when T commits, it will still be true that $rt(x) = wt(x) = TS(T)$

20.40 State which of the following operations has a compensating operation.

a. Give all employees a 10% raise.

Solution:

Yes. Give all employees a 9.09.09% decrease.

b. Give all employees making less than $10,000 a 10% raise.

Solution:

No. There is no way for a compensating operation to know whether or not to give a decrease to an employee whose salary after the original operation is, for example, $10,100.

c. Set the value of a particular item to 12.

Solution:

No. There is no way for a compensating operation to know what the value of the item was before the original operation

d. Insert a new tuple with key 1111 into the database, and set the value of one of its attributes to 12.

Solution:

Yes. Delete the tuple with key 1111.

e. Set the value of a particular item to the square of its original value.

Solution:

No. There is no way for a compensating operation to know what the sign of the original value was.

20.41 Assume an operation p has a compensating operation p^{-1}. Show that if operation q commutes with p, then p^{-1} commutes with q.

Solution:

Since q commutes with p, the schedule p, q, p^{-1} is equivalent to the schedule q, p, p^{-1}. But starting in all initial states, this schedule leaves the database in the same final state as the schedule p, p^{-1}, q, which implies that p^{-1} commutes with q.

20.42 Give an example of a schedule of consistent transactions that is not serializable but maintains the correctness of any integrity constraint that might conceivably be associated with the database.

Solution:
The schedule contains a lost update. For example,

$$r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$$

T_2's update is lost, and the result of the schedule is the same as if only T_1 had executed and T_2 had never executed. Since T_1 is assumed to be consistent, the schedule maintains all possible integrity constraints. However, the final database might be incorrect. since T_2's update has been lost

21

Isolation in Relational Databases

EXERCISES

21.1 Suppose that the transaction processing system of your university contains a table in which there is one tuple for each currently registered student.

a. Estimate how much disk storage is required to store this table.

Solution:
10,000 tuples * 100 bytes/tuple = 1 megabyte.

b. Give examples of transactions that must lock this entire table if a table-locking concurrency control is used.

Solution:
Find all students who are registered in at least one course.

21.2 Consider an INSERT statement that inserts tuples into table T and that contains a nested SELECT statement having predicate P in its WHERE clause. Assume P is a simple predicate that does not contain a nested SELECT statement and consists of a conjunction of clauses of the form (`Attribute` *op* `constant`). Describe the predicate associated with T.

Solution:
Let `P'` is the predicate associated with T. A subset of the attributes named in `P` are associated with attributes of T through the SELECT list of the nested SELECT statement and the attribute list of the INSERT statement. Let `A` be such an attribute and let `A'` be the corresponding attribute of T. Then the conjunct of `P` describing `A` is a conjunct of `P'` with `A'` substituted for `A`. Furthermore, if the schema for T specifies a default value, *def*, for an attribute, `B`, not contained in the attribute list of the INSERT statement, `P'` contains a conjunct (`B` = '*def*').

21.3 Choose a set of transactions for an application of your choice (other than a banking or student registration system). For each isolation level weaker than SERIALIZABLE, give an example of a schedule that produces an erroneous situation.

Solution:
Airline reservation system:

REPEATABLE READ

(SELECT *and print list* : *passengers on flight* 100)$_1$
(*Make new reservation on flight* 100)$_2$
(*Read and print item* : *number of passengers on flight* 100)$_1$

Printout is inconsistent

READ COMMITTED

(*read* : *seat* 5B *is empty*)$_1$
(*read* : *seat* 5B *is empty*)$_2$ (*reserve seat* 5B *for passenger* 10)$_2$ *commit*$_2$
((*reserve seat* 5B *for passenger* 127)$_1$

Seat 5B is reserved for two passengers.

READ UNCOMMITTED

(*write request item* : *passenger requests exotic meal for seat* 5B)$_1$
(*read request item*; *write order item* : *order meal from caterer*)$_2$ *commit*$_2$
abort$_1$

Meal has been ordered that passenger does not want.

21.4 Assume that transactions are executed at REPEATABLE READ. Give an example in which a phantom occurs when a transaction executes a SELECT statement that specifies the value of the primary key in the WHERE clause.

Solution:
The requested tuple is not there when the SELECT statement is executed but is inserted by a concurrent transaction before the transaction commits.

21.5 Assume that transactions are executed at REPEATABLE READ. Give an example in which a phantom occurs when a transaction executes a DELETE statement to delete a set of tuples satisfying some predicate, P.

Solution:
After the DELETE statement is executed, a concurrent transaction inserts a new tuple that satisfies P.

21.6 Assume that transactions are executed at REPEATABLE READ. Give an example in which an UPDATE statement executed by one transaction causes a phantom in an UPDATE statement executed in another.

Solution:
The UPDATE statement in the first transaction updates all tuples that satisfy predicate P, and the UPDATE statement in the second updates a tuple in such a way that its new value satisfies P.

21.7 Consider a schema with two tables, TABLE1 and TABLE2, each having three attributes, `attr1`, `attr2`, and `attr3`, and consider the statement

```
SELECT T1.attr1, T2.attr1
FROM Table1 T1, Table2 T2
WHERE T1.attr2 = T2.attr2 AND T1.attr3 = 5
          AND T2.attr3 = 7
```

Give an INSERT statement that might cause a phantom.

Solution:

```
INSERT INTO T1(attr1, attr2, attr3)
VALUES (?, ?, 5)
```

21.8 Explain the difference between a nonrepeatable read and a phantom. Specifically, give an example of a schedule of the SQL statements of two transactions that illustrate the two cases. Specify an isolation level in each case.

Solution:
To illustrate a non-repeatable read consider two transactions running at READ COMMITTED:

SELECT_1(get sum of balances in all of Mary's accts);

UPDATE_2(post interest in all accts);

Commit_2;

SELECT_1(get sum of balances in all of Mary's accts);

To illustrate a phantom consider two transactions running at REPEATABLE READ:

SELECT_1(get sum of balances in all of Mary's accts);

INSERT_1(new account for Mary);

COMMIT_2;

SELECT_1(get sum of balances in all of Mary's accts);

In the first case, the second SELECT returns the same set of tuples as the first, but their values have changed. In the second case, the second SELECT returns a new tuple that satisfies the read predicate, but the values of the old tuples that satisfy that predicate have not changed.

21.9 For each of the locking implementations of the isolation levels, state whether IS locks are required and when they can be released.

Solution:
READ UNCOMMITTED: No read locks are required and hence no intention to read locks.

READ COMMITTED: Read locks are short term, and for each short term read lock on a tuple, a short term IS lock must be obtained on the table.

REPEATABLE READ: Read locks are long term, and for each long term read lock on a tuple, a long term IS lock must be obtained on the table.

SERIALIZABLE: If index locking is used, a long term IS lock on the table must be acquired. If index locking is not used, an S lock is acquired on the table and no IS lock is needed.

21.10 The following procedure has been proposed for obtaining read locks.

> Whenever an SQL statement reads a set of rows that satisfies some predicate in a table, the system first gets an IS lock on the table containing the rows and then gets an S lock on each of the rows.

Explain why this procedure allows phantoms.

Solution:
An INSERT or DELETE statement gets a non-conflicting IX lock on the table and X locks on the tuples inserted or modified. The inserted tuples might satisfy the predicate but since they are new there would be no lock conflict to delay the insert. Similarly the initial value of a modified tuple might not satisfy the predicate and so it would be possible to obtain an X

21.11 Give an example of a schedule produced by a read-only multiversion concurrency control in which the read/write transactions serialize in commit order while the read-only transactions serialize in a different order.

Solution:

$$r_1(x)\ w_2(y)\ w_3(z)\ commit_3\ commit_2\ r_1(y)\ commit_1$$

T_1 does not see the new value of y written by T_2 and hence serializes before T_2 even though it commits later.

21.12 Give an example of a schedule of read/write requests that is accepted by a multiversion concurrency control in which transaction T_1 starts after transaction T_2 commits, yet T_1 precedes T_2 in the serial order. Such a schedule can have the following nonintuitive behavior (even though it is serializable): you deposit money in your bank account; your transaction commits; later you start a new transaction that reads the amount in your account and finds that the amount you deposited is not there. (*Hint:* The schedule is allowed to contain additional transactions.)

Solution:

$$r_3(x)\ w_2(x)\ commit_2\ r_1(y)\ w_3(y)\ commit_3\ r_1(x)\ commit_1$$

The order is T_1 T_3 T_2. Therefore the last read of T_1 on x does not see the write of x made by T_2. Thus T_1 is before T_2 even though it started after T_2 committed. (Note that the serialization order is the reverse of the commit order.)

21.13 Show that the schedule shown in Figure 21.13 for incorrect execution at SNAPSHOT isolation can also occur when executing at OPTIMISTIC READ COMMITTED (Section 21.2.1) and will also be incorrect.

Solution:
The schedule can occur at OPTIMISTIC READ COMMITTED because when each transaction performs its write, the item it wrote was not changed since it was read.

21.14 Give examples of schedules that would be accepted at

a. SNAPSHOT isolation but not REPEATABLE READ

Solution:

$$r_1(x)\ r_2(x)\ w_2(x)\ commit_2\ w_1(y)\ commit_1$$

At REPEATABLE READ, T_2 would not be allowed to write x since T_1 would have a read lock on it.

b. SERIALIZABLE but not SNAPSHOT isolation

Solution:

$$r_1(x)\ r_2(y)\ w_1(x)\ commit_1\ r_1(x)\ w_2(x)\ commit_2$$

This schedule is serializable, but would not be allowed at SNAPSHOT isolation. At SNAPSHOT, T_2's read of x would give the value before T_1 wrote it, and then T_2 would not be allowed to commit because T_1 was the first committer that wrote x.

21.15 A particular read-only transaction reads data that was entered into the database during the previous month and uses that data to prepare a report. What is the weakest isolation level at which this transaction can execute? Explain.

Solution:
READ UNCOMMITTED, because no other transaction can change the data.

21.16 Explain why a read-only transaction consisting of a single SELECT statement that uses an INSENSITIVE cursor can always execute correctly at READ COMMITTED.

Solution:
It sees a snapshot of committed data.

21.17 Suppose that a locking implementation of REPEATABLE READ requires that a transaction obtain an X lock on a table when a write is requested. When a read is requested, a transaction is required to obtain an IS lock on the table and an S lock on the tuples returned. Show that phantoms cannot occur.

Solution:
The INSERT statement requires an X lock on the table, and it cannot obtain that lock if the select statement has an IS lock on the table.

21.18 Consider an isolation level implemented using long-duration granular locks on rows and tables. Under what conditions can phantoms occur?

Solution:
If a SELECT statement, using a read predicate, gets long term read locks on the rows that satisfy that predicate, those locks do not prevent another transaction from inserting a row that also satisfies that predicate.

21.19 Show that the algorithm for concurrently accessing a B^+tree in which operations that only search the tree use lock coupling and operations that modify the tree handle write latches in a two-phase fashion guarantees that all operations are carried out successfully.

Solution:
Since modifying operations use two-phase locking and searches do not modify the tree, modifying operations are serializable. Furthermore, since searches are read-only, they too are serializable. Hence, our only concern is that a modifying operation does not serialize with a search. Consider the interaction of a search, S, and a modifying operation, M. If M starts at the root first, S will have to wait until M completes before it starts, so they are serializable. Suppose S starts at the root first. Lock coupling guarantees that S never releases a latch until it acquires the next latch on the search path. Hence, it always holds a latch and as a result, is always at least one step ahead of M. As a result, in any read/write conflict between operations of S and M, S always precedes M and hence S and M are serializable.

21.20 In Section 21.3.1 we described an algorithm that allowed the concurrent execution of a search for a unique key and an insert of a row in a B^+ tree. Extend the algorithm to range searches and deletes.

Solution:
The search algorithm uses sibling pointers when it reaches the leaf level to find keys in the range. Read latches are obtained on additional leaf nodes. The delete algorithm upgrades the update latch on the leaf page, $p1$ containing the first key to be deleted to a write latch and deletes the key(s). If the sibling node, $p2$, might also contain keys in the range, the operation backs up to the parent of both, p, and then acquires an update latch on $p2$. If $p2$ contains keys in the range the update latch is upgraded to a write latch and the keys are deleted. The update latch on p is upgraded to a write latch if $p1$ and $p2$ are merged.

21.21 a. Give an example of a schedule of two transactions in which a two-phase locking concurrency control causes one of the transactions to wait but a SNAPSHOT isolation control aborts one of the transactions.

Solution:
The schedule as handled by a two-phase locking concurrency control:

$$r_1(x)\ w_1(x)\ r_2(y)\ \mathit{Request}[w_2(x)]\ \mathit{commit}_1\ \mathit{Execute}[w_2(x)]\ \mathit{commit}_2$$

The schedule as handled by a SNAPSHOT isolation control:

$$r_1(x)\ w_1(x)\ r_2(y)\ w_2(x)\ \mathit{commit}_1\ \mathit{Request}[\mathit{commit}_2]\ \mathit{abort}_2$$

b. Give an example of a schedule of two transactions in which a two-phase locking concurrency control aborts one of the transactions (because of a deadlock) but a SNAPSHOT isolation control allows both transactions to commit.

Solution:

$r_1(x)\ r_2(y)\ w_1(y)\ w_2(x)$

21.22 Explain why SNAPSHOT-isolated schedules do not exhibit dirty reads, dirty writes, lost updates, nonrepeatable reads, and phantoms.

Solution:
Dirty Reads: Reads only committed values.

Dirty Writes: First committer wins.

Lost Updates: First committer wins

Non-Repeatable Reads: Always returns items from same snapshot

Phantoms: Always returns items from same snapshot (but note discussion in text).

21.23 Consider an application consisting of transactions that are assigned different isolation levels. Prove that if transaction T is executed at SERIALIZABLE and transaction T' is any other transaction, then T either sees all changes made by T' or it sees none.

Solution:
Transactions executing at all the isolation levels use long term write locks, and transactions executing at SERIALIZABLE use long term read and write locks. Hence all operations of T are serializable with respect to the write operations of T'. Hence T either sees all the changes made by T' or none.

21.24 All of the transactions in some particular application write all of the data items they read. Show that if that application executes under SNAPSHOT isolation, all schedules of committed transactions will be serializable.

Solution:
If two concurrently executing transactions both commit at SNAPSHOT isolation, they have written disjoint sets of data items. If they have written all the items they have read, then they have also read disjoint sets of data items. Therefore all their operations have been on disjoint sets of data items, and hence all operations commute — thus they are serializable.

21.25 Consider the schedule of two bank withdrawal transactions shown in Figure 21.13 for which SNAPSHOT isolation leads to an inconsistent database. Suppose that the bank encodes, as an integrity constraint in the database schema, the business rule "The sum of the balances in all accounts owned by the same depositor must be nonnegative." Then that particular schedule cannot occur.

Although the integrity constraint is now maintained, the specification of a particular transaction might assert that when the transaction commits, the database state satisfies a stronger condition. Give an example of a stronger condition that a

withdrawal transaction might attempt to impose when it terminates and a schedule of two such transactions at SNAPSHOT isolation that causes them to behave incorrectly.

Solution:
Each account has $10 initially, and each transaction is trying to withdraw $5, but is trying to maintain the condition that the total in the two accounts is at least $12. Note this is not a business rule or an integrity constraint — the final database is consistent, just incorrect.

21.26 The following multiversion concurrency control has been proposed.

> Reads are satisfied using the (committed) version of the database that existed when the transaction made its first read request. Writes are controlled by long-duration write locks on tables.

Does the control always produce serializable schedules? If not, give a nonserializable schedule it might produce.

Solution:
The control can produce schedules with both write skew and lost updates. Assume x and y are in different tables.

Write Skew:

$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(x)\ w_2(y)$

Lost Update:

$r_1(x)\ r_2(x)\ w_1(x)\ commit_1\ w_2(x)$

21.27 We have given two different implementations of the READ COMMITTED isolation level: the locking implementation in Section 21.2 and the read-consistency implementation in Section 21.5. Give an example of a schedule in which the two implementations produce different results.

Solution:
A SELECT statement using a non-insensitive cursor, followed by some FETCH statements. In the Read Consistency version, all fetches are from the same snapshot of the database. In the locking implementation, different fetches might come from different snapshots.

21.28 The granular locking protocol can exhibit a deadlock between two transactions, one of which executes a single SELECT statement and the other a single UPDATE statement. For example, suppose that one transaction contains the single SELECT statement

```
SELECT COUNT (P.Id)
FROM EMPLOYEE P
WHERE P.Age = '27'
```

which returns the number of employees whose age is 27, and the other contains the single UPDATE statement

```
UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Department = 'Adm'
```

which gives all employees in the administration a 10% raise. Assume that there are indices on both `Department` and `Age` and that the tuples corresponding to the department `Adm` are stored in more than one page as are those corresponding to age 27. Show how a deadlock might occur at isolation levels other than READ UNCOMMITTED.

Solution:
At isolation levels SERIALIZABLE, REPEATABLE READ, and READ COMMITTED, the first transaction, T_1, gets an IS lock on the table and an S lock on the first page containing tuples corresponding to Age 27. Meanwhile, the second transaction, T_2, gets an IX lock on the table and an X lock on the first page containing tuples corresponding to the Administration Department. If the Department has 27 year old employees, it might be the case that later T_1 requests an S lock on a page T_2 has locked, and T_2 requests an X lock on a page T_1 has locked. A deadlock occurs.

At isolation level READ UNCOMMITTED transactions do not get read locks. However, since each statement must execute in an isolated fashion, internal locks on the same items must be acquired by T_1 and held until the statement has been completely processed. Therefore the deadlock could still occur.

21.29 Give an example of a schedule executing at SNAPSHOT isolation in which two transactions each introduce a phantom that is not seen by the other transaction, resulting in incorrect behavior. Assume that the data items referred to in the description of SNAPSHOT isolation are rows.

Solution:
p is a predicate over a particular table and x and y are rows satisfying p.

$$\mathsf{SELECT}_1(p)\ \mathsf{SELECT}_2(p)\ \mathsf{INSERT}_1(x)\ \mathsf{INSERT}_2(y)$$

An integrity constraint states that there can be no more than n items in the database satisfying predicate p. Each transaction reads the tuples satisfying p finds there are exactly $(n-1)$ and therefore inserts a new tuple satisfying p. At the end there are $(n+1)$ tuples satisfying p

21.30 In an Internet election system, each voter is sent a PIN in the mail. When that voter wants to vote at the election web site, she enters her PIN and her vote, and then a voting transaction is executed.

In the voting transaction, first the PIN is checked to verify that it is valid and has not been used already, and then the vote tally for the appropriate candidate is incremented. Two tables are used: One contains the valid PINs together with an indication of whether or not each PIN has been used and the other contains the names of the candidates and the vote tally for each. Discuss the issues involved in selecting an appropriate isolation level for the voting transaction. Discuss the issues involved in selecting appropriate isolation levels if a new (read-only) transaction is introduced that outputs the entire vote tally table.

Solution:
The only SQL statements in the voting transaction will be an UPDATE statement that checks and updates the PIN table and an UPDATE statement that updates the tally table. Since write locks are always long duration at all isolation levels, schedules of voting transactions will be serializable at any isolation level.

A transaction that outputs the tally table should run at READ COMMITTED and use an insensitive cursor in order to output a consistent snapshot of the table.

21.31 An airlines database has two tables: FLIGHTS, with attributes `flt_num, plane_id, num_reserv`; and PLANES, with attributes `plane_id,` and `num_seats`.

The attributes have the obvious semantics. A reservation transaction contains the following steps:

```
        SELECT F.plane_id, F.num_reserv
        INTO :p, :n
        FROM FLIGHTS F
        WHERE F.flt_num = :f
A.      SELECT P.num_seats
        INTO :s
        FROM PLANES P
        WHERE P.plane_id = :p
B.       . . . check that n < s . . .
C.      UPDATE FLIGHTS F
        SET F.num_reserv = :n + 1
        WHERE F.flt_num = :f
D.      COMMIT
```

Assume that each individual SQL statement is executed in isolation, that the DBMS uses intention locking and sets locks on tables and rows, and that host variable `f` contains the number of the flight to be booked. The transaction should not overbook the flight.

a. Assuming that the transaction is run at READ COMMITTED, what locks are held at points A, B, and D?

Solution:
A - No locks

B - No locks

D - X lock on row in FLIGHTS, IX lock on FLIGHTS

b. The database can be left in an incorrect state if concurrently executing reservation transactions that are running at READ COMMITTED are interleaved in such a way that one transaction is completely executed at point B in the execution of another. Describe the problem.

Solution:
Lost update causing database to incorrectly record number of passengers and flight might be overbooked

c. In an attempt to avoid the problem described in (b), the SET clause of the UPDATE statement is changed to `F.num_reserv = F.num_reserv + 1`. Can reservation transactions now be run correctly at READ COMMITTED? Explain.

Solution:
Flight can still be overbooked, but database correctly records number of passengers

d. Assuming that the transaction is run at REPEATABLE READ and that the tables are accessed through indices, what table locks are held at points A, B, and D?

Solution:
A - S lock on row in FLIGHTS, IS lock on FLIGHTS

B -The locks named in (A), plus S lock on row in PLANES, IS lock on PLANES

D - X lock on row in FLIGHTS, IX lock on FLIGHTS, S lock on row in PLANES, IS lock on PLANES

e. What problem does the interleaving of (b) cause at REPEATABLE READ? Explain.

Solution:
Deadlock

f. Does the interleaving of (b) cause an incorrect state if the transaction (either version) is run using SNAPSHOT isolation? Explain.

Solution:
No. If two transactions attempt to update the same tuple, one will be aborted.

g. To keep track of each passenger, a new table, PASSENGER, is introduced that has a row describing each passenger on each flight with attributes `name, flt_num, seat_id`. SQL statements are appended to the end of the transaction (1) to read the `seat_id`'s assigned to each passenger on the flight specified in `f` and (2) to insert a row for the new passenger that assigns an empty seat to that passenger. What is the weakest ANSI isolation level at which the transaction can be run without producing an incorrect state (i.e., two passengers in the same seat)? Explain.

Solution:
REPEATABLE READ. No phantom is possible since only one transaction can add a tuple to PASSENGER for a particular flight at a time (since a transaction must hold an X lock on the row in FLIGHTS before inserting a row in PASSENGER).

21.32 Two transactions run concurrently, and each might either commit or abort. The transactions are chosen from the following:

$$
\begin{aligned}
T_1 &: \; r_1(x) \; w_1(y) \\
T_2 &: \; w_2(x) \\
T_3 &: \; r_3(y) \; w_3(x) \\
T_4 &: \; r_4(x) \; w_4(x) \; w_4(y)
\end{aligned}
$$

In each of the following cases, state (yes or no) whether the resulting schedule is always serializable and recoverable. If the answer is no, give an example of a schedule that is either not serializable or not recoverable.

a. T_1 and T_2 both running at READ UNCOMMITTED

Solution:

No.

$$w_2(x) \quad r_1(x) \quad w_1(y) \quad c_1 \quad a_2$$

Not recoverable

b. T_2 and T_2 both running at READ UNCOMMITTED

Solution:

Yes

c. T_1 and T_2 both running at READ COMMITTED

Solution:

Yes

d. T_1 and T_3 both running at READ COMMITTED

Solution:

No

$$r_1(x) \quad r_3(y) \quad w_1(y) \quad w_3(x)$$

Not serializable

e. T_1 and T_3 both running at SNAPSHOT isolation

Solution:

No

$$r_1(x) \quad r_3(y) \quad w_1(y) \quad w_3(x)$$

Not serializable

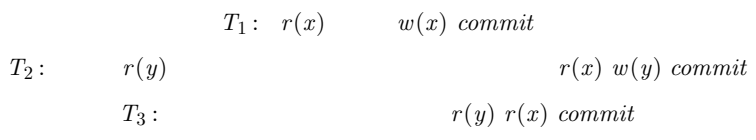
f. T_1 and T_4 both running at SNAPSHOT isolation

Solution:

Yes

21.33 Give an example of a schedule that could be produced at SNAPSHOT isolation in which there are two transactions that execute concurrently but do not have the same snapshot number and do not see the same snapshot of the database. (*Hint:* the schedule can contain more than two transactions.)

Solution:



T_2 and T_3 are concurrent but T_3 has a snapshot number one greater than that of T_2 because T_1 committed before T_3 started. Thus T_3 reads the same value of y that T_2 does but reads a different value of x

21.34 A database has two tables:

STUDENT(`Id`, `Name`, $\cdots$)—`Id` and `Name` are both unique
REGISTERED(`Id`, `CrsCode`, `Credit`, $\cdots$)—contains one row for each course each student is taking this semester

A transaction type, T, has two SQL statements, $S1$ followed by $S2$ (with local computations between them):

```
S1  SELECT   SUM(R.Credits), S.Id
    INTO     :sum, :id
    FROM     STUDENT S, REGISTERED R
    WHERE    S.Name = 'Joe' AND S.Id = R.Id
    GROUP BY S.Name, S.Id
```

```
S2  UPDATE   Registered
    SET      Credits = Credits + 1
    WHERE    Id = :id AND CrsCode = :crs
```

$S1$ returns the total number of credits for which Joe is registered, together with his Id. T maintains the integrity constraint "no student shall register for more than 20 credits." If Joe has less than 20 credits, T executes $S2$ to increment the number of credits for which Joe has registered in a particular course. Suppose Joe executes two instances of T concurrently at the following isolation levels. In each case say whether or not the named violation of the constraint can occur and, if the answer is yes, explain how (e.g., what locks are or are not held).

a. READ COMMITTED

lost update

Solution:
no

violation of the integrity constraint

Solution:
yes - T_1 and T_2 both get short term shared locks on all Joe's rows in REGISTERED and calculate the total number of credits. Assume it is 19. Then

T_1 gets a long term exclusive lock on one of Joe's rows, increments it and commits. T_2 then does the same on another (possibly the same) row.

deadlock

Solution:
no

b. REPEATABLE READ

lost update

Solution:
no

violation of the integrity constraint

Solution:
no

deadlock

Solution:
yes - Both T_1 and T_2 get long term shared locks on all Joe's rows in REGISTERED. Then T_1 requests an exclusive lock on one of those rows and T_2 requests an exclusive lock on another (possibly the same).

c. SNAPSHOT

lost update

Solution:
no

violation of the integrity constraint

Solution:
yes - T_1 and T_2 both execute from the same version. Then they execute $S2$ using different values of :crs.

deadlock

Solution:
no

22

Atomicity and Durability

EXERCISES

22.1 Describe the contents of each of the following log records and how that record is used (if at all) in rollback and in recovery from crashes and media failure.

a. Abort record

Solution:
When a transaction aborts, it rolls back its updates and then writes an *abort record* to the log. The *abort record* contains the transaction's Id. Using these records, the identity of the transactions active at the time of the crash can be determined by the recovery procedure.

b. Begin record

Solution:
To avoid a complete backward scan, when T is initiated a *begin record* containing its transaction Id is appended to the log. When a transaction is aborted the backward scan for update records can be stopped when T's begin record is encountered.

c. Begin dump record

Solution:
Before starting a dump, a *begin dump* record is appended to the log. This is used to locate the checkpoint record at which roll forward during pass 2 of the media recovery procedure must start.

d. Checkpoint record

Solution:
The system periodically writes a *checkpoint record* to the log listing the identities of currently active transactions. The recovery process must (at least) scan backward to the most recent checkpoint record inorder to determine which transactions were active at the time of the crash.

e. Commit record

Solution:
When a transaction commits, it writes a *commit record* to the log. The *commit record*

contains the transaction's Id. Using these records, the identity of the transactions active at the time of the crash can be determined by the recovery procedure. A transaction is committed when this record is stored on the (non-volatile) log.

f. Compensation log record

Solution:
*Compensation log record*s are used to record the reversal of updates during abort processing.

g. Completion record

Solution:
Indicates, in a deferred update system, that updates in the intentions list of a transaction have been transferred to the database. On recovery, the log update records of incomplete transactions must be used to install their updates in the database.

h. Redo record

Solution:
With physical logging the *redo record* contains the new value of the data item that was updated (the after image). It is used to update the database after a crash.

i. Savepoint record

Solution:
Each time a transaction declares a savepoint, a *savepoint record*, containing the transaction's identity and the identity of the savepoint, is written to the log. To roll back to a specific savepoint, the log is scanned backwards to the specified savepoint's record.

j. Undo record

Solution:
With physical logging the *undo record* contains the old value of the data item that was updated (before image). If the transaction aborts, the *undo record* is used to restore the item to its original value.

22.2 Suppose that the concurrency control uses table locks and that a transaction performs an operation that updates the value of one attribute of one tuple in one table. Does the update record have to contain images of the entire table or just the one tuple?

Solution:
Only images of the tuple have to be stored.

22.3 Suppose that a dirty page in the cache has been written by two active transactions and that one of the transactions commits. Describe how the caching procedure works in this case.

Solution:
The write-ahead feature implies that the page cannot be written until the update records of both transactions have been stored in the log. Assuming update records contain after-images, the page does not have to be written out except as dictated by the rule relating to the use of checkpoint records.

22.4 Suppose that the database system crashes between the time a transaction commits (by appending a commit record to the log) and the time it releases its locks. Describe how the system recovers from this situation.

Solution:
No special procedures are required. Since the lock table is volatile, the system has no way of remembering the status of locks and so locks are automatically released.

22.5 Explain why the log buffer need not be flushed when an abort record is appended to it.

Solution:
If the system crashes before the log buffer is flushed, that transaction will be aborted anyway because there is no commit or abort record for it in the (non-volatile) log.

22.6 Explain why the LSN need not be included in pages stored in the database when physical logging is used together with a cache and log buffer.

Solution:
With physical logging the LSN is used to enforce the write-ahead policy and is not used during recovery. Its value lies in determining the relationship between updates to a page after it has been brought into the cache and the corresponding update records in the log. Update records corresponding to updates to the page during its previous stay in the cache must have been moved to mass storage before the page was swapped out. Hence, it is sufficient to associate the LSN with each page in the cache, but it need not be stored in the page. When a page is brought into the cache it is clean and the associated LSN can be initialized to a null value.

22.7 Suppose that each database page contained the LSN of the commit record of the last transaction that has committed and written a database item in the page, and suppose that the system uses the policy that it does not flush the page from the cache until the LSN of the oldest record in the log buffer is greater than the LSN of the page. Will the write-ahead policy be enforced?

Solution:
No. An item updated by an uncommitted transaction might be contained in the page.

22.8 The second step of the sharp checkpoint recovery procedure is as follows:
The log is scanned forward from the checkpoint. The after-images in all update records are used to update the corresponding items in the database. Assuming a locking concurrency control in which locks are held until commit time, show that the updates can be performed in either of the following orders:

a. As each update record is encountered in the forward scan, the corresponding database update is performed (even though the update records for different transactions are interleaved in the log).

Solution:
Because that is the order in which the operations took place.

b. During the forward scan, the update records for each transaction are saved in volatile memory, and the database updates for each transaction are all done at once when the commit record for that transaction is encountered during the forward scan.

Solution:
Because the transactions can be serialized in commit order

22.9 In the sharp checkpoint recovery procedure, explain whether or not the system needs to obtain locks when it is using the after-images in the log to update the database.

Solution:
No. No transactions are executed while recovery is taking place.

22.10 Consider the following two-pass strategy for crash recovery using a sharp checkpoint and physical logging: (1) The first pass is a backward pass in which active transactions are rolled back. Active transactions are identified as described in Section 22.2. The pass extends at least as far as the begin record of the oldest active transaction or the most recent checkpoint record, whichever is earlier in the log. As update records for these transactions are encountered in the scan, their before-images are applied to the database. (2) The second pass is a forward pass from the most recent checkpoint record to roll forward, using after-images, all changes made by transactions that completed since the checkpoint record was written (compensation log records are processed in the same way as ordinary update records so that aborted transactions are handled properly). Does the procedure work?

Solution:
Yes. The first pass rolls back all the updates made by transactions that were active when the crash occurred. The second pass does (or redoes) all the changes made by transactions that committed since the last checkpoint. (those changes might or might not have yet been stored in the database.)

22.11 In order for logical logging to work, a logical database operation must have a logical inverse operation. Give an example of a database operation that has no inverse. Suggest a procedure involving logical logging that can handle this case.

Solution:
An operation that changes an item x to $f(x)$, where $f()$ is some function with no known inverse (such as $x := 5$) is an example. A combination of physical and logical logging can be used. Physical logging is used for operations that have no inverse, logical logging is used for others. Each update record identifies the type of undo/redo that it uses.

22.12 Consider using the crash recovery procedure described in Section 22.2 (intended for physical logging) when logical logging is used. Explain how the procedure has to be modified to handle crashes that occur during recovery. Assume that the effect of each update is confined to a single page.

Solution:
Problem: The crash might occur during Phase 3. It makes a difference during the second execution of recovery (after the crash during Phase 3) whether the application of an inverse operation to an updated item has already been performed. Hence, LSNs must be used to determine whether an operation should be applied to a page.

22.13 Explain why, in a deferred-update system, the write-ahead feature that is a part of immediate-update systems is not used when a database item is updated.

Solution:
Neither the write-ahead feature nor a before-image in the update record is required in this case since the database item is not updated until after the transaction commits.

22.14 Explain why in a deferred-update system, the system does not first copy the intentions list into the database and then append the commit record to the log.

Solution:
Because the intentions list and the update records do not contain before-images. If the system should crash after the database has been updated, but before the commit record has been appended to the log, there would be no way to roll the transaction back.

22.15 Assume that the system supports SNAPSHOT isolation. Describe how a sharp (nonfuzzy) dump could be taken without shutting down the system.

Solution:
Run a transaction that reads each item in the database, and a snapshot will be returned. That snapshot could be used as the dump.

22.16 a. Explain how the log is implemented in your local DBMS.
b. Estimate the time in milliseconds to commit a transaction in your local DBMS.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

22.17 The LSN stored in a page of the database refers to an update record in the log describing the most recent update to the page. Suppose that a transaction has performed the last update to a page and later aborts. Since its update to the page is reversed, the LSN in the page no longer refers to the appropriate update record. Why is this not a problem in the description of logging in the text?

Solution:
When the update is reversed, a compensation log record is written to the log and its LSN is recorded in the page. Since the UNDO can be viewed as simply another write, the page's LSN is correct.

22.18 An airlines reservation system has demanding performance and availability standards. Do the following play a role in enhancing performance? Do they enhance availability? Explain your answers.

a. Page cache

Solution:
Enhances performance (page I/O avoided), not availability

b. Log buffer

Solution:
Enhances performance (page I/O avoided), not availability

c. Checkpoint record

Solution:
Enhances availability (recovery is speeded up), not performance

d. Physiological logging

Solution:
Enhances performance slightly (log records are smaller), but executing logical operations during recovery might increase recovery time

e. Mirrored disk

Solution:
Enhances availability

PART SIX

Distributed Applications and the Web

23

Architecture of Transaction Processing Systems

EXERCISES

23.1 Explain the advantages to a bank in providing access to its accounts database only through stored procedures such as `deposit()` and `withdraw()`.

Solution:

1. Correctness: The procedures are provided by the bank, which can do extensive quality assurance on them. Since the procedures are maintained by the bank site, they are protected from being altered by intruders. Users can be prohibited from submitting individual SQL statements, which might corrupt the database.
2. Efficiency: The SQL statements in the procedures can be prepared in advance and therefore can be executed more efficiently.
3. Authorization: The service provider can more easily authorize users to perform particular application-specific functions.
4. Throughput: By offering a more abstract service, the amount of information that must be communicated through the network is reduced. Only the input and output arguments of the procedure must be communicated.

23.2 Explain why the three-level organization of a transaction processing system (including transaction servers) is said to be scalable to large enterprise-wide systems. Discuss issues of cost, security, maintainability, authentication, and authorization.

Solution:
Authentication: Can be performed on a dedicated serve.r

Authorization: Can be quite specific at the transaction level.

Cost: As the number of users increase, additional servers can be added as needed.

Security: Applications procedures are more secure on a application server. Transaction procedures are more secure on a transaction server.

Maintainability: Application procedures reside in a single place and hence can be updated more easily.

Communication: As the number of users increases, the communication scales approximately linearly.

23.3 Explain what happens in a three-level architecture for a transaction processing system if the presentation server crashes while the transaction is executing.

Solution:
The presentation server is usually not a participant in the transaction. If it fails, the transaction can commit. Later when it restarts, it can ask the coordinator what happened.

23.4 Explain why the *cancel* button on an ATM does not work after the *submit* button has been pressed.

Solution:
The transaction at the bank has started its execution and might have already committed. One might say that the ATM is in a prepared state.

23.5 Explain the advantages of including a transaction server in the architecture of a transaction processing system.

Solution:

1. Separating the stored procedures from the application program has the advantages of decreasing communication costs and providing a more abstract interface to the application program.
2. Executing the stored procedures on a separate transaction server instead of in the DBMS has the advantage of reducing the load on the DBMS, thus providing increased scalability for large numbers of transactions.

23.6 Give an example of a transaction processing system you use that is implemented as a distributed system with a centralized database.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

23.7 Give an example in which a transaction in a distributed database system does not commit atomically (one database manager it accessed commits, and another aborts) and leaves the database in an inconsistent state.

Solution:
Transferring money between accounts at two different banks. The Withdraw at one bank commits, and the Deposit at the other bank aborts—too bad, you lost the money.

23.8 Explain whether the system you are using for your project can be characterized as TP-Lite or TP-Heavy.

Solution:
This is a hands-on problem. It's solution is not meant to be included in the manual.

23.9 List five issues that arise in the design of heterogeneous distributed transaction processing systems that do not arise in homogeneous distributed systems.

Solution:

1. The DBMSs are supplied by different vendors and speak different dialects of SQL.
2. Some of the DBMSs do not perform the two-phase commit protocol.
3. The systems use different low-level communication protocols.
4. At a higher level, some systems use RPCs and others use peer-to-peer communication.
5. Some of the legacy systems are so old that the source code for the transactions has disappeared from the face of the earth.

23.10 Explain the difference between a TP monitor and a transaction manager.

Solution:
A TP monitor is collection of middleware components (usually including a transaction manager) that are useful for building TP-heavy systems. It provides a set of application-independent services that are needed in a transaction processing system, but are not usually provided by an operating system.

A transaction manager is a middleware component that coordinates the execution of transactions, including the two-phase commit

23.11 Give three examples of servers, other than transaction servers, database servers, and file servers, that might be called by an application server in a distributed transaction processing system.

Solution:
queue server; event server; log server; print server

23.12 Describe the architecture of the student registration system used by your school.

23.13 State two ways in which transactional remote procedure calls differ from ordinary remote procedure calls.

Solution:

1. TRPCs contain the name of the transaction performing the call.
2. When a TRPC performs its first call on a new server, the stub notifies the coordinator of the name of that server.

23.14 Suppose that a transaction uses TRPC to update some data from a database at a remote site and that the call successfully returns. Before the transaction completes, the remote site crashes. Describe informally what should happen when the transaction requests to commit.

Solution:
The TRPC has notified the coordinator that the remote site is a participant in the transaction. Therefore when the coordinator performs the two-phase commit protocol, it will attempt to contact that site. When it finds that the site has crashed, it will abort the transaction.

23.15 Explain the difference between the `tx_commit()` command used in the X/Open API and the COMMIT statement in embedded SQL.

Solution:
The COMMIT statement in embedded SQL is a local command requesting that the current DBMS be committed. The `tx_commit()` statement is a request to commit a global transaction, that is to perform a two-phase commit of all the DBMSs that the global transaction has accessed.

23.16 Propose an implementation of distributed savepoints using the `tx` and `xa` interfaces to the transaction manager. Assume that each subtransaction (including the transaction as a whole) can declare a savepoint, and that when it does so, it forces its children to create corresponding savepoints. When a (sub)transaction rolls back to a savepoint, its children are rolled back to their corresponding savepoints.

Solution:
The transaction manager, TM, keeps track of the hierarchical relationships among the subtransactions of a distributed transaction. When a subtransaction, T, creates a child, T', at a resource manager, RM, RM provides TM with a savepoint and a rollback callback (part of `xa`). To create a savepoint, T calls the `create_savepoint` procedure (part of `tx`) of TM. TM in turn calls RM at its savepoint callback requesting RM to create a corresponding savepoint for T'. The relationship between a particular savepoint in T and the corresponding savepoint in T' is maintained at TM. A rollback call from T to TM to a particular savepoint of T causes TM to call RM at its rollback callback causing it to rollback to the corresponding savepoint.

23.17 Give three advantages of using an application server architecture in a client server system.

Solution:
Cheaper, more maintainable; more secure; better authorization; more reliable.

23.18 Give an example of an event, different from that given in the text, in which the callback function should not be part of the transaction.

Solution:
An Internet site saves the information about a user even if the user's transaction should abort.

23.19 Explain how peer-to-peer communication can be used to implement remote procedure calling.

Solution:
Send a message requesting some services and wait for the return message that the task was completed.

23.20 Explain some of the authentication issues involved in using your credit card to order merchandise over the Internet.

Solution:

1. How does the merchant know you are the owner of the card.
2. How do you know that the merchant is who it claims to be and is not three students in a dorm room whose main goal is to steal your card number.

23.21 Implement a Web tic-tac-toe game in which the display is prepared by a presentation server on your browser and the logic of the game is implemented within a servlet on the server.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

23.22 Print out the file of cookies for your local Web browser.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

23.23 Consider a three-tiered system interfacing to a centralized DBMS, in which $n1$ presentation servers are connected to $n2$ application servers, which in turn are connected to $n3$ transaction servers. Assume that for each transaction, the application server, on average, invokes k procedures, each of which is executed on an arbitrary transaction server, and that each procedure, on average, executes s SQL statements that must be processed by the DBMS. If, on average, a presentation server handles r requests per second (each request produces exactly one transaction at the application server), how many SQL statements per second are processed by the DBMS and how many messages flow over each communication line?

Solution:
$r * n1 * k * s$ statements per second are processed by the DBMS.

Each line connecting a presentation server to an application server handles $2 * r$ messages per second

Each application server is connected to $n1/n2$ presentation servers and therefore invokes $r * k * (n1/n2)$ procedures per sec. Assuming that an application server is connected to all transaction servers and that it randomly chooses a transaction server to execute a procedure, the number of invocations over a particular line connecting the application server to a transaction server is $r * k * (n1/n2)/n3$, and hence the number of messages over the line is twice that number.

The number of statements transmitted over a line connecting a transaction server to the DBMS is $r * n1 * k * s/n3$, and hence the number of messages over the line is twice that number.

23.24 In Section 23.10 we discuss an optimistic concurrency control that uses a different validation check than the concurrency control described in Section 20.9.2. Give an example of a schedule that would be allowed by the validation check of Section 23.10 but not by the validation check of Section 20.9.2.

Solution:

$$r_1(x), r_2(y), w_2(y), c_2, r_2(y), w_1(y)c_1$$

24

Implementing Distributed Transactions

EXERCISES

24.1 Describe the recovery procedure if a cohort or coordinator crashes at the following states within the two-phase commit protocol:

a. Before the coordinator sends the *prepare message*

Solution:
The transaction aborts.

b. After a cohort has voted but before the coordinator has decided commit or abort

Solution:
If the coordinator crashes, when it recovers, it aborts the transaction. If a cohort crashes, when it recovers, it contacts the coordinator to find out the status

c. After the coordinator has decided to commit but before the cohort has received the *commit message*

Solution:
If the coordinator crashes, when it recovers, it resends commit messages to all the cohorts. If a cohort crashes, when it recovers, it contacts the coordinator to find out the status

d. After the cohort has committed, but before the coordinator has entered the completion record in the log

Solution:
If the coordinator crashes, when it recovers, it resends commit messages to all the cohorts. If a cohort crashes, when it recovers, it does nothing — it has already committed.

24.2 Explain why a cohort does not have to force an abort record to the log during the two-phase commit protocol.

Solution:
The cohort might decide independently to abort (no prepare record is forced in this case) but crashes before the abort record it writes is durable. The coordinator will decide abort whether or not an vote message is received. The standard recovery

procedures at the cohort site will cause the subtransaction to be aborted independent of whether an abort record is found.

The other case which causes an abort record to be written is if the cohort votes ready but receives an *abort* message from the coordinator. If the cohort crashes before the abort record is durable, the presumed abort property will result in the subtransaction being aborted anyway.

24.3 Explain how the fuzzy dump recovery procedure must be expanded to deal with prepared records in the log (assuming the site engages in the two-phase commit protocol).

Solution:
A transaction in the prepared state must be treated as a committed transaction on recovery from media failure. However, items it has updated must be kept locked until the uncertain period completes in case the transaction has been aborted by the coordinator.

24.4 Describe the two-phase commit protocol when one or more of the database managers is using an optimistic concurrency control.

Solution:
The validation procedure is performed after the *prepare* message has been received. If validation is successful the cohort votes ready, otherwise it votes aborting.

24.5 Describe the presumed abort feature in the case in which multiple domains are involved.

Solution:
If an intermediate transaction manager crashes and is restarted and if there is a prepared record but no commit record for some transaction in its log, the transaction record for that transaction is restored to volatile memory. Then any transaction manager can use the usual presumed abort feature. If any cohort in a prepared state asks its transaction manager the state of the transaction, and that transaction manager does not find a transaction record for that transaction in its volatile memory, it can presume that transaction has aborted.

24.6 Describe how the two-phase commit protocol can be expanded to deal with a cohort site that uses a timestamp-ordered concurrency control.

Solution:
The protocol described in the text assumes that cohorts use strict two-phase locking controls and hence that transactions are serialized in commit order. In order to achieve serialization in commit order, when a site using a time-stamp ordered concurrency control receives a prepare message for a transaction T, it checks to see whether T has a time-stamp later than all committed or prepared subtransactions. If so, it votes ready; if not it votes aborting.

24.7 Give schedules of distributed transactions executing at two different sites such that the commit order is different at each site but the global schedule is serializable.

Solution:

Site 1:	$w_1(x)$	$Commit_1$	$w_2(y)$	$Commit_2$	
Site 2:	$w_1(a)$		$w_2(b)$	$Commit_2$	$Commit_1$

24.8 Phase 1 of the *Extended Two-Phase Commit Protocol* is identical to Phase 1 of the two-phase commit protocol we have described. Phase 2 of the extended protocol is as follows:

> *Phase Two.* If the coordinator received at least one "aborting" vote during Phase 1, it decides to abort, deallocates the transaction record from its volatile memory, sends an *abort message* to all cohorts that voted "ready," and terminates the protocol. If all votes are "ready," the coordinator enters its uncertain period, forces a *willing_to_commit record* (which contains a copy of the transaction record) to its log, and sends a *commit message* to each cohort. Note that the transaction is not yet committed.
>
> If a cohort receives an *abort message*, it decides to abort, rolls back any changes it made to the database, appends an abort record to its log, releases all locks, and terminates. If a cohort receives a *commit message*, it decides to commit, forces a commit record to its log, releases all locks, sends a *done message* to the coordinator, and terminates the protocol.
>
> The coordinator waits until it receives the first *done message* from a cohort. Then it decides to commit, forces a commit record to its log, and waits until it receives a *done message* from all cohorts. Then it writes a completion record to its log, deletes the transaction record from its volatile memory, and terminates.

Describe timeout and restart procedures for this protocol. Show that this protocol blocks only when the coordinator and at least one cohort crash (or a partition occurs) or when all cohorts crash. Give an example where the transaction aborts even though all cohorts voted "ready."

Solution:

The uncertain period for a cohort starts when it votes and ends when it receives a *commit* or *abort* message. If a cohort times out or is restarted in its uncertain period, it first sttempts to contact the coordinator (as in the conventional two-phase commit) or one of the cohorts. If any have decided, it makes the same decision. If it can contact all the cohorts and none have decided, all decide abort. (This is the situation where the transaction aborts even though all the cohorts voted *ready*.) If it cannot contact all the cohorts and all cohorts it can contact are undecided, it blocks. (Blocking occurs only when the coordinator and at least one cohort has crashed or there is a partition.)

The uncertain period for the coordinator starts when it sends its first *commit* message and ends when it receives its first *done* message. If the coordinator times out or is restarted in its uncertain period, it checks its log. If it does not find a *willing_to_commit* record, it terminates the protocol. If it does find such a record, it sends *commit* messages to all cohorts listed in the contained transaction record. If a cohort has already decided commit, it responds with a *done* message and the coordinator commits; if it has decided abort it informs the coordinator that it has aborted and the coordinator aborts. If the coordinator cannot contact a cohort that has decided, it blocks. (Blocking occurs only when all cohorts have crashed or there is a partition.)

24.9 Design a logging, timeout, and restart procedure for the linear commit protocol. Do not assume the existence of a separate coordinator module. Assume that all communication between cohorts is carried on along the chain.

Solution:
The rightmost cohort keeps a log in which it stores commit records for distributed transactions. Each cohort forces a prepare record prior to sending a ready message to its neighbor. On recovery, a cohort in its uncertain period sends a query message down the chain. The coordinator uses the presumed abort property in replying to the query.

24.10 Consider a distributed transaction processing system that uses a serial validation optimistic concurrency control at each site and a two-phase commit protocol. Show that deadlocks are possible under these conditions.

Solution:
A subtransaction enters its validation phase when the prepare message arrives. It votes ready or abort based on the result of the validation test, but cannot start the write phase until the commit or abort message arrives in Phase 2 of the commit protocol, preventing other (sub)transactions at that site from entering validation. If T_1 and T_2 both have cohorts at sites A and B and both start the commit protocol at approximately the same time, it is possible that T_{1A} enters validation at site A and T_{2B} enter validation at site B. A deadlock results if

1. T_1 cannot complete, since T_{1A} cannot exit from its validation phase until T_{1B} completes validation and T_{1B} is waiting for T_{2B}.
2. T_2 cannot complete, since T_{2B} cannot exit from its validation phase until T_{2A} completes validation and T_{2A} is waiting for T_{1A}.

24.11 Prove that if all sites use optimistic concurrency controls and if a two-phase commit protocol is used, distributed transactions are globally serializable.

Solution:
The argument is the same as that used to demonstrate global serializability when each site uses a strict two-phase locking protocol.

24.12 If a cohort in a distributed transaction has performed only read operations, the two-phase commit protocol can be simplified. When the cohort receives the *prepare message*, it gives up its locks and terminates its participation in the protocol. Explain why this simplification of the protocol works correctly.

Solution:
The cohort terminates it participation in the protocol before it is determined whether the distributed transaction, T, commits or aborts. Ordinarily, if T aborts, the cohort must rollback the subtransaction, but with a read-only transaction, there are no changes to roll back. It makes no difference whether the cohort releases locks when the prepare message is received or when the protocol terminates (as would be the case without the simplification) because any other transaction that writes an item that the cohort had read would have to follow T in the global ordering in either case.

24.13 Consider the following atomic commit protocol that attempts to eliminate the blocking that occurs in the two-phase commit protocol. The coordinator sends a *prepare message* to each cohort containing the addresses of all cohorts. Each cohort sends its vote directly to all other cohorts. When a cohort receives the votes of all other cohorts it decides to commit or abort in the usual way.

a. Assuming no failures, compare the number of messages sent in this protocol and in the two-phase commit protocol. Assuming that all messages take the same fixed amount of time to deliver, which protocol would you expect to run faster? Explain.

Solution:
Assume there are n cohorts. The two-phase commit protocol uses $4n$ messages and takes 4 message times. The new protocol uses n prepare messages plus $n(n-1)$ vote messages for a total of n^2 messages and takes 2 message times

b. Does the protocol exhibit blocking when failures occur?

Solution:
Yes. If a cohort that has voted ready does not get a vote from another cohort, it is blocked.

24.14 The *kill-wait* concurrency control of Exercise 20.32 is based on locking. When it is used in a distributed system, it is referred to as the *wound-wait* protocol. We assume that a distributed transaction uses RPC to communicate among cohorts so that when the two-phase commit protocol starts, all cohorts have completed. The *kill* primitive is replaced by a *wound* primitive.

> If the cohort of transaction, T_1, at some site makes a request that conflicts with an operation of the cohort of an active transaction, T_2, at that site, then
>
> **if** $TS(T_1) < TS(T_2)$ **then** *wound* T_2 **else** *make* T_1 *wait*

where *wound* T_2 means that T_2 is aborted (as in the *kill-wait* protocol), unless T_2 has entered the two-phase commit protocol, in which case T_1 waits until T_2 completes the protocol.

Explain why this protocol prevents a global deadlock among transactions.

Solution:
Since RPC is the mode of communication, when T_2 has entered the two-phase commit protocol, it has completed at all sites and therefore is not waiting for resources held by T_1 (or any other transaction) at any site. Because the protocol is based on locking, once a transaction enters the two-phase commit protocol it will complete (assuming no failures). Therefore a deadlock is not possible among active transactions.

24.15 Suppose the nested transaction model were extended so that subtransactions were distributed over different sites in a network. At what point in the execution of a distributed nested transaction would a cohort enter the prepared state? Explain your reasoning.

Solution:
If the conditional commit implies that a vote message is sent to the parent then the child must enter the prepared state at that time. Alternatively, if the coordinator polls all subtransactions that have conditionally committed when the nested transaction as a whole completes then the child need not enter the prepared state until it is polled.

24.16 In the text we state that if, in a distributed database system, each site uses a strict two-phase locking concurrency control (all locks are held until commit time) and the system uses a two-phase commit protocol, transactions will be globally serializable.

Does the result also hold if the concurrency controls are not strict—read locks are released early—but are two phase?

Solution:
No. T_{1A} might release a read lock on x at site A and T_{2A} might subsequently write x. A similar situation might exist at site B on item y except that the subtransaction of T_1 and T_2 might be ordered in the opposite way. Hence the serialization graph for T_1 and T_2 has a cycle, but do not delay each other. Both might complete, although their subtransactions are ordered differently at the two sites.

24.17 Explain how to implement synchronous-update replication using triggers.

Solution:
When an update is made to a replica, a trigger is activated. The trigger can be considered immediately or at commit time. Trigger execution updates all other replicas and completes before the transaction commits.

24.18 Design a quorum consensus protocol in which, instead of a timestamp field, each item has a version number field, which is updated whenever the item is written.

Solution:
The protocol is the same as the one in the text except that each item has a version number field, which is incremented every time the item is written. In any read quorum, the replica with the highest version number is the current value of the item.

24.19 Consider a quorum consensus protocol in which an item is stored as five replicas and the size of each read and write quorum is three. Give a schedule that satisfies the following conditions:

> Three different transactions write to the item. Then two of the replica sites fail, leaving only three copies—*all of which contain different values*.

Explain why the protocol continues to execute correctly.

Solution:

T_1:	w_1	w_2	w_3					
T_2:				w_3	w_4	w_5		
T_3:							w_1	w_3	w_5
Site failures							s_1 *fails*		s_3 *fails*

The three remaining copies are at s_2 (which was last updated by T_1), s_4 (which was last updated by T_2), and s_5 (which was last updated by T_3). Each of the copies has a different value, but the value at s_5 was updated last and is the correct value.

24.20 Consider a quorum consensus protocol in which an item is stored as n replicas, and the size of read and write quorums are p and q respectively.

a. What is the maximum number of replica sites that can fail and still have the protocol work correctly?

Solution:
The minimum of $n - p$ and $n - q$.

b. What is the minimum value that p and q can have such that $p = q$?

Solution:
The smallest integer greater than $n/2$

c. Select p and q so that the maximum number of replica sites can fail and still have the protocol work correctly. For this selection, how many sites can fail?

Solution:
p and q are both selected to be the smallest integer greater than $n/2$. The number of sites that can fail is then the largest integer less than $n/2$. For example, if $n = 7$, p and q are both equal to 4, and 3 sites can fail.

24.21 The quorum consensus replication algorithm requires that the timestamps of transactions be consistent with their commit order. In the absence of synchronized clocks, this requirement is not easily met. Propose an alternate algorithm for tagging replicas that can be used with quorum consensus. (*Hint:* Instead of assigning timestamps to transactions, assign version numbers to individual replicas.)

Solution:
When an item is created, each replica is assigned the version number 0. When a transaction writes a new value to an item the new value of the version number of all the replicas in the set is one more than the maximum of all the old version numbers in the set. Since read and write quorums intersect all write quorums, the replica with the maximum version number in any read quorum must have the most recent value of the item.

24.22 Describe an application of replicated data items in which serializability is not needed.

Solution:
Theater Ticketing System in which each site is allocated some seats, and databases are periodically compared and synchronized.

24.23 In what way is the checkbook you keep at home for your checking account like an asynchronous-update replication system?

Solution:
It is asynchronous because the changes you make in the check register at home are not synchronized with the (corresponding) changes the bank makes to your account item in its database. Hopefully, you synchronize the numbers once a month when you get the bank statement.

24.24 Give an example of a nonserializable schedule produced by a primary copy asynchronous-update replication system.

Solution:

T_1:	$w(x_p)$	$w(y_p)$	*commit*			
T_2:				$r(x_p)$	$r(y_s)$	
T_{ru}:						$w(y_s)$

24.25 The following variant of the primary copy asynchronous-update replication protocol has been proposed for totally replicated systems.

a. A transaction executing at site A updates only the replicas at A before it commits (it needed to access no other site since replication is total).
b. After the transaction commits, a second transaction is initiated to update the primary copies of all items updated at A.

c. After the transaction in step (b) has completed, each primary site propagates the update made at that site to all secondaries (including the one at site A). Updates made to a primary copy by several transactions are propagated to secondaries in the order in which the primary was updated.

Explain why, in step (b), all primaries must be updated in a single transaction and, in step (c), the update is propagated to site A.

Solution:
The goal of the protocol is mutual consistency of all replicas. All primaries must be updated in a single transaction to make sure that updates to all primaries made by concurrent transactions occur in the same order. The update must be propagated back to A to make sure that secondary copies at A are updated in the same order as the primaries. For example, suppose sites A and B both have secondary replicas of x and transactions T_1 and T_2 executing at A and B respectively update x concurrently. If in step (c) the primary site did not send updates to all sites, then when the protocol completed the value of x at A would be the value written by T_2 and at B the value written by T_1.

24.26 Explain how triggers can be used to implement primary copy replication.

Solution:
A transaction, T_1, updates only the primary copy of the replicated data. When T_1 commits, a trigger initiates the update of the other replica

24.27 Describe a design of a logging protocol for distributed savepoints.

Solution:
The transaction manager, *TM*, provides a `tx_create_savepoint` procedure that can be called from the application. *TM* knows the identity of all resource managers participating in the transaction. It calls those resource managers using the `xa` interface to request that they each create a local savepoint for the transaction and return the savepoint's identity. *TM* then writes a savepoint record to its log and returns a distributed savepoint identifier to the application. *TM* also provides a `tx_rollback` procedure that can be called by the application. The application provides a distributed savepoint identifier as an argument to the procedure. The procedure locates the savepoint record in its log and calls each resource manager, requesting it to rollback to the appropriate local

24.28 In the presumed commit version of the two-phase commit protocol described in the text, when the protocol starts, the coordinator forces a start record to its log. Describe a situation in which, if the coordinator had not forced this record to its log, the protocol would not work correctly.

Solution:
The coordinator crashes during the first phase of the protocol after sending *prepare* messages to a number of the cohorts. Some of the cohorts that received the message have voted ready and some have voted aborting. After the coordinator crashes and is restarted, it finds no information about the transaction in its log. Later one of the cohorts that had voted ready (and had timed out while waiting for a response) asks the coordinator the state of the transaction. Since the coordinator finds no transaction

record for the transaction in its volatile memory, it reports, incorrectly, that the transaction has committed (some cohorts have already aborted).

k

25

Web Services

EXERCISES

25.1 SOAP extensibility relates to the use of intermediaries and SOAP headers to add features to a Web service that is invoked using SOAP messaging. Give an example of an extension and how an intermediary and header would be used to implement it.

Solution:
SlowHawk might introduce a frequent flyer feature to enhance its business. Previously, a client simply invoked the operation `reserveOp` to make a reservation. The operation's parameters might include the client's name and address and the date and number of the flight. Assuming billing is handled separately by an intermediary fulfilling the role "billing", the client's credit card information would be passed in a header block whose `role` attribute has value "billing".

In the new system `reserveOp` is to be invoked in exactly the same way, but a new intermediary is introduced to handle frequent flyers. Clients wishing to use frequent flyer miles include a new header block whose `role` attribute has value "freqf". The block contains data items such as account number and PIN. Messages are routed through the new intermediary prior to arriving at the billing intermediary. The new intermediary scans the header looking for frequent flyer blocks. If no such block is present the message is simply passed on to the next intermediary and `reserveOp` is invoked as before. If the intermediary finds such a block, it deletes it, looks up the client's account balance and determines if sufficient frequent flyer miles are present. If the answer is yes, it decrements the balance by the appropriate amount and either deletes the header addressed to the billing intermediary or inserts a new header addressed to that intermediary indicating that the client's credit card should not be charged. (In the latter case changes must be made at the billing node to accomodate the new feature.) In either case the final destination is ignorant of the fact the the frequent flyer feature has been added.

25.2 Give the declaration of a Java method that corresponds to the RPC SOAP messages in Figures 25.3 and 25.4.

Solution:

```
public static fareInfo getFareOp(string CustId, string destination,
         date departureDate, date returnDate) {
     .......
      }
```

where `fareInfo` is declared as

```
public class fareInfo {
     string custId
     float tripNo
     float cost
 }
```

25.3 Give an example of a Java object that can be serialized into the parts of the messages in the WSDL`getFareOperation`.

Solution:

```
public class Itinerary {
     string custId
     float tripNo
     string destination
     date departureDate
     date returnDate
     float cost
     .......
 }
```

25.4 Propose a declaration for `receiveResMsg`, and give RPC/literal and document/literal bindings for `hotelCallbackPT`.

Solution:
The following declarations can be used for an RPC/literal binding. `gs` is the prefix assigned to the target namespace of the document containing these declarations and the other declarations relevant to GoSlow (port types, operations)

```
<message name=‘‘receiveResMsg’’>
     <part name=‘‘custId’’ type=‘‘xs.string’’/>
     <part name=‘‘result’’ type=‘‘xs:string’’/>
     <part name=‘‘cost’’ type=‘‘xs:float’’/>
</message>
```

```
<binding name=‘‘hotelCallbackRpcLitBinding’’ type =‘‘gs:hotelCallbackPT> ■
     <soap:binding style=‘‘rpc’’
          transport=‘‘http://schemas.xmlsoap.org/soap/http’’/>
     <operation name=‘‘gs:receiveResOp’’>
          <input>
               <soap:body
                    use=‘‘literal’’
                    namespace=‘‘http://www.goslow.com/wsdl/trips’’/> ■
          </input>
     </operation>
</binding>
```

A multipart message with a document/literal binding must use an element attribute in the part declarations. The element name becomes the tag of the child of `Body` (in the SOAP message) that carries the part. The types of the elements would be as given above (e.g., the type of `costElem` would be `float`).

```
<message name=‘‘receiveResMsg’’>
     <part name=‘‘custId’’ element=‘‘gs:custIdElem’’/>
     <part name=‘‘result’’ element=‘‘gs:resultElem’’/>
     <part name=‘‘cost’’ element=‘‘gs:costElem’’/>
</message>

<binding name=‘‘hotelCallbackDocLitBinding’’ type =‘‘gs:hotelCallbackPT> ■
     <soap:binding style=‘‘document’’
          transport=‘‘http://schemas.xmlsoap.org/soap/http’’/>
     <operation name=‘‘gs:receiveResOp’’>
          <input>
               <soap:body
                    use=‘‘literal’’
                    namespace=‘‘http://www.goslow.com/wsdl/trips’’/> ■
          </input>
     </operation>
</binding>
```

25.5 SlowHawk wants to provide an asynchronous version of its service to give fare information. One asynchronous message requests fare information, and a second asynchronous message provides the fare information. Design WSDL specifications for these two services, and then design a BPEL specification for the business process consisting of these two services.

Solution:
For simplicity we will assume that SlowHawk's only client is GoSlow. Hence, when an asynchronous request for fare information arrives, SlowHawk knows to whom to reply. It does this using a callback port type provided by GoSlow. Let GoSlow's port type be named `asynchRespPT` and its (one-way) operation `asynchRespOp`.

SlowHawk's asynchronous service will be hosted at a new port type:

```
<portType name=‘‘asynchGetFarePT’’>
    <operation name=‘‘asynchGetFareOp’’>
        <input message=‘‘gs:itineraryMsg’’/>
    </operation>
</portType>
```

A new partner link type to connect the two processes through the new port types is required.

```
<plnk:partnerLinkType name=‘‘asynchGetFarePLT’’>
    <plnk:role name=‘‘airline’’>
        <plnk:portType name=‘‘gs:asynchGetFarePT’’/>
    </plnk:role>
    <plnk:role name=‘‘travelAg’’>
        <plnk:portType name=‘‘gs:asynchRespPT’’/>
    </plnk:role>
</plnk:partnerLinkType>
```

SlowHawk must then declare (in its BPEL program) a partner link which asserts its role in the message exchange.

```
<partnerLinks>
    <partnerLink name=‘‘asynchGetFarePL’’
        partnerLinkType=‘‘gs:asynchGetFarePLT’’
        myRole=‘‘airline’’ partnerRole=‘‘travelAg’’/>
</partnerLinks>
```

Finally, the receive/reply pattern of communication used for synchronous requests in SlowHawk is replaced by a receive to accept the request from GoSlow and an invoke to return the reply.

```
<receive partnerLink=‘‘asynchGetFarePL’’
    portType=‘‘asynchGetFarePT’’
    operation=‘‘asynchGetFareOp’’ variable=‘‘asynchGetFareVar’’/>
    <!-- body of handler for asynchronous request goes here -->
<invoke partnerLink=‘‘asynchGetFarePL’’
    portType=‘‘asynchRespPT’’
```

```
        operation=‘‘asynchRespOp’’ inputVariable=‘‘asynchRespVar’’/>
```

25.6 SlowHawk provides versions of the `getFarePT` (Section 25.5) that use the HTTP GET and POST protocols. Show a possible binding for each of these protocols. (You might have to look on the Web to get additional information not in the text.)

Solution:

```
<service name = "GetFareService">
    <port name = "GetFareGETPort" binding="gq:GetFareGETBinding">
        <soap:address location =
            "http://www.SlowHawk.com/fareservice/"/>
    </port>
</service>

<binding name = "GetFareGETBinding" type = "tns:GetFarePT">
    <http:binding verb = "get"
    <operation name = "getFareOperation">
        <http: location = "/getFareLocation"/>
        <input>
            <http:urlEncoded/>
        </input>
        <output>
            <mime:content type = "text/xml"/>
        </output>
    </operation>
</binding>
```

```
<service name = "GetFareService">
    <port name = "GetFarePOSTPort" binding="gq:GetFarePOSTBinding">
        <soap:address location =
            "http://www.SlowHawk.com/fareservice/"/>
    </port>
</service>

<binding name = "GetFarePOSTBinding" type = "tns:GetFarePT">
    <http:binding verb = "POST"
    <operation name = "getFareOperation">
        <http: location = "/postFareLocation"/>
        <input>
            <mime:content type = "application/x-www-form-urlencoded"/>
        </input>
```

```
            <output>
                <mime:content type = "text/xml"/>
            </output>
        </operation>
    </binding>
```

25.7 A stock brokerage company offers a `getQuote` service in which a requester sends a `getQuoteRequest` message containing a stock symbol and the brokerage company responds with a `GetQuoteResponse` message containing the current quotation for that stock. Suppose all that is known is the name of the brokerage company? Describe informally the sequence of UDDI queries you would use to find out the details of the RPC SOAP messages needed to invoke that service.

Solution:

Use `find_business` based on the name to obtain information about that brokerage company. Then use `get_serviceDetail`, `get_bindingDetail`, and `get_tModelDetail` to get the details of the binding

25.8 Two BPEL processes are said to be **compatible** if there are no communication deadlocks: every send in one process is received by the other, and every receive in one process has a corresponding send in the other. Give an example of two noncompatible processes for which every receive in one of the processes has a corresponding send in the other, but there is a communication deadlock.

Solution:

Process A:

```
<receive PartnerLink=''Process B'' ..../>
<invoke PartnerLink=''Process B'' .../>
```

Process B:

```
<receive PartnerLink=''Process A''..../>
<invoke PartnerLink=''Process A''.../>
```

Note that each process waits for a message from the other process. The system is deadlocked.

25.9 Design the outline of a BPEL process that the vendor can use to describe the following business process:

> When you buy something over the Internet, you send the vendor the name, catalog number, and price of the item you want. Then the vendor sends you a form to fill out, containing the name, catalog number, and price of the item and requesting your name, address, credit card number, and expiration date. You return the form to the vendor. The vendor then sends your name, credit card number, and the

price of the item to the credit card company to be approved. After it is approved, the vendor sends you a confirmation message. Then it sends a message to the warehouse to ship the item.

Solution:

Outline of vendor BPEL process

```
<receive PartnerLink=''you''..stuff to buy../>
<invoke PartnerLink=''you''..form to fill out../>
<receive PartnerLink=''you''..form filled out../>
<invoke PartnerLink=''credit card company''..approval request../>
<receive PartnerLink=''credit card company''..approval../>
<invoke PartnerLink=''you''..confirmation message../>
<invoke PartnerLink=''warehouse''..ship the stuff../>
```

25.10 Explain which parts of the business process in the previous example should be part of an ACID transaction (Section 25.8).

Solution:

Sending the approval request to the credit company, receiving thrie reply, sending you the confirmation message, and sending the message to the warehouse to ship the item.

25.11 Design the outlines of BPEL processes for two sites that are performing the following bidding workflow:

Site 1 decides to offer an item for sale at some offering price and sends that offer in a message to site 2. When site 2 receives that message, it decides whether it wants to accept that offer or make a new bid (at some lower price). It sends that information, either new bid or accept, in a message to site 1. When site 1 receives that message, it decides whether it wants to accept that bid or make a new offer (at a higher offering price). It then sends that information in a message to site 2. The bidding continues, sending messages back and forth, until one of the sites, s_a decides to accept a bid or offer and sends that information in a message to the other site, s_b. After sending that accept message, s_a exits the process, and after receiving the accept message, s_b also exits the process.

Solution:

The solution uses the BPEL `while` construct, which we did not discuss in the text. The outline of the processes at sites 1 and 2 are given below. First site 1 sends site 2 an initial bid (in `bid1Var`) and enters a `while` loop. Then, within the loop, site 1 receives a reply from site 2, which either accepts the bid (with the `acceptOp` operation) or makes a new bid (with the `newBidOp` operation containing the value of the new bid in `bid2Var`). If site 2 made a new bid, site 1 sends it to its customer, `cust1`. The customer either accepts the bid or makes a new bid, which site 2 then sends to site 1 and then goes around the while loop again. The two processes use the

following WSDL declarations. The file containing the declarations assigns the prefix `n` to its target namespace.

```
<portType name=‘‘negPT’’>
     <operation name=‘‘bidOp’’>
          <input message=‘‘n:bidMsg’’/>
     </operation>
     <operation name=‘‘acceptOp>
          <input message=‘‘n:acceptMsg’’/>
     </operation>
</portType>

<portType name=‘‘custToNegPT’’>
     <operation name=‘‘custBidOp’’>
          <input message=‘‘n:custBidMsg’’/>
     </operation>
     <operation name=‘‘custAcceptOp>
          <input message=‘‘n:custAcceptMsg’’/>
     </operation>
</portType>

<plnk:partnerLinkType name=‘‘negPLT’’>
     <plnk:role name=‘‘initiator’’>
          <plnk:portType name=‘‘n:negPT’’/>
     </plnk:role>
     <plnk:role name=‘‘negPartner’’>
          <plnk:portType name=‘‘n:negPT’’/>
     </plnk:role>
</plnk:partnerLinkType>

<plnk:partnerLinkType name=‘‘custToNegPLT’’>
     <plnk:role name=‘‘negotiator’’>
          <plnk:portType name=‘‘n:custToNegPT’’/>
     </plnk:role>
     <plnk:role name=‘‘customer’’>
          <plnk:portType name=‘‘c:customerPT’’/>
     </plnk:role>
</plnk:partnerLinkType>
```

The prefix `ng` in the processes refers to the WSDL document containing these declarations. We assume that each customer supports a port type, `customerPT`, with operation `getNextBidOp` that a negotiator can invoke to relay the latest bid received from its negotiating partner. The negotiating process at site 1, the initiator, declares the following partnerLinks:

```
<partnerLinks>
    <partnerLink name=‘‘site2’’ partnerLinkType=‘‘ng:negPLT’’
        myRole=‘‘initiator’’
        partnerRole=‘‘negPartner’’/>
    <partnerLink name=‘‘custToNeg’’
        partherLinkType=‘‘ng:custToNegPLT’’
        myRole=‘‘negotiator’’
        partnerRole=‘‘customer’’/>
</partnerLinks>
```

The following sequence activity is contained in the body of the process at site 1.

```
<sequence>
    <invoke partnerLink=‘‘site2’’ portType=‘‘ng:negPT’’
        operation=‘‘bidOp’’ inputVariable=‘‘bid1Var’’/>
    <assign>
        <copy>
            <from expression = ‘‘’notAccept’’’/>
            <to variable = ‘‘status’’/>
        </copy>
    </assign>
    <while condition = ‘‘status = ’notAccept’’’>
        <pick>
            <onMessage partnerLink=‘‘site2’’
                portType=‘‘ng:negPT’’
                operation=‘‘acceptOp’’ variable=‘‘...’’>
                <assign>
                    <copy>
                        <from expression = ‘‘’accept’’’/>
                        <to variable = ‘‘status’’/>
                    </copy>
                </assign>
            </onMessage>
            <onMessage partnerLink=‘‘site2’’
                portType=‘‘ng:negPT’’
                operation = ‘‘bidOp’’ variable = ‘‘bid2Var’’>
                <sequence>
                    <invoke partnerLink=‘‘custToNeg’’
                        portType=‘‘c:customerPT’’
                        operation=‘‘getNextBidOp’’
                        inputVariable=‘‘bid2Var’’/>
                    <pick>
                        <onMessage partnerLink=‘‘custToNeg’’
                            portType=‘‘ng:custToNegPT’’
```

```
                                        operation=‘‘custAcceptOp’’
                                        variable=‘‘...’’>
                                        <assign>
                                            <copy>
                                                <from expression=‘‘’accept’’’/>
                                                <to variable=‘‘status’’/>
                                            </copy>
                                        </assign>
                                    </onMessage>
                                    <onMessage partnerLink=‘‘custToNeg’’
                                        portType=‘‘ng:custToNegPT’’
                                        operation=‘‘custBidOp’’
                                        variable=‘‘bid1Var’’>
                                        <invoke partnerLink=‘‘site2’’
                                            portType=‘‘ng:negPT’’
                                            operation=‘‘bidOp’’
                                            inputVariable=‘‘bid1Var’’/>
                                    </onMessage>
                                </pick>
                            </sequence>
                        </onMessage>
                    </pick>
                </while>
                <!-- handle the completion of the negotiation -->
            </sequence>
```

The outline of the BPEL process at site 2 is similar except that the roles are reversed and site 2 does not send the initial bid message. The following partner links are declared (once again the prefix `ng` refers to the WSDL document)

```
<partnerLinks>
    <partnerLink name=‘‘site1’’ partnerLinkType=‘‘ng:negPLT’’
        myRole=‘‘negPartner’’
        partnerRole=‘‘initiator’’/>
    <partnerLink name=‘‘custToNeg’’
        partherLinkType=‘‘ng:custToNegPLT’’
        myRole=‘‘negotiator’’
        partnerRole=‘‘customer’’/>
</partnerLinks>
```

and the following sequence activity is contained in the body of the process.

```
<sequence>
```

```
<assign>
     <copy>
          <from expression = ‘‘’notAccept’’’/>
          <to variable = ‘‘status’’/>
     </copy>
</assign>
<while condition = ‘‘status = ’notAccept’’’>
     <pick>
          <onMessage partnerLink = ‘‘site1’’
               portType=‘‘ng:negPT’’
               operation = ‘‘acceptOp’’ variable=‘‘...’’>
               <assign>
                    <copy>
                         <from expression = ‘‘’accept’’’/>
                         <to variable = ‘‘status’’/>
                    </copy>
               </assign>
          </onMessage>
          <onMessage partnerLink=‘‘site1’’
               portType=‘‘ng:negPT’’
               operation=‘‘bidOp’’ variable=‘‘bid1Var’’>
               <sequence>
                    <invoke partnerLink = ‘‘custToNeg’’
                         portType=‘‘c:customerPT’’
                         operation=‘‘getNextBidOp’’
                         inputVariable=‘‘bid1Var’’/>
                    <pick>
                         <onMessage partnerLink=‘‘custToNeg’’
                              portType=‘‘ng:custToNegPT’’
                              operation =‘‘custAcceptOp’’
                              variable=‘‘...’’>
                              <assign>
                                   <copy>
                                        <from expression = ‘‘’accept’’’/>
                                        <to variable = ‘‘status’’/>
                                   </copy>
                              </assign>
                         </onMessage>
                         <onMessage partnerLink=‘‘custToNeg’’
                              portType=‘‘ng:custToNegPT’’
                              operation=‘‘custBidOp’’
                              variable=‘‘bid2Var’’>
                              <invoke partnerLink = ‘‘site1’’
                                   portType=‘‘ng:negPT’’
                                   operation=‘‘bidOp’’
```

```
                                                    inputVariable=‘‘bid2Var’’/>
                                        </onMessage>
                                    </pick>
                                </sequence>
                            </onMessage>
                        </pick>
                    </while>
                    <!-- handle the completion of the negotiation -->
                </sequence>
```

25.12 Explain why an assignment of an endpoint reference to `myRole` of a `partnerLink` makes no sense.

Solution:
The endpoint reference associated with `myRole` should have the address of the BP that contains the declaration of the `partnerLink`. Since this should never change, it makes no sense to assign a different value.

25.13 Show the changes to GoSlow that will be required if customers invoke `makeTripOp` using a one-way pattern and pass an endpoint reference for a callback.

Solution:
`itineraryMsg` has an additional part:

```
<part name=‘‘custERef’’ type=‘‘wsa:EndpointReferenceType’’/>
```

Each customer must support a callback port type to receive a response from GoSlow:

```
<portType name=‘‘custCallbackPT’’>
    <operation name=‘‘custCallbackOp’’>
        <input message=‘‘itineraryRespMsg’’/>
    </operation>
</portType>
```

The partner link type for connecting the customer to Goslow must be changed to indicate that both ends of the connection support port types:

```
<plnk:partnerLinkType name=‘‘custGoSlowLT’’>
    <plnk:role name=‘‘travelService’’>
        <plnk:portType name=‘‘gs:tripPT’’/>
    </plnk:role>
    <plnk:role name=‘‘cust’’>
        <plnk:portType name=‘‘gs:custCallbackPT’’/>
    </plnk:role>
</plnk:partnerLinkType>
```

The partner link declaration in the customer must now declare a value for `myRole` (`myRole=‘‘cust’’` (previously it only needed to declare that the value for `partnerRole` was `travelService`). The partner link declaration in GoSlow becomes:

```
<partnerLink name=‘‘customer’’ partnerLinkType=‘‘custGoSlowLT’’>
    myrole=‘‘travelService’’ partnerRole=‘‘cust’’/>
</partnerLink>
```

Assuming, as in the text, that GoSlow receives the invocation message from the customer in variable `itineraryVar`, prior to responding to the customer Goslow must assign the end point reference contained in `itineraryMsg` to its partner link:

```
<assign>
    <copy>
        <from variable=‘‘itineraryVar’’ part=‘‘custERef’’/>
        <to partnerLink=‘‘customer’’/>
    </copy>
</assign>
```

Finally, the statement in GoSlow that returns a response to the customer becomes an `invoke` (instead of a `reply`):

```
<invoke partnerLink=‘‘customer’’ portType=‘‘custCallbackPT’’
    operation=‘‘custCallbackOp’’ variable=‘‘itineraryRespVar’’/>
```

25.14 Give an example of a business activity (Section 25.8) involving a travel agency in which a `compensate` message will be sent after an activity has completed.

Solution:

The agency asks a number of airlines to make quotes and reservations. It then accepts the lowest quote and sends the other airlines `compensate` messages.

25.15 Scope $S3$ is nested in scope $S2$ which is nested in scope $S1$. Assume that fault handlers for fault f exist in each scope and that compensation handlers $C2$ and $C3$ have been declared in $S2$ and $S3$. In each of the following cases, either explain how the specified sequence of events might happen or why the sequence is impossible

a. Suppose f is raised in $S2$ after $S3$ has exited normally and $C2$ is entered before $C3$.

Solution:
Cannot happen. The only way $C2$ can be entered is as the result of a call from the fault handler in $S1$. But $S2$ must have exited abnormally, so $C2$ is not installed.

b. Suppose f is raised in $S2$ after $S3$ has exited normally and $C3$ is entered.

Solution:
The fault handler in $S2$ calls $C3$.

c. Suppose f is raised in $S1$ after $S2$ has exited normally and $C2$ is entered before $C3$.

Solution:
The fault handler in $S1$ invokes $C2$, which in turn invokes $C3$.

d. Suppose f is raised in $S1$ after $S2$ has exited normally and $C3$ is entered before $C2$.

Solution:
Cannot happen. The only way $C3$ can be entered in this case is via a call from $C2$.

e. Suppose f is raised in $S1$ after $S2$ has exited normally, no handler for f has been declared in $S1$. Describe the sequence of events that occurs.

Solution:
The default fault handler for f in $S1$ will be called. Since $C3$ has been declared in $S2$, it will be called and it might invoke $C3$. The default handler will reraise f in the immediately enclosing scope.

f. Suppose f is raised in $S1$ after $S2$ has exited normally, no handler for f has been declared in $S1$, and no compensation handler has been declared in $S2$. Describe the sequence of events that occurs.

Solution:
The default fault handler for f in $S1$ will be called. Since there is no explicitly declared compensation handler for $S2$ it will reraise f in the immediately enclosing scope.

g. Suppose f is raised in $S1$ after $S2$ has exited normally, no handler for f has been declared in $S1$, and no compensation handler has been declared in $S2$. Describe the sequence of events that occurs.

Solution:
The default fault handler for f in $S1$ will be called. Since there is no explicitly declared compensation handler for $S2$ it will reraise f in the immediately enclosing scope.

25.16 Correlation sets can be associated with `invoke` statements (as well as `receive` statements) using

```
<invoke partnerLink="···" portType="···"
     operation="···" inputVariable="···">
     <correlations>
          <correlation set="···" initiate="yes/no"/>
     <correlations>
</invoke>
```

Give a circumstance under which it might be useful to do this.

Solution:
A customer might initiate an interaction with an instance of service $S1$. In order to service the request the instance in $S1$ requests service from $S2$. A sequence of messages will have to be exchanged between the two instances and the correlation properties

for the sequence are different than those used for correlating the messages between the customer and $S1$. When $S1$ invokes $S2$ for the first time it must initialize the correlation set at its end, so that when (the newly created instance at) $S2$ responds the response will be directed to the correct instance of $S1$.

26

Security and Electronic Commerce

EXERCISES

26.1 Discuss some security issues involved in executing transactions over the Internet.

Solution:
An intruder can eavesdrop on messages, change messages, pretend to be someone else, send false messages.

An intruder can read/change the database information on a server, read/change the information on your browser, deny service by sending a large number of messages to a server.

26.2 Anyone who uses a computer keyboard should be able to easily solve the following simple substitution cipher:

Rsvj ;ryyrt od vjsmhrf yp yjr pmr pm oyd tohjy pm yjr lrunpstf/

Solution:
Each letter is changed to the one on its right on the keyboard.

26.3 Explain why, in general, short encryption keys are less secure than long keys.

Solution:
Because it is easier for an imposter to guess a short key or to try all keys of a particular (short) length.

26.4 Why is it necessary, in the Kerberos protocol, to include S in the message sent from KS to C (i.e., message $M2$)? Describe an attack that an intruder can use if S is not included.

Solution:
I can intercept the first message (C,S) sent in the clear from C to KS and substitute the message (C,I). Thereafter C will think I is S and send private information to I.

26.5 Explain how timestamps are used to defend against a replay attack in a security protocol.

Solution:
The participant will not accept two different messages with the same timestamp.

26.6 Explain how nonces are used to increase security when encrypting messages that are short or that include fields for which the plaintext might be known to an intruder.

Solution:
It is more difficult for the intruder to guess part of the plaintext, which would make it easier to determine the key.

26.7 In a system using public-key cryptography, site B wants to fool C by impersonating A. B waits until A requests to communicate with B. A does this by sending an "I want to communicate" message to B, stating its name (A) and encrypted with B's public key. Then B springs the trap. It sends an "I want to communicate" message to C claiming it is A and encrypted with C's public key. To ensure that it is actually communicating with A, C replies (to B) with a message obtained by encrypting a large random number, N, with A's public key. If C gets a response containing $N + 1$ encrypted with C's public key, it would like to conclude that the responder is A because only A could have decrypted C's message. C gets such a response. However, this conclusion is wrong because the response comes from B. Explain how this could have happened. (*Hint:* The protocol can be corrected if the encrypted text of each message includes the name of the sender.)

Solution:
C replied to B (thinking it was A). When B gets the encrypted message, containing N, it cannot decrypt it, but it sends the message to A using the same protocol. A decrypts the message and encrypts $N + 1$ with B's public key since it is trying to complete the protocol with B. B decrypts the message, re-encrypts it with C's public key, and sends it to C.

If the protocol requires that the encrypted text of each message includes the name of the sender, A would know the message it received from B was constructed by C and not by B.

26.8 Suppose an intruder obtains a copy of a merchant's certificate.

a. Explain why the intruder cannot simply use that certificate and pretend he is the merchant.

Solution:
Because the certificate contains the merchant's public key. Since the intruder does now know the merchant's private key, it cannot decrypt any message sent to the merchant.

b. Explain why the intruder cannot replace the merchant's public key with his own in the certificate.

Solution:
Because the intruder cannot sign this fake certificate using the CA's private key.

26.9 Suppose that you use the SSL protocol and connect to a merchant site, M. The site sends you M's certificate. When the SSL protocol completes, how can you be sure that the new session key can be known only to M (perhaps an intruder has sent you a copy of M's certificate)? Can you be sure that you are connected to M?

Solution:
Only M can decrypt the session key you have created since you encrypted it with M's

public key obtained from a valid certificate. Although you cannot be sure that you are connected to M, you can find out by sending a message encrypted with the new session key. If the reply to that message implies that the site decrypted your message, the site must be M.

26.10 Using your local Internet Browser

a. Describe how you can tell when you are connected to a site that is using the SSL protocol.

Solution:
The address has the prefix https and there is a closed lock in the corner of the screen.

b. Suppose you are connected to a site that is using the SSL protocol. Describe how you can determine the name of the CA that supplied the certificate used by that site.

Solution:
Click on the closed lock.

c. Determine how many bits are in the keys that your browser uses for SSL encryption.

Solution:
Most browsers use 128 bits. That can also be determined by first clicking on the closed lock.

26.11 Suppose that you obtained a certificate of your own. Explain how you could use that certificate to deal with situations in which an intruder might steal your credit card number.

Solution:
You use a protocol that requires that a merchant accept a credit card only if it comes with a certificate and that the credit card number is always encrypted with your private key. Thus only you, and not an intruder, could have sent it. Effectively, you are signing your credit card number.

26.12 Suppose that an intruder puts a virus on your computer that alters your browser. Describe two different ways that the intruder could then impersonate some server site S that you might attempt to communicate with—even though you use the SSL protocol—and obtain your credit card number.

Solution:
Substitute the intruder's public key for one of the CA's and then forge and use a certificate for S signed with the intruder's public key. Alternately, cause the browser to encrypt all messages with the intruder's public key.

26.13 A merchant using the SSL protocol (without SET) might implement a credit card transaction as follows: The customer purchases an item, and the merchant asks him to send his credit card information encrypted using the session key established with the SSL protocol. When the merchant receives that information, she initiates a separate transaction with the credit card company to have the purchase approved. When that transaction commits, the merchant commits the transaction with the customer.

Explain the similarities and differences of this protocol to SET.

Solution:
Similarities:

1. Using a certificate to authenticate the server;
2. Having a trusted third party.
3. Sending the credit card in encrypted form, so an intruder cannot read it.

Differences:

1. In SET, the merchant does not get to see the credit card number.
2. In SET, each customer has its own certificate; but In SSL, customer does not have its own certificate;
3. The automic commit protocol used by SET is the linear commit protocol; while SSL does not use an atomic commit protocol.
4. In the SSL only protocol, the merchant could cheat and specify an arbitrary charge in its transaction with the credit card company.

26.14 Assume that the merchant in the SET protocol is dishonest. Explain why he cannot cheat the customer.

Solution:
The merchant cannot cheat the customer because the dual signature prevents him from changing the details of the order, determining the credit card number, or submitting the same charges to the credit card company again.

26.15 Explain why a trusted third party is used in the certified delivery protocol.

Solution:
To recover from a situation in which the merchant does not send the key, sends the wrong key, or sends the wrong goods.

26.16 Describe a restart procedure that the merchant's computer can use to deal with crashes during the SET protocol.

Solution:
When the merchant restarts, if there is no prepare or commit record in its log, it aborts; if there is a prepare record, but no commit record, it asks the gateway what happened. If there is a commit record, but no complete record, it sends a commit record to the customer.

26.17 Explain why MD_2 in the SET protocol (Section 26.11) must be a part of the dual signature.

Solution:
When G wants to verify that m_1 was prepared by C it does not have m_2. Hence, without MD_2 it cannot compute $f(MD_1 \circ MD_2)$.

26.18 Explain why a forger could not simply submit an arbitrary random number as a token in the electronic cash protocol.

Solution:
When the bank decrypted it, it would not satisfy $r()$

26.19 Assume that, in the anonymous electronic cash protocol, the bank is honest but the customer and the merchant might not be.

a. After receiving the tokens from the bank, the customer later claims that she never received them. Explain what the bank should then do and why it is correct.

Solution:
The bank sends the customer another copy of the tokens. Even if the customer is dishonest and did receive the tokens the first time, she cannot spend the same tokens twice.

b. After receiving the message containing the purchase order and the tokens from the customer, the merchant claims never to have received the message. Explain what the customer should do and why.

Solution:
Resend the message. The merchant cannot deposit the tokens twice.

26.20 Describe the methods used in each of the following protocols to prevent a replay attack:

a. Kerberos authentication

Solution:
Authenticator

b. SET

Solution:
Transaction number

c. Electronic cash

Solution:
The bank keeps a list of tokens that have been deposited and will not deposit the same token twice.

A

An Overview of Transaction Processing

EXERCISES

A.1 State which of the following schedules are serializable.

a. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_2(x)\ r_1(y)$

Solution:
Yes. all transactions just read.

b. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_2(x)\ r_1(y)$

Solution:
No. Transaction T_1 is before Transaction T_2 on x and after Transaction T_2 on y.

c. $r_1(x)\ w_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ r_2(y)$

Solution:
Yes. Transaction T_1 and Transaction T_2 write and read different variables.

d. $r_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ w_1(x)\ w_2(y)$

Solution:
Yes. Transaction T_1 and Transaction T_2 write and read different variables.

e. $w_1(x)\ r_2(y)\ r_1(z)\ r_3(z)\ r_1(x)\ w_2(y)$

Solution:
Yes. Transaction T_1 and Transaction T_2 write and read different variables.

A.2 In the Student Registration System, give an example of a schedule in which a deadlock occurs.

Solution:
Consider the concurrent execution of a registration transaction, T_1, and a deregistration transaction, T_2, for the same course and assume the DBMS uses page locking:

T_2 first deletes the row in Transcript describing the student registered in that course and then decrements CurReg in Course. The registration transaction, T_1, first updates Course, and then inserts a tuple in Transcript.

The schedule is:

$$w_2(x)\ w_1(y)\ w_2(y)\ w_1(x)$$

A deadlock might result since T_1 and T_2 obtain their locks in the opposite order.

A.3 Give an example of a schedule that might be produced by a nonstrict two-phase locking concurrency control that is serializable but not in commit order.

Solution:

$$w_1(x)\ r_2(x)\ w_2(z)\ commit_2\ commit_1$$

Transaction T_2 is after transaction T_1.

A.4 Give an example of a transaction processing system (other than a banking system) that you have interacted with, for which you had an intuitive expectation that the serial order was the commit order.

Solution:
Airlines reservation system, or credit card system.

A.5 Suppose that the transaction processing system of your university contains a table in which there is one tuple for each current student.

a. Estimate how much disk storage is required to store this table.

Solution:
10,000 tuples * 100 bytes/tuple = 1 megabyte.

b. Give examples of transactions in the student registration system that have to lock this entire table if a table locking concurrency control is used.

Solution:
Find all students who are registered in at least one course.

A.6 Give an example of a schedule at the READ COMMITTED isolation level in which a lost update occurs.

Solution:
Suppose a transaction reads the value of an item, x, and, on the basis of the value read, writes a new value back to x. A deposit transaction in a banking system has this pattern of behavior, where x is the balance of the account being credited. At the READ COMMITTED isolation level, such a deposit transaction releases its read lock before acquiring a write lock, two deposit transactions on the same account can be interleaved as illustrated in the following schedule

$$r_1(x)\ r_2(x)\ w_2(x)\ commit_2\ w_1(x)\ commit_1$$

The effect of T_2 is lost since the value written by T_1 is based on the original value of x, rather than the value written by T_2. Thus if T_1 was attempting to deposit $5 and T_2 was attempting to deposit $10, the final database has only been incremented by $5, so T_2's update has been lost.

A.7 Give examples of schedules at the REPEATABLE READ isolation level in which a phantom is inserted after a SELECT statement is executed and

a. The resulting schedule is nonserializable and incorrect.

Solution:

b. The resulting schedule is serializable and hence correct.

Solution:

A.8 Give examples of schedules at the SNAPSHOT isolation that are

a. Serializable and hence correct.

Solution:

$$r_1(x)\ r_1(y)\ r_2(y)\ r_2(z)\ w_1(x)\ w_2(z)$$

is serializable in either order.

b. Nonserializable and incorrect.

Solution:

$$r_1(x)\ r_1(y)\ r_2(x)\ r_2(y)\ w_1(y)\ w_2(x)$$

is not serializable. T_1 and T_2 read the same snapshot, but since they write different data items, they are both allowed to commit.

A.9 Give examples of schedules that would be accepted at

a. SNAPSHOT isolation but not REPEATABLE READ.

Solution:

$$r_1(x)\ r_1(y)\ w_1(x)\ r_2(y)\ r_2(x)\ w_1(y)\ commit_2\ commit_1$$

A deadlock would occur at REPEATABLE READ.

b. SERIALIZABLE but not SNAPSHOT isolation. (*Hint:* T_2 performs a write after T_1 has committed.)

Solution:

$$r_1(x)\ r_1(y)\ r_2(y)\ w_1(x)\ commit_1\ r_2(x)\ w_2(x)\ commit_2$$

This schedule is serializable, but would not be allowed in SNAPSHOT isolation, because both concurrent transactions updated x.

A.10 Give an example of a schedule of two transactions in which a two-phase locking concurrency control

a. makes one of the transactions wait, but a control implementing SNAPSHOT isolation aborts one of the transactions.

Solution:

$$r_1(x)\ w_1(x)\ r_2(y)\ w_2(x)\ commit_2$$

A two-phase locking concurrency control would make T_2 wait, but a SNAPSHOT isolation control would abort T_1, since T_2 was the first committer.

b. aborts one of the transactions (because of a deadlock), but a control implementing SNAPSHOT isolation allows both transactions to commit.

Solution:

$$r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)$$

A.11 A particular read-only transaction reads data entered into the database during the previous month and uses it to prepare a report. What is the weakest isolation level at which this transaction can execute? Explain.

Solution:
READ UNCOMMITTED, because no other transaction can change the data.

A.12 What intention locks must be obtained by a read operation in a transaction executing at REPEATABLE READ when the locking implementation given in Section A.1.5 is used?

Solution:
An IS lock on the table (and an S lock on the tuples returned)

A.13 Explain how the commit of a transaction is implemented within the logging system.

Solution:
When a transaction commits, it writes a commit record to the log. If it aborts, it rolls back its updates and then writes an abort record to the log. Using these records, the backward scan can record the identity of transactions that completed prior to the crash, and ignore their update records as they are subsequently encountered. If, during the backward scan, the first record relating to T is an update record, T was active when the crash occurred and must be aborted.

A transaction is committed if and only if its commit record has been successfully appended to the log.

A.14 Explain why the write-ahead feature of a write-ahead log is needed.

Solution:
The write-ahead feature of a write-ahead log means the update record is appended to the log before the database is updated.

Suppose that the database is updated before the update record is appended to the log. If the system should crash between the update and the log entry, the recovery process has no way of rolling the transaction back. If, on the other hand, the update record is appended first, this problem is avoided.

A.15 Explain why a cohort in the two-phase commit protocol cannot release locks acquired by the subtransaction until its uncertain period terminates.

Solution:
The cohort must block because the coordinator might decide (or might have already decided) commit or abort, and the cohort must make the same decision.

A.16 Two distributed transactions execute at the same two sites. Each site uses a strict two-phase locking concurrency control, and the entire system uses a two-phase commit protocol. Give a schedule for the execution of these transactions in which the commit order is different at each site but the global schedule is serializable.

Solution:

Site 1:	$w_1(x)$	$Commit_1$		$w_2(y)$		$Commit_2$	
Site 2:	$w_1(a)$		$w_2(b)$		$Commit_2$		$Commit_1$

It takes a long time for the commit message for transaction 1 to get to Site 2

A.17 Give an example of an incorrect schedule that might be produced by an asynchronous-update replication system.

Solution:
Transaction T_1 updates replicas x_1 and x_2 and replicas y_1 and y_2 asynchronously. T_2 reads x_1 after the T_1's update is made and y_1 before T_1's update is made, thus seeing an inconsistent database.

A.18 Explain how to implement synchronous-update replication using triggers.

Solution:
When an update is made to a replica, a trigger is fired, which updates all the other replicas before the transaction commits.

B

Requirements and Specifications

EXERCISES

B.1 Prepare a Requirements Document for a simple calculator.

Solution:

1. Objectives:
   a. Use buttons to input numbers and designate operator to perform calculations on those numbers
   b. Display the outcome of the calculations on a screen
2. Information to be contained in the system:
   a. Buttons representing the 10 numerals. a decimal point, the operators, +, -, ,∗, /, sqrt, and =.
   b. Screen that can display 10 numerals, a decimal point between any two numerals, and a minus sign
   c. On/off button
   d. Internal storage to store intermediate results.
3. Interactions with the System
   a. When the system is turned on, the display shows 0.
   b. The user can input up to ten numerals with or without a decimal point, and the resulting number is shown on the screen
   c. The user can input an operator
   If the operator is sqrt, the result is displayed on the screen
   d. The user can input up to ten numerals with or without a decimal point the the resulting number is shown on the screen

   The remainder is similar and is omitted.
4. Error Situations
   a. More than 10 numerals are entered by the user. The system . . .
   b. The result of an operation is more than 10 numerals. The system . . .
   c. The result of an operation is too small (less than can be expressed with 10 zeros followed by a decimal). The system . . .
   d. The user attempts to take the sqrt of a negative number. The system . . .

   The remainder is similar and is omitted.

B.2 According to the Requirements Document for the Student Registration System, one session can include a number of use cases. Later, during the design, we will decompose each use case into one or more transactions. The ACID properties apply to all transactions, but a session that involves more than one transaction might not be isolated or atomic. For example, the transactions of several sessions might be interleaved. Explain why the decision was made not to require sessions to be isolated and atomic. Why is a session not one long transaction?

Solution:
Locks would be held too long, delaying other people. Thus the interaction with the user is not part of the transaction. After the user inputs the information required to perform one activity (for example, register for a course), a transaction is executed to perform that activity. If the user then inputs additional information to perform another activity, another transaction is executed to perform that activity. Thus the two activities are not executed as one atomic unit, but this would be perfectly acceptable for this application.

B.3 Suppose that the database in the Student Registration System satisfies all the integrity constraints given in Section IV of the Requirements Document Outline (Section B.2). Is the database necessarily correct? Explain.

Solution:
No. For example a student might have registered for a course and the database was not updated at all! The database is consistent because it was consistent before the registration transaction started, and it was not changed at all during the registration transaction — but the database is not correct, because it does not contain any information about the student that just registered.

B.4 The Requirements Document for the Student Registration System does not address security issues. Prepare a section on security issues, which might be included in a more realistic Requirements Document.

Solution:
Some security issues – assuming passwords are used:

1. How do users get passwords
2. What if a user forgets her password
3. Are the passwords sent over the network in encrypted form
4. How are the user's passwords stored in the database — are they encrypted?
5. How does the system remember between use cases that the user has logged in with her password.
6. Does the information sent by transactions have to be encrypted — the password information, other information?

B.5 In the resolution of issue 2 in Section B.3, the statement was made that new prerequisites do not apply to courses offered in the next semester. How can a Registration Transaction know whether or not a prerequisite is "new"?

Solution:
A date-added attribute can be added to each prerequisite.

B.6 Suppose that the Student Registration System is to be expanded to include graduation clearance. Describe some additional items that must be stored in the database. Describe some additional integrity constraints.

Solution:
New items might include the graduation requirements and the clearance status of each student. Perhaps the graduation requirements for a student depend on the year the student started at the school. In that case, the new items might be the year the student started at the school, and the graduation requirements that were in effect in each year. An integrity constraint might be that the status of a student is 'cleared' only if that student satisfies the graduation clearance requirements.

B.7 Prepare a Specification Document for a simple calculator.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

B.8 Prepare a Specification Document for the controls of a microwave oven.

Solution:
This is a hands-on problem for which the solution is not meant to be given in this manual.

B.9 Specify a use case for the Student Registration System that assigns a room to a course to be taught next semester.

Solution:
Assign Room.
Purpose. Assign a room for a course to be taught next semester.
Actor. A professor (actually the Director of the program offering the course).
Input. A course number and a room number.
Result. The room is assigned to the course, and an appropriate message is displayed. If a room had previously been assigned to the course, the new assignment shall replace the old assignment.
Exception. If the capacity of the room is less than the maximum allowed enrollment for the course, the room assignment shall not take place and an appropriate message shall be displayed.

C

Design, Coding, and Testing

EXERCISES

C.1 Prepare a Design Document and Test Plan for a simple calculator.

C.2 Explain why

a. Black box testing cannot usually test all aspects of the specifications.

Solution:
There are too many things to test. For example, if the specification involves integers, we cannot test for all possible integers

b. Glass box testing cannot usually test all execution paths through the code. (This does not mean that glass box testing cannot visit all lines and visit all branches of the code.)

Solution:
There are too many paths. For example, if there are N two-way branches, there are 2^N paths.

C.3 Explain why concurrent systems (such as operating systems) are difficult to test. Explain why transaction processing applications, even though they are concurrent, do not have these same difficulties.

Solution:
There are an astronomical number of interleavings, which are hard to replicate, so if the system fails it is difficult to reproduce the situation that caused the failure.
The TP Monitor ensures that all schedules are atomic, isolated, and durable, and hence these properties do not have to be tested.

C.4 Explain the advantages of incremental system development from the viewpoint of

a. The managers of the enterprise sponsoring the project

Solution:
The managers have a working version of the system in a short time. They can evaluate its usefulness and suggest changes for the next iteration.

b. The project manager

Solution:
The project manager can make a more realistic schedule when only a relatively

small part of the project is to be built. The project manager can more efficiently deal with the changes the managers will suggest. It is easier to test the system in incremental steps. Risk is minimized.

c. The implementation team

Solution:
The implementation team can more easily prepare the required documentation for the project in incremental steps. Design, coding, and testing are all easier.

C.5 In the design of the registration transaction given in Section C.7.2, some of the required checks are performed in the schema and some in the transaction program.

a. Which of the checks performed in the program can be performed in the schema?

Solution:
All of the checks can be performed in the schema.

1. The course is offered the following semester.

2. The student is not already registered for the course, is not currently enrolled in the course, and has not completed the course with a grade of C or better.

3.The student is not already registered for another course scheduled at the same time.

4. The total number of credits taken by the student the following semester will not exceed 20.

5. The student has completed all of the prerequisites for the course with a grade of C or better (or is currently enrolled in some prerequisites).

b. For each such check, change the schema to perform that check.

C.6 Rewrite the program for the registration transaction in Section C.7.3

a. Using stored procedures for the procedures that perform the registration checks
b. Using one stored procedure that performs all of the checks that are performed by individual procedures in the figure
c. In C and embedded SQL
d. In C and ODBC

C.7 Evaluate the coding style used in the program for the registration transaction in Section C.7.3.

C.8 Prepare a test plan for the registration transaction given in Section C.7.2.

C.9 For the deregistration transaction in the Student Registration System

a. Prepare a design.
b. Write a program.
c. Prepare a test plan.