CHAPTER 1

Introduction

Solutions to Practice Exercises

1.1 Two disadvantages associated with database systems are listed below.

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

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

1.2 Programming language classification:

- Procedural: C, C++, Java, Basic, Fortran, Cobol, Pascal
- Non-procedural: Lisp and Prolog

Note: Lisp and Prolog support some procedural constructs, but the core of both these languages is non-procedural.

In theory, non-procedural languages are easier to learn, because they let the programmer concentrate on *what* needs to be done, rather than *how* to do it. This is not always true in practice, especially if procedural languages are learned first.

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

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

- Create/initialize the database.

1.4 Let *tgrid* be a two-dimensional integer array of size $n \times m$.

a.
- The physical level would simply be $m \times n$ (probably consecutive) storage locations of whatever size is specified by the implementation (e.g., 32 bits each).
- The conceptual level is a grid of boxes, each possibly containing an integer, which is *n* boxes high by *m* boxes wide.
- There are $2^{m \times n}$ possible views. For example, a view might be the entire array, or particular row of the array, or all *n* rows but only columns 1 through *i*.

b.
- Consider the following Pascal declarations:

  type *tgrid* = **array**[1..*n*, 1..*m*] **of integer**;
  var *vgrid1*, *vgrid2* : *tgrid*

  Then *tgrid* is a schema, whereas the value of variables *vgrid1* and *vgrid2* are instances.
- To illustrate further, consider the schema **array**[1..2, 1..2] **of integer**. Two instances of this scheme are:

1	16
7	89

17	90
412	8

CHAPTER 2

Relational Model

Solutions to Practice Exercises

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

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

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

If people may not work for any company:

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

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

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

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

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

2.3 **a.**
$$employee \leftarrow \Pi_{person\_name, street, ''Newtown''}(\sigma_{person\_name=\text{“Jones”}}(employee)) \cup (employee - \sigma_{person\_name=\text{“Jones”}}(employee))$$

b. The update syntax allows reference to a single relation only. Since this update requires access to both the relation to be updated (*works*) and the *manages* relation, we must use several steps. First we identify the tuples of *works* to be updated and store them in a temporary relation (t_1). Then we create a temporary relation containing the new tuples (t_2). Finally, we delete the tuples in t_1, from *works* and insert the tuples of t_2.

$$t_1 \leftarrow \Pi_{works.person\_name, company\_name, salary}(\sigma_{works.person\_name = manager\_name}(works \times manages))$$
$$t_2 \leftarrow \Pi_{person\_name, company\_name, 1.1 * salary}(t_1)$$
$$works \leftarrow (works - t_1) \cup t_2$$

CHAPTER 3

SQL

Solutions to Practice Exercises

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

a. Note: this is not the same as the total number of accidents in 1989. We must count people with several accidents only once.

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

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

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

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

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

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

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

3.2 **a.** Query:

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

b. If people may work for several companies, the following solution will only list those who earn more than $10,000 per annum from "First Bank Corporation" alone.

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

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

c. The following solution assumes that all people work for exactly one company.

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

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

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

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

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

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

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

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

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

Below is a solution using standard SQL.

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

f. Query:

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

g. Query:

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

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

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

b. Query:

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

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

SQL-92 provides a **case** operation (see Exercise 3.5), using which we give a more concise solution:

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

3.4 Query:

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

3.5 We use the **case** operation provided by SQL-92:

a. To display the grade for each student:

```
select student_id,
       (case
                when score < 40 then 'F',
                when score < 60 then 'C',
                when score < 80 then 'B',
                else 'A'
        end) as grade
from marks
```

b. To find the number of students with each grade we use the following query, where *grades* is the result of the query given as the solution to part 0.a.

select *grade*, **count**(*student_id*)
from *grades*
group by *grade*

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

3.7 To insert the tuple ("Johnson", 1900) into the view *loan_info*, we can do the following:

$borrower \leftarrow (\text{"Johnson"}, \perp_k) \cup borrower$

$loan \leftarrow (\perp_k, \perp, 1900) \cup loan$

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

CHAPTER 4

Advanced SQL

Solutions to Practice Exercises

4.1 Query:

```
create table loan
    (loan_number     char(10),
     branch_name     char(15),
     amount          integer,
     primary key (loan_number),
     foreign key (branch_name) references branch)


create table borrower
    (customer_name  char(20),
     loan_number    char(10),
     primary key (customer_name, loan_number),
     foreign key (customer_name) references customer,
     foreign key (loan_number) references loan)
```

Declaring the pair *customer_name, loan_number* of relation *borrower* as primary key ensures that the relation does not contain duplicates.

4.2 Query:

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

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

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

```
ppcreate table manages
    (person_name    char(20),
     manager_name  char(20),
     primary key (person_name),
     foreign key (person_name) references employee,
     foreign key (manager_name) references employee)
```

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

a. check condition for the *works* table:

```
check((employee_name, company_name) in
        (select  e.employee_name, c.company_name
         from    employee e, company c
         where   e.city = c.city
        )
)
```

b. check condition for the *works* table:

```
check(
        salary < all
                (select   manager_salary
                 from     (select manager_name, manages.employee_name as emp_name,
                                  salary as manager_salary
                           from works, manages
                           where works.employee_name = manages.manager_name)
                 where    employee_name = emp_name
                )
)
```

The solution is slightly complicated because of the fact that inside the **select** expression's scope, the outer *works* relation into which the insertion is being performed is inaccessible. Hence the renaming of the *employee_name* attribute to *emp_name*. Under these circumstances, it is more natural to use assertions.

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

4.4 The assertion_name is arbitrary. We have chosen the name *perry*. Note that since the assertion applies only to the Perryridge branch we must restrict attention to only the Perryridge tuple of the *branch* relation rather than writing a constraint on the entire relation.

```
create assertion perry check
   (not exists (select *
                from branch
                where branch_name = 'Perryridge' and
                      assets ≠ (select sum (amount)
                                from loan
                                where branch_name = 'Perryridge')))
```

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

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

CHAPTER 5

Other Relational Languages

Solutions to Practice Exercises

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

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

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

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

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

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

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

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

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

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

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

i.

r	*A*	*B*
	P.	*17*

ii. $query\,(X)$:- $r\,(X,\ 17)$

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

i.

r	*A*	*B*
	_*a*	_*b*

s	*A*	*C*
	_*a*	_*c*

result	*A*	*B*	*C*
P.	_*a*	_*b*	_*c*

ii. *query(X, Y, Z) :- r(X, Y), s(X, Z)*

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

i.

r	*A*	*B*
	_*a* _*c*	>_*s* _*s*

s	*A*	*C*
	P._*a*	_*c*

ii. $query\,(X)$:- $s\,(X,\, Y),\ r\,(X,\, Z),\ r\,(Y,\, W),\ Z > W$

5.4 **a.** Query:

$query\,(X)$:- $p\,(X)$
$p\,(X)$:- $manages\,(X,$ "Jones"$)$
$p\,(X)$:- $manages\,(X,\, Y),\ p\,(Y)$

b. Query:

query(X, C) :- p(X), employee(X, S, C)
p(X) :- manages(X, "Jones")
p(X) :- manages(X, Y), p(Y)

c. Query:

query(X, Y) :- p(X, W), p(Y, W)
p(X, Y) :- manages(X, Y)
p(X, Y) :- manages(X, Z), p(Z, Y)

d. Query:

query(X, Y) :- *p(X, Y)*
p(X, Y) :- *manages(X, Z), manages(Y, Z)*
p(X, Y) :- *manages(X, V), manages(Y, W), p(V, W)*

5.5 A Datalog rule has two parts, the *head* and the *body*. The body is a comma separated list of *literals*. A *positive literal* has the form $p(t_1,\ t_2, \ldots,\ t_n\ )$ where p is the name of a relation with n attributes, and $t_1,\ t_2, \ldots,\ t_n$ are either constants or variables. A *negative literal* has the form $\neg p(t_1,\ t_2, \ldots,\ t_n)$ where p has n attributes. In the case of arithmetic literals, p will be an arithmetic operator like $>$, $=$ etc.

We consider only safe rules; see Section 5.4.4 for the definition of safety of Datalog rules. Further, we assume that every variable that occurs in an arithmetic literal also occurs in a positive non-arithmetic literal.

Consider first a rule without any negative literals. To express the rule as an extended relational-algebra view, we write it as a join of all the relations referred to in the (positive) non-arithmetic literals in the body, followed by a selection. The selection condition is a conjunction obtained as follows. If $p_1\,(X,\ Y),\ p_2\,(Y,\ Z)$ occur in the body, where p_1 is of the schema $(A,\ B)$ and p_2 is of the schema $(C,\ D)$, then $p_1.B\ =\ p_2.C$ should belong to the conjunction. The arithmetic literals can then be added to the condition.

As an example, the Datalog query

query(X, Y) :- *works(X, C, S1), works(Y, C, S2), S1 > S2, manages(X, Y)*

becomes the following relational-algebra expression:

$$E_1\ =\ \sigma_{(w1.company\_name = w2.company\_name\ \wedge\ w1.salary > w2.salary\ \wedge\ manages.person\_name = w1.person\_name\ \wedge\ manages.manager\_name = w2.person\_name)} (\rho_{w1}(works)\ \times\ \rho_{w2}(works)\ \times\ manages)$$

Now suppose the given rule has negative literals. First suppose that there are no constants in the negative literals; recall that all variables in a negative literal must also occur in a positive literal. Let $\neg q(X, Y)$ be the first negative literal, and let it be of the schema $(E,\ F)$. Let E_i be the relational algebra expression obtained after all positive and arithmetic literals have been handled. To handle this negative literal, we generate the expression

$$E_j\ \ =\ \ E_i \bowtie (\Pi_{A_1, A_2}(E_i) - q)$$

where A_1 and A_2 are the attribute names of two columns in E_i which correspond to X and Y respectively.

Now let us consider constants occurring in a negative literal. Consider a negative literal of the form $\neg q(a, b, Y)$ where a and b are constants. Then, in the above expression defining E_j we replace q by $\sigma_{A_1 = a \wedge A_2 = b}(q)$.

Proceeding in a similar fashion, the remaining negative literals are processed, finally resulting in an expression E_w.

Finally the desired attributes are projected out of the expression. The attributes in E_w corresponding to the variables in the head of the rule become the projection attributes.

Thus our example rule finally becomes the view:

create view *query* **as**
$$\Pi_{w1.person\_name,\, w2.person\_name}(E_2)$$

If there are multiple rules for the same predicate, the relational-algebra expression defining the view is the union of the expressions corresponding to the individual rules.

The above conversion can be extended to handle rules that satisfy some weaker forms of the safety conditions, and where some restricted cases where the variables in arithmetic predicates do not appear in a positive non-arithmetic literal.

CHAPTER 6

Database Design and the E-R Model

Solutions to Practice Exercises

6.1 See Figure 6.1

6.2 See Figure 6.2.
In the answer given here, the main entity sets are *student*, *course*, *course_offering*, and *instructor*. The entity set *course_offering* is a weak entity set dependent on *course*. The assumptions made are :

a. A class meets only at one particular place and time. This E-R diagram cannot model a class meeting at different places at different times.
b. There is no guarantee that the database does not have two classes meeting at the same place and time.

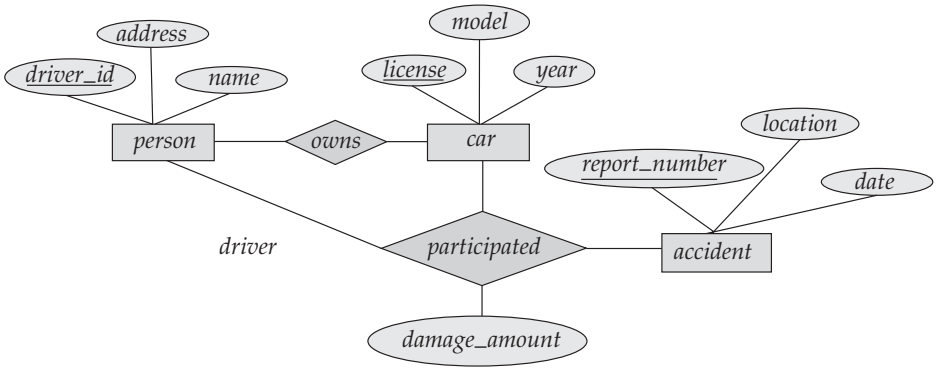


Figure 6.1 E-R diagram for a car insurance company.

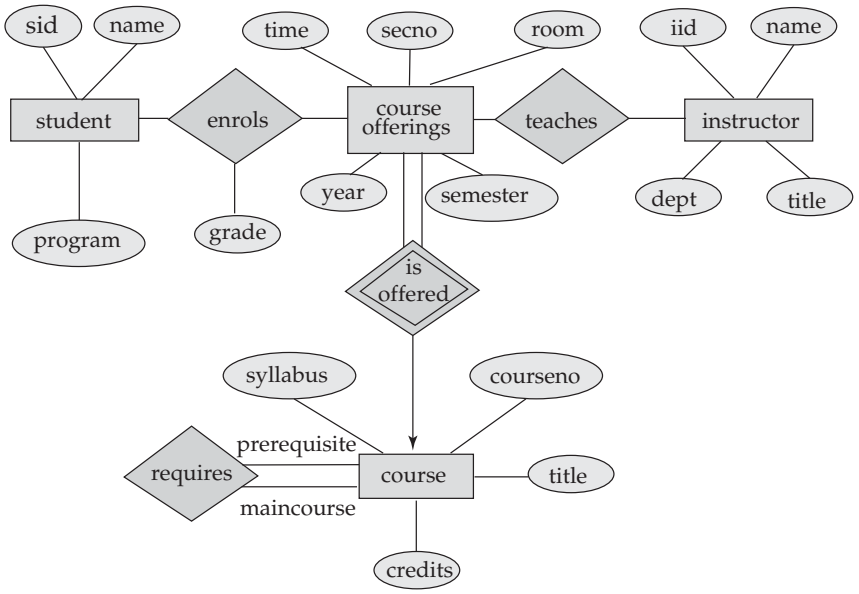


Figure 6.2 E-R diagram for a university.

6.3 **a.** See Figure 6.3
b. See Figure 6.4

6.4 See Figure 6.5

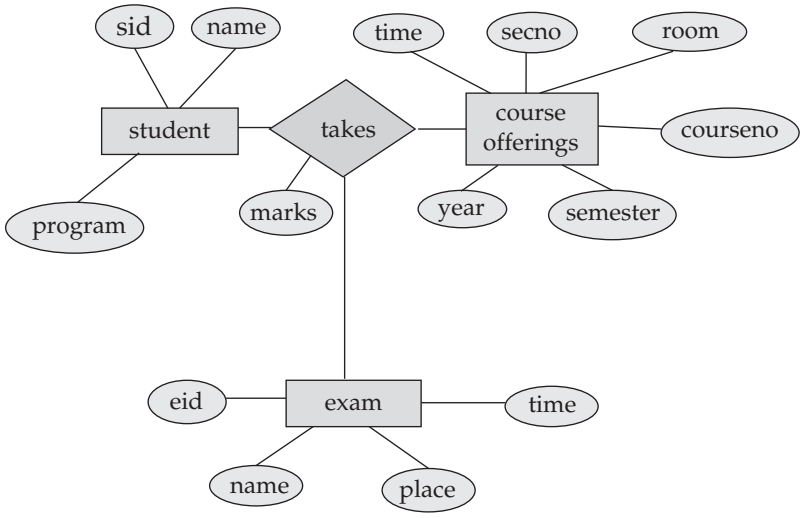


Figure 6.3 E-R diagram for marks database.

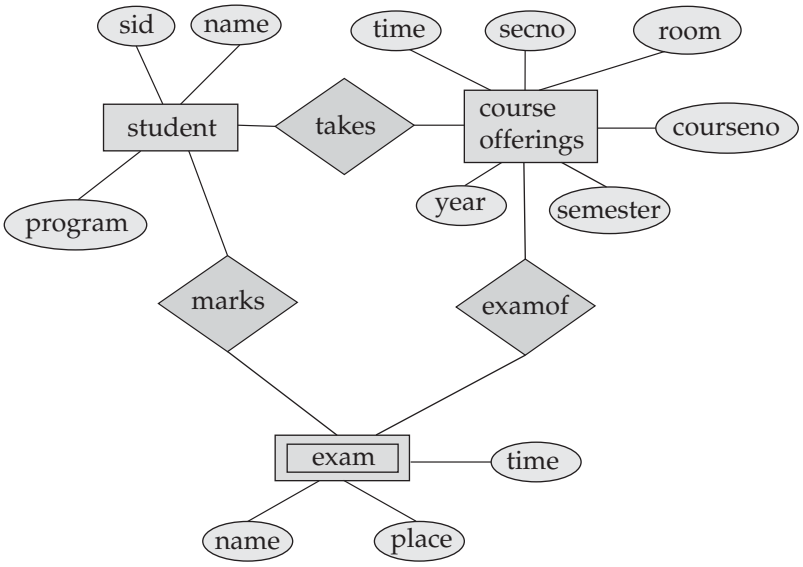


Figure 6.4 Another E-R diagram for marks database.

6.5 By using one entity set many times we are missing relationships in the model. For example, in the E-R diagram in Figure 6.6: the students taking classes are the same students who are athletes, but this model will not show that.

6.6 **a.** See Figure 6.7

b. The additional entity sets are useful if we wish to store their attributes as part of the database. For the *course* entity set, we have chosen to include three attributes. If only the primary key (*c_number*) were included, and if courses have only one section, then it would be appropriate to replace the *course* (and *section*) entity sets by an attribute (*c_number*) of *exam*. The reason it is undesirable to have multiple attributes of *course* as attributes of *exam* is that it would then be difficult to maintain data on the courses, particularly if a course has no exam or several exams. Similar remarks apply to the *room* entity set.

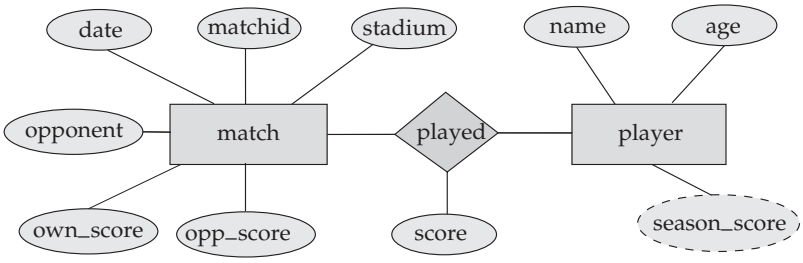


Figure 6.5 E-R diagram for favourite team statistics.

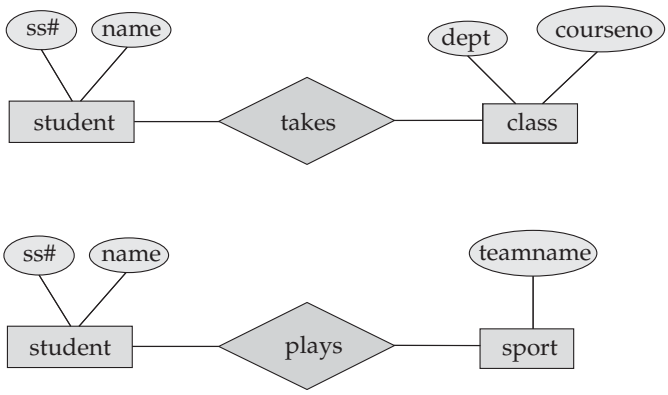


Figure 6.6 E-R diagram with entity duplication.

6.7 **a.** The criteria to use are intuitive design, accurate expression of the real-world concept and efficiency. A model which clearly outlines the objects and relationships in an intuitive manner is better than one which does not, because it is easier to use and easier to change. Deciding between an attribute and an entity set to represent an object, and deciding between an entity set and relationship set, influence the accuracy with which the real-world concept is expressed. If the right design choice is not made, inconsistency and/or loss of information will result. A model which can be implemented in an efficient manner is to be preferred for obvious reasons.

b. Consider three different alternatives for the problem in Exercise 6.2.

- See Figure 6.8
- See Figure 6.9
- See Figure 6.10

Each alternative has merits, depending on the intended use of the database. Scheme 6.8 has been seen earlier. Scheme 6.10 does not require a separate entity for *prerequisites*. However, it will be difficult to store all the prerequi-

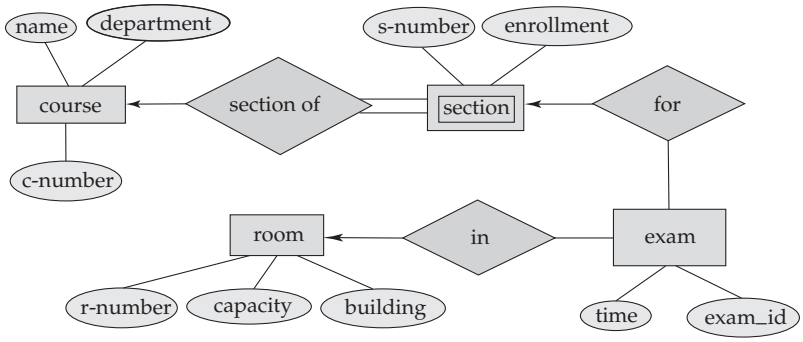


Figure 6.7 E-R diagram for exam scheduling.

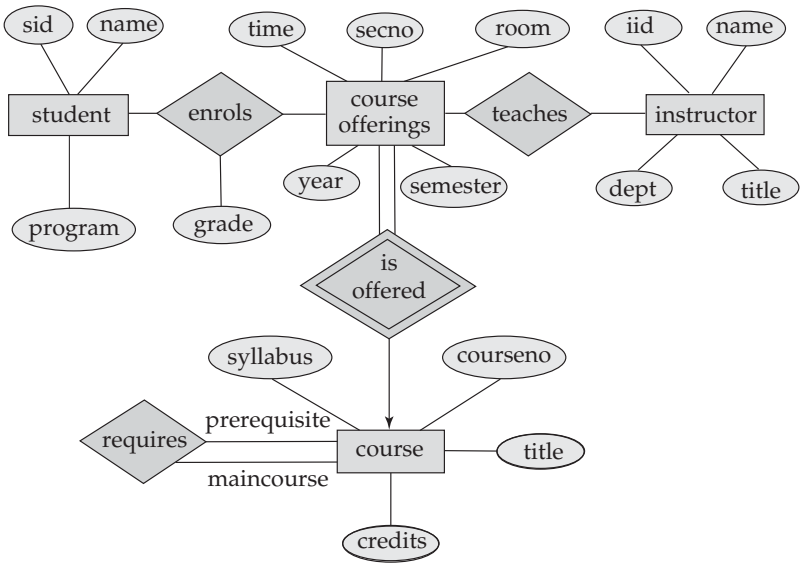


Figure 6.8 E-R diagram for University(a) .

sites(being a multi-valued attribute). Scheme 6.9 treats prerequisites as well as classrooms as separate entities, making it useful for gathering data about prerequisites and room usage. Scheme 6.8 is in between the others, in that it treats prerequisites as separate entities but not classrooms. Since a registrar's office probably has to answer general questions about the number of classes a student is taking or what are all the prerequisites of a course, or where a specific class meets, scheme 6.9 is probably the best choice.

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

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

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

b. See Figure 6.11. The idea is to introduce total participation constraints between E and the relationships R_A, R_B, R_C so that every tuple in E has a relationship with A, B and C.

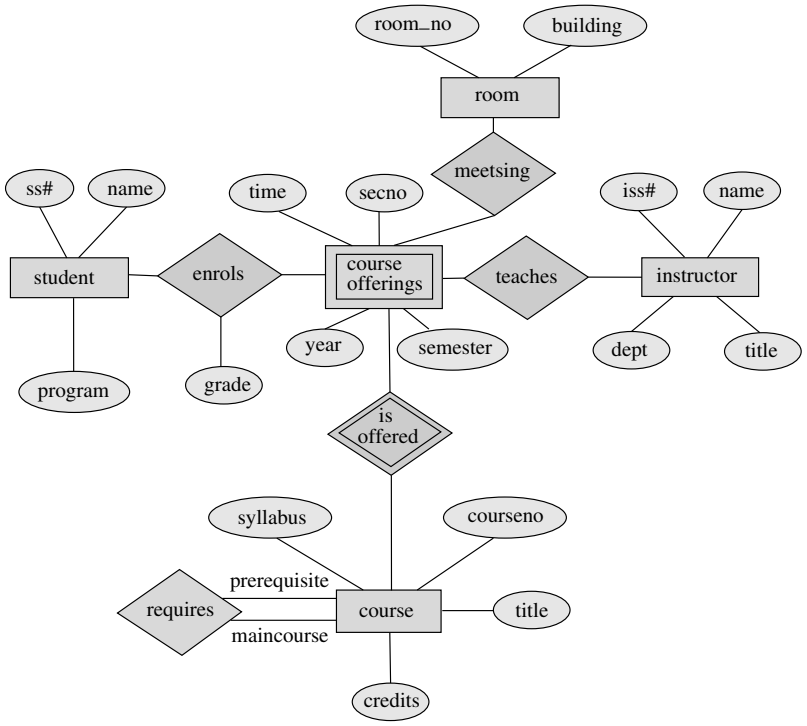


Figure 6.9 E-R diagram for University(b).

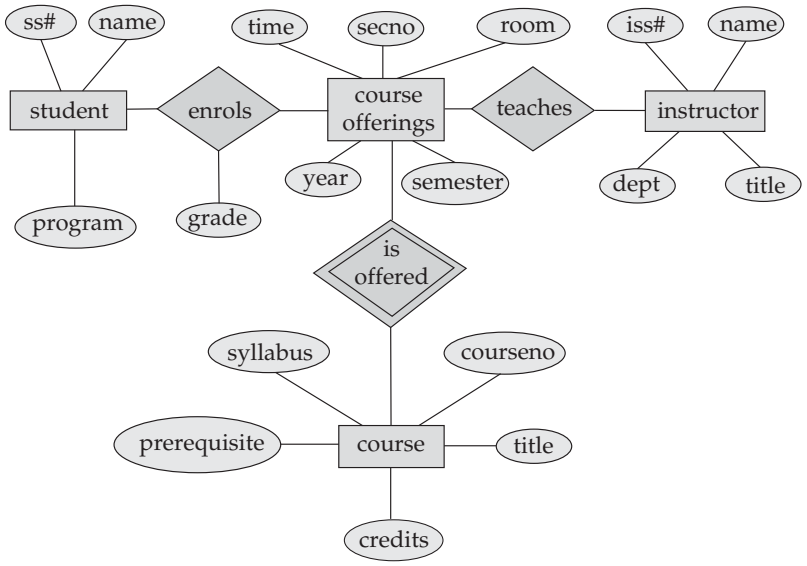


Figure 6.10 E-R diagram for University(c).

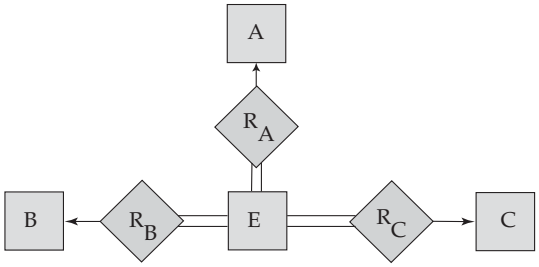


Figure 6.11 E-R diagram to Exercise 6.9b.

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

d. Consider E as a weak entity set and R_A, R_B and R_C as its identifying relationship sets. See Figure 6.12.

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

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

6.12 In this example, we assume that both banks have the shared identifiers for customers, such as the social security number. We see the general solution in the next exercise.

Each of the problems mentioned does have potential for difficulties.

a. *branch_name* is the primary-key of the *branch* entity set. Therefore while merging the two banks' entity sets, if both banks have a branch with the same name, one of them will be lost.

b. customers participate in the relationship sets *cust_banker*, *borrower* and *depositor*. While merging the two banks' *customer* entity sets, duplicate tuples

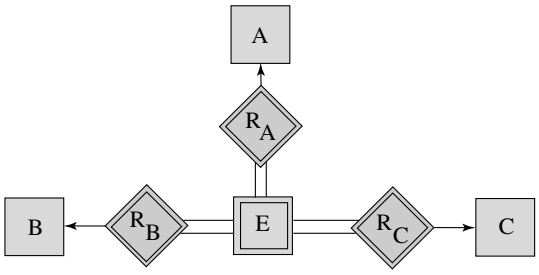


Figure 6.12 E-R diagram to Exercise 6.9d.

of the same customer will be deleted. Therefore those relations in the three mentioned relationship sets which involved these deleted tuples will have to be updated. Note that if the tabular representation of a relationship set is obtained by taking a union of the primary keys of the participating entity sets, no modification to these relationship sets is required.

c. The problem caused by *loans* or *accounts* with the same number in both the banks is similar to the problem caused by branches in both the banks with the same *branch_name*.

To solve the problems caused by the merger, no schema changes are required. Merge the *customer* entity sets removing duplicate tuples with the same *social_security* field. Before merging the *branch* entity sets, prepend the old bank name to the *branch_name* attribute in each tuple. The *employee* entity sets can be merged directly, and so can the *payment* entity sets. No duplicate removal should be performed. Before merging the *loan* and *account* entity sets, whenever there is a number common in both the banks, the old number is replaced by a new unique number, in one of the banks.

Next the relationship sets can be merged. Any relation in any relationship set which involves a tuple which has been modified earlier due to the merger, is itself modified to retain the same meaning. For example let 1611 be a loan number common in both the banks prior to the merger, and let it be replaced by a new unique number 2611 in one of the banks, say bank 2. Now all the relations in *borrower*, *loan_branch* and *loan_payment* of bank 2 which refer to loan number 1611 will have to be modified to refer to 2611. Then the merger with bank 1's corresponding relationship sets can take place.

6.13 This is a case in which the schemas of the two banks differ, so the merger becomes more difficult. The identifying attribute for persons in the US is *social-security*, and in Canada it is *social-insurance*. Therefore the merged schema cannot use either of these. Instead we introduce a new attribute *person_id*, and use this uniformly for everybody in the merged schema. No other change to the schema is required. The values for the *person_id* attribute may be obtained by several ways. One way would be to prepend a country code to the old *social-security* or *social-insurance* values ("U" and "C" respectively, for instance), to get the corresponding *person_id* values. Another way would be to assign fresh numbers starting from 1 upwards, one number to each *social-security* and *social-insurance* value in the old databases.

Once this has been done, the actual merger can proceed as according to the answer to the previous question. If a particular relationship set, say *borrower*, involves only US customers, this can be expressed in the merged database by specializing the entity-set *customer* into *us_customer* and *canada_customer*, and making only *us_customer* participate in the merged *borrower*. Similarly *employee* can be specialized if needed.

CHAPTER 7

Relational-Database Design

Solutions to Practice Exercises

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

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

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

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

7.4 To prove that:

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

Following the hint, we derive:

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

7.5 Proof using Armstrong's axioms of the Pseudotransitivity Rule:
if $\alpha \rightarrow \beta$ and $\gamma\beta \rightarrow \delta$, then $\alpha\gamma \rightarrow \delta$.

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

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

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

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

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

7.7 The given set of FDs F is:

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

The left side of each FD in F is unique. Also none of the attributes in the left side or right side of any of the FDs is extraneous. Therefore the canonical cover F_c is equal to F.

7.8 The algorithm is correct because:

- If A is added to $result$ then there is a proof that $\alpha \rightarrow A$. To see this, observe that $\alpha \rightarrow \alpha$ trivially so α is correctly part of $result$. If $A \notin \alpha$ is added to $result$ there must be some FD $\beta \rightarrow \gamma$ such that $A \in \gamma$ and β is already a subset of $result$. (Otherwise $fdcount$ would be nonzero and the **if** condition would be false.) A full proof can be given by induction on the depth of recursion for an execution of **addin**, but such a proof can be expected only from students with a good mathematical background.
- If $A \in \alpha^+$, then A is eventually added to $result$. We prove this by induction on the length of the proof of $\alpha \rightarrow A$ using Armstrong's axioms. First observe that if procedure **addin** is called with some argument β, all the attributes in β will be added to $result$. Also if a particular FD's *fdcount* becomes 0, all the attributes in its tail will definitely be added to $result$. The base case of the proof, $A \in \alpha \Rightarrow A \in \alpha^+$, is obviously true because the first call to **addin** has the argument α. The inductive hypotheses is that if $\alpha \rightarrow A$ can be proved in n steps or less then $A \in result$. If there is a proof in $n+1$ steps that $\alpha \rightarrow A$, then the last step was an application of either reflexivity, augmentation or transitivity on a fact $\alpha \rightarrow \beta$ proved in n or fewer steps. If reflexivity or augmentation was used in the $(n+1)^{st}$ step, A must have been in $result$ by the end of the n^{th} step itself. Otherwise, by the inductive hypothesis $\beta \subseteq result$. Therefore the dependency used in proving $\beta \rightarrow \gamma$, $A \in \gamma$ will have $fdcount$ set to 0 by the end of the n^{th} step. Hence A will be added to $result$.

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

7.9 **a.** The query is given below. Its result is non-empty if and only if $b \rightarrow c$ does not hold on r.

select b
from r
group by b
having count(**distinct** c) > 1

b.

```
create assertion b-to-c check
    (not exists
        (select b
         from r
         group by b
         having count(distinct c) > 1
        )
    )
```

7.10 Consider some tuple t in u.

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

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

By the definition of natural join,

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

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

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

Since t is any arbitrary tuple in u,

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

7.11 The dependency $B \rightarrow D$ is not preserved. F_1, the restriction of F to (A, B, C) is $A \rightarrow ABC$, $A \rightarrow AB$, $A \rightarrow AC$, $A \rightarrow BC$, $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow A$, $B \rightarrow B$, $C \rightarrow C$, $AB \rightarrow AC$, $AB \rightarrow ABC$, $AB \rightarrow BC$, $AB \rightarrow AB$, $AB \rightarrow A$, $AB \rightarrow B$, $AB \rightarrow C$, AC (same as AB), BC (same as AB), ABC (same as AB). F_2, the restriction of F to (C, D, E) is $A \rightarrow ADE$, $A \rightarrow AD$, $A \rightarrow AE$, $A \rightarrow DE$, $A \rightarrow A$, $A \rightarrow D$, $A \rightarrow E$, $D \rightarrow D$, E (same as A), AD, AE, DE, ADE (same as A). $(F_1 \cup F_2)^+$ is easily seen not to contain $B \rightarrow D$ since the only FD in $F_1 \cup F_2$ with B as the left side is $B \rightarrow B$, a trivial FD. We shall see in Practice Exercise 7.13 that $B \rightarrow D$ is indeed in F^+. Thus $B \rightarrow D$ is not preserved. Note that $CD \rightarrow ABCDE$ is also not preserved.

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

7.12 Let F be a set of functional dependencies that hold on a schema R. Let $\sigma = \{R_1, R_2, \ldots, R_n\}$ be a dependency-preserving 3NF decomposition of R. Let X be a candidate key for R.

Consider a legal instance r of R. Let $j = \Pi_X(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$. We want to prove that $r = j$.

We claim that if t_1 and t_2 are two tuples in j such that $t_1[X] = t_2[X]$, then $t_1 = t_2$. To prove this claim, we use the following inductive argument –
Let $F' = F_1 \cup F_2 \cup \ldots \cup F_n$, where each F_i is the restriction of F to the schema R_i in σ. Consider the use of the algorithm given in Figure 7.9 to compute the closure of X under F'. We use induction on the number of times that the for loop in this algorithm is executed.

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

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

Our claim implies that the size of $\Pi_X(j)$ is equal to the size of j. Note also that $\Pi_X(j) = \Pi_X(r) = r$ (since X is a key for R). Thus we have proved that the size of j equals that of r. Using the result of Practice Exercise 7.10, we know that $r \subseteq j$. Hence we conclude that $r = j$.

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

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

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

is in BCNF as is

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

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

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

7.14 Suppose R is in 3NF according to the textbook definition. We show that it is in 3NF according to the definition in the exercise. Let A be a nonprime attribute in

R that is transitively dependent on a key α for R. Then there exists $\beta \subseteq R$ such that $\beta \rightarrow A$, $\alpha \rightarrow \beta$, $A \notin \alpha$, $A \notin \beta$, and $\beta \rightarrow \alpha$ does not hold. But then $\beta \rightarrow A$ violates the textbook definition of 3NF since

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

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

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

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

7.15 Referring to the definitions in Practice Exercise 7.14, a relation schema R is said to be in 3NF if there is no non-prime attribute A in R for which A is transitively dependent on a key for R.

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

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

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

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

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

7.16 The relation schema $R = (A, B, C, D, E)$ and the set of dependencies

$$\begin{aligned} A &\rightarrow\!\!\!\!\rightarrow BC \\ B &\rightarrow\!\!\!\!\rightarrow CD \\ E &\rightarrow\!\!\!\!\rightarrow AD \end{aligned}$$

constitute a BCNF decomposition, however it is clearly not in 4NF. (It is BCNF because all FDs are trivial).

CHAPTER 8

Application Design and Development

Solutions to Practice Exercises

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

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

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

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

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

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

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

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

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

The active rules for this insertion are given below –

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

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

Note that if the execution binding was *deferred* (instead of immediate), then the result of the join of the set of new tuples of *account* with the set of new tuples of *depositor* would have been inserted by *both* active rules, leading to duplication of the corresponding tuples in *branch_cust*.

The deletion of a tuple from *branch_cust* is similar to insertion, except that a deletion from either *depositor* or *account* will cause the natural join of these relations to have a lesser number of tuples. We denote the newly deleted set of tuples by qualifying the relation name with the keyword **deleted**.

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

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

8.5 Query:

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

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

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

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

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

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

8.7 A scheme for storing passwords would be to encrypt each password, and then use a hash index on the user-id. The user-id can be used to easily access the encrypted password. The password being used in a login attempt is then encrypted and compared with the stored encryption of the correct password. An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist!

CHAPTER 9

Object-Based Databases

Solutions to Practice Exercises

9.1 For this problem, we use table inheritance. We assume that **MyDate**, **Color** and **DriveTrainType** are pre-defined types.

```
create type Vehicle
    (vehicle_id integer,
     license_number char(15),
     manufacturer char(30),
     model char(30),
     purchase_date MyDate,
     color Color)

create table vehicle of type Vehicle

create table truck
    (cargo_capacity integer)
    under vehicle

create table sportsCar
    (horsepower integer
     renter_age_requirement integer)
    under vehicle

create table van
    (num_passengers integer)
    under vehicle
```

```
create table offRoadVehicle
    (ground_clearance real
     driveTrain DriveTrainType)
    under vehicle
```

9.2 **a.** No Answer.

b. Queries in SQL:1999.

i. Program:

```
select ename
from emp as e, e.ChildrenSet as c
where 'March' in
        (select birthday.month
         from c
        )
```

ii. Program:

```
select e.ename
from emp as e, e.SkillSet as s, s.ExamSet as x
where s.type = 'typing' and x.city = 'Dayton'
```

iii. Program:

```
select distinct s.type
from emp as e, e.SkillSet as s
```

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

```
create type Name
    (first_name varchar(15),
     middle_initial char,
     last_name varchar(15))
create type Street
    (street_name varchar(15),
     street_number varchar(4),
     apartment_number varchar(7))
create type Address
    (street Street,
     city varchar(15),
     state varchar(15),
     zip_code char(6))
create table customer
    (name Name,
     customer_id varchar(10),
     address Adress,
     phones  char(7)  array[10],
     dob date)
```

method integer *age*()

b. **create function** *Name* (*f* **varchar**(15), *m* **char**, *l* **varchar**(15))
returns *Name*
begin
set *first_name* = *f*;
set *middle_initial* = *m*;
set *last_name* = *l*;
end
create function *Street* (*sname* **varchar**(15), *sno* **varchar**(4), *ano* **varchar**(7))
returns *Street*
begin
set *street_name* = *sname*;
set *street_number* = *sno*;
set *apartment_number* =*ano*;
end
create function *Address* (*s Street*, *c* **varchar**(15), *sta* **varchar**(15), *zip* **varchar**(6))
returns *Address*
begin
set *street* = *s*;
set *city* = *c*;
set *state* =*sta*;
set *zip_code* =*zip*;
end

9.4 **a.** The schema definition is given below. Note that backward references can be addedbut they are not so important as in OODBS because queries can be written in SQL and joins can take care of integrity constraints.

create type *Employee*
(*person_name* **varchar(30)**,
street **varchar(15)**,
city **varchar(15)**)
create type *Company*
(*company_name* **varchar(15)**,
(*city* **varchar(15)**)
create table *employee* of *Employee*
create table *company* of *Company*
create type *Works*
(*person* **ref**(*Employee*) **scope** *employee*,
comp **ref**(*Company*) **scope** *company*,
salary **int**)
create table *works* of *Works*
create type *Manages*
(*person* **ref**(*Employee*) **scope** *employee*,
(*manager* **ref**(*Employee*) **scope** *employee*)

create table *manages* of *Manages*

b. i. **select** *comp*− >*name*
from *works*
group by *comp*
having count(*person*) ≥ **all(select count(***person*)
from *works*
group by *comp*)

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

iii. **select** *comp*− >*name*
from *works*
group by *comp*
having avg(*salary*) > **(select avg(***salary*)
from *works*
where *comp*− >*company_name*="First Bank Corporation")

9.5 **a.** A computer-aided design system for a manufacturer of airplanes:
An OODB system would be suitable for this. That is because CAD requires complex data types, and being computation oriented, CAD tools are typically used in a programming language environment needing to access the database.

b. A system to track contributions made to candidates for public office:
A relational system would be apt for this, as data types are expected to be simple, and a powerful querying mechanism is essential.

c. An information system to support the making of movies:
Here there will be extensive use of multimedia and other complex data types. But queries are probably simple, and thus an object relational system is suitable.

9.6 An entity is simply a collection of variables or data items. An object is an encapsulation of data as well as the methods (code) to operate on the data. The data members of an object are directly visible only to its methods. The outside world can gain access to the object's data only by passing pre-defined messages to it, and these messages are implemented by the methods.

CHAPTER 10

XML

Solutions to Practice Exercises

10.1 **a.** The XML representation of data using attributes is shown in Figure 10.1.
b. The DTD for the bank is shown in Figure 10.2.

10.2 Query:

```
<!DOCTYPE db [
    <!ELEMENT emp (ename, children*, skills*)>
    <!ELEMENT children (name, birthday)>
    <!ELEMENT birthday (day, month, year)>
    <!ELEMENT skills (type, exams+)>
    <!ELEMENT exams (year, city)>
    <!ELEMENT ename( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT day( #PCDATA )>
    <!ELEMENT month( #PCDATA )>
    <!ELEMENT year( #PCDATA )>
    <!ELEMENT type( #PCDATA )>
    <!ELEMENT city( #PCDATA )>
] >
```

10.3 Code:

```
/db/emp/skills/type
```

```
<bank>
    <account account-number="A-101" branch-name="Downtown"
            balance="500">
    </account>
    <account account-number="A-102" branch-name="Perryridge"
            balance="400">
    </account>
    <account account-number="A-201" branch-name="Brighton"
            balance="900">
    </account>
    <customer customer-name="Johnson" customer-street="Alma"
            customer-city="Palo Alto">
    </customer>
    <customer customer-name="Hayes" customer-street="Main"
            customer-city="Harrison">
    </customer>
    <depositor account-number="A-101" customer-name="Johnson">
    </depositor>
    <depositor account-number="A-201" customer-name="Johnson">
    </depositor>
    <depositor account-number="A-102" customer-name="Hayes">
    </depositor>
</bank>
```

Figure 10.1 XML representation.

```
<!DOCTYPE bank [
    <!ELEMENT account >
    <!ATTLIST account
        account-number ID #REQUIRED
        branch-name CDATA #REQUIRED
        balance CDATA #REQUIRED >
    <!ELEMENT customer >
    <!ATTLIST customer
        customer-name ID #REQUIRED
        customer-street CDATA #REQUIRED
        customer-street CDATA #REQUIRED >
    <!ELEMENT depositor >
    <!ATTLIST depositor
        account-number IDREF #REQUIRED
        customer-name IDREF #REQUIRED >
] >
```

Figure 10.2 The DTD for the bank.

10.4 Query:

```
for $b in distinct (/bank/account/branch-name)
return
<branch-total>
      <branch-name> $b/text() </branch-name>
      let $s := sum (/bank/account[branch-name=$b]/balance)
      return <total-balance> $s </total-balance>
</branch-total>
```

10.5 Query:

```
      <lojoin>
      for $b in /bank/account,
            $c in /bank/customer,
            $d in /bank/depositor
      where $a/account-number = $d/account-number
            and $c/customer-name = $d/customer-name
      return <cust-acct> $c $a </cust-acct>
      |
      for $c in /bank/customer,
      where every $d in /bank/depositor satisfies
      (not ($c/customer-name=$d/customer-name))
      return <cust-acct> $c </cust-acct>
      </lojoin>
```

10.6 The answer in XQuery is

```
<bank-2>
      for $c in /bank/customer
      return
            <customer>
                  <customer-name> $c/* </customer-name>
                  for $a in $c/id(@accounts)
                  return $a
            </customer>
</bank-2>
```

10.7 Realtion schema:

book (bid, title, year, publisher, place)
article (artid, title, journal, year, number, volume, pages)
book_author (bid, first_name,last_name, order)
article_author (artid, first_name,last_name, order)

10.8 The answer is shwn in Figure 10.3.

```
nodes(1,element,bank,–)
nodes(2,element,account,–)
nodes(3,element,account,–)
nodes(4,element,account,–)
nodes(5,element,customer,–)
nodes(6,element,customer,–)
nodes(7,element,depositor,–)
nodes(8,element,depositor,–)
nodes(9,element,depositor,–)
child(2,1) child(3,1) child(4,1)
child(5,1) child(6,1)
child(7,1) child(8,1) child(9,1)
nodes(10,element,account-number,A-101)
nodes(11,element,branch-name,Downtown)
nodes(12,element,balance,500)
child(10,2) child(11,2) child(12,2)
nodes(13,element,account-number,A-102)
nodes(14,element,branch-name,Perryridge)
nodes(15,element,balance,400)
child(13,3) child(14,3) child(15,3)
nodes(16,element,account-number,A-201)
nodes(17,element,branch-name,Brighton)
nodes(18,element,balance,900)
child(16,4) child(17,4) child(18,4)
nodes(19,element,customer-name,Johnson)
nodes(20,element,customer-street,Alma)
nodes(21,element,customer-city,Palo Alto)
child(19,5) child(20,5) child(21,5)
nodes(22,element,customer-name,Hayes)
nodes(23,element,customer-street,Main)
nodes(24,element,customer-city,Harrison)
child(22,6) child(23,6) child(24,6)
nodes(25,element,account-number,A-101)
nodes(26,element,customer-name,Johnson)
child(25,7) child(26,7)
nodes(27,element,account-number,A-201)
nodes(28,element,customer-name,Johnson)
child(27,8) child(28,8)
nodes(29,element,account-number,A-102)
nodes(30,element,customer-name,Hayes)
child(29,9) child(30,9)
```

Figure 10.3 Relational Representation of XML Data as Trees.

10.9 **a.** The answer is shown in Figure 10.4.

b. Show how to map this DTD to a relational schema.

part(partid,name)
subpartinfo(partid, subpartid, qty)

Attributes partid and subpartid of subpartinfo are foreign keys to part.

c. No answer

```
<parts>
    <part>
        <name> bicycle </name>
        <subpartinfo>
            <part>
                <name> wheel </name>
                <subpartinfo>
                    <part>
                        <name> rim </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> spokes </name>
                    </part>
                    <qty> 40 </qty>
                </subpartinfo>
                <subpartinfo>
                    <part>
                        <name> tire </name>
                    </part>
                    <qty> 1 </qty>
                </subpartinfo>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> brake </name>
            </part>
            <qty> 2 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> gear </name>
            </part>
            <qty> 3 </qty>
        </subpartinfo>
        <subpartinfo>
            <part>
                <name> frame </name>
            </part>
            <qty> 1 </qty>
        </subpartinfo>
    </part>
</parts>
```

Figure 10.4 Example Parts Data in XML.

CHAPTER 11

Storage and File Structure

Solutions to Practice Exercises

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

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

i. Write the information onto the first physical block.
ii. When the first write completes successfully, write the same information onto the second physical block.
iii. The output is declared completed only after the second write completes successfully.

During recovery, each pair of physical blocks is examined. If both are identical and there is no detectable partial-write, then no further actions are necessary. If one block has been partially rewritten, then we replace its contents with the contents of the other block. If there has been no partial-write, but they differ in content, then we replace the contents of the first block with the contents of the second, or vice versa. This recovery procedure ensures that a write to stable storage either succeeds completely (that is, updates both copies) or results in no change.

The requirement of comparing every corresponding pair of blocks during recovery is expensive to meet. We can reduce the cost greatly by keeping track of block writes that are in progress, using a small amount of nonvolatile RAM. On recovery, only blocks for which writes were in progress need to be compared.

b. The idea is similar here. For any block write, the information block is written first followed by the corresponding parity block. At the time of

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

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

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

Under MRU, some unused blocks may remain in memory forever. In practice, MRU can be used only in special situations like that of the nested-loop strategy discussed in example 0.a

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

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

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

11.5 (We use "↑ i" to denote a pointer to record "i".)
The original file of Figure 11.8.

header				↑ 1
record 0	A-102	Perryridge	400	
record 1				↑ 4
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4		Perryridge		↑ 6
record 5	A-201		900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

a. The file after **insert** (Brighton, A-323, 1600).

header				↑ 4
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

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

header				↑ 2
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2				↑ 4
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

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

c. The file after **insert** (Brighton, A-626, 2000).

header				↑ 4
record 0	A-102	Perryridge	400	
record 1	A-323	Brighton	1600	
record 2	A-626	Brighton	2000	
record 3	A-101	Downtown	500	
record 4				↑ 6
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

11.6 Instance of relations:

course relation

course_name	room	instructor	
Pascal	CS-101	Calvin, B	c_1
C	CS-102	Calvin, B	c_2
Lisp	CS-102	Kess, J	c_3

course_name	student_name	grade
Pascal	Carper, D	A
Pascal	Merrick, L	A
Pascal	Mitchell, N	B
Pascal	Bliss, A	C
Pascal	Hames, G	C
C	Nile, M	A
C	Mitchell, N	B
C	Carper, D	A
C	Hurly, I	B
C	Hames, G	A
Lisp	Bliss, A	C
Lisp	Hurly, I	B
Lisp	Nile, M	D
Lisp	Stars, R	A
Lisp	Carper, D	A

Block 0 contains: c_1, e_1, e_2, e_3, e_4, and e_5
Block 1 contains: c_2, e_6, e_7, e_8, e_9 and e_{10}
Block 2 contains: c_3, e_{11}, e_{12}, e_{13}, e_{14}, and e_{15}

11.7 **a.** Everytime a record is inserted/deleted, check if the usage of the block has changed levels. In that case, update the corrosponding bits. Note that we don't need to access the bitmaps at all unless the usage crosses a boundary, so in most of the cases there is no overhead.

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

CHAPTER 12

Indexing and Hashing

Solutions to Practice Exercises

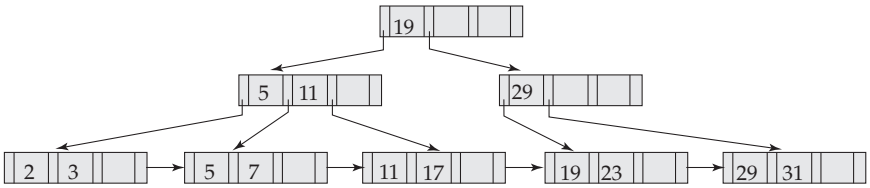
12.1 Reasons for not keeping several search indices include:

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

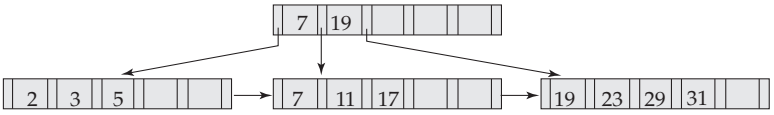
12.2 In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but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

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

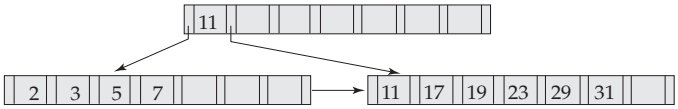
a.



b.

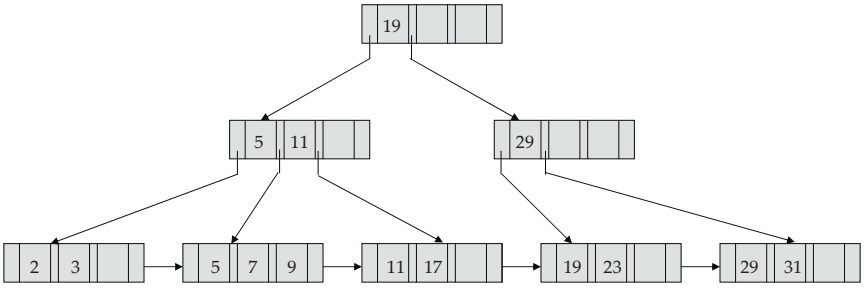


c.

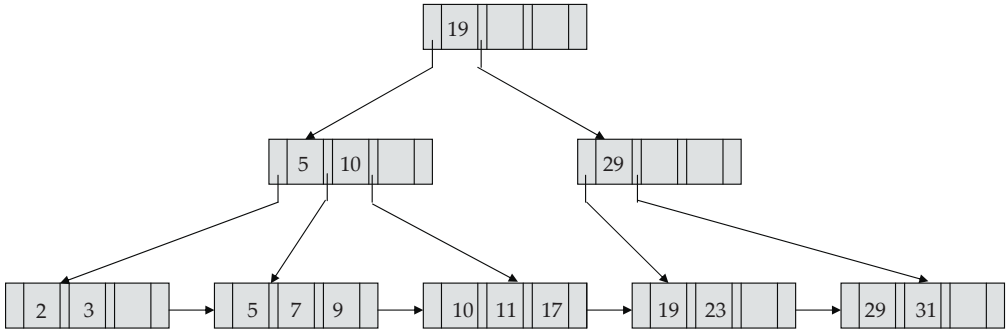


12.4 With structure 12.3.a:

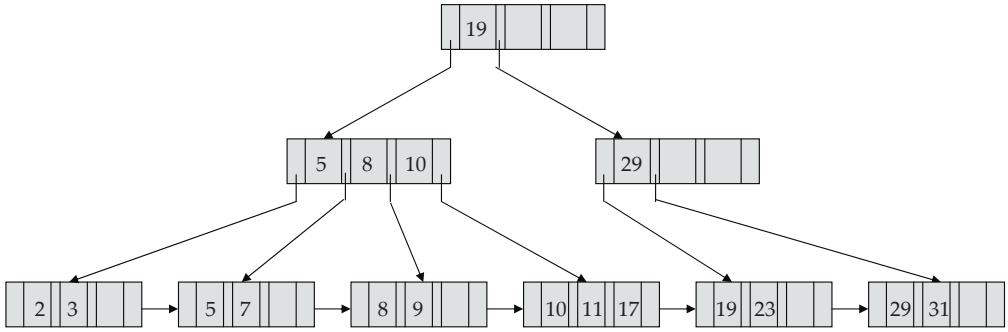
Insert 9:



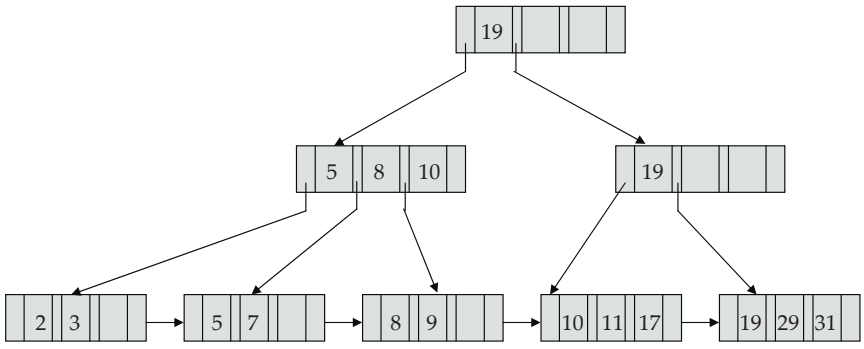
Insert 10:



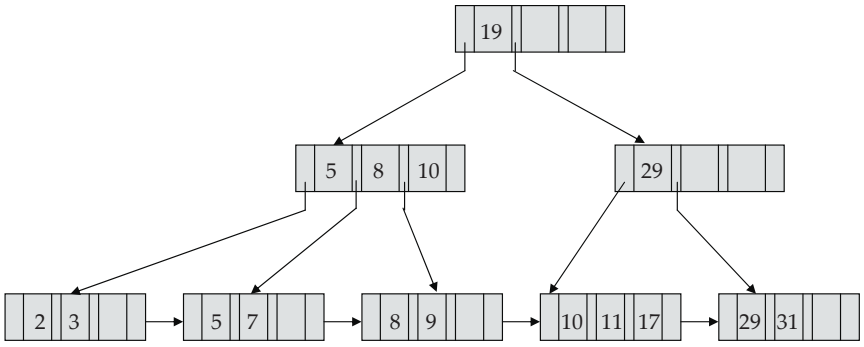
Insert 8:



Delete 23:

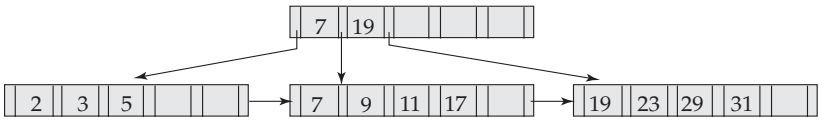


Delete 19:

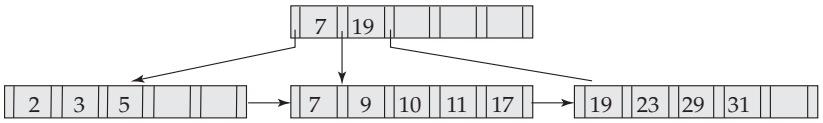


With structure 12.3.b:

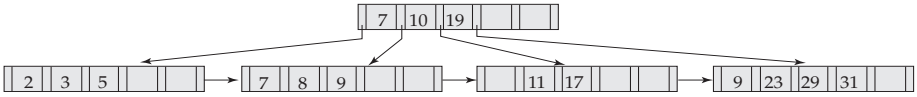
Insert 9:



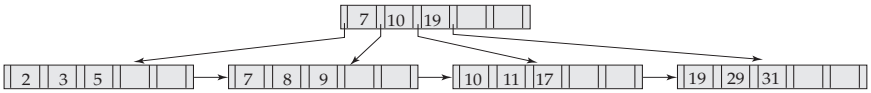
Insert 10:



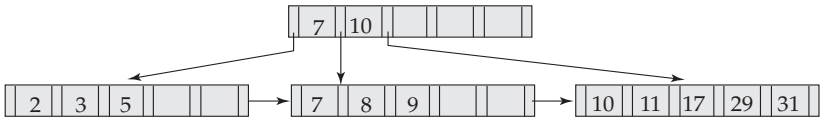
Insert 8:



Delete 23:

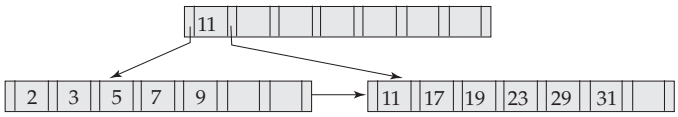


Delete 19:

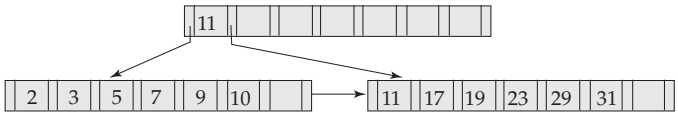


With structure 12.3.c:

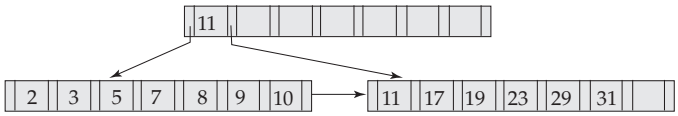
Insert 9:



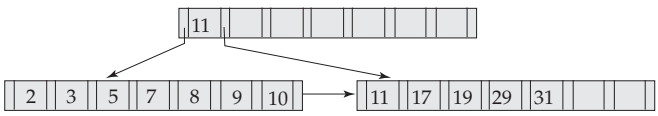
Insert 10:



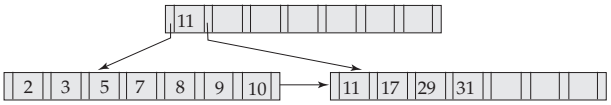
Insert 8:



Delete 23:



Delete 19:



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

12.6 The algorithm for insertion into a B-tree is:

Locate the leaf node into which the new key-pointer pair should be inserted. If there is space remaining in that leaf node, perform the insertion at the correct location, and the task is over. Otherwise insert the key-pointer pair conceptually into the correct location in the leaf node, and then split it along the middle. The middle key-pointer pair does not go into either of the resultant nodes of the split operation. Instead it is inserted into the parent node, along with the tree pointer to the new child. If there is no space in the parent, a similar procedure is repeated.

The deletion algorithm is:

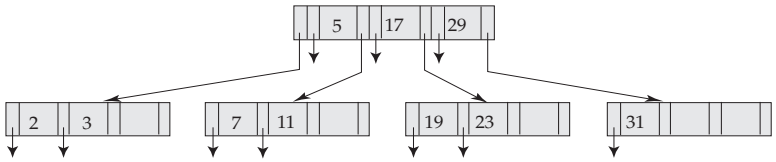
Locate the key value to be deleted, in the B-tree.

a. If it is found in a leaf node, delete the key-pointer pair, and the record from the file. If the leaf node contains less than $\lceil n/2 \rceil - 1$ entries as a result of this deletion, it is either merged with its siblings, or some entries are redistributed to it. Merging would imply a deletion, whereas redistribution would imply change(s) in the parent node's entries. The deletions may ripple upto the root of the B-tree.

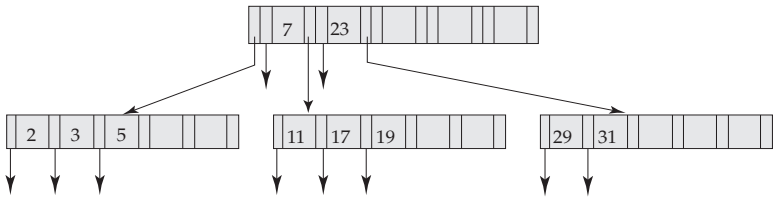
b. If the key value is found in an internal node of the B-tree, replace it and its record pointer by the smallest key value in the subtree immediately to its right and the corresponding record pointer. Delete the actual record in the database file. Then delete that smallest key value-pointer pair from the subtree. This deletion may cause further rippling deletions till the root of the B-tree.

Below are the B-trees we will get after insertion of the given key values. We assume that leaf and non-leaf nodes hold the same number of search key values.

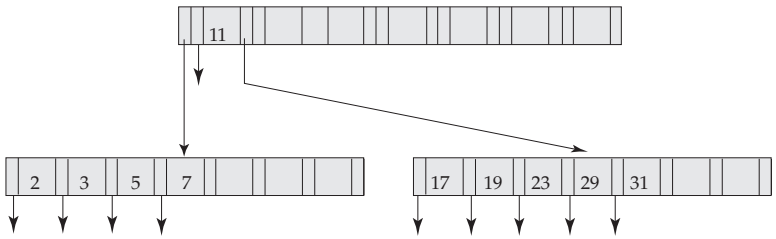
a.



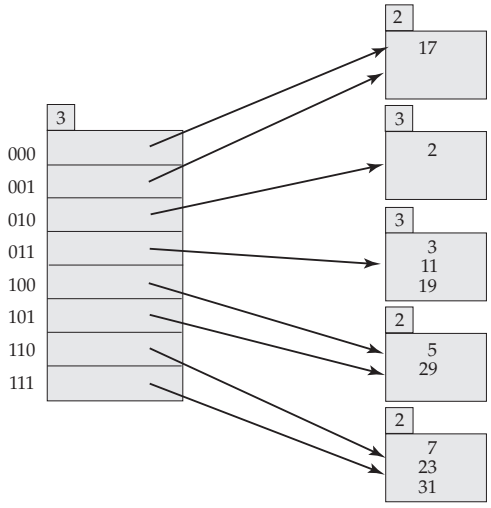
b.



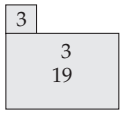
c.



12.7 Extendable hash structure

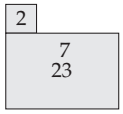


12.8 **a.** Delete 11: From the answer to Exercise 12.7, change the third bucket to:

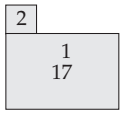


At this stage, it is possible to coalesce the second and third buckets. Then it is enough if the bucket address table has just four entries instead of eight. For the purpose of this answer, we do not do the coalescing.

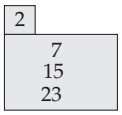
b. Delete 31: From the answer to 12.7, change the last bucket to:



c. Insert 1: From the answer to 12.7, change the first bucket to:



d. Insert 15: From the answer to 12.7, change the last bucket to:



12.9 Let `i` denote the number of bits of the hash value used in the hash table. Let **bsize** denote the maximum capacity of each bucket.

```
delete(value K_1)
begin
    j = first i high-order bits of h(K_1);
    delete value K_1 from bucket j;
    coalesce(bucket j);
end

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

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

    coalesce(bucket j);
end
```

Note that we can only merge two buckets at a time. The common hash prefix of the resultant bucket will have length one less than the two buckets merged. Hence we look at the buddy bucket of bucket `j` differing from it only at the last bit. If the common hash prefix of this bucket is not i_j, then this implies that the buddy bucket has been further split and merge is not possible.

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

12.10 If the hash table is currently using `i` bits of the hash value, then maintain a count of buckets for which the length of common hash prefix is exactly `i`.

Consider a bucket `j` with length of common hash prefix i_j. If the bucket is being split, and i_j is equal to `i`, then reset the count to 1. If the bucket is being

split and i_j is one less that i, then increase the count by 1. It the bucket if being coalesced, and i_j is equal to $i$then decrease the count by 1. If the count becomes 0, then the bucket address table can be reduced in size at that point.

However, note that if the bucket address table is not reduced at that point, then the count has no significance afterwards. If we want to postpone the reduction, we have to keep an array of counts, i.e. a count for each value of common hash prefix. The array has to be updated in a similar fashion. The bucket address table can be reduced if the i^{th} entry of the array is 0, where i is the number of bits the table is using. Since bucket table reduction is an expensive operation, it is not always advisable to reduce the table. It should be reduced only when sufficient number of entries at the end of count array become 0.

12.11 We reproduce the account relation of Figure 12.25 below.

A-217	Brighton	750
A-101	Downtown	500
A-1 10	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Bitmaps for *branch_name*

Brighton	1	0	0	0	0	0	0	0	0
Downtown	0	1	1	0	0	0	0	0	0
Mianus	0	0	0	1	0	0	0	0	0
Perryridge	0	0	0	0	1	1	1	0	0
Redwood	0	0	0	0	0	0	0	1	0
Round hill	0	0	0	0	0	0	0	0	1

Bitmaps for *balance*

L_1	0	0	0	0	0	0	0	0	0
L_2	0	0	0	0	1	0	0	0	1
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0

where, level L_1 is below 250, level L_2 is from 250 to below 500, L_3 from 500 to below 750 and level L_4 is above 750.

To find all accounts in Downtown with a balance of 500 or more, we find the union of bitmaps for levels L_3 and L_4 and then intersect it with the bitmap for Downtown.

Downtown	0	1	1	0	0	0	0	0	0
L_3	0	1	1	1	0	0	1	1	0
L_4	1	0	0	0	0	1	0	0	0
$L_3 \cup L_4$	1	1	1	1	0	1	1	1	0
Downtown	0	1	1	0	0	0	0	0	0
Downtown $\setminus (L_3 \cup L_4)$	0	1	1	0	0	0	0	0	0

Thus, the required tuples are A-101 and A-110.

12.12 No answer

CHAPTER 13

Query Processing

Solutions to Practice Exercises

13.1 Query:

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

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

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

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

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

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

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

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

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

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

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

c. Merge-join:
Assuming that r_1 and r_2 are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil log_{M-1}(1500/M) \rceil + 2) + 800(2\lceil log_{M-1}(800/M) \rceil + 2)$ disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

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

13.4 If there are multiple tuples in the inner relation with the same value for the join attributes, we may have to access that many blocks of the inner relation for each tuple of the outer relation. That is why it is inefficient. To reduce this cost we can perform a join of the outer relation tuples with just the secondary index leaf entries, postponing the inner relation tuple retrieval. The result file obtained is then sorted on the inner relation addresses, allowing an efficient physical order scan to complete the join.

Hybrid merge–join requires the outer relation to be sorted. The above algorithm does not have this requirement, but for each tuple in the outer relation it needs to perform an index lookup on the inner relation. If the outer relation is much larger than the inner relation, this index lookup cost will be less than the sorting cost, thus this algorithm will be more efficient.

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

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

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

c. This query is equivalent to the query

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

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

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

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

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

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

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

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

We assume that for any value of T, all tuples in r with that T value fit in memory, and consider the general case at the end. Partition the relation r on attributes in T such that each partition fits in memory (always possible because of our assumption). Consider partitions one at a time. Build a hash table on the tuples, at the same time collecting all distinct T values in a separate hash table. For each value of T, Now, for each value V_T of T, each value s of S, probe the hash table on (V_T, s). If any of the values is absent, discard the value V_T, else output the value V_T.

In the case that not all r tuples with one value for T fit in memory, partition r and s on the S attributes such that the condition is satisfied, run the algorithm on each corresponding pair of partitions r_i and s_i. Output the intersection of the T values generated in each partition.

13.9 Seek overhead is reduced, but the the number of runs that can be merged in a pass decreases potentially leading to more passes Should choose a value of b_b that minimizes overall cost.

CHAPTER 14

Query Optimization

Solutions to Practice Exercises

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

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

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

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

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

The above two equations imply the given equivalence.

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

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

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

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

Using similar reasoning, we can also conclude that

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

The above two equations imply the given equivalence.

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

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

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

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

Using similar reasoning, we can also conclude that

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

The above two equations imply the given equivalence.

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

14.2 **a.** $R = \{(1,2)\},\ S = \{(1,3)\}$

The result of the left hand side expression is $\{(1)\}$, whereas the result of the right hand side expression is empty.

b. $R = \{(1,2),(1,5)\}$

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

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

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

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

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

i. σ

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

ii. Π

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

iii. $\times$

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

iv. $\bowtie$

The output will be the same as a cross product followed by a selection.

v. $-$

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

vi. $\cup$

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

vii. $\cap$

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

b. All the equivalence rules 1 through 7.b of section 14.2.1 hold for the multiset version of the relational-algebra defined in the first part.

There exist equivalence rules which hold for the ordinary relational-algebra, but do not hold for the multiset version. For example consider the rule :-

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

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

14.4

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

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

the intermediate relation is about the same size as r_1 or r_2. Then r_3 is joined to this result.

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

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

c. This query is equivalent to the query:

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

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

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

14.8 Consider the dynamic programming algorithm given in Section 14.4.2. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x = 2$. Thus the total cost is less than 3^n.

14.9 The derivation of time taken is similar to the general case, except that instead of considering $2^{k+1}-2$ subsets of size less than or equal to k for A, we only need to consider $k+1$ subsets of size exactly equal to k. That is because the right hand operand of the topmost join has to be a single relation. Therefore the total cost for finding the best join order for all subsets of size $k+1$ is $\binom{n}{k+1}(k+1)$, which is equal to $n\binom{n-1}{k}$. Summing over all k from 1 to $n-1$ using the binomial expansion of $(1+x)^{n-1}$ with $x = 1$, gives a total cost of less than $n2^{n-1}$.

14.10 **a.** The nested query is as follows:

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

b. The decorrelated query is as follows:

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

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

```
select  ···
from    L1
where   P1 and
        A1 op
        (select f(A2)
        from L2
        where P2)
```

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

```
create table t1 as
        select f(A2),V
        from L2
        where P2^1
        group by V
select  ···
from    L1, t1
where   P1 and P2^2 and
        A1 op t1.A2
```

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

CHAPTER 15

Transactions

Solutions to Practice Exercises

15.1 Even in this case the recovery manager is needed to perform roll-back of aborted transactions.

15.2 There are several steps in the creation of a file. A storage area is assigned to the file in the file system, a unique i-number is given to the file and an i-node entry is inserted into the i-list. Deletion of file involves exactly opposite steps.

For the file system user in UNIX, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor though, many of the internal file system actions need to have transaction semantics. All the steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

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

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

15.5 Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practise are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

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

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

CHAPTER 16

Concurrency Control

Solutions to Practice Exercises

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

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

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

16.2 **a.** Lock and unlock instructions:

T_{31}: **lock-S**(A)
read(A)
lock-X(B)
read(B)
if $A\ =\ 0$
then $B\ :=\ B\ +\ 1$
write(B)
unlock(A)
unlock(B)

T_{32}: **lock-S**(B)
read(B)
lock-X(A)
read(A)
if $B = 0$
then $A := A + 1$
write(A)
unlock(B)
unlock(A)

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

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

The transactions are now deadlocked.

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

16.4 The proof is in Buckley and Silberschatz, "Concurrency Control in Graph Protocols by Using Edge Locks," Proc. ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, 1984.

16.5 Consider the tree-structured database graph given below.

Schedule possible under tree protocol but not under 2PL:

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

Schedule possible under 2PL but not under tree protocol:

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

16.6 The proof is in Kedem and Silberschatz, "Locking Protocols: From Exclusive to Shared Locks," JACM Vol. 30, 4, 1983.

16.7 The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979.

16.8 The proof is in Kedem and Silberschatz, "Controlling Concurrency Using Locking Protocols," Proc. Annual IEEE Symposium on Foundations of Computer Science, 1979.

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

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

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

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

16.13 In the concurrency control scheme of Section 16.3 choosing **Start**(T_i) as the timestamp of T_i gives a subset of the schedules allowed by choosing **Validation**(T_i) as the timestamp. Using **Start**(T_i) means that whoever started first must finish first. Clearly transactions could enter the validation phase in the same order in which they began executing, but this is overly restrictive. Since choosing **Validation**(T_i) causes fewer nonconflicting transactions to restart, it gives the better response times.

16.14

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

sions adds space and time overheads though, therefore plain 2PL may be preferable in low conflict situations.

16.15 A transaction waits on *a.* disk I/O and *b.* lock acquisition. Transactions generally wait on disk reads and not on disk writes as disk writes are handled by the buffering mechanism in asynchronous fashion and transactions update only the in-memory copy of the disk blocks.

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

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

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

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

16.16 Consider two transactions T_1 and T_2 shown below.

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

Let TS(T_1) < TS(T_2) and let the timestamp test at each operation except write(q) be successful. When transaction T_1 does the timestamp test for write(q) it finds that TS(T_1) < R-timestamp(q), since TS(T_1) < TS(T_2) and R-timestamp(q) = TS(T_2). Hence the writeoperation fails and transaction T_1 rolls back. The cascading results in transaction T_2 also being rolled back as it uses the value for item p that is written by transaction T_1.

If this scenario is exactly repeated every time the transactions are restarted, this could result in starvation of both transactions.

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

16.18 Note: The tree-protocol of Section 16.1.5 which is referred to in this question, is different from the multigranularity protocol of Section 16.4 and the B^+-tree concurrency protocol of Section 16.9.

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

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

CHAPTER 17

Recovery System

Solutions to Practice Exercises

17.1

- The recovery scheme using a log with deferred updates has the following advantages over the recovery scheme with immediate updates:
  - **a.** The scheme is easier and simpler to implement since fewer operations and routines are needed, i.e., no UNDO.
  - **b.** The scheme requires less overhead since no extra I/O operations need to be done until commit time (log records can be kept in memory the entire time).
  - **c.** Since the old values of data do not have to be present in the log-records, this scheme requires less log storage space.
- The disadvantages of the deferred modification scheme are :
  - **a.** When a data item needs to accessed, the transaction can no longer directly read the correct page from the database buffer, because a previous write by the same transaction to the same data item may not have been propagated to the database yet. It might have updated a local copy of the data item and deferred the actual database modification. Therefore finding the correct version of a data item becomes more expensive.
  - **b.** This scheme allows less concurrency than the recovery scheme with immediate updates. This is because write-locks are held by transactions till commit time.
  - **c.** For long transaction with many updates, the memory space occupied by log records and local copies of data items may become too high.

17.2 The first phase of recovery is to undo the changes done by the failed transactions, so that all data items which have been modified by them get back the

values they had before the *first* of the failed transactions started. If several of the failed transactions had modified the same data item, forward processing of log-records for undo-list transactions would make the data item get the value which it had before the *last* failed transaction to modify that data item started. This is clearly wrong, and we can see that reverse prcessing gets us the desired result.

The second phase of recovery is to redo the changes done by committed transactons, so that all data items which have been modified by them are restored to the value they had after the *last* of the committed transactions finished. It can be seen that only forward processing of log-records belonging to redo-list transactions can guarantee this.

17.3 Interactive transactions are more difficult to recover from than batch transactions because some actions may be irrevocable. For example, an output (write) statement may have fired a missile, or caused a bank machine to give money to a customer. The best way to deal with this is to try to do all output statements at the end of the transaction. That way if the transaction aborts in the middle, no harm will be have been done.

17.4

- Consider the a bank account A with balance \$100. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. The log records corresponding to the updates of A by transactions T_1 and T_2 would be $< T_1, A, 100, 110 >$ and $< T_2, A, 110, 120 >$ resp.

  Say, we wish to undo transaction T_1. The normal transaction undo mechanism will replaces the value in question—A in this example—by the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism the resulting balance would be \$100 and we would, in effect, undo both transactions, whereas we intend to undo only transaction T_1.
- Let the erroneous transaction be T_e.
  - ☐ Identify the latest checkpoint, say C, in the log before the log record $< T_e$, *START*$>$.
  - ☐ Redo all log records starting from the checkpoint C till the log record $< T_e$, *COMMIT*$>$. Some transaction—apart from transaction T_e—would be active at the commit time of transaction T_e. Let S_1 be the set of such transactions.
  - ☐ Rollback T_e and the transactions in the set S_1.
  - ☐ Scan the log further starting from the log record $< T_e$, *COMMIT*$>$ till the end of the log. Note the transactions that were started after the commit point of T_e. Let the set of such transactions be S_2. Re-execute the transactions in set S_1 and S_2 logically.

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

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

17.5 This is implemented by using `mprotect` to initially turn off access to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler accociated with protection violation then requests a write lock on the page, and after the lock is acquired, it writes the initial contents (before-image) of the page to the log. It then uses `mprotect` to allow write access to the page by the process, and finally allows the process to continue. When the transaction is ready to commit, and before it releases the lock on the page, it writes the contents of the page (after-image) to the log. These before- and after- images can be used for recovery after a crash.

This scheme can be optimized to not write the whole page to log for undo logging, provided the program pins the page in memory.

17.6 We can maintain the LSNs of such pages in an array in a separate disk page. The LSN entry of a page on the disk is the sequence number of the latest log record reflected on the disk. In the normal case, as the LSN of a page resides in the page itself, the page and its LSN are in consistent state. But in the modified scheme as the LSN of a page resides in a separate page it may not be written to the disk at a time when the actual page is written and thus the two may not be in consistent state.

If a page is written to the disk before its LSN is updated on the disk and the system crashes then, during recovery, the page LSN read from the LSN array from the disk is older than the sequence number of the log record reflected to the disk. Thus some updates on the page will be redone unnecessarily but this is fine as updates are idempotent. But if the page LSN is written to the disk to before the actual page is written and the system crashes then some of the updates to the page may be lost. The sequence number of the log record corresponding to the latest update to the page that made to the disk is older than the page LSN in the LSN array and all updates to the page between the two LSNs are lost.

Thus the LSN of a page should be written to the disk only after the page has been written and; we can ensure this as follows: before writing a page containing the LSN array to the disk, we should flush the corresponding pages to the disk. (We can maintain the page LSN at the time of the last flush of each page in the buffer separately, and avoid flushing pages that have been flushed already.)

CHAPTER 18

Data Analysis and Mining

Solutions to Practice Exercises

18.1 query:

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

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

select *student*, **sum**(*marks*) **as** *total*,
    rank() **over** (**order by** (*total*) **desc**) **as** *trank*
from S
groupby *student*
having *trank* $\leq n$

18.3 query:

select *t1.account-number*, *t1.date-time*, *sum(t2.value)*
from *transaction* as *t1*, *transaction* as *t2*
where *t1.account-number* = *t2.account-number* **and**
    t2.date-time < *t1.date-time*
groupby *t1.account-number*, *t1.date-time*
order by *t1.account-number*, *t1.date-time*

18.4 query:

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

18.5 In a destination-driven architecture for gathering data, data transfers from the data sources to the data-warehouse are based on demand from the warehouse, whereas in a source-driven architecture, the transfers are initiated by each source.

The benefits of a source-driven architecture are

- Data can be propagated to the destination as soon as it becomes available. For a destination-driven architecture to collect data as soon as it is available, the warehouse would have to probe the sources frequently, leading to a high overhead.
- The source does not have to keep historical information. As soon as data is updated, the source can send an update message to the destination and forget the history of the updates. In contrast, in a destination-driven architecture, each source has to maintain a history of data which have not yet been collected by the data warehouse. Thus storage requirements at the source are lower for a source-driven architecture.

On the other hand, a destination-driven architecture has the following advantages.

- In a source-driven architecture, the source has to be active and must handle error conditions such as not being able to contact the warehouse for some time. It is easier to implement passive sources, and a single active warehouse. In a destination-driven architecure, each source is required to provide only a basic functionality of executing queries.

- The warehouse has more control on when to carry out data gathering activities, and when to process user queries; it is not a good idea to perform both simultaneously, since they may conflict on locks.

18.6 Consider the following pair of rules and their confidence levels :

No.	Rule	Conf.
1.	$\forall\ persons\ \ P,\ 10000 < P.salary \leq 20000 \Rightarrow P.credit = good$	60%
2.	$\forall\ persons\ \ P,\ 20000 < P.salary \leq 30000 \Rightarrow P.credit = good$	90%

The new rule has to be assigned a confidence-level which is between the confidence-levels for rules 1 and 2. Replacing the original rules by the new rule will result in a loss of confidence-level information for classifying persons, since we cannot distinguish the confidence levels of perople earning between 10000 and 20000 from those of people earning between 20000 and 30000. Therefore we can combine the two rules without loss of information only if their confidences are the same.

18.7 query:

select *store-id*, *city*, *state*, *country*,
date, *month*, *quarter*, *year*,
sum(*number*), **sum**(*price*)
from *sales*, *store*, *date*
where *sales.store-id* = *store.store-id* **and**
sales.date = *date.date*
groupby rollup(*country*, *state*, *city*, *store-id)*,
rollup(*year*, *quarter*, *month*, *date*)

CHAPTER 19

Information Retrieval

Solutions to Practice Exercises

19.1 We do not consider the questions containing neither of the keywords as their relevance to the keywords is zero. The number of words in a question include stop words. We use the equations given in Section 19.2.1 to compute relevance; the log term in the equation is assumed to be to the base 2.

Q#	#wo--rds	# "SQL"	#"rela--tion"	"SQL" term freq.	"relation" term freq.	"SQL" relv.	"relation" relv.	Tota relv.
1	84	1	1	0.0170	0.0170	0.0002	0.0002	0.0004
4	22	0	1	0.0000	0.0641	0.0000	0.0029	0.0029
5	46	1	1	0.0310	0.0310	0.0006	0.0006	0.0013
6	22	1	0	0.0641	0.0000	0.0029	0.0000	0.0029
7	33	1	1	0.0430	0.0430	0.0013	0.0013	0.0026
8	32	1	3	0.0443	0.1292	0.0013	0.0040	0.0054
9	77	0	1	0.0000	0.0186	0.0000	0.0002	0.0002
14	30	1	0	0.0473	0.0000	0.0015	0.0000	0.0015
15	26	1	1	0.0544	0.0544	0.0020	0.0020	0.0041

19.2 Let S be a set of n keywords. An algorithm to find all documents that contain at least k of these keywords is given below :

This algorithm calculates a reference count for each document identifier. A reference count of i for a document identifier d means that at least i of the keywords in S occur in the document identified by d. The algorithm maintains a

list of records, each having two fields – a document identifier, and the reference count for this identifier. This list is maintained sorted on the document identifier field.

```
initialize the list L to the empty list;
for (each keyword c in S) do
begin
   D := the list of documents identifiers corresponding to c;
   for (each document identifier d in D) do
      if (a record R with document identifier as d is on list L) then
            R.reference_count := R.reference_count + 1;
      else begin
            make a new record R;
            R.document_id := d;
            R.reference_count := 1;
            add R to L;
      end;
end;
for (each record R in L) do
   if (R.reference_count >= k) then
      output R;
```

Note that execution of the second *for* statement causes the list D to "merge" with the list L. Since the lists L and D are sorted, the time taken for this merge is proportional to the sum of the lengths of the two lists. Thus the algorithm runs in time (at most) proportional to n times the sum total of the number of document identifiers corresponding to each keyword in S.

19.3 No answer

19.4 No answer

19.5 No answer

CHAPTER 20

Database System Architectures

Solutions to Practice Exercises

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

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

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

A fully distributed system might seem attractive in the presence of powerful clients, but client-server systems still have the advantage of simpler concurrency control and recovery schemes to be implemented on the server alone, instead of having these actions distributed in all the machines.

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

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

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

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

20.4 Since the part which cannot be parallelized takes 20% of the total running time, the best speedup we can hope for has to be less than 5.

20.5 With the central server, each site does not have to remember which site to contact when a particular data item is to be requested. The central server alone needs to remember this, so data items can be moved around easily, depending on which sites access which items most frequently. Other house-keeping tasks are also centralized rather than distributed, making the system easier to develop and maintain. Of course there is the disadvantage of a total shutdown in case the server becomes unavailable. Even if it is running, it may become a bottleneck because every request has to be routed via it.

CHAPTER 21

Parallel Databases

Solutions to Practice Exercises

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

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

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

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

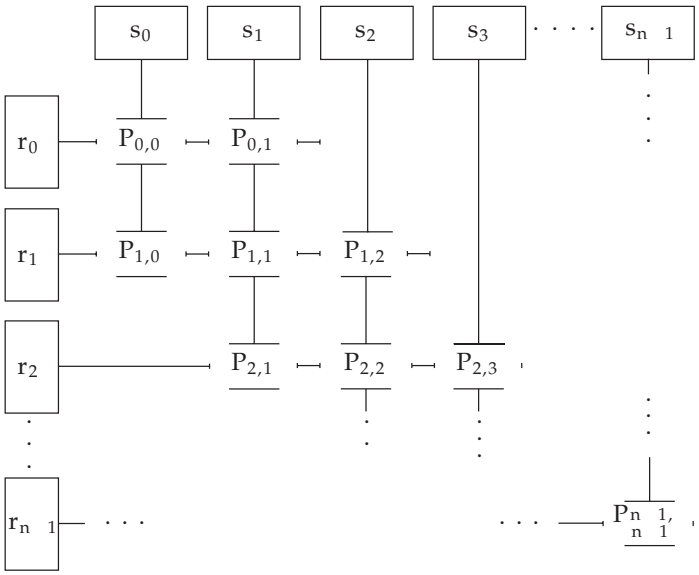
b. With a few large queries, intra-query parallelism is essential to get fast response times. Given that there are large number of processors and disks, only intra-operation parallelism can take advantage of the parallel hardware – for queries typically have few operations, but each one needs to process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

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

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

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

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



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

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

b. Let the histogram ranges be called $h_1, h_2, \ldots, h_h$, and the partitions $p_1, p_2, \ldots, p_p$. Let the frequencies of the histogram ranges be $n_1, n_2, \ldots, n_h$. Each partition should contain N/p tuples, where $N = \Sigma_{i=1}^{h} n_i$.

To construct the load balanced partitioning vector, we need to determine the value of the k_1^{th} tuple, the value of the k_2^{th} tuple and so on, where $k_1 = N/p$, $k_2 = 2N/p$ etc, until k_{p-1}. The partitioning vector will then be $[k_1, k_2, \ldots, k_{p-1}]$. The value of the k_i^{th} tuple is determined as follows. First determine the histogram range h_j in which it falls. Assuming all values in

a range are equally likely, the k_i^{th} value will be

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

where

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

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

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

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

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

CHAPTER 22

Distributed Databases

Solutions to Practice Exercises

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

22.2 **a.** The types of failure that can occur in a distributed system include
   i. Computer failure (site failure).
   ii. Disk failure.
   iii. Communication failure.

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

22.3 A proof that 2PC guarantees atomic commits/aborts inspite of site and link failures, follows. The main idea is that after all sites reply with a <**ready** T> message, only the co-ordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a site can happen only after it ascertains the co-ordinator's decision, either directly from the co-ordinator, or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

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

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

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

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

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

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

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

22.4 Site A cannot distinguish between the three cases until communication has resumed with site B. The action which it performs while B is inaccessible must be correct irrespective of which of these situations has actually occurred, and must be such that B can re-integrate consistently into the distributed system once communication is restored.

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

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

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

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

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of mes-

sages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

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

22.7 In remote backup systems all transactions are performed at the primary site and the data is replicated at the remote backup site. The remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.

The distributed systems offer greater availability by having multiple copies of the data at different sites whereas the remote backup systems offer lesser availability at lower cost and execution overhead.

In a distributed system, transaction code runs at all the sites whereas in a remote backup system it runs only at the primary site. The distributed system transactions follow two-phase commit to have the data in consistent state whereas a remote backup system does not follow two-phase commit and avoids related overhead.

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

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

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

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

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

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

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

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

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

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

22.11 The result is as follows.

$r \ltimes s =$

A	B	C
1	2	3
5	3	2

22.12 The reasons are:

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

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

CHAPTER 23

Application Development and Administration

Solutions to Practice Exercises

23.1 No answer.

23.2 No answer.

23.3 No answer.

23.4 **a.** Let there be 100 transactions in the system. The given mix of transaction types would have 25 transactions each of type A and B, and 50 transactions of type C. Thus the time taken to execute transactions only of type A is 0.5 seconds and that for transactions only of type B or only of type C is 0.25 seconds. Given that the transactions do not interfere, the total time taken to execute the 100 transactions is $0.5 + 0.25 + 0.25 = 1$ second. i.e, the average overall transaction throughput is 100 *transactions per second*.

b. One of the most important causes of transaction interference is lock contention. In the previous example, assume that transactions of type A and B are update transactions, and that those of type C are queries. Due to the speed mismatch between the processor and the disk, it is possible that a transaction of type A is holding a lock on a "hot" item of data and waiting for a disk write to complete, while another transaction (possibly of type B or C) is waiting for the lock to be released by A. In this scenario some CPU cycles are wasted. Hence, the observed throughput would be lower than the calculated throughput.

Conversely, if transactions of type A and type B are disk bound, and those of type C are CPU bound, and there is no lock contention, observed throughput may even be better than calculated.

Lock contention can also lead to deadlocks, in which case some transaction(s) will have to be aborted. Transaction aborts and restarts (which may

also be used by an optimistic concurrency control scheme) contribute to the observed throughput being lower than the calculated throughput.

Factors such as the limits on the sizes of data-structures and the variance in the time taken by book-keeping functions of the transaction manager may also cause a difference in the values of the observed and calculated throughput.

23.5 In the absence of an anticipatory standard it may be difficult to reconcile between the differences among products developed by various organizations. Thus it may be hard to formulate a reactionary standard without sacrificing any of the product development effort. This problem has been faced while standardizing pointer syntax and access mechanisms for the ODMG standard.

On the other hand, a reactionary standard is usually formed after extensive product usage, and hence has an advantage over an anticipatory standard - that of built-in pragmatic experience. In practice, it has been found that some anticipatory standards tend to be over-ambitious. SQL-3 is an example of a standard that is complex and has a very large number of features. Some of these features may not be implemented for a long time on any system, and some, no doubt, will be found to be inappropriate.

CHAPTER 24

Advanced Data Types and New Applications

Solutions to Practice Exercises

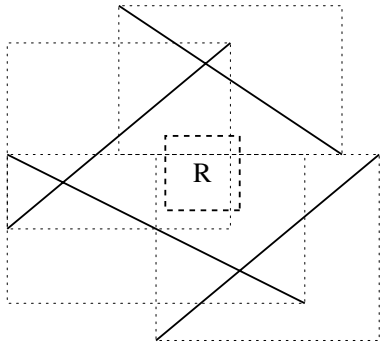
24.1 A temporal database models the changing states of some aspects of the real world. The time intervals related to the data stored in a temporal database may be of two types - *valid time* and *transaction time*. The valid time for a fact is the set of intervals during which the fact is true in the real world. The transaction time for a data object is the set of time intervals during which this object is part of the physical database. Only the transaction time is system dependent and is generated by the database system.

Suppose we consider our sample bank database to be bitemporal. Only the concept of valid time allows the system to answer queries such as - "What was Smith's balance two days ago?". On the other hand, queries such as - "What did we record as Smith's balance two days ago?" can be answered based on the transaction time. The difference between the two times is important. For example, suppose, three days ago the teller made a mistake in entering Smith's balance and corrected the error only yesterday. This error means that there is a difference between the results of the two queries (if both of them are executed today).

24.2 The given query is not a range query, since it requires only searching for a point. This query can be efficiently answered by a B-tree index on the pair of attributes (x, y).

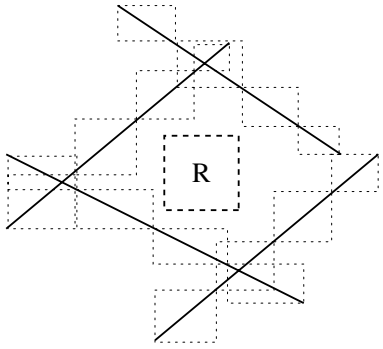
24.3 Suppose that we want to search for the nearest neighbor of a point P in a database of points in the plane. The idea is to issue multiple region queries centered at P. Each region query covers a larger area of points than the previous query. The procedure stops when the result of a region query is non-empty. The distance from P to each point within this region is calculated and the set of points at the smallest distance is reported.

24.4 Large bounding boxes tend to overlap even where the region of overlap does not contain any information. The following figure:



shows a region R within which we have to locate a segment. Note that even though none of the four segments lies in R, due to the large bounding boxes, we have to check each of the four bounding boxes to confirm this.

A significant improvement is observed in the follwoing figure:



where each segment is split into multiple pieces, each with its own bounding box. In the second case, the box R is not part of the boxes indexed by the R-tree. In general, dividing a segment into smaller pieces causes the bounding boxes to be smaller and less wasteful of area.

24.5 Following is a recursive procedure for computing spatial join of two R-trees.

```
SpJoin(node n1, node n2)
begin
    if(the bounding boxes of n1 and n2 do not intersect)
        return;
    if(both n1 and n2 are leaves)
        output all pairs of entries (e1, e2) such that
            e1 ∈ n1 and e2 ∈ n2, and e1 and e2 overlap;
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    if(n1 is not a leaf)
        NS1 = set of children of n1;
    else
        NS1 = { n1 };
    for each ns1 in NS1 and ns2 in NS2;
        SpJoin(ns1, ns2);
end
```

24.6 The concepts of RAID can be used to improve reliability of the broadcast of data over wireless systems. Each block of data that is to be broadcast is split into *units* of equal size. A checksum value is calculated for each unit and appended to the unit. Now, parity data for these units is calculated. A checksum for the parity data is appended to it to form a parity unit. Both the data units and the parity unit are then broadcast one after the other as a single transmission.

On reception of the broadcast, the receiver uses the checksums to verify whether each unit is received without error. If one unit is found to be in error, it can be reconstructed from the other units.

The size of a unit must be chosen carefully. Small units not only require more checksums to be computed, but the chance that a burst of noise corrupts more than one unit is also higher. The problem with using large units is that the probability of noise affecting a unit increases; thus there is a tradeoff to be made.

24.7 We can distinguish two models of broadcast data. In the case of a pure broadcast medium, where the receiver cannot communicate with the broadcaster, the broadcaster transmits data with periodic cycles of retransmission of the entire data, so that new receivers can catch up with all the broadcast information. Thus, the data is broadcast in a continuous cycle. This period of the cycle can be considered akin to the worst case rotational latency in a disk drive. There is no concept of seek time here. The value for the cycle latency depends on the

application, but is likely to be at least of the order of seconds, which is much higher than the latency in a disk drive.

In an alternative model, the receiver can send requests back to the broadcaster. In this model, we can also add an equivalent of disk access latency, between the receiver sending a request, and the broadcaster receiving the request and responding to it. The latency is a function of the volume of requests and the bandwidth of the broadcast medium. Further, queries may get satisfied without even sending a request, since the broadcaster happened to send the data either in a cycle or based on some other receivers request. Regardless, latency is likely to be at least of the order of seconds, again much higher than the corresponding values for a hard disk.

A typical hard disk can transfer data at the rate of 1 to 5 megabytes per second. In contrast, the bandwidth of a broadcast channel is typically only a few kilobytes per second. Total latency is likely to be of the order of seconds to hundreds or even thousands of seconds, compared to a few milliseconds for a hard disk.

24.8 Let C be the computer onto which the central database is loaded. Each mobile computer (host) i stores, with its copy of each document d, a version-vector – that is a set of version numbers $V_{d,i,j}$, with one entry for each other host j that stores a copy of the document d, which it could possibly update.

Host A updates document 1 while it is disconnected from C. Thus, according to the version vector scheme, the version number $V_{1,A,A}$ is incremented by one.

Now, suppose host A re-connects to C. This pair exchanges version-vectors and finds that the version number $V_{1,A,A}$ is greater than $V_{1,C,A}$ by 1, (assuming that the copy of document 1 stored host A was updated most recently only by host A). Following the version-vector scheme, the version of document 1 at C is updated and the change is reflected by an increment in the version number $V_{1,C,A}$. Note that these are the only changes made by either host.

Similarly, when host B connects to host C, they exchange version-vectors, and host B finds that $V_{1,B,A}$ is one less than $V_{1,C,A}$. Thus, the version number $V_{1,B,A}$ is incremented by one, and the copy of document 1 at host B is updated.

Thus, we see that the version-vector scheme ensures proper updating of the central database for the case just considered. This argument can be very easily generalized for the case where multiple off-line updates are made to copies of document 1 at host A as well as host B and host C. The argument for off-line updates to document 2 is similar.

CHAPTER 25

Advanced Transaction Processing

Solutions to Practice Exercises

25.1 **a.** The tasks in a workflow have dependencies based on their status. For example the starting of a task may be conditional on the outcome (such as commit or abort) of some other task. All the tasks cannot execute independently and concurrently, using 2PC just for atomic commit.

b. Once a task gets over, it will have to expose its updates, so that other tasks running on the same processing entity don't have to wait for long. 2PL is too strict a form of concurrency control, and is not appropriate for workflows.

c. Workflows have their own consistency requirements; that is, failure-atomicity. An execution of a workflow must finish in an *acceptable termination state*. Because of this, and because of early exposure of uncommitted updates, the recovery procedure will be quite different. Some form of logical logging and compensation transactions will have to be used. Also to perform forward recovery of a failed workflow, the recovery routines need to restore the state information of the scheduler and tasks, not just the updated data items. Thus simple WAL cannot be used.

25.2

- Loading the entire database into memory in advance can provide transactions which need high-speed or realtime data access the guarantee that once they start they will not have to wait for disk accesses to fetch data. However no transaction can run till the entire database is loaded.
- The advantage in loading on demand is that transaction processing can start rightaway; however transactions may see long and unpredictable delays in disk access until the entire database is loaded into memory.

25.3 A high-performance system is not necessarily a real-time system. In a high performance system, the main aim is to execute each transaction as quickly as

possible, by having more resources and better utilization. Thus average speed and response time are the main things to be optimized. In a real-time system, speed is not the central issue. Here *each* transaction has a deadline, and taking care that it finishes within the deadline or takes as little extra time as possible, is the critical issue.

25.4 In the presence of long-duration transactions, trying to ensure serializability has several drawbacks:-

a. With a waiting scheme for concurrency control, long-duration transactions will force long waiting times. This means that response time will be high, concurrency will be low, so throughput will suffer. The probability of deadlocks is also increased.

b. With a time-stamp based scheme, a lot of work done by a long-running transaction will be wasted if it has to abort.

c. Long duration transactions are usually interactive in nature, and it is very difficult to enforce serializability with interactiveness.

Thus the serializability requirement is impractical. Some other notion of database consistency has to be used in order to support long duration transactions.

25.5 Each thread can be modeled as a transaction T which takes a message from the queue and delivers it. We can write transaction T as a multilevel transaction with subtransactions T_1 and T_2. Subtransaction T_1 removes a message from the queue and subtransaction T_2 delivers it. Each subtransaction releases locks once it completes, allowing other transactions to access the queue. If transaction T_2 fails to deliver the message, transaction T_1 will be undone by invoking a compensating transaction which will restore the message to the queue.

25.6 Consider the advanced recovery algorithm of Section 17.8. The redo pass, which repeats history, is the same as before. We discuss below how the undo pass is handled.

- **Recovery with nested transactions**:

  Each subtransaction needs to have a unique TID, because a failed subtransaction might have to be independently rolled back and restarted.

  If a subtransaction fails, the recovery actions depend on whether the unfinished upper-level transaction should be aborted or continued. If it should be aborted, all finished and unfinished subtransactions are undone by a backward scan of the log (this is possible because the locks on the modified data items are not released as soon as a subtransaction finishes). If the nested transaction is going to be continued, just the failed transaction is undone, and then the upper-level transaction continues.

  In the case of a system failure, depending on the application, the entire nested-transaction may need to be aborted, or, (for e.g., in the case of long duration transactions) incomplete subtransactions aborted, and the nested transaction resumed. If the nested-transaction must be aborted, the rollback can be done in the usual manner by the recovery algorithm, during the undo pass. If the nested-transaction must be restarted, any incomplete

subtransactions that need to be rolled back can be rolled back as above. To restart the nested-transaction, state information about the transaction, such as locks held and execution state, must have been noted on the log, and must restored during recovery. Mini-batch transactions (discussed in Section 23.1.8) are an example of nested transactions that must be restarted.

- **Recovery with multi-level transactions**:
  In addition to what is done in the previous case, we have to handle the problems caused by exposure of updates performed by committed subtransactions of incomplete upper-level transactions. A committed subtransaction may have released locks that it held, so the compensating transaction has to reacquire the locks. This is straightforward in the case of transaction failure, but is more complicated in the case of system failure.

  The problem is, a lower level subtransaction a of a higher level transaction A may have released locks, which have to be reacquired to compensate A during recovery. Unfortunately, there may be some other lower level subtransaction b of a higher level transaction B that started and acquired the locks released by a, before the end of A. Thus undo records for b may precede the operation commit record for A. But if b had not finished at the time of the system failure, it must first be rolled back and its locks released, to allow the compensating transaction of A to reacquire the locks.

  This complicates the undo pass; it can no longer be done in one backward scan of the log. Multilevel recovery is described in detail in David Lomet, "MLR: A Recovery Method for Multi-Level Systems", ACM SIGMOD Conf. on the Management of Data 1992, San Diego.

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

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

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

25.8 **a.** The same system as in the answer to Exercise 25.7 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.