Solutions to selected exercises

An Introduction to Database Systems

Bipin C. Desai

BytePress

Solutions to Selected exercises

from

An Introduction to Database Systems

Bipin C. Desai

Concordia University

Montreal

BytePress

Limit of Liability/Disclaimer of Warranty:

The author and the publisher have taken care to prepare this book. However, there is no warranty of the accuracy, completeness or presentation of the latest version/generation of any system discussed in this book. The reader must be aware of the fact that software systems often have multiple bugs and are not well thought out, and are usually suitable for limited situations and/or data combinations. Hence the user must be responsible for the appropriate application of any technique and use of any software or code examples.

Furthermore, there is no assurance whatsoever of the possible usefulness or commercialization of any programs, scripts and examples given in this book.

Any references given are based on their existence at the time of writing and the authors and the publishers do not endorse them or imply any usefulness of the information found therein. The reader must be aware that any web site cited may change, disappear or change their terms of service.

Published by: Electronic Publishing BytePress.com Inc.
ISBN: 978-1-98-839210-3

Dedication

To my family

Bipin C. Desai

Table of Contents

Preface

The purpose of this guide is to assist the student of an introductory course in database system in conjunction with the author's text Introduction to Database Systems. It is suggested that the student try the exercises given a the end of the chapters before consulting the solutions given here and compare their the solutions.

The organization of this guide follows the suggested plan diagrammed in the preface of the text. Solutions for selected end of chapter exercises are given. The exercises in some of the chapters are of a general nature and their solutions are not included. The chapters on File organization and Higher order normal forms would not be covered in many undergraduate syllabus: and Chapters 8 and 9, are often considered obsolete, and not covered in these courses. Hence the solution to the exercises in these chapters are not provided.

Bipin C. DESAI

1. Basic Concepts

Objectives: This chapter introduces the student to the following concepts and gives overview of a DBMS system.

Concept of modelling for database
Concept of entities and their attributes and keys
Concept of relationship and their attributes
Concept of data integration for sharing
Three level architecture for a database system
Mapping between levels and data independence
Components and structure of a DBMS
Pros and cons of a DBMS

Solution to selected exercises

2. Data Models

Objectives: This chapter introduces the student to the following concepts and gives and overview of different database models.

Concept of data associations and introduction of the concept of functional dependency
Concept of relationships among entities
Entity-Relationship model and its use
Concept of aggregation, generalization and specialization
Introduction to relational, network and hierarchical models
Comparison of these models.

In this chapter we look at the method of modelling entities, and the interrelations of these entities. We introduce the concept of association amongst various attributes of an entity and the relationships among entities. We, also, introduce the data models used in database applications. They differ from each other in the methods used to represent the relationships among entities.

Solution to selected exercises

3. FILE ORGANIZATION

Objectives: This is an optional chapter and may be skipped in most of the programs where, a course in File Systems is a prerequisite or co-requisite to the course in database systems. The chapter introduces the student to the following concepts and gives overview of file systems.

.Characteristics of storage devices
.The components of a file
.Basic file access and primary key retrieval
.Serial, Sequential, and Index-Sequential Files
.Concept of Multi-level Indexing
.Concept of hashing and Direct File organization
.Extensible Hashing
.Secondary Key Retrieval
.Inverted Index Files
.Multi-list Files
.Cellular Files
.Ring Files
.Tree Structured files
. B^+-tree and B-tree and their comparison

In this chapter, we focus on a number of methods used to organize files and the issues involved in the choice of a method. File organization deals with the structure of data on secondary storage devices. In designing the structure the designer is concerned with the access time involved in the retrieval of records based on primary or secondary keys, as well as the techniques involved in updating data. We discuss the following file organization schemes: sequential, index sequential, multi-list, direct, extensible hashing, and tree structured. The general principles involved in these schemes are presented, while not delving into the implementation issues under a specific operating system.

Solution to selected exercises

3.3. (a) Each entry in the bucket will have a key value of 10 bytes and block address of 5 bytes for a total of 15 bytes. This means that we can have a maximum of 66 entries in a block of a bucket. Since there are 1000 buckets. each bucket has 1000 entries and the total number of blocks per bucket is 16. On the average half of these buckets have to be accessed to find an existing record and the actual record would require another access for a total of 9 accesses per record.

3.3. (b) In the index sequential organization, there would be one entry per file block. Each file block contains 5 records, hence there will be a total of 200000 index entries. Since each index block could have 66 entries, the total number of index blocks is 3031. A binary search will require 12 accesses in the index block followed by an access to the file for the actual record, for a total of 13 accesses.

In the index sequential organization, there would be one entry per file block. Each file block contains 5 records, hence there will be a total of 200000 index entries. Since each index block could have 66 entries, the total number of index blocks is 3031. A binary search will require 12 accesses in the index block followed by an access to the file for the actual record, for a total of 13 accesses.

3.3. (c) Each index (internal) node of the B$^+$-tree index would be able to contain a maximum of 66 key values and 67 pointers. We assume that the B$^+$-tree is a dense index, hence each key value is in the leaf node. The number of lowest level internal nodes is 15152. At the next level there will be 230 nodes. Then on the following level, we would have 4 nodes and there would be one node at the root level. The total number of nodes is therefore, 1 + 4 + 230 + 15152 = 15387, and the height of the tree is 4.

3.3. (d) The number of lowest level internal nodes is 30304. At the next level there will be 919 nodes. On the succeeding level, we would have 28 nodes and there would be one node at the root level. The total number of nodes is therefore, 1 + 28 + 919 + 30304 = 31252, and the height of the tree is still 4.

3.4. (a) Since there are ten million records and 10000 buckets, the number of entries per bucket is 10,000,000/10,000 = 1000. Since this represents half the capacity of the bucket, each bucket is to have a capacity for 2000 entries. Each entry consists of a key value and a block address and requires 10 bytes. Hence, the size of the bucket is 20,000 bytes, or, it requires 2 physical blocks. Since hashing is used, the first block access would find the correct bucket and the next access would find the appropriate record.

3.4. (b) The number of data blocks is given by:

$$2*10*10^6*100/10,000 = 2*10^5$$

Since there is an entry in the index for each block, the number of blocks needed for this level of indexing is given by:

$$2*10^5*10/10,000 = 200$$

The binary search in these number of blocks would need 8 accesses followed by an access to one data block for a total of 9 accesses.

3.4. (c) The number entries in the leaf nodes is $10*10^6$ and assuming a leaf node contains a key value and a block address, there will be 1000 entries per leaf node and 10,000 leaf nodes. This is also the number of entries in the lowest level internal node of the B$^+$-tree. Assuming the nodes are full, there would be 200 nodes at this level and one node at the root level. The height of the tree is two. The number of disk accesses required, is two for the root and the internal node of the B$^+$-tree combined, followed by one for the leaf node, as well as one for the data block containing the actual record.

3.5. (a) With a block size of 5,000 bytes and a record size of 200 bytes, the blocking factor is 25. Hence, the total number of blocks in the file would be 1,000,000/25 = 40,000.

Time to read each block = $25*10^{-3} + 5{,}000/(100 * 10^{3})$
$= 75* 10^{-3}$ sec
Time to process the tape = $40{,}000*75*10^{-3}$ sec
= 3,000 sec

3.5. (b) With a block size of 50,000 bytes Title and a record size of 200 bytes, the blocking factor is 250. Hence, the total number of blocks in the file would be 1,000,000/250 = 4,000.

Time to read each block = $25*10^{-3} + 50{,}000/(100 * 10^{3})$
$= 525* 10^{-3}$ sec

Time to process the tape = $4{,}000*525*10^{-3}$ sec
= 2,100 sec

3.7.
Since 80% of the tape is to be used to record data, each block would be of a 2.4 inch length giving a blocking factor of 120.

3.11. (a)

Directory

Dept	Head
CHEM	4
COMP	1
ELEC	3
ENGL	5
PHYS	2

Advisor	Head
ACIAN R	4
BROST A	5
JONES A	2
MARTIN R	13
NEWELL J	11
SMITH F	1
WAGNER B	3

Status	Head
F2	1
F3	2
I1	3
I2	8
I3	9
P1	13
P2	5
P3	11

3.11. (b)

File

Rec#	Name	Id	Dept	Ptr	Advisor	Ptr	Status	Ptr
1	MICROSLAW	3634592	COMP	8	SMITH F	10	F2	6
2	PASSASLO	3894336	PHYS	7	JONES A	6	F3	4
3	PRONOVOST	6888954	ELEC	9	WAGNER B	6	I1	⊥
4	LOANNIDES	3518445	CHEM	6	ACIAN R	⊥	F3	7
5	MACIOCIA	7564019	ENGL	11	BROST A	⊥	P2	10
6	CHO BYUNG	2566984	CHEM	14	JONES A	7	F2	12
7	CANNON	7868286	PHYS	⊥	JONES A	8	F3	⊥
8	BERGEROM	2736849	COMP	10	JONES A	⊥	I2	14
9	ABOND	7382943	ELEC	15	WAGNER B	12	I3	15
10	HAMMERBELL	6792839	COMP	12	SMITH F	14	P2	⊥
11	LANGEVIN	2768736	ENGL	⊥	NEWELL J	⊥	P3	⊥
12	PELLERIN	6689184	COMP	13	WAGNER B	⊥	F2	⊥
13	ROBERT	3707939	COMP	⊥	MARTIN R	⊥	P1	⊥
14	SHARPE	9877546	CHEM	⊥	SMITH F	15	I2	⊥
15	PETIT	2742619	ELEC	⊥	SMITH F	⊥	I3	⊥

For *Dept* = COMP we access record 1, and make the following entry in the DONTAG list: <record 1, *Advisor* = Smith F, 10>; <record 1, *Status* = F2, 6>. We find the next record for *Dept* = COMP list to be 8 and access this record. No entries are made in the DONTAG list. The next record to be accessed for *Dept* = COMP list is 10. Here we make the following entry in the DONTAG list: <record 10, *Advisor* = Smith F, 14>. The next record to be accessed for *Dept* = COMP list is 12. Here we make the following entry in the DONTAG list: <record 12, *Status* = F2, $\perp$> The last record to be accessed for *Dept* = COMP list is 13. No entries are made in the DONTAG list. At the end of traversing this list, the DONTAG list contains the following entries:

<record 1, *Advisor* = Smith F, 10>
<record 1, *Status* = F2, 6>
<record 10, *Advisor* = Smith F, 14>
<record 12, *Status* = F2, $\perp$>

Now the list for *Status* = F2 is to be traversed. The first record from the directory is found to be 1. Before accessing this record, the DONTAG list is consulted and it is found that there is an entry in it for this record which also indicates that the next record for *Status* = F2 to be 6. Since from the DONTAG list we find that the record has not been previously accessed, we access it. Processing this record we find that the DONTAG list does not have to be updated and we discover, further, that the next record having the same value for *Status* is 12. However, the DONTAG list entry <record 12, *Status* = F2, $\perp$> indicates that this record was already accessed and there are no further records in this list.

Now the list for *Advisor* = Smith F is to be processed starting with the record 1. Consulting the DONTAG list, we conclude that this record was already accessed and that the next record in this list is 10. Again the DONTAG list tells us that this last record was already processed earlier and that we now have to access and process record 14. Since there is no entry for record 14 in the DONTAG list, we access it and find that the next record to be accessed is record 15. This last record is the tail of the list and we have accessed all records satisfying the query.

3.11. (c)

Directory with an entry per cell

Dept	Head
CHEM	4,14
COMP	1,8,10,13
ELEC	3,9,15
ENGL	5,11
PHYS	2,7

Advisor	Head
ACIAN R	4
BROST A	5
JONES A	2,6,7
MARTIN R	13
NEWELL J	11
SMITH F	1,10,14
WAGNER B	3,9,12

Status	Head
F2	1,6,12
F3	2,4,7
I1	3
I2	8,14
I3	9,15
P1	13
P2	5,10
P3	11

Cellular File

Rec#	Name	Id	Dept	Ptr	Advisor	Ptr	Status	Ptr
1	MICROSLAW	3634592	COMP	⊥	SMITH F	⊥	F2	⊥
2	PASSASLO	3894336	PHYS	⊥	JONES A	⊥	F3	⊥
3	PRONOVOST	6888954	ELEC	⊥	WAGNER B	⊥	I1	⊥
4	LOANNIDES	3518445	CHEM	6	ACIAN R	⊥	F3	⊥
5	MACIOCIA	7564019	ENGL	⊥	BROST A	⊥	P2	⊥
6	CHO BYUNG	2566984	CHEM	⊥	JONES A	⊥	F2	⊥
7	CANNON	7868286	PHYS	⊥	JONES A	8	F3	⊥
8	BERGEROM	2736849	COMP	⊥	JONES A	⊥	I2	⊥
9	ABOND	7382943	ELEC	⊥	WAGNER B	⊥	I3	⊥
10	HAMMERBELL	6792839	COMP	12	SMITH F	⊥	P2	⊥
11	LANGEVIN	2768736	ENGL	⊥	NEWELL J	⊥	P3	⊥
12	PELLERIN	6689184	COMP	⊥	WAGNER B	⊥	F2	⊥
13	ROBERT	3707939	COMP	⊥	MARTIN R	⊥	P1	⊥
14	SHARPE	9877546	CHEM	⊥	SMITH F	15	I2	⊥
15	PETIT	2742619	ELEC	⊥	SMITH F	⊥	I3	⊥

3.16. File before modifications:

Cyl. 41 — Sectors

Surf.	1	2	3	4
00	Tr.Index	A_1, A_4	A_5, A_6, A_9	A_{10}, A_{13}
01	A_{17}, A_{18}	A_{20}	A_{28}, A_{29}	A_{30}, A_{31}, A_{36}
02	A_{42}, A_{43}	A_{45}, A_{46}, A_{48}	A_{51}, A_{52}, A_{56}	A_{59}, A_{61}
03	A_{75}, A_{76}, A_{78}	A_{79}, A_{80}	A_{83}	A_{89}, A_{91}
04	A_{93}, A_{94}	A_{96}, A_{98}	A_{100}	A_{120}, A_{125}
05	⊥	⊥	⊥	⊥

Initial track index

Prime Key	Address	Overflow Key	Address
A_{13}	41002	A_{13}	⊥
A_{36}	41011	A_{36}	⊥
A_{61}	41021	A_{61}	⊥
A_{91}	41031	A_{91}	⊥
A_{125}	41041	A_{125}	⊥

File after modifications

Cyl. 41

Surf.	Sectors 1	2	3	4
00	Tr.Index	A_1, A_2, A_3	A_5, A_6	A_9, A_{10}, A_{13}
01	A_{17}, A_{18}	A_{20}	A_{28}, A_{29}	A_{30}, A_{31}, A_{33}
02	A_{41}, A_{42}, A_{43}	A_{45}, A_{46}, A_{48}	A_{51}, A_{52}, A_{54}	A_{56}, A_{59}, A_{60}
03	A_{75}, A_{76}, A_{78}	A_{79}, A_{80}	A_{82}, A_{84}	A_{89}, A_{91}
04	A_{93}, A_{94}	A_{95}, A_{96}, A_{98}	A_{100}	$A_{120}, A_{122}, A_{124}$
05	A_{34}, *A_{36}, *A_{61}	*A_{125}	$\perp$	$\perp$

Track index after file modifications

Prime Key	Address	Overflow Key	Address
A_{13}	41002	A_{13}	$\perp$
A_{33}	41011	A_{36}	41051
A_{60}	41021	A_{61}	41051
A_{91}	41031	A_{91}	$\perp$
A_{124}	41041	A_{125}	41052

4. The Relational Model

Objectives: This chapter introduces the student to the Relational data model and relational algebra and calculus. The following concepts are introduced:

Concept of attributes, domains, tuples, and relations
Operation on relations
Integrity rules
Relation Schemes
Representing relations
Relational Algebra and operations
Relational Calculus
Tuple Calculus
Domain Calculus
Comparison of Relational Algebra and Relational Calculus

Solution to selected exercises

4.1. Relations P and Q are as follows:

P

A	B	C	D
a1	b2	c2	d2
a2	b1	c1	d2
a1	b1	c2	d1
a2	b1	c2	d2
a1	b2	c1	d2
a3	b1	c2	d1
a2	b1	c2	d1
a1	b3	c2	d2

Q

B	C	D
b1	c1	d2
b3	c1	d2
b2	c2	d1
b3	c2	d2

4.1.1. Find the projection of Q on the attributes (B,C).

$\pi_{B,C}Q$

B	C
b1	c1
b3	c1
b2	c2
b3	c2

4.1.2. Find the natural join of P and Q on the common attributes.

$P \bowtie Q$

A	B	C	D
a2	b1	c1	d2
a1	b3	c2	d2

4.1.3. Divide P by the relation that is obtained by first selecting those tuples of Q where the value of B is either b1 or b2 and then projecting Q on the attributes (C,D).

$\pi_{C,D}\sigma_{B=b1 \vee B=b2}Q$

C	D
c1	d2
c2	d1

$P \div \pi_{C,D}\sigma_{B=b1 \vee B=b2}Q$

A	B
a2	b1

4.2 The relational database scheme is given as:

PARTS(<u>*P#*</u>,*Name*,*Colour*)

SUPPLIER(*S#*,*Name*,*Address*)
CAN_SUPPLY(*S#*,*P#*,*Quality*)
SUPPLY(*S#*,*P#*,*Price*,*Qty*)

The relations CAN_SUPPLY and SUPPLY contain foreign keys *S#*, *P#*. Presence of foreign keys requires the maintenance of referential integrity. The addition, deletion and modification of tuples must ensure this integrity.

4.3.(a)

$$\pi_{(\text{SUPPLIER.S\#},\text{SUPPLIER.Name},\text{SUPPLIER.Address},\text{SUPPLY.Price})}(X)$$

where X is given by

$$\pi_{\text{PARTS.P\#}}\sigma_{\text{PARTS.Name='bolts'}}(\text{PARTS}) \bowtie \text{SUPPLY} \bowtie \text{SUPPLIER}$$

```
{s|∃t,u,v(t∈SUPPLIER ∧ u∈PARTS ∧ v∈SUPPLY
          ∧ u[Name] = 'bolts'
          ∧ t[S#] = v[S#]
          ∧ u[P#] = v[P#]
          ∧ s[S#] = t[S#]
          ∧ s[Name] = t[Name]
          ∧ s[Price] = v[Price])}
```

4.3 (b)
Let Y = PARTS $\bowtie$ SUPPLY $\bowtie$SUPPLIER
Now find the unary relation R containing the *S#* of suppliers who supply bolts costing less than \$0.01.

R = $\pi_{(\text{SUPPLIER.}S\#)}(\sigma_{(\text{PARTS.}Name=\text{'bolts'} \wedge \text{SUPPLY.}Price<.01)}Y$

The details of the parts supplied by these SUPPLIERs is obtained as follows:
$\pi_{(\text{PARTS.}P\#,\text{PARTS.}Name,\text{PARTS.}Colour,\text{R.}S\#)}(\sigma_{\text{CAN\_SUPPLY.}Quality>x}Z)$

where Z is the relation obtained by a natural join of R, CAN_SUPPLY, and PARTS.

```
{x|∃ r,s,t(r ∈ SUPPLIER  ∧ s ∈ PARTS ∧ t ∈ SUPPLY
            ∧ t[Price] < 0.01
            ∧ s[Name] = 'bolts'
            ∧ s[P#] = t[P#]
            ∧ r[S#] = t[S#]
            ∧ ∃ u,v(u ∈ CAN_SUPPLY ∧ v ∈ PARTS
            ∧ u[S#] = r[S#]
            ∧ u[Quality] > x
            ∧ u[P#] = v[P#]
            ∧ x[S#] = u[S#]
            ∧ x[P#] = v[P#]
            ∧ x[Name] = v[Name]
            ∧ x[Colour] = v[Colour])))
```

4.4. (a)

$$\pi_{S\#}\text{ENROLL} \bowtie (\sigma_{(\text{TEACH.}Prof=\text{'Smith'} \vee \text{TEACH.}Prof=\text{'Jones'})}\text{TEACH})$$

$$\{s \mid \exists e,t(e \in \text{ENROLL} \wedge t \in \text{TEACH} \wedge (t[Prof] = \text{'Smith'} \vee t[Prof] = \text{'Jones'})$$
$$\wedge\ e[C\#] = t[C\#] \wedge e[Section] = t[Section] \wedge s[S\#] = e[S\#]\}$$

4.4. (b)

$$\pi_{S\#}(\text{ENROLL} \bowtie \text{TEACH} \bowtie \text{ADVISE})$$

$$\{s \mid \exists e,t,a\ (e \in \text{ENROLL} \wedge t \in \text{TEACH} \wedge a \in \text{ADVISE} \wedge e[C\#] = t[C\#] \wedge$$
$$e[Section] = t[Section] \wedge t[Prof] = a[Prof] \wedge a[S\#] = e[S\#] \wedge s[S\#] = a[S\#]\}$$

4.4. (c)
Let TEACH1 and TEACH2 be copies of the relation TEACH.
Let R = TEACH1 x TEACH2, then

$$S=\sigma_{(\text{TEACH1.}Prof=\text{TEACH2.}Prof \wedge \text{TEACH1.}C\#=\text{TEACH2.}C\# \wedge \text{TEACH1.}Section \neq \text{TEACH2.}Section)}(R)$$

The required response is given by $\pi_{\text{TEACH1.}Prof}S$

$$\{p \mid \exists t1, t2\ (t1 \in \text{TEACH} \wedge t2 \in \text{TEACH} \wedge t1[C\#] = t2[C\#] \wedge t1[Section] \neq t2[Section]$$
$$\wedge\ t1[Prof] = t2[Prof] \wedge p[Prof] = t2[Prof])$$

4.4. (d)
A way to tackle this rather complex query is to break it down into a set of simpler queries and then deal with them individually. This approach is illustrated below:

- Find the set of courses that Mr. Doe has passed: PASS(*C#*)

- Find the courses that Mr. Doe cannot do: CANNOTDO(*C#*)

- Subtract the above set of courses from the set of all courses to get those that he can do.
  CANDO(*C#*)

- Some of these may have been already completed. Therefore, to find the required response subtract from the above set of courses that can be done by Mr. Doe, those that he has passed.

- Find the set of courses that Mr. Doe has passed: PASS(*C#*)

$$\text{PASS}(C\#)=\pi_{C\#}((\sigma_{\text{GRADES.}Grade \neq \text{'F'}}\text{GRADES}) \bowtie (\sigma_{\text{STUDENT.}Sname=\text{'John Doe'}}\text{STUDENT}))$$

-The courses being offered are given by the projection of TEACH on *C#*.

-To find the courses that Mr. Doe can do we find those courses for which he has the required prerequisite. This is obtained by performing the Cartesian product of the courses being offered with PASS. The attribute of PASS being renamed *Pre_C#*. Let us call this relation HAS_PRE_REQ(*C#*, *Pre_C#*)

$$\text{HAS\_PRE\_REQ}(C\#, Pre\_C\#) = (\pi_{\text{TEACH}.C\#}\text{TEACH}) \times \text{PASS}[Pre\_C\#]$$

Now the courses that Mr. Doe cannot do is given by:

$$\text{CANNOTDO}(C\#) = \pi_{C\#}(\text{PRE\_REQ} - \text{HAS\_PRE\_REQ})$$

Courses he can do is then given by:

$$\text{CANDO}(C\#) = (\pi_{C\#}\text{TEACH}) - \text{CANNOTDO}$$

Subtracting the courses already completed, we get the courses for which Mr. Doe can now enrol:

$\text{CAN\_ENROLL}(C\#) = \text{CANDO}(C\#) - \text{PASS}(C\#)$.

4.5. (a) Let us first find the relation X as follows:
$X = \sigma_{(Conductor=\text{'Letitia Melody'})}Y$

where Y(*Conductor,Composition,Player,Instrument)* is given as :
Y= CONDUCTS ⋈ REQUIRES ⋈ PLAYS
Then the list of players and their instruments that can be part of the orchestra when Letitia Melody conducts is given by:

$$\pi_{(Player,Instrument)}X$$

$\{z \mid \exists c,r,p(c \in \text{CONDUCTS} \wedge r \in \text{REQUIRES} \wedge p \in \text{PLAYS}$

$\wedge\, c[Conductor] = \text{'Letitia Melody'} \wedge c[Composition] = r[Composition]$

$\wedge\, r[Instrument] = p[Instrument] \wedge z[Player] = p[Player] \wedge z[Instrument] = p[Instrument])\}$

Note: The schema of z define in the TRC query,

4.5 (b) Let TEMP be the relation defined below:
$\text{TEMP} = \sigma_{Conductor=\text{'Letitia Melody'}}\text{CONDUCTS}$

Then the players who like the composition they are likely to play is given by:

$\pi_{\text{LIKES.}Player}$(REQUIRES $\bowtie$ PLAYS $\bowtie$ LIKES $\bowtie$ TEMP)

{x | ∃c,r,p,l(c ∈ CONDUCTS ∈ r ∈ REQUIRES ∧ p ∈ PLAYS ∧ l ∈ LIKES
∧ c[*Conductor*] = 'Letitia Melody' ∧ c[*Composition*] = r[*Composition*]∧ l[*Player*] = x[*Player*]
∧ c[*Composition*] = l[*Composition*] ∧ r[*Instrument*] = p[*Instrument*] ∧ l[*Player*] = p[*Player*])}

4.7 (a) Select tuples from rel_1 such that the attribute B has either the value B_1 or B_2.
4.7 (b) $\sigma_{B='B1' \vee B='B2'}(rel_1)$

4.7(c) $\{t \mid t \in rel_1 \wedge (t[B] = 'B_1' \vee t[B] = 'B_2')\}$

4.10. "Get complete details of employees working on a Database project."

{s | s ∈ EMPLOYEE ∧ ∃u,t(t ∈ PROJECT ∧ t[*Project_Name*] = 'Database'
∧ u ∈ ASSIGNED_TO ∧ u[*Project#*] = t[*Project#*] ∧ s[*Emp#*] = u[*Emp#*])}

The above can be written using the identity ∃xA(x) = ¬∀x(¬A(x)) as follows:

{s | s ∈ EMPLOYEE ∧ ¬∀u,t (t ∉ PROJECT ∨ t[*Project_Name*] ≠ 'Database'
∨ u ∉ASSIGNED_TO ∨ u[*Project#*] ≠ t[*Project#*] ∨ s[*Emp#*] ≠ u[*Emp#*])}

The query "Get complete details of employees working on **all** Database projects" can be expressed as follows:

{s | s ∈ EMPLOYEE ∧ ∀t (t ∉ PROJECT ∨ t[Project-Name] ≠ 'Database'
∨ ∃u(u ∈ ASSIGNED_TO ∧ u[Project#] = t[Project#] ∧ s [Emp#] = u[Emp#])}

The above can be written using the negating both of the identity ∃xA(x) = ¬∀x(¬A(x)
i.e., ¬(∃xA(x))= ¬(¬∀x(¬A(x))) which is:
∀x(A(x)) = ¬∃x(¬A(x))as follows:

{s | s ∈ EMPLOYEE ∧ ¬∃t(¬{t ∉ PROJECT ∨ t[Project-Name] ≠ 'Database'
∨ ∃u(u ∈ ASSIGNED_TO ∧ u[Project#] = t[Project#] ∧ s [Emp#] = u[Emp#])}]

{s | s ∈ EMPLOYEE ∧ ¬∃t(t ∈ PROJECT ∧ t[*Project_Name*] = 'Database'
∧¬∃ u (u ∈ASSIGNED_TO ∧ u[*Project#*] = t[*Project#*] ∧ s[*Emp#*] = u[*Emp#*])}

"List the complete details of employees working on both COMP353 and COMP354."

{s | s ∈ EMPLOYEE ∧ $\exists u_1, u_2$ (u_1 ∈ ASSIGNED_TO

$\wedge\ u_2 \in \text{ASSIGNED\_TO} \wedge u_1[\mathit{Emp\#}] = u_2[\mathit{Emp\#}]$
$\wedge\ s[\mathit{Emp\#}] = u_1[\mathit{Emp\#}] \wedge u_1[\mathit{Project\#}] = \text{'COMP353'}$
$\wedge\ u_2[\mathit{Project\#}] = \text{'COMP354')}\}$

Interchanging the quantifiers using $\exists x A(x) = \neg\forall x(\neg A(x))$ we get:

$\{s \mid s \in \text{EMPLOYEE} \wedge \neg\forall u_1,u_2\ (u_1 \notin \text{ASSIGNED\_TO}$
$\vee\ u_2 \notin \text{ASSIGNED\_TO} \vee u_1[\mathit{Emp\#}] \neq u_2[\mathit{Emp\#}]$
$\vee s[\mathit{Emp\#}] \neq u_1[\mathit{Emp\#}] \vee u_1[\mathit{Project\#}] \neq \text{'COMP353'}$
$\vee\ u_2[\mathit{Project\#}] \neq \text{'COMP354')}\}$

Exercise: modify the above query to read "List the complete details of employees working on either 'COMP353' or COMP354 or both."

"Get employee numbers of employees, excluding employee 107, who works on at least one project that employee 107 works on".

$\{t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge \exists s\ (s \in \text{ASSIGNED\_TO} \wedge s[\mathit{Emp\#}] = 107$
$\wedge\ \exists u(u \in \text{ASSIGNED\_TO} \wedge s[\mathit{Project\#}] = u[\mathit{Project\#}]$
$\wedge\ u[\mathit{Emp\#}] \neq 107 \wedge t[\mathit{Emp\#}] = u[\mathit{Emp\#}])\}$

Interchanging the quantifiers using $\exists x A(x) = \neg\forall x(\neg A(x))$ we get:

$\{t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge \neg\forall s\ (s \notin \text{ASSIGNED\_TO} \vee s[\mathit{Emp\#}] \neq 107$
$\vee \neg\exists u(u \in \text{ASSIGNED\_TO} \wedge s[\mathit{Project\#}] = u[\mathit{Project\#}]$
$\wedge\ u[\mathit{Emp\#}] \neq 107 \wedge t[\mathit{Emp\#}] = u[\mathit{Emp\#}])\}$

"Get employee numbers of employees who do not work on project COMP453".

$\{\ t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\neg\exists u(u \in \text{ASSIGNED\_TO} \wedge u[\mathit{Project\#}] = \text{'COMP453'} \wedge t[\mathit{Emp\#}] = u[\mathit{Emp\#}])\}$

Interchanging the quantifiers using $\exists x A(x) = \neg\forall x(\neg A(x))$ we get:

$\{\ t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge \forall u\ (u \notin \text{ASSIGNED\_TO}$
$\vee\ u[\mathit{Project\#}] \neq \text{'COMP453'} \vee t[\mathit{Emp\#}] \neq u[\mathit{Emp\#}])\}$

"Compile a list of employee numbers of employees who work on all projects."

$\{ t[Emp\#] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\forall p(p \in \text{PROJECT} \rightarrow \exists u(u \in \text{ASSIGNED\_TO}$
$\wedge p[Project\#] = u[Project\#] \quad \wedge t[Emp\#] = u[Emp\#]))\}$

This can be re-written $f \rightarrow g$ can be replaced by $\neg f \vee g$:

$\{ t[Emp\#] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\forall p(p \notin \text{PROJECT} \vee \exists u(u \in \text{ASSIGNED\_TO}$
$\wedge p[Project\#] = u[Project\#]$
$\wedge t[Emp\#] = u[Emp\#]))\}$

Interchanging the quantifiers using $\forall x(A(x)) = \neg\exists x(\neg A(x))$, we get:
$\{ t[Emp\#] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\neg\exists p(p \in \text{PROJECT} \wedge \neg\exists u(u \in \text{ASSIGNED\_TO}$
$\wedge p[Project\#] = u[Project\#] \quad \wedge t[Emp\#] = u[Emp\#]))\}$

"Get employee numbers of employees, not including employee 107, who work on at least one project that employee 107 works on".

$\{ t[Emp\#] \mid t \in \text{ASSIGNED\_TO} \wedge$
$\exists s,u (s \in \text{ASSIGNED\_TO} \wedge u \in \text{ASSIGNED\_TO}$
$\wedge s[Project\#] = u[Project\#] \wedge s[Emp\#] = 107$
$\wedge t[Emp\#] \neq 107 \wedge t[Emp\#] = u[Emp\#])\}$

After interchanging the quantifiers, we get:

$\{ t[Emp\#] \mid t$ é $\text{ASSIGNED\_TO} \wedge$
$\neg\forall s,u (s \neq \text{ASSIGNED\_TO} \vee u \notin \text{ASSIGNED\_TO}$
$\vee s[Project\#] \neq u[Project\#] \vee s[Emp\#] \neq 107$
$\vee t[Emp\#] = 107 \vee t[Emp\#] \neq u[Emp\#])\}$

4.12 (a) Acquire details of the projects for each employee by name.
$\pi_{\text{EMPLOYEE.}EmpName\text{,PROJECT.}Project\#\text{,PROJECT.}Project\_Name\text{,PROJECT.}Cheif\_Architect}(X)$

Here the relation X is given as: $\text{ASSIGNED\_TO} \bowtie \text{EMPLOYEE} \bowtie \text{PROJECT}$

4.12 (b) Compile the names of projects to which employee 107 is assigned.

Let X be the relation as : $(\sigma_{\text{EMPLOYEE.}EmpN\#=107}(\text{ASSIGNED\_TO})) \bowtie \text{PROJECT}$

Then the project names are obtained as: $\pi_{\text{PROJECT.}Project\_Name}(X)$

4.12 (c) Access all employees assigned to projects whose chief architect is employee 109.

The required employee numbers are given by:

$$\pi_{\text{EMPLOYEE.}Emp\#}((\text{ASSIGNED\_TO}) \bowtie (\text{X}))$$

where the relation X is given by:

$$\text{X} = \pi_{\text{PROJECT.}Project\#}(\sigma_{\text{PROJECT.}Cheif\_Architect=109}(\text{PROJECT}))$$

4.12 (d) Derive the list of employees who are assigned to **all** projects where employee 109 is the chief architect.

The list is given by ASSIGNED_TO ÷ X, where X is obtained as follows:

$$\text{X} = \pi_{.Project\#}(\sigma_{Cheif\_Architect=109}(\text{PROJECT}))$$

4.12 (e) Get all project names to which employee 107 is not assigned.
Let X be given by:

$$\text{X} = \pi_{Project\#}(\sigma_{Emp\#=107}(\text{ASSIGNED\_TO}))$$

and let Y. the project numbers where 107 is not assigned is given by:

$$\text{Y} = \pi_{Project\#}\text{PROJECT} - \text{X}$$

Then the required response is given by Z where Z is:

$$\text{Z} = \pi_{\text{PROJECT.}Project\_\text{Name}}(\text{PROJECT} \bowtie \text{Y})$$

4.12 (f) Get complete details of employees who are assigned to projects not assigned to employee 107.
Let X be given by:

$$\text{X} = \pi_{\text{PROJECT.}Project\#}(\sigma_{\text{ASSIGNED\_TO.}Emp\#=107}(\text{ASSIGNED\_TO}))$$

and let Y be given by:

$$\text{Y} = \text{ASSIGNED\_TO} \bowtie (\pi_{\text{PROJECT.}Project\#}\text{PROJECT} - \text{X})$$

Then the requires response is given by:

$$\pi_{\text{EMPLOYEE.}Emp\#\text{EMPLOYEE.}EmpName}(\text{EMPLOYEE} \bowtie \text{Y})$$

4.13 (a) Acquire details of the projects for each employee by name.

$$\{e[EmpName],p \mid \exists\, e,a,p(e \in \text{EMPLOYEE} \wedge a \in \text{ASSIGNED\_TO} \wedge p \in \text{PROJECT} \wedge a[Project\#] = p[Project\#] \wedge a[Emp\#] = e[Emp\#]\}$$

4.13 (b) Compile the names of project to which employee 107 is assigned.

$$\{p[Project\_Name] \mid \exists\, a,p\ (a \in \text{ASSIGNED\_TO} \wedge p \in \text{PROJECT} \wedge a[Emp\#] = 107 \wedge a[Project\#] = p[Project\#]\ )\}$$

4.13 (c)   Access all employees assigned to projects whose chief architect is employee 109.

$\{a[\mathit{Emp\#}] \mid \exists\ a,p\ (a \in \text{ASSIGNED\_TO} \wedge p \in \text{PROJECT}$
$\wedge\ p[\mathit{Cheif\_Architect}] = 109 \wedge p[\mathit{Project\#}] = a[\mathit{Project\#}]\ )\}$

4.13 (d)   Derive the list of employees who are assigned to all projects where employee 109 is the chief architect.

$\{t[\mathit{Emp\#}] \mid t \in \text{ASSIGNED\_TO} \wedge \forall p(p \in \text{PROJECT} \vee p[\mathit{Cheif\_Architect}] \neq 109$
$\vee \exists\ a\ (a \in \text{ASSIGNED\_TO} \wedge p[\mathit{Project\#}] = a[\mathit{Project\#}] \wedge t[\mathit{Emp\#}] = a[\mathit{Emp\#}]\ ))\}$

4.13 (e)   Get all project names to which employee 107 is not assigned.

$\{\ p[\mathit{Project\_Name}] \mid p \in \text{PROJECT} \wedge \neg\exists a\ (a \in \text{ASSIGNED\_TO}$
$\wedge\ p[\mathit{Project\#}] = a[\mathit{Project\#}] \wedge a[\mathit{Emp\#}] = 107\ )\}$

4.13 (f) Get complete details of employees who are assigned to projects not assigned to employee 107

$\{\ e \mid e \in \text{EMPLOYEE} \wedge \exists\ e1,a,p\ (e1 \in \text{EMPLOYEE} \wedge a \in \text{ASSIGNED\_TO}$
$\wedge\ p \in \text{PROJECT} \wedge e[\mathit{Emp\#}] = e1[\mathit{Emp\#}] \wedge e1[\mathit{Emp\#}] = a[\mathit{Emp\#}]$
$\wedge\ p[\mathit{Project\#}] = a[\mathit{Project\#}] \wedge \neg$Éa1$(a1 \in \text{ASSIGNED_TO}$
$\wedge\ a1[\mathit{Project\#}] = p[\mathit{Project\#}] \wedge a[\mathit{Emp\#}] = 107\ ))\}$

4.14 (a) Acquire details of the projects for each employee by name.

$\{\langle m,p,n,c\rangle \mid \exists\ e1,p1(\langle e1,m\rangle \in \text{EMPLOYEE} \wedge \langle p1,e1\rangle \in \text{ASSIGNED\_TO}$
$\wedge\ \langle p1,n,c\rangle \in \text{PROJECT} \wedge p = p1)\}$

4.14 (b) Compile the names of projects to which employee 107 is assigned.

$\{\langle n\rangle \mid \exists\ p,e,p1,c(\langle p,e\rangle \in \text{ASSIGNED\_TO}$
$\wedge\ \langle p1,n,c\rangle \in \text{PROJECT} \wedge p = p1 \wedge e = 107)\}$

4.14 (c) Access all employees assigned to projects whose chief architect is employee 109.

$\{\langle e\rangle \mid \exists\ p, n,c\ (\langle p,e\rangle \in \text{ASSIGNED\_TO}$
$\wedge\ \langle p,n,c\rangle \in \text{PROJECT} \wedge c = 109)\}$

4.14 (d) Derive the list of employees who are assigned to all projects where employee 109 is the chief architect.

$\{\langle e\rangle \mid \forall p,n,c(\langle p,n,c\rangle \notin \text{PROJECT} \vee c \neq 109$
$\vee \exists\ p1(\langle p1,e\rangle \in \text{ASSIGNED\_TO} \wedge p = p1\ )))\}$

4.14 (e) Get all project names to which employee 107 is not assigned.

$\{\langle n\rangle \mid \exists\ p,c\ (\langle p,n,c\rangle \in \text{PROJECT}$
$\wedge\ \neg\exists\ p1,e(\langle p1,e\rangle \in \text{ASSIGNED\_TO} \wedge p1 = p \wedge e = 107\ )\}$

4.14 (f) Get complete details of employees who are assigned to projects not assigned to employee 107.

$\{\ \langle e,m\rangle \mid \exists\ p,n,c,p1,e1(\langle e,m\rangle \in \text{EMPLOYEE}$
$\wedge\ \langle p,n,c\rangle \in \text{PROJECT}$
$\wedge\ (\langle p1,e1\rangle \in \text{ASSIGNED\_TO} \wedge e = e1 \wedge p1 = p$
$\wedge\ (\ \neg\exists p2,e2(\langle p2,e2\rangle \in \text{ASSIGNED\_TO}$
$\wedge\ p2 = p \wedge e2 = 107\ ))\}$

5. Relational Database Manipulation

Objectives: This chapter introduces the student to the commercial data manipulation languages. We look at the main features of SQL, QUEL, and QBE and illustrate their usage. It is normal to cover details of SQL, the most common of these languages. In addition, some versions of QBE are also implemented in many commercial DBMSs and therefore the student should be familiarized with its concepts. The concept of using SQL and QUEL embedded in HLL is presented too. A comparison of SQL and QUEL with their shortcomings is included.

The following features of SQL are discussed:

Data definition facilities: create table, alter table, create index, drop table, drop index statements
Data manipulation facilities: select, delete, insert, and update statements
Method of specifying predicates and joins in SQL
Use of arithmetic and aggregate operators
Method of specifying joins in SQL
Nested queries and manipulating sets in SQL
Specifying quantifiers in SQL
Creating views in SQL

The following features of QUEL are discussed:

Data definition facilities: create, index, modify, and destroy statements
Data manipulation facilities: retrieve, range, delete, append, and replace statements
Method of specifying predicates and joins
Aggregation in QUEL
Use of temporary relations in QUEL to implement the SQL nested query feature
Creating views in QUEL
The basic data retrieval, aggregation and update features of QBE are discussed:

Solution to selected exercises

5.1. SQL

(a) List all students taking courses with Smith or Jones.

```
select S#,Sname
from STUDENT, ENROLL, TEACH
where STUDENT.S# = ENROLL.S# and
        ENROLL.Section = TEACH.Section and
        (TEACH.Prof = 'Smith' or TEACH.Prof = 'Jones')
```

(b) List all students taking at least one course that their advisor teaches.

```
select ADVISE.S#
from ENROLL, TEACH, ADVISE
where ENROLL.Section = TEACH.Section and
        TEACH.Prof = ADVISE.Prof and
        ENROLL.S# = ADVISE.S# and
        ENROLL.C# = TEACH.C#
```

(c) List those professors who teach more than one section of the same course.

```
select t1.Prof
from TEACH t1, TEACH t2
where t1.Prof = t2.Prof and
        t1.C# = t2.C# and
        t1.Section ¤ t2.Section
```

(d) List the courses that student "John Doe" can enrol in, i.e., has passed the necessary prerequisite courses but not the course itself.

As before, this query is resolved by breaking it down into a set of simpler queries:

(i) Find the courses John Doe cannot do,
(ii) Find the courses John Doe can do,
(iii) Find courses John Doe can enrol-in.

(i) Let us first create a temporary relation TEMP1(*C#*) and store the courses that John Doe has passed in it as follows:

```
insert into TEMP1
select C#
from STUDENT, GRADES
```

where *Sname* = 'John Doe' **and**
Grade ≠ 'F' **and** STUDENT.*S#* = GRADES.*S#*

Now let us find the Cartesian product of the courses offered and the courses passed, to find those courses for which he has the necessary prerequisites. Save the result into another relation TEMP2(*C#*, *Pre_C#*).

```
insert into TEMP2
    select TEACH.C#, Pre_C# = TEMP1.C#
    from TEACH, TEMP1
```

Now let us find the set of courses that he cannot do and store it into the temporary relation TEMP3(*C#*):

```
insert into TEMP3
    select C#
    from PRE_REQ
    where not exists
        (select *
        from TEMP2
        where PRE_REQ.C# = TEMP2.C# and
        PRE_REQ.Pre_C# = TEMP2.Pre_C#)
```

(ii) Now let us find the courses that John Doe can do and store these in a temporary relation TEMP4(*C#*) as follows:

```
insert into TEMP4
    (select C#
    from TEACH)   minus
    (select C#
    from TEMP3)
```

(iii) Now we can find the courses that he can enrol-in as:

```
(select C#
from TEMP4)   minus
    (select C#
    from TEMP1)
```

QUEL

(a) List all students taking courses with Smith or Jones.

```
range of s is STUDENT
range of e is ENROLL
range of t is TEACH
retrieve (s.S#, s.Sname)
where s.S# = e.S# and
```

e.*C#* = t.*C#* **and**
e.*Section* = t.*Section* **and**
(t.*Prof* = 'Smith' **or**
t.*Prof* = 'Jones')

(b) List all students taking at least one course that their advisor teaches.

range of a **is** ADVISE
range of e **is** ENROLL
range of t **is** TEACH
retrieve (a.*S#*)
where e.*C#* = t.*C#* **and**
e.*Section* = t.*Section* **and**
t.*Prof* = a.*Prof* **and**
e.*S#* = a.*S#*

(c) List those professors who teach more than one section of the same course.

range of t1 **is** TEACH
range of t2 **is** TEACH
retrieve (t1.*Prof*)
where t1.*Prof* = t2.*Prof* **and**
t1.*C#* = t2.*C#* **and**
t1.*Section* ¤ t2.*Section*

5.2

CONDUCTS (*Conductor*, *Composition*)
REQUIRES (*Composition*, *Instrument*)
PLAYS (*Player*, *Instrument*)
LIKES (*Player*, *Composition*)

SQL

(a) List the players and their instruments that can be part of the orchestra when Letitia Melody conducts.

select *Player*, *Instrument*
from CONDUCTS REQUIRES PLAYS
where CONDUCTS.*Composition* = REQUIRES.*Composition* **and**
REQUIRES.*Instrument* = PLAYS.*Instrument* **and**
Conductor = 'Letitia Melody'

(b) From the above list of players, identify those who like the composition they are likely to play.

select LIKES.*Player*
from CONDUCTS, REQUIRES, PLAYS, LIKES
where CONDUCTS.*Composition* = REQUIRES.*Composition* **and**

CONDUCTS.*Composition* = LIKES.*Composition* **and**
REQUIRES.*Instrument* = PLAYS.*Instrument* **and**
PLAYS.*Player* = LIKES.*Player*

QUEL

(a) List the players and their instruments that can be part of the orchestra when Letitia Melody conducts.

range of c **is** CONDUCTS
range of r **is** REQUIRES
range of p **is** PLAYS
retrieve (p.*Player*,p. *Instrument*)
where c.*Composition* = r.*Composition* **and**
r.*Instrument* = p.*Instrument* **and**
c.*Conductor* = 'Letitia Melody'

(b) From the above list of players, identify those who would like the composition they are likely to play.

range of c **is** CONDUCTS
range of r **is** REQUIRES
range of p **is** PLAYS
range of l **is** LIKES
retrieve (l.*Player*)
where c.*Composition* = r.*Composition* **and**
r.*Instrument* = p.*Instrument* **and**
c.*Composition* = r.*Composition* **and**
c.*Composition* = l.*Composition* **and**
c.*Conductor* = 'Letitia Melody'

QBE

(a) List the players and their instruments that can be part of the orchestra when Letitia Melody conducts.

CONDUCTS	*Conductor*	*Composition*
	Letitia Melody	<u>C0</u>

REQUIRES	*Composition*	*Instrument*
	<u>C0</u>	<u>IN</u>

PLAYS	*Player*	*Instrument*
	P.PL	P.IN

(b) From the above list of players, identify those who would like the composition they are likely to play.

CONDUCTS	*Conductor*	*Composition*
	Letitia Melody	CO

REQUIRES	*Composition*	*Instrument*
	CO	IN

PLAYS	*Player*	*Instrument*
	P.PL	P.IN

LIKES	*Player*	*Composition*
	P.PL	CO

5.3

Acquire details of the projects for each employee by name.

select *Emp#*, *EmpName*, *Project_Name*
from ASSIGNED_TO, EMPLOYEE, PROJECT
where ASSIGNED_TO.*Project#* = PROJECT.*Project#* **and**
ASSIGNED_TO.*Emp#* = EMPLOYEE.*Emp#*

Compile the names of project where employee 107 is assigned.

select *Project_Name*
from ASSIGNED_TO, PROJECT

where ASSIGNED_TO.*Project#* = PROJECT.*Project#* **and**
EMPLOYEE.*Emp#* = 107

Access all employees assigned to projects whose chief architect is employee 109.

```
select a.Emp#
from  ASSIGNED_TO a, PROJECT p
where a.Project# = p.Project# and
      p.Cheif_Architect = 109
```

Derive the list of employees who are assigned to all projects where employee 109 is the chief architect.

```
select a1.Emp#
from  ASSIGNED_TO a1
where (select a2.Project#
      from  ASSIGNED_TO a2
      where a1.Emp# = a2.Emp#)
      contains
            (select p.Project#
            from  PROJECT p
            where p.Cheif_Architect = 109)
```

or

```
select a1.Emp#
from  ASSIGNED_TO a1
where not exists
      (select *
      from  PROJECT p
      where p.Cheif_Architect = 109 and
      not exists
      (select *
      from  ASSIGNED_TO a2
      where a1.Emp# = a2.Emp# and
      a2.Project#  = p.Project# ))
```

Get all project names to which employee 107 is not assigned.

```
select Project_Name
from  PROJECT
where Project# not in
      (select Project#
      from  ASSIGNED_TO
      where Emp# = 107)
```

or

```
select Project_Name
from  PROJECT p
where not exists
          (select *
          from  ASSIGNED_TO a
            where p.Project# = a.Project# and
            Emp# = 107)
```

or

```
(select p.Project_Name
from  PROJECT p
where Project# in
          (select distinct p1.Project#
          from  PROJECT p1)
          minus
          (select distinct a.Project#
          from  ASSIGNED_TO a
          where a.Emp# = 107))
```

Get complete details of employees who are assigned to projects not assigned to employee 107.

```
select e.Emp# e.EmpName
from  EMPLOYEE e, PROJECT p
where e.Project# = p.Project#
          and not exists
          (select *
          from  ASSIGNED_TO a
          where a.Project# = p.Project# and
          a.Emp# = 107)
```

5.5.

SQL

```
select a,b
from REL1
where b = 'B1' or b = 'B2'
```

QUEL

```
range of r is REL1
retrieve (r.all)
where r.b = 'B1' or r.b = 'B2'
```

QBE

REL1	*A*	*B*
P.	<u>X</u>	B1
	<u>Y</u>	B2

5.6.

SQL

(a) List all modules that use the HEAPSORT and BINARY_SEARCH modules.

select c1.*Module*
from CONSISTS_OF c1, CONSISTS_OF c2
where c1.*Module* = c2.*Module* **and**
c1.*Sub_Module* = 'HEAPSORT' **and**
c2.*Sub_Module* = 'BINARY_SEARCH'

(b) List employees that were involved in the development of all modules that use the HEAPSORT and BINARY_SEARCH modules.

select distinct *Employee*
from DEVELOPED_BY, CONSISTS_OF c1, CONSISTS_OF c2
where c1.*Module* = c2.*Module* **and**
c1.*Sub_Module* = 'HEAPSORT' **and**
c2.*Sub_Module* = 'BINARY_SEARCH' **and**
c1.*Module* = DEVELOPED_BY.*Module*

QUEL

(a) List all modules that use the HEAPSORT and BINARY_SEARCH modules.

range of c1 **is** CONSISTS_OF
range of c2 **is** CONSISTS_OF
retrieve (c1.*Module*)
where c1.*Module* = c2.*Module* **and**
c1.*Sub_Module* = 'HEAPSORT' **and**
c2.*Sub_Module* = 'BINARY_SEARCH'

(b) List employees that were involved in the development of all modules that use the HEAPSORT and BINARY_SEARCH modules.

range of c1 **is** CONSISTS_OF
range of c2 **is** CONSISTS_OF
range of d **is** DEVELOPED_BY
retrieve (d.*Employee*)

```
where  c1.Module = c2.Module and
       c1.Sub_Module = 'HEAPSORT' and
       c2.Sub_Module = 'BINARY_SEARCH' and
       c1.Module = d.Module
```

The above query does not list employees who are involved indirectly with the development of HEAPSORT or BINARY_SEARCH. One level of indirection can be obtained as shown below and a modification can be used to get two level of indirection. A multilevel indirection is not expressible in relational algebra calculus and hence in SQL or QUEL.

```
select distinct Employee
from DEVELOPED_BY, CONSISTS_OF c1, CONSISTS_OF c2
where  c1.Sub_Module = c2.Module and
       (c2.Sub_Module = 'HEAPSORT' or
       c2.Sub_Module = 'BINARY_SEARCH') and
       c1.Module = DEVELOPED_BY.Module
```

```
range of c1 is CONSISTS_OF
range of c2 is CONSISTS_OF
range of d is DEVELOPED_BY
retrieve (d.Employee)
where  c1.Sub_Module = c2.Module and
       (c2.Sub_Module = 'HEAPSORT'  or
       c2.Sub_Module = 'BINARY_SEARCH') and
       c1.Module = d.Module
```

5.8.

SQL

```
update EMPLOYEE
set Pay_Rate = 1.05 * Pay_Rate
where Empl_No in
       (select Empl_No
       from DUTY_ALLOCATION
       where Posting_No = 7 and
              Shift = 3)
```

QUEL

```
range of e is EMPLOYEE
range of d is DUTY_ALLOCATION
replace e (Pay_Rate = 1.05 * Pay_Rate)
       where e.Empl_No = d.Empl_No and
       d.Posting_No = 7 and
```

d.*Shift* = 3

5.10.

(i) Get Emp# of employees working on project numbered COMP353.

select *Emp#*
from ASSIGNED_TO
where *Project#* = 'COMP353'

(ii) Get details of employees(name and number) working on project COMP353.

select EMPLOYEE.*Emp#*, *EmpName*
from ASSIGNED_TO, EMPLOYEE
where EMPLOYEE.*Emp* = ASSIGNED_TO.*Emp* **and**
Project# = 'COMP353'

(iii) Get details of employees working on all Database projects"

The following gives employees working on at-least one Database project

select EMPLOYEE.*Emp#*, *EmpName*
from ASSIGNED_TO, EMPLOYEE, PROJECT
where *Project_Name* = 'Database' **and**
PROJECT.*Project#* = ASSIGNED_TO.*Project#* **and**
EMPLOYEE.*Emp* = ASSIGNED_TO.*Emp*

To get details for employees working on all Database projects we use the following query.

select e.*Emp#*, e.*EmpName*
from EMPLOYEE e
where e.*Emp#* **in**
(**select** a1.*Emp#*
from ASSIGNED_TO a1
where (**select distinct** a2.*Project#*
from ASSIGNED_TO a2
where a1.*Emp#* = a2.*Emp#*)
contains
(**select** p.*Project#*
from PROJECT p
where p.*Project_Name* = 'Database')

or

```
select e.Emp#, e.EmpName
from EMPLOYEE e
where e.Emp# in
        (select a1.Emp#
        from ASSIGNED_TO a1
        where not exists
                (select p.Project#
                from PROJECT p
                where p.Project_Name = 'Database' and
                 not exists
                (select *
                from ASSIGNED_TO a2
                where  a2.Project# = p.Project# and
                a1.Emp# = a2.Emp#)))
```

(iv) Get details of employees working on both COMP353 and COMP354.

```
select Emp#, EmpName
from EMPLOYEE
where Emp# in
        (select a1.Emp#
        from ASSIGNED_TO a1
        where (select distinct a2.Project#
                from ASSIGNED_TO a2
                where a1.Emp# = a2.Emp#)
                contains
                        (select distinct a3.Project#
                        from ASSIGNED_TO a3
                        where a3.Project# = 'COMP353' or
                        a3.Project# = 'COMP354'))
```

(v) Get employee number of employees who work on at least all those projects that employee 107 works on.

```
select a1.Emp#
from ASSIGNED_TO a1
where (select distinct a2.Project#
        from ASSIGNED_TO a2
        where  a1.Emp# = a2.Emp#)
        contains
                (select distinct a3.Project#
                from ASSIGNED_TO a3
                where a3.Emp# = 107)
```

or

```
select a1.Emp#
from ASSIGNED_TO a1
where not exists
          (select *
          from ASSIGNED_TO a2
          where  a2.Emp# = 107 and not exists
                    (select  *
                    from ASSIGNED_TO a3
                    where a3.Emp# = a1.Emp# and
                    a3.Project# = a1.Project# ))
```

(vi) Get employee number of employees who do not work on project COMP453.

```
select distinct Emp#
from ASSIGNED_TO a1
minus
(select distinct Emp#
from ASSIGNED_TO
where  Project# = a1.'COMP453')
```

or

```
select a1.Emp#
from ASSIGNED_TO a1
where not exists
          (select *
          from ASSIGNED_TO a2
          where  a2.Project# = 'COMP453'and
          a2.Emp# = a1.Emp#)
```

(vii) Get employee number of employees who work on all projects.

```
select a1.Emp#
from ASSIGNED_TO a1
where (select distinct a2.Project#
          from ASSIGNED_TO a2
          where  a1.Emp# = a2.Emp#)
          contains
                    (select p.Project#
                    from PROJECT p)
```

or

```
select a1.Emp#
from ASSIGNED_TO a1
where not exists
          (select *
          from PROJECT p
```

where not exists
(**select** *
from ASSIGNED_TO a2
where a2.*Project#* = p.*Project#* **and**
a1.*Emp#* = a2.*Emp#*))

(viii) Get employee number of employees who work on at least one project that employee 107 works on.

select a1.*Emp#*
from ASSIGNED_TO a1
where *Emp#* ¤ 107 **and**
Project# in
(**select distinct** a2.*Project#*
from ASSIGNED_TO a2
where a2.*Emp#* = 107)

5.11.

(i) Get Emp# of employees working on project number COMP353.

range of a **is** ASSIGNED_TO
retrieve (a.*Emp#*)
where a.*Project#* = 'COMP353'

(ii) Get details of employees(name and number) working on project COMP353.

range of e **is** EMPLOYEE
range of a **is** ASSIGNED_TO
retrieve (e.*Emp#*, e.*EmpName*)
where e.*Emp* = a.*Emp* **and**
a.*Project#* = 'COMP353'

(iii) Get details of employees working on all Database projects.

The following query finds employees who are working on any one Database project:

range of a **is** ASSIGNED_TO
range of e **is** EMPLOYEE
range of p **is** PROJECT
retrieve (e.*Emp#*, e.*EmpName*)
where p.*Project_Name* = 'Database' **and**
p.*Project#* = a.*Project#* **and**
e.*Emp#* = a.*Emp#*

To find employees who are working on all Database projects, we use the following:

```
range of a1 is ASSIGNED_TO
range of a2 is ASSIGNED_TO
range of p is PROJECT
retrieve (a1.Emp#)
where any (p.Project# by a1.Emp#
            where p.Project_Name = 'Database' and
            any (a2.Project# by a1.Emp#, p.Project#
                where a1.Emp# = a2.Emp# and
                a2.Project# = p.Project#) = 0) = 0
```

(iv) Get details of employees working on both COMP353 and COMP354.

```
range of a1 is ASSIGNED_TO
range of a2 is ASSIGNED_TO
range of a3 is ASSIGNED_TO
range of e is EMPLOYEE
retrieve (e.Emp#, e.EmpName)
where e.Emp# = a1.Emp# and
    any (a2.Project# by a1.Emp#
    where (a2.Project# = COMP353 or
        a2.Project# = COMP354)  and
    any (a3.Project# by a1.Emp#, a2.Project#
        where a1.Emp# = a3.Emp# and
        a2.Project# = a3.Project#) = 0 ) = 0 )
```

(v) Get employee number of employees who work on at least all those projects that employee 107 works on.

```
range of a1 is ASSIGNED_TO
range of a2 is ASSIGNED_TO
range of a3 is ASSIGNED_TO
retrieve (a1.Emp#)
where a1.Emp# ≠ 107 and
    any (a2.Project# by a1.Emp#
    where a2.Emp# = 107 and
        any (a3.Project# by a1.Emp#, a2.Project#
            where a3.Project# and
            a1.Emp# = a3.Emp#) = 0 ) = 0
```

(vi) Get employee number of employees who do not work on project COMP453.

```
range of a1 is ASSIGNED_TO
range of a2 is ASSIGNED_TO
retrieve (a1.Emp#)
where any (a2.Emp# by a1.Emp#
    where a1.Emp# = a2.Emp# and
        a2.Project# = COMP453) = 0
```

(vii) Get employee number of employees who work on all projects.

range of a1 **is** ASSIGNED_TO
range of a2 **is** ASSIGNED_TO
range of p **is** PROJECT
retrieve (a1.*Emp#*)
where any (p.*Project#* **by** a1.*Emp#*
    where any (a2.*Project#* **by** a1.*Emp#*, p.*Project#*
        where a1.*Emp#* = a2.*Emp#* **and**
    a2.*Project#* = p.*Project#*) = 0) = 0

(viii) Get employee number of employees who work on at least one project that employee 107 works on.

range of a1 **is** ASSIGNED_TO
range of a2 **is** ASSIGNED_TO
retrieve (a1.*Emp#*)
where a1.*Emp#* ¤ 107 **and**
    a2.*Project#* = a1.*Project#* **and**
    a2.*Emp#* = 107

5.14. Using SQL, get the *Empl_No*, *Skill*, and average chef's pay rate for the EMPLOYEE relation shown in Figure 5.6.

Consider the temporary relation TEMP1(*Empl_No*, *Skill*) as follows:

insert into TEMP1
    select *Empl_No*, *Skill*
    from EMPLOYEE

Consider the temporary relation TEMP2(*Pay_Rate*) as follows:

insert into TEMP2
    select avg(*Pay_Rate*)
    from EMPLOYEE
    where *Skill* = 'chef'

Now the required response can be derived as:

select *

from TEMP1, TEMP2

5.17

(i) Give the names of the players who played as forwards in 1987 with the franchise "Blades".

(a) SQL

select f.*Name*
from FORWARD f
where f.*Franchise_Name* = 'Blades' **and**
    f.*Year* = 1987

(b) QUEL

range of f **is** FORWARD
retrieve (f.*Name*)
where f.*Franchise_Name* = 'Blades' **and**
    f.*Year* = 1987

(ii) Find the names of all the goalies who played with the forward Ozzy Xavier over the span of his hockey career.

(a) SQL

select g.*Name*
from FORWARD f, GOAL g
where f.*Name* = 'Ozzy Xavier' **and**
    f.*Year* = g.*Year* **and**
        f.*Franchise_Name* = g.*Franchise_Name*

(b) QUEL

range of f **is** FORWARD
range of g **is** GOAL
retrieve (g.*Name*)
where f.*Name* = 'Ozzy Xavier' **and**
        f.*Year* = g.*Year* **and**
        f.*Franchise_Name* = g.*Franchise_Name*

(iii) List forwards and the franchises for those forwards who had at least 50 goals in both of the years 1985 and 1986. A player must have at least 50 goals in both the years, however may be with two different franchises.

(a) SQL

select f.*Name*, f.*Franchise_Name*, f1.*Franchise_Name*
from FORWARD f, FORWARD f1
where f.*Name* = f1.*Name* **and**

```
f.Year = 1985 and
f1.Year = 1986 and
f.Goals >= 50 and
f1.Goals >= 50
```

(b) QUEL

```
range of f is FORWARD
range of f1 is FORWARD
retrieve (f.Name, f.Franchise_Name, f1.Franchise_Name)
where f.Name = f1.Name and
    f.Year = 1985 and
        f1.Year = 1986 and
        f.Goals >= 50 and
        f1.Goals >= 50
```

(iv) Give the complete details of players who played for the same franchises that Ozzy Xavier did over his career. However, they may not necessarily have played in the same year or as forwards.

(a) SQL

```
select *
from PLAYER p
where p.Name in
        ((select f1.Name
        from FORWARD f1
        where   f1.Name ≠ 'Ozzy Xavier' and
        not exists
                (select *
                from FORWARD f2
                where f2.Name = 'Ozzy Xavier' and not exists
                        (select *
                        from FORWARD f3
                        where f3.Name = f1.Name and
                        f3.Franchise_Name = f1.Franchise_Name )))
union
(select g.Name
from GOAL g
where not exists
        (select *
        from FORWARD f2
        where f2.Name = 'Ozzy Xavier' and not exists
                (select  *
                from GOAL g1
                where g.Name = g1.Name and
                  g.Franchise_Name = g1.Franchise_Name ))))
```

(b) QUEL

range of f1 **is** FORWARD
range of f2 **is** FORWARD
range of f3 **is** FORWARD
retrieve into TEMP(f1.*Name*)
where f1.*Name* ≠ 'Ozzy Xavier' **and**
    any (f2.*Franchise_Name* **by** f1.*Name*
    where f2.*Name* = 'Ozzy Xavier' **and**
    any (f3.*Franchise_Name* **by** f1.*Name*, f2.*Franchise_Name*
        where f2.*Franchise_Name* = f3.*Franchise_Name* **and**
            f1.*Name* = f3.*Name*) = 0) = 0
range of g1 **is** GOAL
range of f2 **is** FORWARD
range of g3 **is** GOAL
retrieve into TEMP(g1.*Name*)
where any (f2.*Franchise_Name* **by** g1.*Name*
where f2.*Name* = 'Ozzy Xavier' **and**
any (g3.*Franchise_Name* **by** g1.*Name*, f2.*Franchise_Name*
        where f2.*Franchise_Name* = g3.*Franchise_Name* **and**
            g1.*Name* = g3.*Name*) = 0) = 0
range of t **is** TEMP
range of p **is** PLAYER
retrieve (p.**all**)
where p.*Name* = t.*Name*

(v) Compile the list of goalies who played, during their career, for franchises in St. Louis, Edmonton and Paris. A goalie should be listed if and only if he had played in all three cities.

(a) SQL

select g1.*Name*
from GOAL g1, GOAL g2, GOAL g3, TEAM t1, TEAM t2 ,TEAM t3
where g1.*Name* = g2.*Name* **and** g1.*Name* = g3.*Name* **and**
g1.*Franchise_Name* = t1.*Franchise_Name* **and**
g2.*Franchise_Name* = t2.*Franchise_Name* **and**
g3.*Franchise_Name* = t3.*Franchise_Name* **and**
t1.*City* = 'St. Louis' **and**
t2.*City* = 'Edmonton' **and**
t3.*City* = 'Paris' **and**
g1.*Year* = t1.*Year* **and**
g2.*Year* = t2.*Year* **and**
g3.*Year* = t3.*Year*

(b) QUEL

range of g1 **is** GOAL
range of g2 **is** GOAL
range of g3 **is** GOAL
range of t1 **is** TEAM
range of t2 **is** TEAM
range of t3 **is** TEAM
retrieve (g1.*Name*)
where g1.*Name* = g2.*Name* **and** g1.*Name* = g3.*Name* **and**
g1.*Franchise_Name* = t1.*Franchise_Name* **and**
g2.*Franchise_Name* = t2.*Franchise_Name* **and**
g3.*Franchise_Name* = t3.*Franchise_Name* **and**
t1.*City* = 'St. Louis' **and**
t2.*City* = 'Edmonton' **and**
t3.*City* = 'Paris' **and**
g1.*Year* = t1.*Year* **and**
g2.*Year* = t2.*Year* **and**
g3.*Year* = t3.*Year*

6. Relational Database Design

Objectives: This chapter introduces the student to the following concepts:
Relation scheme
Anomalies in database as a result of bad design and normal forms
Concept of decomposition of a relation scheme
Concept of universal relation
Functional dependency and logical implication
Inference axioms
Concept of closures: of a set of FDs, of a set of attributes under a set of FDs
Membership of a FD in the closure of a set of FDs
Non-redundant and minimum covers
Concept of Full Functional, Partial and Transitive dependencies
Aim of relational database design: content and dependency preservation
Concept of un-normalized relation and the first, second, third normal forms
Concept of lossless and lossy decomposition
Concept of dependency-preserving decomposition
Algorithm to verify if a decomposition is: lossless, dependency-preserving
Algorithm for deriving a lossless and dependency-preserving third normal form relation database
Concept of the Boyce Codd normal form
Algorithm for decomposing into a lossless-join Boyce Codd normal form

Solution to Selected Exercises:

6.1.

The FDs in the set F are already left-reduced. In the set of FDs **F**=$\{A \rightarrow B, BC \rightarrow D, D \rightarrow BC, DE \rightarrow \emptyset\}$, the $DE \rightarrow \emptyset$ is redundant since its RHS is $\emptyset$. However, if the FD is included to indicate that there is some form of non-functional dependency, we may leave it in. Another reason to leave-in this FD is to include the attribute E which does not appear in any other FD in the set F. Writing the remaining FDs in the simple form we get:

F'=$\{A \rightarrow B, BC \rightarrow D, D \rightarrow B, D \rightarrow C\}$.

None of these FDs are redundant hence this set forms a canonical cover.

$\mathbf{F}_c=\{A \rightarrow B, BC \rightarrow D, D \rightarrow B, D \rightarrow C\}$.

Using $\mathbf{F}_c$ we get the following decompositions

R1 = (*E*)
R2 = (*AB*)
R3 = (*BCD*)
R4 = (*BD*)
R5 = (*CD*)

Since ADE is a key of R, we modify R1 to (*ADE*) and in this way keep the attributes *DE* together. R4 and R5 may be combined into a single relation scheme (*BCD*) which already exists as R3!

6.2.

Given**:** **R**$\{ABCDE\}$ **F**=$\{AB \rightarrow CD, ABC \rightarrow E, C \rightarrow A\}$
$ABC^+ = ABCDE$

Candidate keys: *AB*, *BC*

The relation is in the 1NF since there is a partial dependency in **F**.

6.3.

Given **R**$\{ABCDEF\}$ **F**=$\{ABC \rightarrow DE, AB \rightarrow D, DE \rightarrow ABCF, E \rightarrow C\}$

R is in 1NF. The key of this relation are: *ABC* and *DE*. However, the FDs $AB \rightarrow D$, $E \rightarrow C$ are partial dependencies and hence R is not in 3NF.

A lossless and dependency preserving decomposition of **R** is:

R1{*ABCE*}, **R2**{*ABD*}, **R3**{*ADE*}, **R4**{*BDE*}, **R5**{*DEF*}, **R6**{*CE*}.

6.4.

R{*T*, *C*, *Y*, *G*, *D*, *V*}   FD's{$T \rightarrow C$, $TY \rightarrow G$, $TY \rightarrow D$, $CG \rightarrow V$}

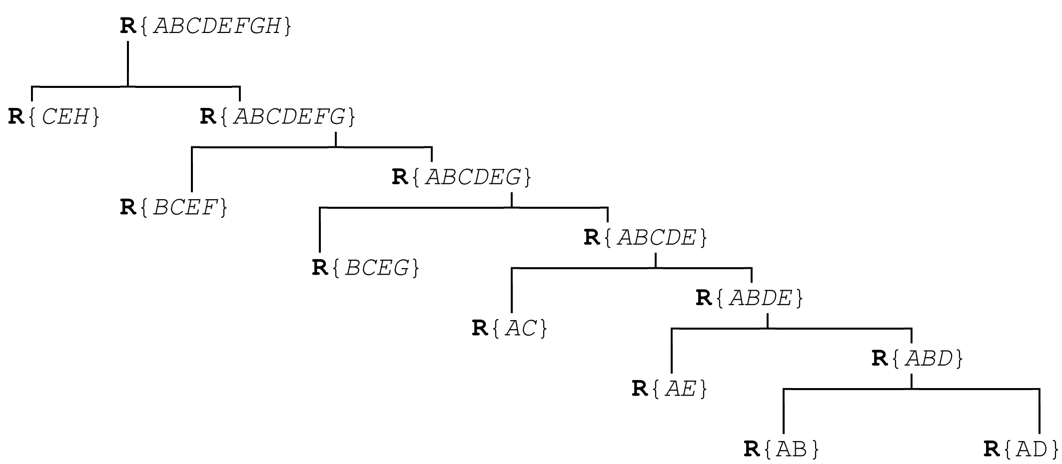
The decomposition of **R** into **R1**{*TCD*} and **R2**{*TGDVY*} is lossless but not dependency preserving. It is lossless since, the common attributes TD forms a superkey of the first relation. It is not dependency preserving since the FD $CG \rightarrow V$ is not preserved in the decomposition.

The decomposition of **R** into **R1**{*TC*}, **R2**{*TGDY*} and **R3**{*CGV*} is a lossless and dependency preserving 3NF decomposition. This decomposition is also in BCNF since each FDs in each relation involve only the superkeys of the decomposed relation.

6.7.

	Left-reduced	Right-reduced	Non-redundant covers	
$A \rightarrow BCD$	$A \rightarrow BCD$	$A \rightarrow D$	$A \rightarrow D$	$A \rightarrow BCD$
$CD \rightarrow E$	$D \rightarrow E$	$D \rightarrow E$	$E \rightarrow D$	$E \rightarrow CD$
$E \rightarrow CD$	$E \rightarrow CD$	$E \rightarrow D$	$D \rightarrow ABCEH$	$D \rightarrow AEH$
$D \rightarrow AH$	$D \rightarrow AH$	$D \rightarrow AH$		
$ABH \rightarrow BD$	$AH \rightarrow BD$	$AH \rightarrow$ Ý		
$DH \rightarrow BC$	$D \rightarrow BC$	$D \rightarrow BC$		

6.8.



This decomposition is not dependency preserving since among others the FD $BCD \rightarrow E$ is not preserved.

6.9.

Given FD set	Left-reduced	Right-reduced	Canonical cover
$I \rightarrow K$	$I \rightarrow K$	$I \rightarrow K$	$I \rightarrow BCDEFGJK$
$AI \rightarrow BFG$	$I \rightarrow BFG$	$I \rightarrow BFG$	$K \rightarrow AH$
$IC \rightarrow ADE$	$I \rightarrow ADE$	$I \rightarrow DE$	
$BIG \rightarrow CJ$	$I \rightarrow CJ$	$I \rightarrow CJ$	
$K \rightarrow AH$	$K \rightarrow AH$	$K \rightarrow AH$	

The decomposition of **R** into **R1**<{*BCDEFGIJK*}, {$I \rightarrow BCDEFGJK$ }>, and **R2**<{*AHK*}, {$K \rightarrow AH$}> is both lossless and dependency preserving. Furthermore, this decomposition is also in BCNF.

6.10.

Given set	Left-reduced	Right-reduced	Canonical cover
$A \rightarrow BCDE$	$A \rightarrow BCDE$	$A \rightarrow C$	$A \rightarrow C$
$B \rightarrow ACDE$	$B \rightarrow ACDE$	$B \rightarrow C$	$B \rightarrow C$
$C \rightarrow ABDE$	$C \rightarrow ABDE$	$C \rightarrow ABDE$	$C \rightarrow ABDE$

The decomposition of **R** into **R1**{*AC*}, **R2**{*BC*}, **R3**{*CDE*} is lossless. To preserve dependeny we may decomose **R** into **R1**{*AC*}, **R2**{*BC*}, **R3**{*ABCDE*}. However, this requires some duplication.

6.13.

$BCD^+ = ABCDEF$.

6.17.

Under the modified assumption TEACHES is not in 2NF, since *Room_Cap*, a non-prime attribute is not dependent on the key of the relation. Its decomposition into COURSE_DETAILS and ROOM_DETAILS is a 3NF decomposition which is both lossless and dependency preserving.

6.18.

The decomposition is lossy since the final version of the TABLE_LOSSY shown below, does not have any row with all α's.

	A	*B*	*C*	*D*	*E*
R1	α_A	α_B	α_C	α_D	β_{1E}
R2	β_{2A}	α_B	α_C	α_D	β_{2E}
R3	β_{3A}	β_{3B}	α_C	α_D	α_E

6.22.

With only two atomic attributes, we can say that the relation is in BCNF form and, therefore in 3NF form.

6.23.

Since A is a candidate key, we can deduce that the FD $A \rightarrow BCD$ is satisfied. This means that the relation is at least in the 2NF. However, it may have a transitive dependency such as $B \rightarrow C$ and hence may not be in any higher normal form.

7. Synthesis Approach and Higher Order Normal Form

Objectives: This is an optional chapter for a first course in database systems. The chapter introduces the student to the synthesis approach to 3NF relational database design. We then turn our attention to the higher order normal forms. The concept of multi-valued dependency and axioms which involve both functional and multi-valued dependencies are examined. The fourth normal form and a lossless decomposition algorithm for it is given. The concept of join dependency and a normal form for it is introduced. Finally, we introduce a scheme whereby all general constraints could be enforced via domain and key constraint, and the associated normal form, known as domain key normal form.

Solution to selected exercises

7.1.

R1 = {*AB*}, R2 = {*BCD*}, R3 = {*DE*}, R4 = {*ADE*}

7.3.

*[*ACE*, *BD*, *CE*], *[*ABC*, *BCD*, *CDE*], *[*AB*, *BC*, *CD*, *ADE*]

R1 = {*AB*C}, R2 = {*BCD*}, R3 = {*CDE*}

7.4.

(*A*) (*E*) (*F*)

8. The Network Model

Objectives: This chapter as well as the next have a slightly different style than the rest of the text. This has been done to allow these chapters to be studied either with very little help from the instructor or their coverage could be entrusted to a tutor or a T.A. It is expected that the instructor has covered the basic concept of these models in Chapter 2. The chapters at hand use the same database example.
The chapter introduces the student to the following concepts of the network data model:

The use of the DBTG set to express a one-to-many relationship

The restriction of the DBTG set construct

Implementation of the DBTG set

Expressing a many-to-many relationship in the network model

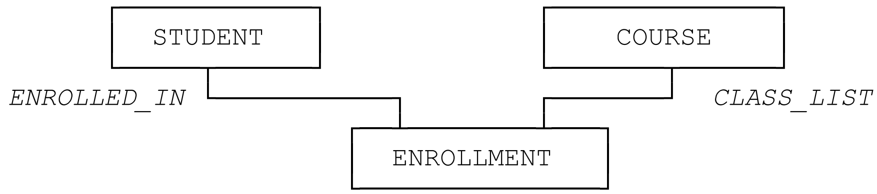
Data definition facility in the network model and different types of set memberships

Data manipulation facility

Concept of currency indicators, status registers, record templates and navigating through the network database.

Solution to selected Exercises

8.4



*Schema name is **SCHOOL***

type STUDENT = *record*
Student_No: *string*;
Name: *string*;
Address: *string*;
end

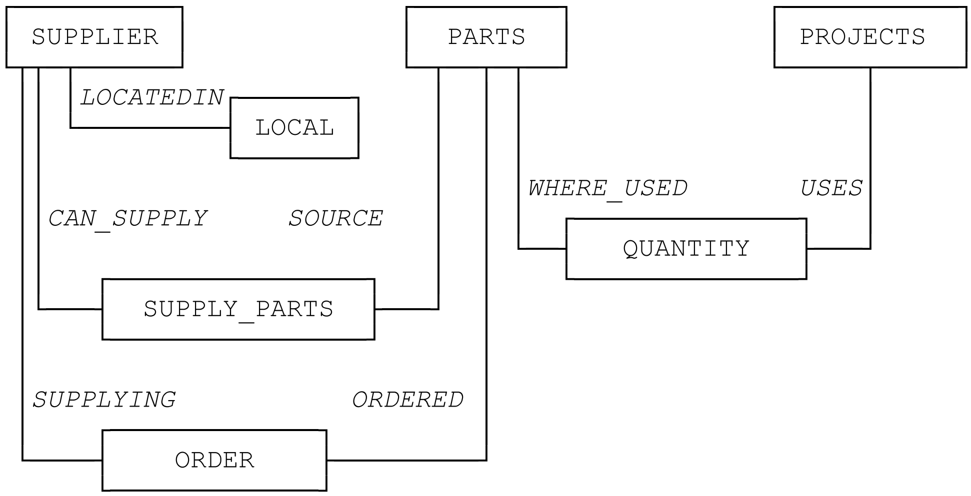
type COURSE = *record*
Course_No: *string*;
Course_Name: *string*;
end;

type ENROLLMENT = *record*
Course_No: *string*;
Student_No: *string*;
end;

set is *ENROLLED_IN*
owner is STUDENT
member is ENROLLMENT **optional manual**
end

set is *CLASS_LIST*
owner is COURSE
member is ENROLLMENT **manual optional**
end

8.7



*Schema name is **SUPPLIER_PARTS-PROJECTS***

type SUPPLIER = *record*
Supplier#: *string*;
Company-Name: *string*;
end

type LOCAL = *record*
City: *string*;
end

type PARTS = *record*
Part#: *string*;
Weight: *integer*;
end;

type PROJECTS = *record*
Project#: *string*;
end;

type QUANTITY = *record*
Project#: *string*;
Part#: *string*;
Quant: *integer*;
end;

type SUPPLY_PARTS = *record*
Supplier#: *string*;
Part#: *string*;
end;

```
type  ORDER = record
        Supplier#: string;
        Part#: string;
        Date_of_Delivery: string;
        end;

set is  LOCATEDIN
        owner is SUPPLIER
        member is LOCAL automatic fixed
        end

set is  WHERE_USED
        owner is PARTS
        member is QUANTITY automatic fixed
        end

set is  USES
        owner is PROJECTS
        member is QUANTITY automatic fixed
        end

set is  CAN_SUPPLY
        owner is SUPPLIER
        member is SUPPLY_PARTS automatic fixed
        end

set is  SOURCE
        owner is PARTS
        member is SUPPLY_PARTS automatic fixed
        end

set is  SUPPLYING
        owner is SUPPLIER
        member is ORDER automatic fixed
        end

set is  ORDERED
        owner is PARTS
        member is ORDER automatic fixed
        end
```

8.7 i.

```
SUPPLIER.Supplier# := supplier1;
find any SUPPLIER using SUPPLIER.Supplier#;
find first SUPPLY_PARTS within CAN_SUPPLY;
while DB_Status  = 0 do
   begin
```

```
     get SUPPLY_PARTS;
     display ( 'Supplier', supplier1 'supplies part# ',
               SUPPLY_PARTS.Part# )
     find next SUPPLY_PARTS within CAN_SUPPLY;
     end
```

8.7 ii.

```
SUPPLIER.Supplier# := supplier1;
find any SUPPLIER using SUPPLIER.Supplier#;
if DB_Status = 0 then get SUPPLIER
if DB_Status = 0 then find first LOCAL within LOCATEDIN;
while DB_Status = 0  do
   begin
    get LOCAL;
    display ( ' Supplier''s ', SUPPLIER.Supplier#,
             'city is', LOCAL.City);
    find next LOCAL within LOCATEDIN;
    end;
```

8.7 iii.

We assume that there is an array parts_list as given below where we will first store the list of all parts supplied by $supplier_1$.

```
parts_list = array [1..max_no_parts] of string;
n := 1;
SUPPLIER.Supplier# := supplier1;
find any SUPPLIER using SUPPLIER.Supplier#;
find first SUPPLY_PARTS within CAN_SUPPLY;
while DB_Status  = 0 do
 begin
  get SUPPLY_PARTS;
  parts_list[n] := SUPPLY_PARTS.Part#
  n := n + 1;
  find next SUPPLY_PARTS within CAN_SUPPLY;
  end
```

Now we use the set *SOURCE* to find at least another supplier who supplies each of these parts as follows:

```
for i := 1 to n do
 begin;
  PARTS.Part# := parts_list[i];
  find any PARTS using PARTS.Part#;
  find first SUPPLY_PARTS within SOURCE;
  found := false;
```

```
while DB_Status = 0  and not found do
 begin
  get SUPPLY_PARTS;
  if SUPPLY_PARTS.Supplier# <> supplier_1 then
    found := true;
    else   find next SUPPLY_PARTS within SOURCE;
  end
if found then
 display (' Another supplier for part ' PARTS.Part#,
              ' is ' ,  SUPPLY_PARTS.Supplier#)
  else display (' No other supplier supplies the part ',
             PARTS.Part#);
end;
```

8.7 iv.

We assume that there is an array parts_list where we will first store the list of all parts supplied by $supplier_1$ (as in the previous example). Now for each such part, we find the set of projects where it is used. The union of all these sets gives the projects where $supplier_1$ may supply. These projects are created in the array projects_list as shown below:

```
projects_list = array[1..max_no_of_projects] of string;
m := 0;
for i := 1 to n do
 begin
   PARTS.Part# = parts_list[i];
   find any PARTS using PARTS.Part#;
   find first QUANTITY within WHERE_USED;
   while DB_Status  = 0 do
     begin
      get QUANTITY;
      found := false;
      j := 1;
      while not found and j < m do
        if projects_list[j] = QUANTITY.Project# then
          found := true
          else  j := j+1;
      if not found then
        begin
          m := m+1;
          projects_list[m] := QUANTITY.Project#;
          end;
      find next QUANTITY within WHERE_USED
      end { while }
   end {for i }
```

8.7 v.

PARTS.*Part#* := $part_1$;
find any PARTS **using** PARTS.*Part#*;
find first SUPPLY_PARTS **within** *SOURCE*;
while **DB_Status** = 0 *do*
begin
get SUPPLY_PARTS;
display (' Supplier is ', SUPPLY_PARTS.*Supplier#*)
find next SUPPLY_PARTS **within** *CAN_SUPPLY*;
end

8.7 vi.

PARTS.*Part#* := $part_1$;
find any PARTS **using** PARTS.*Part#*;
find first QUANTITY **within** *WHERE_USED*;
while **DB_Status** = 0 *do*
begin
get QUANTITY;
display (' Project is ', QUANTITY.*Project#*)
find next QUANTITY **within** *WHERE_USED*;
end

8.8 i. True

8.8 ii. False

8.8 iii. False

8.8 iv. True

8.8 v. False

8.8 vi. True

8.8 vii. True

9. The Hierarchical Data Model

Objectives: As mentioned before, this and the previous chapter have a slightly different style than the rest of the text. This is to allow these chapters to be studied either with very little help from the instructor or their coverage be entrusted to a tutor or a T.A.

The chapter introduces the student to the following concepts of the hierarchical data model:

Concept of ordered tree

Representation of data and relationship using the ordered tree

Representation of a many-to-many relationship in the hierarchical model

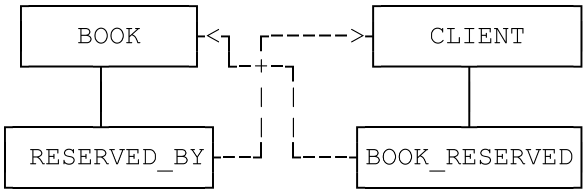
Data definition facilities

Data manipulation in the hierarchical model

Concept of currency indicators, status registers, record templates and navigating through the hierarchical database

Solution to selected exercises

9.3



The paired bi-directional logical relationship, with its associated symmetrical virtual records, is used in the hierarchical model to implement a many-to-many relationship. The many-to-many relationship between clients and the books they reserve may be implemented as shown above:

type BOOK = *record*
Author: *string*;
Title: *string*;
Call_No: *string*;
end

type CLIENT = *record*
Client_No: *integer*;
Name: *string*;
Address: *string*;
end

type RESERVED_BY = *record*
{*Client_No*: *integer*;
Name: *string*;
Address: *string*;}
(* *virtual of logical parent*
CLIENT *in CLIENT_BOOK_TREE*; *)
end

type BOOK_RESERVED = *record*
{*Author*: *string*;
Title: *string*;
Call_No: *string*;}
(* *virtual of logical parent*
BOOK *in BOOK_CLIENT_TREE*; *)
end

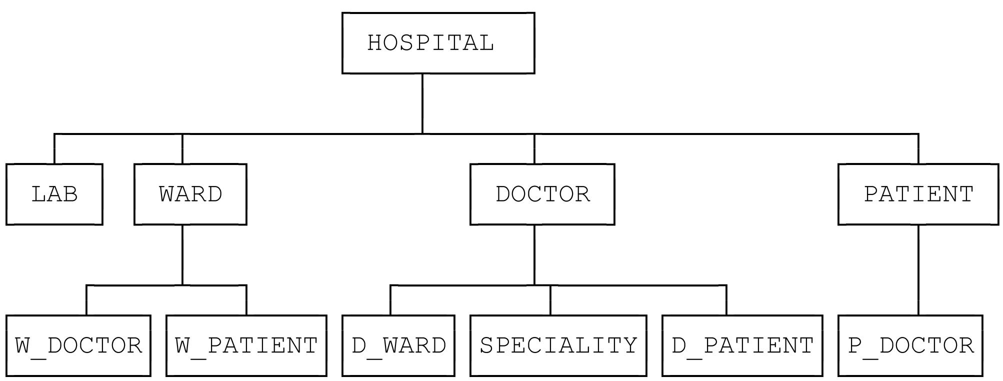
tree is CLIENT_BOOK_TREE
CLIENT *is parent*
BOOK_RESERVED *is child*
end

tree is BOOK_CLIENT_TREE
BOOK *is parent*
RESERVED_BY *is child*
end

9.6

Since the child records are linked directly to the parent record by hierarchical pointers, there is no need for foreign keys.

9.7



tree is HOSPITAL_TREE
HOSPITAL *is parent*
LAB *is child*
WARD *is child*
DOCTOR *is child*
PATIENT *is child*
end

tree is WARD_TREE
WARD *is parent*
W_DOCTOR *is child*
W_PATIENT *is child*
end

tree is DOCTOR_TREE

```
        DOCTOR  is parent
        D_WARD  is child
        SPECIALITY  is child
        D_PATIENT  is child
        end

tree is  PATIENT_TREE
        PATIENT is parent
        P_DOCTOR  is child
        end

type HOSPITAL  = record
        Hospital_Name:  string;
        Address: string;
        Phone_No: string;
        end

type LAB = record
        Lab_Name:  string;
        Room_No: integer;
        Phone_No: string;
        end

type WARD = record
        Ward_Name:  string;
        Capacity: integer;
        end

type DOCTOR = record
        D_Name:  string;
        Current_Status: string;
        end

type PATIENT = record
        P_Name:  string;
        Address: string;
        Phone: string;
        end

 type   W_DOCTOR = record
        {D_Name: string;}
           (* virtual of logical parent
              DOCTOR in DOCTOR_TREE *)
        end

 type   W_PATIENT = record
        {P_Name: string;}
```

```
              (* virtual of logical parent
                 PATIENT in PATIENT_TREE *)
           end

    type   D_WARD = record
           {Ward_Name: string;}
              (* virtual of logical parent
                 WARD in WARD_TREE *)
           end

    type   SPECIALITY = record
           Speciality_Name : string;
           end

    type   D_PATIENT = record
           {P_Name: string;}
              (* virtual of logical parent
                 PATIENT in PATIENT_TREE *)
           end

    type   P_DOCTOR = record
           {D_Name: string;}
              (* virtual of logical parent
                 DOCTOR in DOCTOR_TREE *)
           end
```

9.8 a)

```
get first HOSPITAL;
while DB-Status = 0 do
 begin
  get next within parent LAB where Lab_Name = 'haematology';
  if DB-Status = 0 then display (HOSPITAL.Hospital_Name);
  get next HOSPITAL;
  end
```

9.8 b)

```
get first HOSPITAL;
while DB-Status = 0 do
 begin
  get next within parent WARD where WARD.Capacity > 4;
  while DB-Status = 0 do
    begin
     display (HOSPITAL.Hospital_Name, WARD.Ward_Name );
          get next within parent WARD where WARD.Capacity > 4;
     end
```

```
   get next HOSPITAL
   end
```

9.8 c)

```
get first PATIENT where PATIENT.P_Name = '   given   ';
if DB-Status = 0
then get next within parent P_DOCTOR;
while DB-Status = 0  do
 begin
  display (PATIENT.P_Name, P_DOCTOR.D_Name);
   get next within parent P_DOCTOR;
   end;
```

9.8 d)

```
get first DOCTOR;
while DB-Status = 0 do
 begin
   get next within parent SPECIALITY where Speciality_Name =
                                                          'pediatrics';
   if DB-Status = 0
     then display (DOCTOR.D_Name);
   get next DOCTOR;
   end
```

9.8 e)

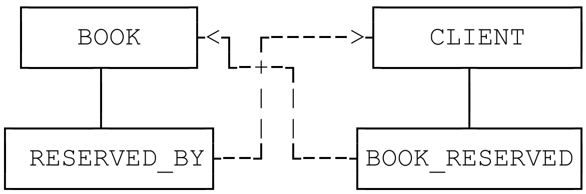
```
no_of_doctors := 0;
get first PATIENT where PATIENT.P_Name = '   given   ';
if DB-Status = 0
then get next within parent P_DOCTOR;
while DB-Status = 0 do
 begin
   no_of_doctors := no_of_doctors + 1;
  display (PATIENT.P_Name, P_DOCTOR.D_Name);
   get next within parent P_DOCTOR;
   end;
display (PATIENT.P_Name, 'Number of Doctors = ', no_of_doctors);
```

9.8 f)

```
DOCTOR.D_Name :=  '   given   ';
DOCTOR.Current_Status'   given   ';
insert (DOCTOR) where (HOSPITAL.Hospital_Name = '  given  ');
for i := 1 to no_of _speciality do
```

begin
get (speciality);
SPECIALITY.*Speciality_Name* := speciality;
insert (SPECIALITY) **where** (DOCTOR.*D_Name* = ' given ');
end

9.3



The paired bi-directional logical relationship, with its associated symmetrical virtual records, is used in the hierarchical model to implement a many-to-many relationship. The many-to-many relationship between clients and the books they reserve may be implemented as shown above:

type BOOK = *record*
Author: *string*;
Title: *string*;
Call_No: *string*;
end

type CLIENT = *record*
Client_No: *integer*;
Name: *string*;
Address: *string*;
end

type RESERVED_BY = *record*
{*Client_No*: *integer*;
Name: *string*;
Address: *string*;}
(* *virtual of logical parent*
CLIENT *in CLIENT_BOOK_TREE*; *)
end

type BOOK_RESERVED = *record*
{*Author*: *string*;
Title: *string*;
Call_No: *string*;}
(* *virtual of logical parent*
BOOK *in BOOK_CLIENT_TREE*; *)
end

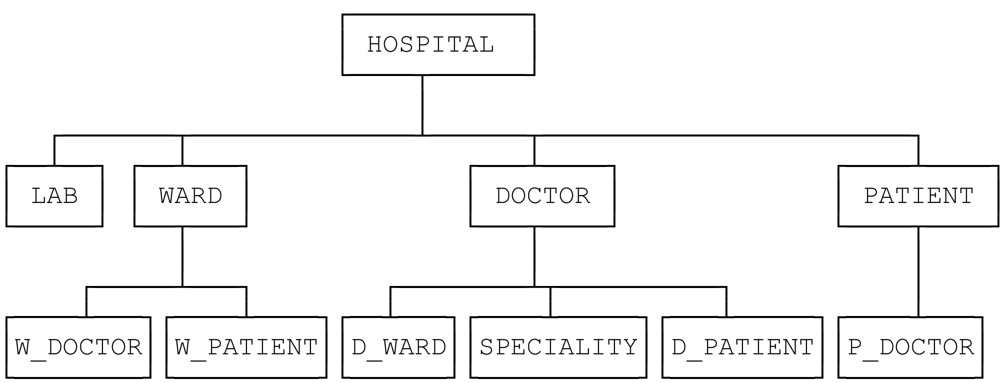
tree is CLIENT_BOOK_TREE
CLIENT *is parent*
BOOK_RESERVED *is child*
end

tree is BOOK_CLIENT_TREE
BOOK *is parent*
RESERVED_BY *is child*
end

9.6

Since the child records are linked directly to the parent record by hierarchical pointers, there is no need for foreign keys.

9.7



tree is HOSPITAL_TREE
HOSPITAL *is parent*
LAB *is child*
WARD *is child*
DOCTOR *is child*
PATIENT *is child*
end

tree is WARD_TREE
WARD *is parent*
W_DOCTOR *is child*
W_PATIENT *is child*
end

tree is DOCTOR_TREE
DOCTOR *is parent*
D_WARD *is child*

```
        SPECIALITY  is child
        D_PATIENT  is child
        end

tree is  PATIENT_TREE
        PATIENT is parent
        P_DOCTOR  is child
        end

type HOSPITAL  = record
        Hospital_Name:  string;
        Address: string;
        Phone_No: string;
        end

type LAB = record
        Lab_Name:  string;
        Room_No: integer;
        Phone_No: string;
        end

type WARD = record
        Ward_Name:  string;
        Capacity: integer;
        end

type DOCTOR = record
        D_Name:  string;
        Current_Status: string;
        end

type PATIENT = record
        P_Name:  string;
        Address: string;
        Phone: string;
        end

 type   W_DOCTOR = record
        {D_Name: string;}
          (* virtual of logical parent
             DOCTOR in DOCTOR_TREE *)
        end

 type   W_PATIENT = record
        {P_Name: string;}
          (* virtual of logical parent
             PATIENT in PATIENT_TREE *)
```

```
        end

    type  D_WARD = record
         {Ward_Name: string;}
            (* virtual of logical parent
               WARD in WARD_TREE *)
         end

    type  SPECIALITY = record
         Speciality_Name : string;
         end

    type  D_PATIENT = record
         {P_Name: string;}
            (* virtual of logical parent
               PATIENT in PATIENT_TREE *)
         end

    type  P_DOCTOR = record
         {D_Name: string;}
            (* virtual of logical parent
               DOCTOR in DOCTOR_TREE *)
         end
```

9.8 a)

```
get first HOSPITAL;
while DB-Status = 0 do
 begin
  get next within parent LAB where Lab_Name = 'haematology';
  if DB-Status = 0 then display (HOSPITAL.Hospital_Name);
  get next HOSPITAL;
  end
```

9.8 b)

```
get first HOSPITAL;
while DB-Status = 0 do
 begin
  get next within parent WARD where WARD.Capacity > 4;
  while DB-Status = 0 do
    begin
     display (HOSPITAL.Hospital_Name, WARD.Ward_Name );
          get next within parent WARD where WARD.Capacity > 4;
     end
   get next HOSPITAL
   end
```

9.8 c)

```
get first PATIENT where PATIENT.P_Name = '   given   ';
if DB-Status = 0
then get next within parent P_DOCTOR;
while DB-Status = 0  do
 begin
  display (PATIENT.P_Name, P_DOCTOR.D_Name);
   get next within parent P_DOCTOR;
   end;
```

9.8 d)

```
get first DOCTOR;
while DB-Status = 0 do
 begin
   get next within parent SPECIALITY where Speciality_Name =
                                                    'pediatrics';
   if DB-Status = 0
     then display (DOCTOR.D_Name);
   get next DOCTOR;
   end
```

9.8 e)

```
no_of_doctors := 0;
get first PATIENT where PATIENT.P_Name = '   given   ';
if DB-Status = 0
then get next within parent P_DOCTOR;
while DB-Status = 0 do
 begin
   no_of_doctors := no_of_doctors + 1;
  display (PATIENT.P_Name, P_DOCTOR.D_Name);
   get next within parent P_DOCTOR;
   end;
display (PATIENT.P_Name, 'Number of Doctors = ', no_of_doctors);
```

9.8 f)

```
DOCTOR.D_Name :=  '   given   ';
DOCTOR.Current_Status'   given   ';
insert (DOCTOR) where (HOSPITAL.Hospital_Name = '  given  ');
for i := 1 to no_of _speciality do
 begin
   get (speciality);
   SPECIALITY.Speciality_Name :=  speciality;
   insert (SPECIALITY) where (DOCTOR.D_Name = '   given   ');
   end
```

10. Query Processing

Objectives: This chapter introduces the student to the following concepts:

In this chapter we focus on different aspects of converting a user's query into a standard form and thence into a plan to be executed against the database to generate a response.

Solution to selected exercises

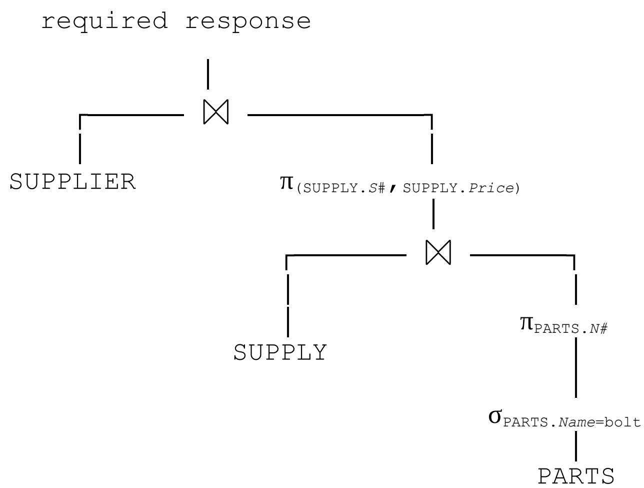
10.2.

(a) Let $X = \pi_{PARTS.P\#}(\sigma_{PARTS.Name=bolt}(PARTS))$ and

$Y = \pi_{(SUPPLY.S\#,SUPPLY.Price)}X$

The required response is given as: $Y \bowtie$ SUPPLIER

The query tree is given as:



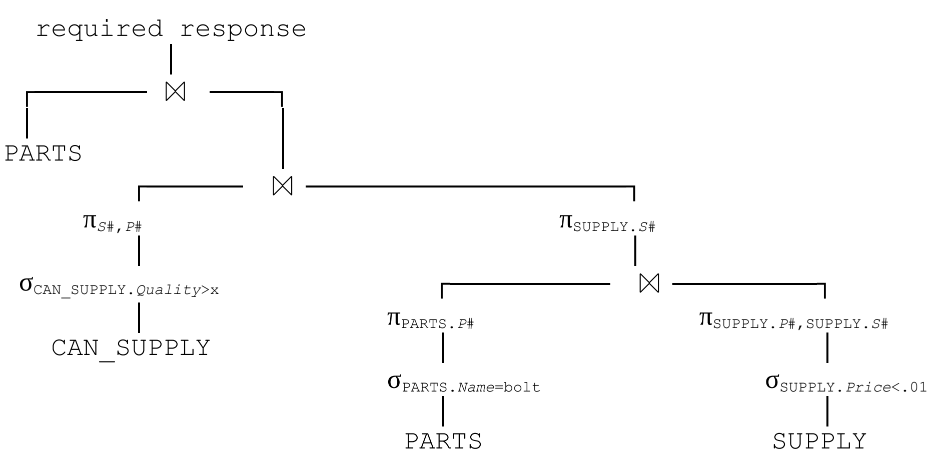
(b) Let

$X = \pi_{PARTS.P\#}(\sigma_{PARTS.Name=bolt}(PARTS))$,

$Y = \pi_{SUPPLY.S\#}(X \bowtie \pi_{SUPPLY.P\#,SUPPLY.S\#}(\sigma_{SUPPLY.Price<.01}(SUPPLY)))$

$Z = Y \bowtie (\pi_{S\#,P\#}(\sigma_{CAN\_SUPPLY.Quality>x}CAN\_SUPPLY))$

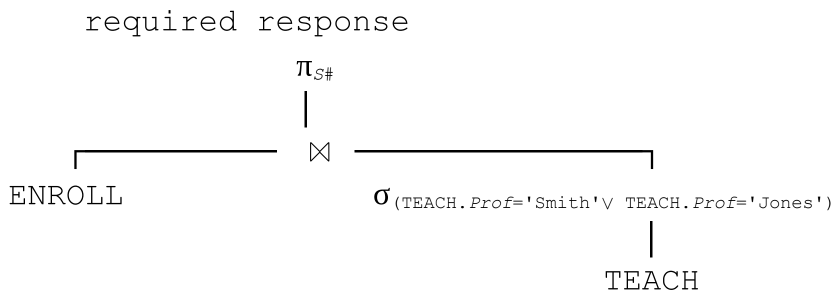
The required response is obtained as: $Z \bowtie$ PARTS



10.3. Repeat exercise 4 from Chapter 4, presenting both an efficient relational algebraic expression and the corresponding query tree.

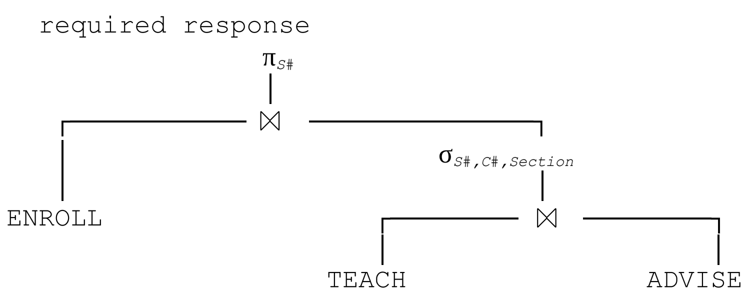
(a)

$\Pi_{S\#}\text{ENROLL} \bowtie (\sigma_{(\text{TEACH.}Prof=\text{'Smith'} \vee \text{TEACH.}Prof=\text{'Jones'})}\text{TEACH})$



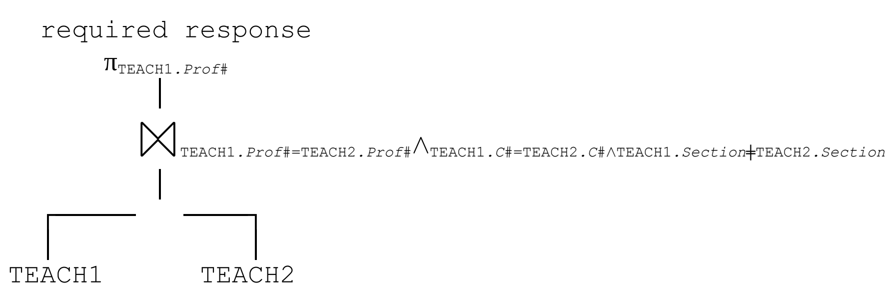
(b)

$\Pi_{S\#}(\text{ENROLL} \bowtie (\sigma_{S\#,C\#,Section}(\text{TEACH} \bowtie \text{ADVISE})))$



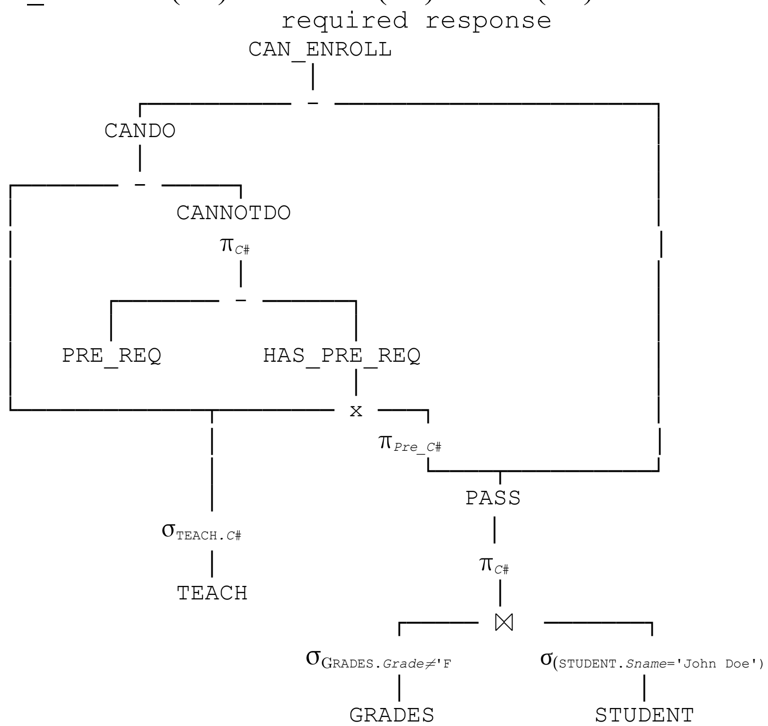
(c)

Let TEACH1 and TEACH2 be copies of the relation TEACH.
Let R = TEACH1 x TEACH2, then
$S=\sigma_{(\text{TEACH1.}Prof\#=\text{TEACH2.}Prof\# \wedge \text{TEACH1.}C\#=\text{TEACH2.}C\# \wedge \text{ TEACH1.}Section \neq \text{TEACH2.}Section)}(R)$
The required response is given by $\Pi_{\text{TEACH1.}Prof\#}S$



(d)

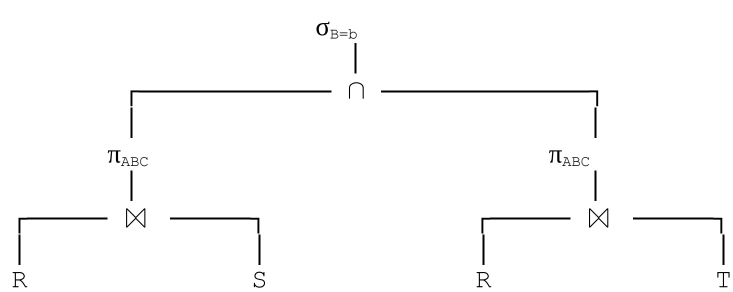
PASS($C\#$)= $\pi_{C\#}(\sigma_{GRADES.Grade \neq 'F'} GRADES \bowtie (\sigma_{(STUDENT.Sname='John\ Doe')}$ STUDENT))
HAS_PRE_REQ($C\#$, $Pre\_C\#$)=($\sigma_{TEACH.C\#}$TEACH) x PASS[$Pre\_C\#$]
CANNOTDO($C\#$) = $\pi_{C\#}$(PRE_REQ - HAS_PRE_REQ)
CANDO($C\#$) = ($\pi_{C\#}$TEACH) - CANNOTDO
CAN_ENROLL($C\#$) = CANDO($C\#$) - PASS($C\#$).



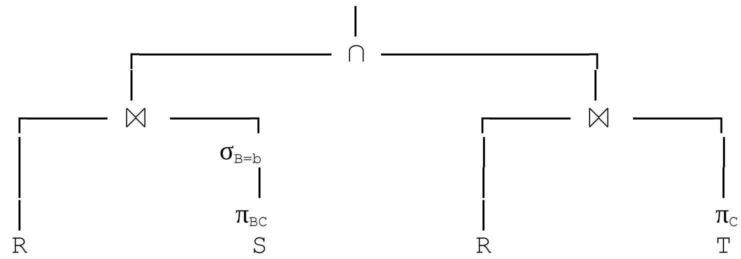
10.5.
With S in the outer loop and R in the inner loop, the number of disc accesses is 1700. If only one buffer is used for R, and the number of buffers for S is increased to 6, then the number of disc accesses can be trimmed down to 1417.

10.7. Given R(A,B,C), S(B,C,D) and T(C,D,E).

(i) $\sigma_{B=b}(\pi_{ABC}(R \bowtie S) \cap \pi_{ABC}(R \bowtie T))$

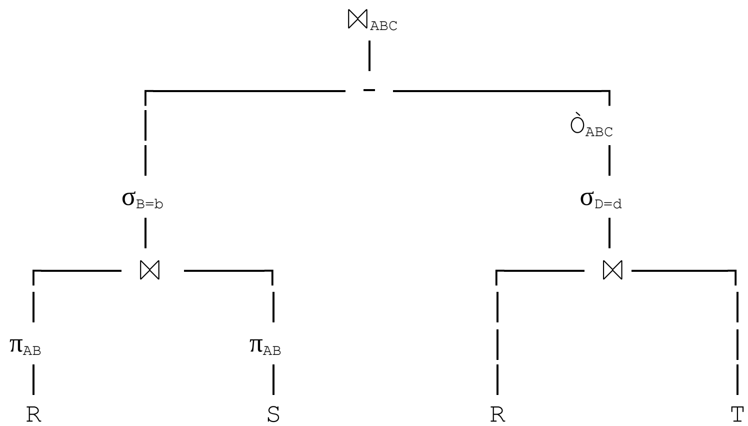


Optimized version required response

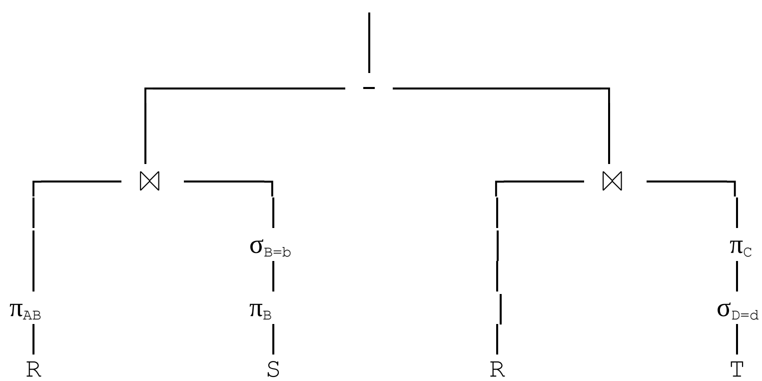


(ii) $\pi_{ABC}(\sigma_{B=b}(\pi_{AB}R)\bowtie\pi_{AB}S) - \pi_{ABC}(\sigma_{D=d}(R\bowtie T)))$

required response

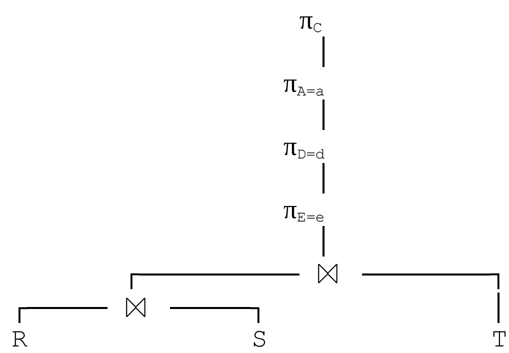


Optimized version required response

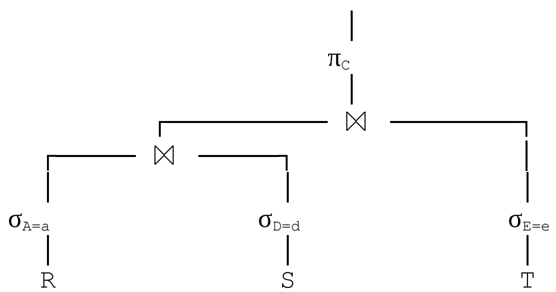


(iii) $\pi_C(\sigma_{A=a}\sigma_{D=d}\sigma_{E=e}(R\bowtie S\bowtie T))$

required response

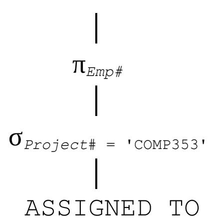


Optimized version required response

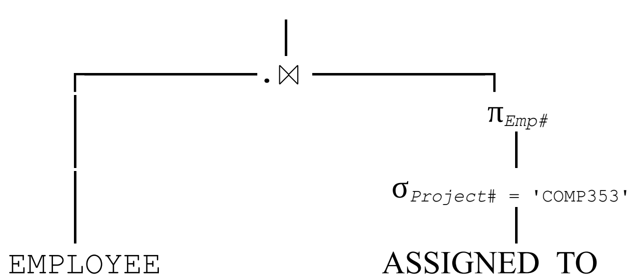


10.9. (i)

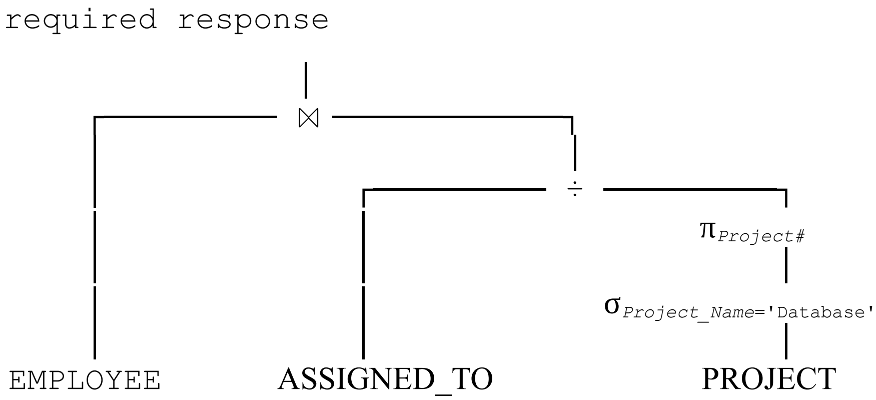
required response



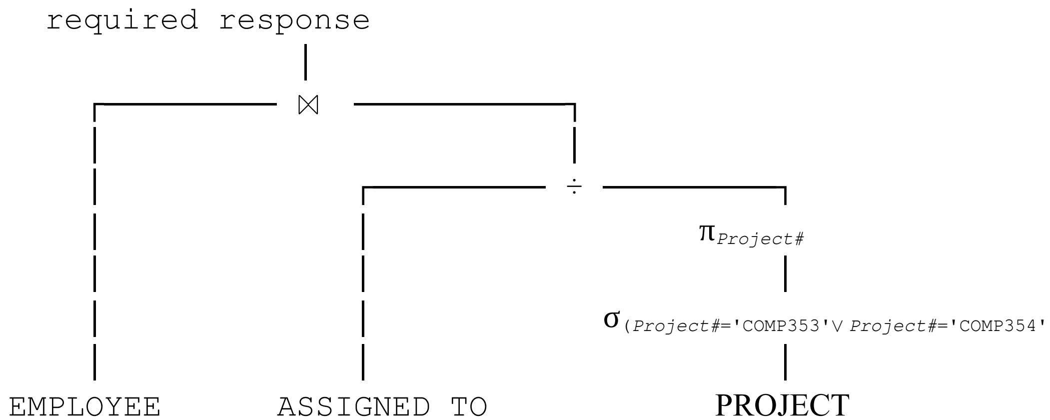
(ii) required response



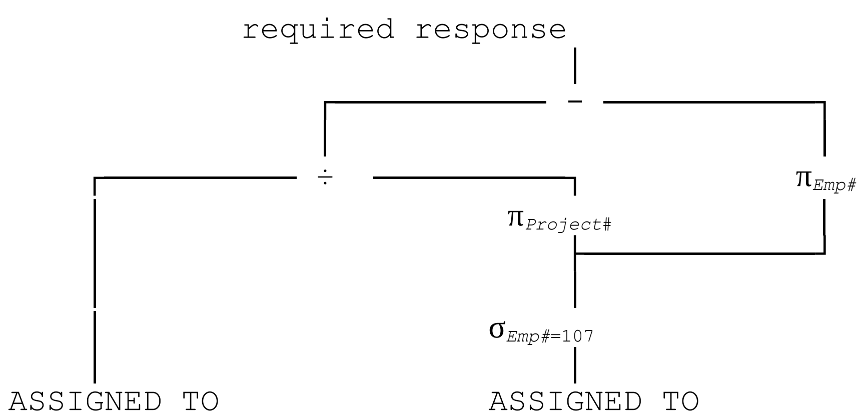
(iii)


required response
⋈
÷
π Project#
σ Project_Name='Database'
EMPLOYEE
ASSIGNED_TO
PROJECT

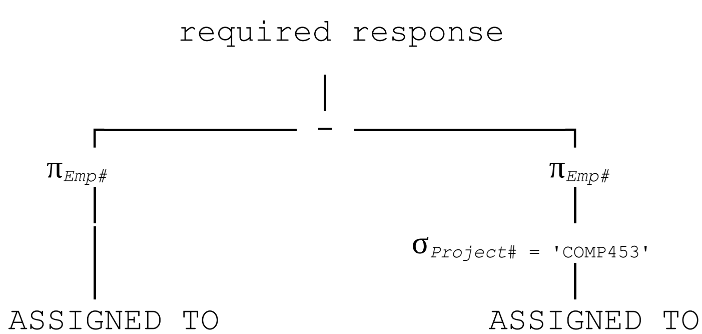
(iv)


required response
⋈
÷
π Project#
σ (Project#='COMP353'∨ Project#='COMP354')
EMPLOYEE
ASSIGNED_TO
PROJECT

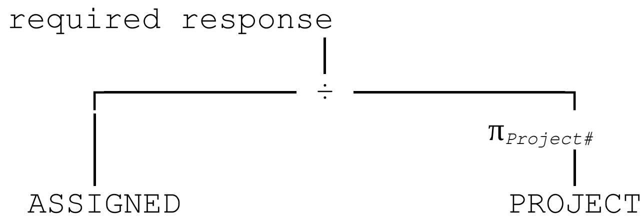
(v)


required response
−
÷
π Emp#
π Project#
σ Emp#=107
ASSIGNED_TO
ASSIGNED_TO

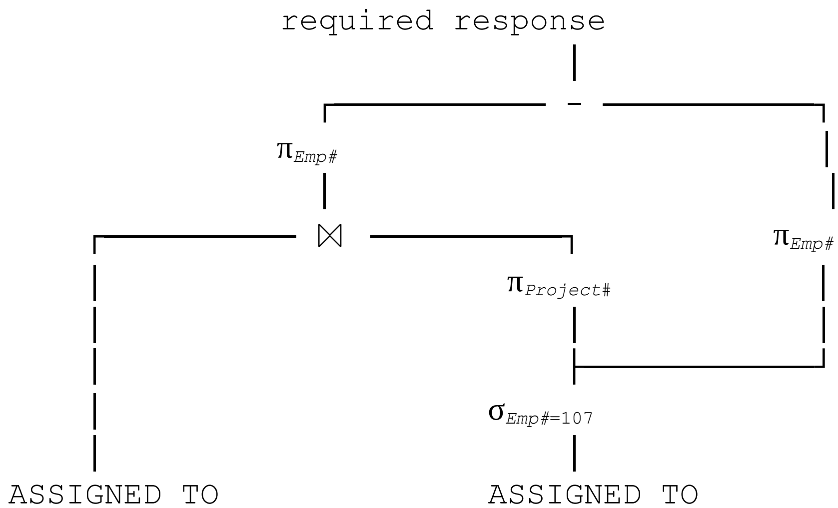
(vi)


required response
−
π Emp#
π Emp#
σ Project# = 'COMP453'
ASSIGNED_TO
ASSIGNED_TO

(vii)


required response
÷
$\pi_{Project\#}$
ASSIGNED
PROJECT

(viii)


required response
−
$\pi_{Emp\#}$
⋈
$\pi_{Project\#}$
$\pi_{Emp\#}$
$\sigma_{Emp\#=107}$
ASSIGNED_TO
ASSIGNED_TO

11. Recovery

A computer system is an electro-mechanical device subject to failures of various types. The reliability problem of the database system is linked to the reliability of the computer system on which it runs. In this chapter we will discuss the recovery of the data contained in a database system following failures of various types. We will include the type of failures that have to be considered from the point of view of providing a reliable system and present the different approaches to database recovery. The types of failures that the computer system is likely to be subjected to include failures of components or subsystems, software failures, power outages, accidents, unforeseen situations, and natural or man-made disasters. Database recovery techniques are methods of making the database fault-tolerant. The aim of the recovery scheme is to allow database operations to be resumed after a failure, with minimum loss of information, at an economically justifiable cost. We will concentrate on the recovery of centralized database systems in this chapter; the recovery issues in a distributed system are presented in chapter 13.

Solution to selected exercises

12. Concurrency Management

Concurrent execution of a number of transactions implies that the operations from these transactions may be interleaved. This is not the same as serial execution of the transactions where each transaction is run to completion before the next transaction is started. Concurrent access to a database by a number of transactions requires some type of concurrency control to preserve the consistency of the database, to ensure that the modifications made by the transactions are not lost, and to guard against transaction reading data that is inconsistent. The serializability criterion is used to test whether an interleaved execution of the operations from a number of concurrent transactions is correct or not. The serializability test consists of generating a precedence graph from a interleaved execution schedule. If the precedence graph is acyclic, then the schedule is serializable, which means that the database will have the same state at the end of the schedule as some serial execution of the transactions. In this chapter, we introduce a number of concurrency control schemes.

Solution to selected exercises

13. Database Security, Integrity & Control

Security in database involves both policies and mechanisms to protect the data in the database and ensure that the data is not accessed, altered or deleted without proper authorization. Integrity implies that any properly authorized access, alteration or deletion of the data in the database does not change the validity of the data. Security and integrity concepts, though distinct, are related. The implementation of both the security and integrity requires that certain controls in the form of constraints must be built into the system. The DBA, in consultation with the security administrators, specifies these controls. The system enforces the controls by monitoring the actions of the users of the database and limiting their actions within the constraints specified for them.

Solution to selected exercises

14. Database Design

Database design process is an iterative process. A number of design methodologies have been developed for use in the process. This chapter offers an informal discussion of the steps involved in designing a database.

Solution to selected exercises

15. Distributed Databases

In this chapter we present distributed database systems. A distributed database can be defined as consisting of a collection of data with different parts of it being under control of a separate DBMS, running on an independent computer system. All such computers are interconnected and each system has autonomous processing capability, serving local applications. Each system participates, as well, in the execution of one or more global applications. Such applications require data from more than one site.

Solution to selected exercises

16. Current Topics in Database Research

In this chapter we present some highlights of the recent advances in database system. The approach used is informal and intuitive. We discuss knowledgebase systems, logic databases, expert systems and the object oriented approach.

Solution to selected exercises

17. Database Machines

In this chapter we discuss a number of approaches used to relieve the main computer system of the burden of running the database management system and handling the superfluous data not required for deriving the response of a user's query.

Solution to selected exercises

Appendix: CopyForward

This document in electronic form, bearing a CopyForward permission, could be used for personal use and/or study, free of charge. Anyone could use it to derive updated versions. The derived version must be published under CopyForward. All authors of the version used to derive the new version must be included in the updated version in the existing order, followed by name(s) of author(s) producing the derived work.

Such derived version must be made available free of charge in electronic form under CopyForward. Any other means of reproduction requires that annual profits(income minus the actual production costs) should be shared with established charitable organizations for children. This annual share must be at least 25% of the profits and the organization being supported must have a very modest administrative charges(20-30% of their annual budget). The 25% of the profits is the minimum and the original creator of the digital content may increase it to up to 40%. The derived contents would be governed by the term of the original creator of contents.
Readers who found a CopyForward content or any derived work useful are encouraged to also make a donation to their favourite children charity. Make sure to choose charity which has very modest administrative charges or some deserving children in your community.
This children's charity contribution requirement of CopyForward is civil and moral! It would be judged in the court of public opinion.

Why yet another intellectual rights protection?

There are number of other copy permission other than the traditional **copyright**. With electronic contents it and software has become difficult to enforce copyright. Software has been opened up under some version of the copyleft (GNU GPL[1]). Another licensing arrangement is the open source licence[2,3]. Yet another version of copyright is the Creative Commons(CC) license. As in CopyForward, CC allows the creator to share, use, and building upon the CCed work but does not allow commercialization.

The document outlining copyleft is over a hundred page long as opposed to CopyForward which is just the para given above.

To the knowledge of the author, there have been no monetary claim litigations regrading the above new forms of copy protection licences. However, looking at the tech-giants that have emerged over the last

1 https://copyleft.org/
2 https://opensource.org/licenses
3 https://en.wikipedia.org/wiki/Open-source_license

few decades, they have taken something that was considered open[4] and have created monopolies, concentration of market shares and deter the creation of alternatives. The types of mobile phones and the number of operating systems is an example[5]. These new tech-barons do not pay a fair percent of their income and none on the accumulated wealth; in this way they keep enriching themselves.
While there is a move to limit the wealth as outlined in Limitarianism[6] the success of even timid moves to impose a minimum income and wealth tax rate is hardly sufficient.

How will CopyForward change?

The author's intent to publish this and other works under CopyForward is to allow the sharing of his effort and with the hope that even if there is commercialization, there is a moral and civic obligation that an appreciable part of the earnings would go to charitable causes for the next generation. It is hoped that if this charitable sharing of profits is not honoured, the public would boycott such commercialization. This is the only effective remedy for greed that exploits others' labour for obscene personal enrichment[7].

4 https://arstechnica.com/gadgets/2019/08/unix-at-50-it-starts-with-a-mainframe-a-gator-and-three-dedicated-researchers/

5 Richard Jensen, Unix at 50: How the OS that powered smartphones started from failurehttps://arstechnica.com/gadgets/2019/08/unix-at-50-it-starts-with-a-mainframe-a-gator-and-three-dedicated-researchers/

6 Ingrid Robeyns, Why Limitarianism? https://onlinelibrary.wiley.com/doi/full/10.1111/jopp.12275

7 Bipin C. Desai. Colonization of the Internet, IDEAS '21: Proceedings of the 25th International Database Engineering & Applications Symposium, https://doi.org/10.1145/3472163.3472179

ISBN 978-1-98-839210-3
12100
9 781988 392103