CHP 8

Relational ALJEBRA

↳ basic set of operations for relational model

UNARY Operations

- ↳ Select σ → sigma
- LIST ← ↳ Project π → Pi
- ↳ Rename ρ → rho

Set Theory Operations

- ↳ Union ∪
- ↳ Intersection ∩
- ↳ Difference −

need to be TYPE COMPATIBLE

- ↳ Cartesian Product x

Binary Operations

- ↳ JOIN ⋈
- ↳ Division ÷ → all, every, each

Additional Operations

- ↳ OUTER JOINS
- ↳ OUTER UNION
- ↳ Aggregate function

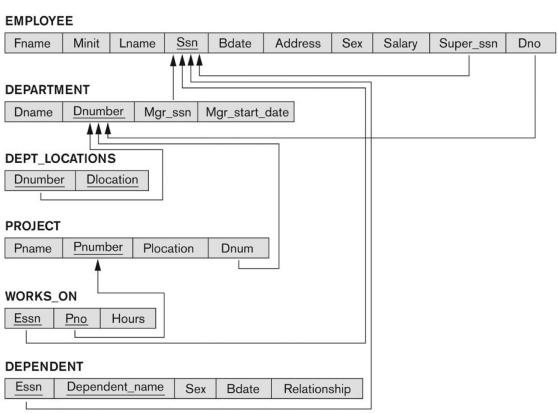
SUM, MAX, MIN, AVG, COUNT

Type Compatible

- ↳ same no of attributes
- ↳ same domain for attributes

$dom(A_i) = dom(B_i)$ for $i = 1, 2, \ldots, n$

The set of operations including SELECT σ, PROJECT π , UNION ∪, DIFFERENCE − , RENAME ρ, and CARTESIAN PRODUCT X is called a *complete set* because any other



UNARY OPERATIONS

1. Select (σ) $\sigma_{<condition\ 1>, <condition\ 2>, \ldots <condition\ n>}(R)$
   ↳ filter out tuples

   - Select the EMPLOYEE tuples whose department number is 4:

     $\sigma_{DNO = 4}$ (EMPLOYEE)
   - Select the employee tuples whose salary is greater than $30,000:

     $\sigma_{SALARY > 30,000}$ (EMPLOYEE)

2. Project (π) $\pi_{<attributes\ list>}(R)$
   ↳ keeps certain columns
   ↳ removes duplicate tuples

e) Select fname, lname, salary (ρ) from Employee E (π) where Dno = 5 AND salary > 30000 (σ)

1. Single relational Algebra expression

   $\pi_{fname, lname, salary}(\sigma_{DNO=5\ AND\ salary>30000}(Employee))$

2. Imediate result relations expression

   DEP5 EMP ← $\sigma_{Dno = 5\ AND\ salary > 30000}(Employee)$

   finalresult ← π_{fname} (DEP5 EMP)

3. Rename (ρ) $\rho_{S(B_1, B_2, \ldots B_n)}(R)$
   S: changes relation name to S; $B_1, B_2, \ldots B_n$: changes column names to $B_1, B_2, \ldots B_n$

   $\rho_{(A_1, B_1)}(Employee(A,B))$ → A (renamed to A_1), B (renamed to B_1)

A_1	B_1	C	D
A	B		

   - If we write:
     - RESULT ← $\pi_{FNAME, LNAME, SALARY}$ (DEP5_EMPS)
     - RESULT will have the *same attribute names* as DEP5_EMPS (same attributes as EMPLOYEE)
   - If we write:
     - RESULT (F, M, L, S, B, A, SX, SAL, SU, DNO) ← $\rho_{RESULT(F,M,L,S,B,A,SX,SAL,SU,DNO)}$ (DEP5_EMPS)
     - The 10 attributes of DEP5_EMPS are *renamed* to F, M, L, S, B, A, SX, SAL, SU, DNO, respectively

   Note: the ← symbol is an assignment operator

Set Theory Operations

1. Union (∪) R ∪ S

↳ Combine R and S → no of columns must be same

↳ no duplicates

↳ must be type compatible

- Example:
  - To retrieve the social security numbers of all employees who either *work in department 5* (RESULT1 below) or *directly supervise an employee who works in department 5* (RESULT2 below)
  - We can use the UNION operation as follows:

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{SSN}(DEP5\_EMPS)$$
$$RESULT2(SSN) \leftarrow \pi_{SUPERSSN}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

  - The union operation produces the tuples that are in either RESULT1 or RESULT2 or both

Some properties of UNION, INTERSECT, and DIFFERENCE

- Notice that both union and intersection are *commutative* operations; that is
  - R ∪ S = S ∪ R, and R ∩ S = S ∩ R
- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are *associative* operations; that is
  - R ∪ (S ∪ T) = (R ∪ S) ∪ T
  - (R ∩ S) ∩ T = R ∩ (S ∩ T)
- The minus operation is not commutative; that is, in general
  - R – S ≠ S – R

2. Intersection (∩) R ∩ S

↳ common tuples of R and S

↳ must be type compatible

3. Difference – R – S

↳ all tuples that are in R but not S

↳ must be type compatible

÷ = all, every

∪ = OR

doesnot = –

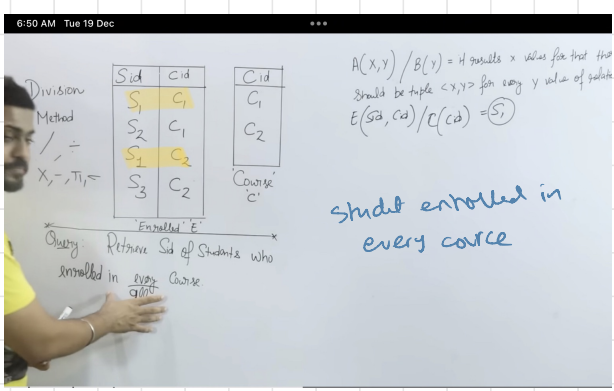
∩ = common = either/or

4. Cartesian Product × $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots B_n)$

↳ combine tuples from 2 diff relations

↳ dont need to be type compatible

- Generally, CROSS PRODUCT is not a meaningful operation
  - Can become meaningful when followed by other operations
- Example (not meaningful):
  - FEMALE_EMPS ← $\sigma_{SEX='F'}$(EMPLOYEE)
  - EMPNAMES ← $\pi_{FNAME, LNAME, SSN}$(FEMALE_EMPS)
  - EMP_DEPENDENTS ← EMPNAMES x DEPENDENT
- EMP_DEPENDENTS will contain every combination of EMPNAMES and DEPENDENT
  - whether or not they are actually related



- To keep only combinations where the DEPENDENT is related to the EMPLOYEE, we add a SELECT operation as follows
- Example (meaningful):
  - FEMALE_EMPS ← $\sigma_{SEX='F'}$(EMPLOYEE)
  - EMPNAMES ← $\pi_{FNAME, LNAME, SSN}$ (FEMALE_EMPS)
  - EMP_DEPENDENTS ← EMPNAMES x DEPENDENT
  - ACTUAL_DEPS ← $\sigma_{SSN=ESSN}$(EMP_DEPENDENTS)
  - RESULT ← $\pi_{FNAME, LNAME, DEPENDENT\_NAME}$ (ACTUAL_DEPS)
- RESULT will now contain the name of female employees and their dependents

Binary Operations

1. Division ÷ R(A) ÷ S(B)

Applied on 2 relations

- DIVISION Operation
  - The division operation is applied to two relations
  - R(Z) ÷ S(X), where X subset Z. Let Y = Z - X (and hence Z = X ∪ Y); that is, let Y be the set of attributes of R that are not attributes of S.
  - The result of DIVISION is a relation T(Y) that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with
    - $t_R[X] = t_s$ *for every tuple* t_s in S.
  - For a tuple t to appear in the result T of the DIVISION, the values in t must appear in R in combination with *every* tuple in S.

R ÷ S

R1 ← π_{SSN} (EMPLOYEE)

2. JOIN ⋈

- Example: Suppose ... name of the manager of each department.
  - To get the manager's name, we need to combine each DEPARTMENT tuple with the EMPLOYEE tuple whose SSN value matches the MGRSSN value in the department tuple.
  - We do this by using the join ⋈ operation.
  - DEPT_MGR ← DEPARTMENT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE
- MGRSSN=SSN is the join condition
  - Combines each department record with the employee who manages the department
  - The join condition can also be specified as DEPARTMENT.MGRSSN= EMPLOYEE.SSN

Equi Join R ⋈<join condition> S

Natural Join R *<condition> S

- Example: To apply a natural join on the DNUMBER attributes of DEPARTMENT and DEPT_LOCATIONS, it is sufficient to write:
  - DEPT_LOCS ← DEPARTMENT * DEPT_LOCATIONS
- Only attribute with the same name is DNUMBER
- An implicit join condition is created based on this attribute:
  DEPARTMENT.DNUMBER=DEPT_LOCATIONS.DNUMBER
- Another example: Q ← R(A,B,C,D) * S(C,D,E)
  - The implicit join condition includes *each pair* of attributes with the same name, "AND"ed together:
    - R.C=S.C AND R.D.S.D
  - Result keeps only one attribute of each such pair:
    - Q(A,B,C,D,E)

- **Q1: Retrieve the name and address of all employees who work for the 'Research' department.**

  RESEARCH_DEPT ← σ DNAME='Research' (DEPARTMENT)
  RESEARCH_EMPS ← (RESEARCH_DEPT ⋈ DNUMBER= DNO EMPLOYEE)
  RESULT ← π FNAME, LNAME, ADDRESS (RESEARCH_EMPS)

- **Q6: Retrieve the names of employees who have no dependents.**

  ALL_EMPS ← π SSN(EMPLOYEE)
  EMPS_WITH_DEPS(SSN) ← π ESSN(DEPENDENT)
  EMPS_WITHOUT_DEPS ← (ALL_EMPS - EMPS_WITH_DEPS)
  RESULT ← π LNAME, FNAME (EMPS_WITHOUT_DEPS * EMPLOYEE)

As a single expression, these queries become:

- **Q1: Retrieve the name and address of all employees who work for the 'Research' department.**

  $\pi_{Fname, Lname, Address}$ (σ Dname= 'Research' (DEPARTMENT ⋈Dnumber=Dno(EMPLOYEE))

- **Q6: Retrieve the names of employees who have no dependents.**

  $\pi_{Lname, Fname}((\pi_{Ssn}$ (EMPLOYEE) − ρ_{Ssn} (π Essn (DEPENDENT))) * EMPLOYEE)

- Use of the Aggregate Functional operation $\mathcal{F}$
  - $\mathcal{F}_{MAX\ Salary}$ (EMPLOYEE) retrieves the maximum salary value from the EMPLOYEE relation
  - $\mathcal{F}_{MIN\ Salary}$ (EMPLOYEE) retrieves the minimum Salary value from the EMPLOYEE relation
  - $\mathcal{F}_{SUM\ Salary}$ (EMPLOYEE) retrieves the sum of the Salary from the EMPLOYEE relation
  - $\mathcal{F}_{COUNT\ SSN,\ AVERAGE\ Salary}$ (EMPLOYEE) computes the count (number) of employees and their average salary
    - Note: count just counts the number of rows, without removing duplicates

- Grouping attribute placed to left of symbol
- Aggregate functions to right of symbol
- DNO $\mathcal{F}_{COUNT\ SSN,\ AVERAGE\ Salary}$ (EMPLOYEE)

Database Design Process

↳ Two main activities

↳ Database Design

↳ Application Design

↳ focuses on programs and interface that access the DB

Overview of Database Design Process

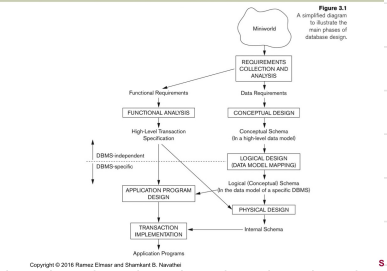


Figure 3.1
A simplified diagram to illustrate the main phases of database design.

Min/Max Cardinality

3NF

ERT

NO Relations

Entity Relationship Diagram (ERD)

conceptual view

chp 3

SDT

Entities set ?

Entity Types

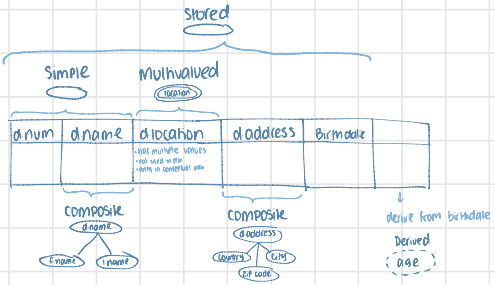
- Strong — Primary key independent — e.g. Employee SSN ?
- Weak — no Primary key dependent on strong entity — e.g. name, birth date ?

Value Sets

- Specifies set of values
  - e.g. name has string upto 15 characters
  - date has value MM-DD-YYYY

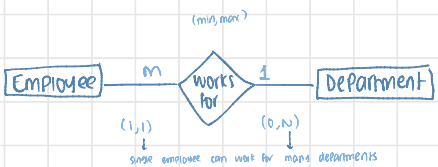
Attributes

- Simple vs Multivalued — multiple values for that attribute
- Simple vs Composite
- Stored vs Derived
- Nested Composite and Multivalued
- Key attribute



Relationships / Associations

- 1 : many (N)
- many (N) : 1
- many (N) : many (N)
- 1 : 1



Structural Constraints

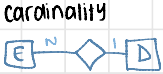
- Participation
  - Full Participation — every employee is in a department
  - Partial Participation — not every department has an employee
- Cardinality → (min , max)
  - partial : 0
  - full : 1

USE ANY ONE

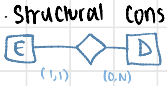
3 ways to represent

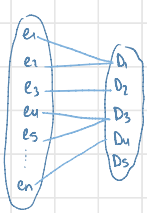
1. Participation

2. Cardinality



3. Structural Constraint (min, max)





corresponding Entity Set

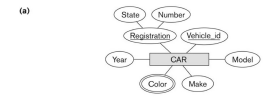


Figure 3.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

(b)

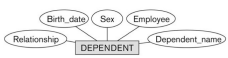
CAR
Registration (Number, State), Vehicle_id, Make, Model, Year, {Color}

CAR_1
((ABC 123, TEXAS), TK629, Ford Mustang, convertible, 2004 {red, black})

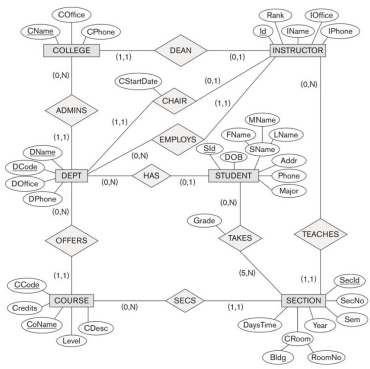
CAR_2
((ABC 123, NEW YORK), WP9872, Nissan Maxima, 4-door, 2005, {blue})

CAR_3
((VSY 720, TEXAS), TD729, Chrysler LeBaron, 4-door, 2002, {white, blue})

⋮



Prelimi
type
da
sho
refin



Indentifying Relationship

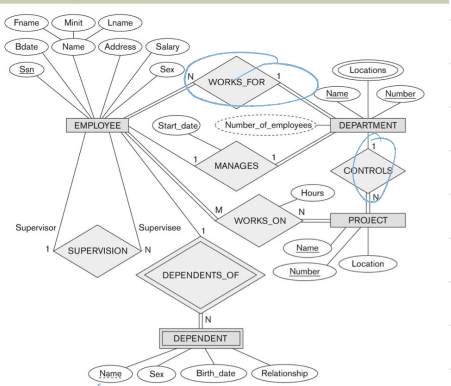
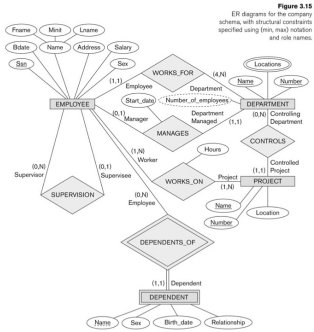


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

declarator
reasoning to decide
NOT PRIMARY KEY

COMPANY ER Schema Diagram using (min, max) notation

Figure 3.15
ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.



Recursive Relationship Type

↳ same participating entity in different roles

e.g employee as manager

employee as worker

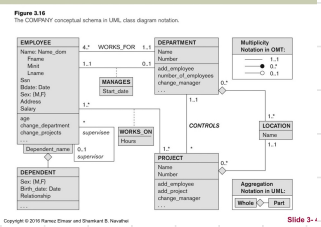
Relationship Type

Relationship Set

↳ current state of relationship type

UML Class Diagrams

UML class diagram for COMPANY database schema



Relationships of Higher Degree

↳ Binary : RS types of degree 2

↳ Ternary : RS types of degree 3

↳ n-ary : RS types of degree n>2

harder to specify constraints

Example of a ternary relationship

3 Binary RS

weak entity

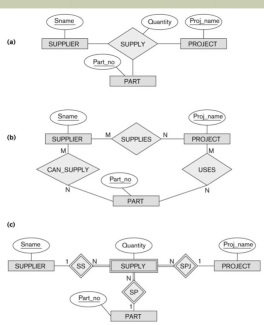


Figure 3.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

Normalization

chp 14 SOT.

Normalization

- used on vague data
- to reduce
  - redundancy
  - null values
    - attribute not applicable/invalid
    - attribute value unknown
    - value known to exist, but unavailable

Relational Database Design

- groupings of attributes to form good relation schemas
- it has 2 levels
  1. Logical level → user view
  2. storage level → base relation

ANOMALIES

2. Update Anomaly
- chances of redundancy

if EMP_id changes update all of them in list

3. Delete Anomaly
- chances to have null value

eg if we delete a Project all emp with that Project will be null vice versa

1. Insert Anomaly
- chances to have null value
  - eg inserting Ali but theres no project yet
- chances of redundancy
  - eg inserting Ali multiple times for diff projects

EXAMPLE OF AN UPDATE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Update Anomaly:
  - Changing the name of project number P1 from "Billing" to "Customer-Accounting" may cause this update to be made for all 100 employees working on project P1.

EXAMPLE OF AN INSERT ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Insert Anomaly:
  - Cannot insert a project unless an employee is assigned to it.
- Conversely
  - Cannot insert an employee unless an he/she is assigned to a project.

EXAMPLE OF A DELETE ANOMALY

- Consider the relation:
  - EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)
- Delete Anomaly:
  - When a project is deleted, it will result in deleting all the employees who work on that project.
  - Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

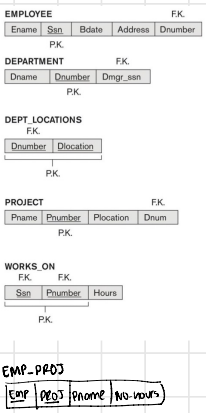
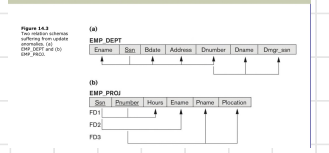


Figure 14.3 Two relation schemas suffering from update anomalies



Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Redundancy Redundancy

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation
123456789	1	32.5	Smith, John B.	ProductX	Bellaire
123456789	2	7.5	Smith, John B.	ProductY	Sugarland
666884444	3	40.0	Narayan, Ramesh K.	ProductZ	Houston
453453453	1	20.0	English, Joyce A.	ProductX	Bellaire
453453453	2	20.0	English, Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong, Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong, Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong, Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong, Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya, Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya, Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar, Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar, Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace, Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace, Jennifer S.	Reorganization	Houston
888665555	20	Null	Borg, James E.	Reorganization	Houston

Decompositions

↳ non additive or lossiessness of the corresponding join → Very imp cant be sacrificed

↳ Preservation of the functional dependencies → may be sacrificed

Figure 14.7 Ruling Out FDs

Note that given the state of the TEACH relation, we can say that the FD: Text → Course may exist. However, the FDs Teacher → Course, Teacher → Text and Couse → Text are ruled out.

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

Functional Dependencies

$X \rightarrow Y$ Text → Cource

X will have diff y every text has diff cource

	X	Y
	SSN	Emp_name
t_1	123	Ishma
t_2	123	Ishma

if $t_1(X) = t_2(X)$

then $t_1(Y) = t_2(Y)$

X = Prime Y = non Prime

all non Prime are determined by Prime attributes

X determines Y

Text → Cource FD

Teacher → Text NO FD

Cource → Text NO FD

Teacher → Cource NO F

eg

SSN → ENAME

P.id → {PNAME, PLOCATION}

{SSN, PNUMBER} → HOURS

Normalization

↳ lower NF to higer NF → Normalization Form

↳ decomposition

↳ no of decomposition = no of join operations

eg 3 NF to 4 NF

De Normalization

↳ higer NF to lower NF

↳ combine

eg 4NF to 3 NF

Emp_name	Emp_id	Dep_id	P_no	P_id	P_location
Ali	123	NULL	NULL	NULL	NULL

⇓ TO 3NF

Emp_name	Emp_id
Ali	123

Dep_id	Emp_id
NULL	123

P_no	P_id	P_location	Emp_id
NULL	NULL	NULL	123

Super key

↳ a set of attributes S

↳ such that

$t_1[S] \neq t_2[S]$

Key K

↳ a super key

↳ such that removal of any attribute from K will cause K to no longer be a superkey

Candidate key

↳ a relation schema has > 1 key

1. Primary key
2. Secondary key

Prime attribute: Primary | Secondary key

NON Prime attribute: not a Primary/secondary key

1 NF

↳ each cell should have single attribute

↳ no nested relation

↳ no multi valued attributes  ✗

↳ no composite attributes  ✗

↳ can have redundant values → 2NF and > NF don't have redundant values

(a) NON 1 NF
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b) SAMPLE
DEPARTMENT

multi valued

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	(Bellaire, Sugarland, Houston)
Administration	4	987654321	(Stafford)
Headquarters	1	888665555	(Houston)

(c) 1 NF with redundancy
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

How to remove multivalued

1. Decomposition
   ↳ seperate table
   ↳ make PK FK to the other table
2. Repeat info for id multiple times

(a) nested relation attribute
EMP_PROJ

Ssn	Ename	Projs	
		Pnumber	Hours

→ composite

(b) sample
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c) Decomposition
EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Normalizing nested r
nested relation at
showing nested re
relations

2NF

↳ no partial dependency

↳ Full functional Dependency

↳ if every non Prime is ↗ on the Primary key

AB → C

composite key

if C partially depends on AB
then normalize
eg C depends only on B but not on A

Examples:
- {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold
- {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency) since SSN -> ENAME also holds

(a) Normalizing to 2NF
EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation
FD1					
FD2					
FD3					

2NF Normalization

EP1

Ssn	Pnumber	Hours
FD1		

EP2

Ssn	Ename
FD2	

EP3

Pnumber	Pname	Plocation
FD3		

(b) normalizing to 3NF
EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn

3NF Normalization

ED1

Ename	Ssn	Bdate	Address	Dnumber

ED2

Dnumber	Dname	Dmgr_ssn

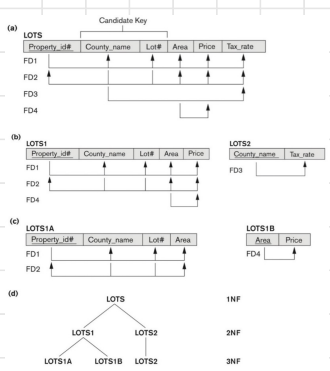
Figur
Normalizing into 2NF a
(a) Normalizing EMP_PI
2NF relations. (b) Nor
EMP_DEPT into 3NF r

Find PK

↳ if all columns same of any attribute
then that attributes is PK

↳ if cant
then create a Sno column
give values 1,2,3... n
make it PK

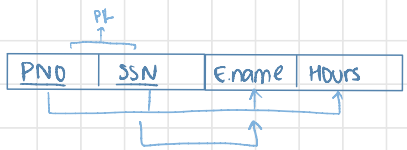
bad practice



2NF

↳ no Partial dependency

If 2F, no partial dependency → so remove composite



Removed composite

R1 → SSN | E.name
(SSN: PK)

R2 → PNO | SSN | hours
(PNO: PK, SSN: FK)

3NF

↳ no transisitive dependency

X → Y → Z

X: PK
Y: non candidate → if candidate
Z: non candidate
X ✓

EMP(SSN, EMP#, Salary)

SSN → EMP# → Salary

EMP#: candidate key ✓

STEPS

if 1 non prime taken from main table into R1, remove that from main table

R1 and R2 born in 2NF and no partial dependency

every tables non prime is fully dependent on prime

Transisitive Dependency

Functional ↑

↳ X → Y → Z

then X → Z

(→ determines)

FULL Dependency

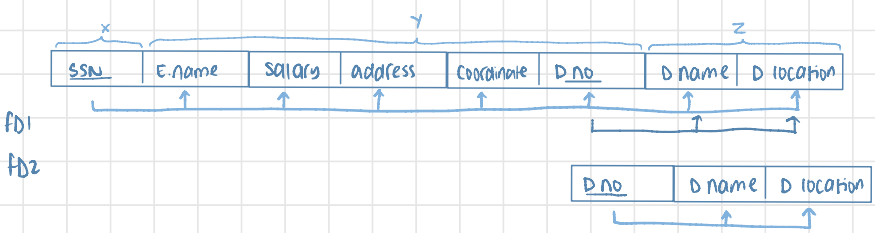
Functional ↑

Partial Dependency

Functional ↑

↳ Composite PK : 2 or more PK

↳ not 2NF



1NF all attributes depend on a key (composite key), Redundancy

2NF all attributes depend on the whole key, No Redundancy, No Partial dependency,

3NF all attributes depend on nothing but the key, No Redundancy, no transisitive dependency

1 NF

Employee

EmployeeId	Name	Address	Phone 1
201	Saghir	R288 Karachi	033255
202	Harris	G25 Yorkshire	033543
203	Maxwell	K87 Surrey	035872
204	Andy	Y78 NewCastle	038896
205	Simon	R288 London	038745
206	Sam	F7 Manchester	031210
207	Jim	R88 London	031247
208	Taylor	A4 Manchester	033351

Primary key

Employee

EmployeeId	Name	Address
201	Saghir	R288 Karachi
202	Harris	G25 Yorkshire
203	Maxwell	K87 Surrey
204	Andy	Y78 NewCastle
205	Simon	R288 London
206	Sam	F7 Manchester
207	Jim	R88 London
208	Taylor	A4 Manchester

Phone

Phone	Name
033255	Saghir
033674	Saghir
033543	Harris
035872	Maxwell
036536	Maxwell
035972	Maxwell
038896	Andy
038745	Simon
031210	Sam
033825	Sam
031247	Jim
033111	Jim
033755	Jim
033351	Taylor
033751	Taylor

This is 1NF

no repeating columns, no repeating values

?

2NF

Result

StudentId	Course	Name	Marks	Teacher
201	Software Architecture	Saghir	85	A
202	Software Design	Harris	90	B
201	Quality Assurance	Saghir	75	G
204	Enlish Language	Andy	63	O
205	History	Simon	74	L
206	Project Managent	Sam	93	G
205	Software Architecture	Simon	70	A
208	Quality Assurance	Taylor	61	G

COMPOSITE KEY

TEACHER partially depends on COMPOSITE KEY

Result

StudentId	Course	Name	Marks
201	Software Architecture	Saghir	85
202	Software Design	Harris	90
201	Quality Assurance	Saghir	75
204	Enlish Language	Andy	63
205	History	Simon	74
206	Project Managent	Sam	93
205	Software Architecture	Simon	70
208	Quality Assurance	Taylor	61

Teacher

Course	Teacher
Software Architecture	A
Software Design	B
Quality Assurance	G
Enlish Language	O
History	L
Project Managent	G

now this is in 2NF

Both tables have no partial dependecny

take away partailly dependent to new table

3NF

Exam

StudentId	Name	ExamType	MaxMarks
201	Saghir	Viva	20
202	Harris	Theroy	100
203	Maxwell	Practical	50
204	Andy	Practical	50
205	Simon	Viva	20
206	Sam	Theroy	100
207	Jim	Theroy	100
208	Taylor	Practical	50

Primary key

MaxMarks transitively depends on Examtype

violating 3NF

Match

Match#	Teams	Ground	Capicity
01	Aus v Eng	MCG	80000
02	Zim v Eng	Hobart	30000
03	SI v Ind	SCG	65000
04	Pak v Wi	MCG	80000
05	SA v NZ	Hobart	30000
06	Ausv Ind	Perth	45000
07	Pak v Ind	Canbera	10000
08	Eng v SI	MCG	80000

v

Exam

StudentId	Name	ExamType
201	Saghir	Viva
202	Harris	Theroy
203	Maxwell	Practical
204	Andy	Practical
205	Simon	Viva
206	Sam	Theroy
207	Jim	Theroy
208	Taylor	Practical

Marks

ExamType	MaxMarks
Viva	20
Theroy	100
Practical	50

Match

Match#	Teams	Ground
01	Aus v Eng	MCG
02	Zim v Eng	Hobart
03	SI v Ind	SCG
04	Pak v Wi	MCG
05	SA v NZ	Hobart
06	Ausv Ind	Perth
07	Pak v Ind	Canbera
08	Eng v SI	MCG

Capicity

Ground	Capicity
MCG	80000
Hobart	30000
SCG	65000
Perth	45000
Cambera	10000

Single User DBMS

↳ one user at a time

e.g home computer

Multi User DBMS

↳ many users can acess system at a time

e.g airline reservation system, banks, supermarket

Multiprogramming: execute multiple processes concurrently → execute some commands of P1, then suspend execute some commands of P2, then suspend resume P1....

Interleaved Processing

↳ concurrent execution of processes

↳ keep CPU busy, when I/O operation required by switching to another process instead of remaining idle

↳ prevents long process from delaying other processes

Parallel Processing

↳ uses multiple CPUs

↳ parallel processing of multiple processes

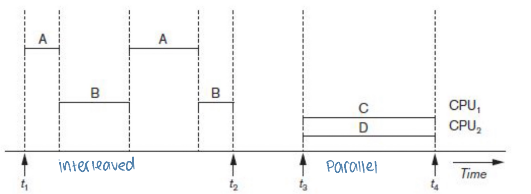


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

Transaction

→ atomic operation either be completed on not done at all

↳ an executing program that forms a logical unit of DB processing

↳ has many operations

1. Insersion
2. Deletion
3. Modification
4. Retreival

↳ Specifying transaction boundaries

1. begin transaction statement
2. end transaction statement

↳ Types of Transaction

1. Read only transaction → only retreive no updation
2. Read write transaction → retreive + updation

Database

↳ a collection of named data items → its size is called Granduality

↳ data items

1. DB Record
2. Disk block (larger unit)
3. Attribute value of a DB record (smaller unit)

Transaction Processing Systems

↳ systems with large databases

↳ has many concurrent users

↳ reauires high availability, fast response time

↳ these are independent of item granduality

Read_item(x)

- Reads a database item named X into a program variable named X

Steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer). The size of the buffer is the same as the disk block size.
3. Copy item X from the buffer to the program variable named X.

Write_item(x)

- Writes the value of program variable X into the database item named X
- Process includes finding the address of the disk block, copying to and from a memory buffer, and storing the updated disk block back to disk

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated disk block from the buffer back to disk (either immediately or at some later point in time).

(a)

T_1
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

→ Read set transaction (X,Y)
→ Write set transaction (X,Y)

(b)

T_2
read_item(X);
$X := X + M$;
write_item(X);

→ Read set transaction (X)
→ Write set transaction (X)

Figure 20.2 Two sample transactions (a) Transaction $T1$ (b) Transaction $T2$

Concurrency Control

↳ transactions by multiple users may execute concurrently leading to acess and updation of same DB items causing an inconsistent DB

↳ Problems

1. Lost Update Problem
2. Temporary Update Problem
3. Incorrect Summary Problem
4. Unrepeatable Read Problem

DBMS Buffers

- DBMS will maintain several main memory data buffers in the database cache
- When buffers are occupied, a buffer replacement policy is used to choose which buffer will be replaced
  - Example policy: least recently used

1. LOST UPDATE PROBLEM → nullifying update of first transaction

↳ T_1, T_2 perform interleaved operation on same data item

↳ resulting in incorrect value of DB item

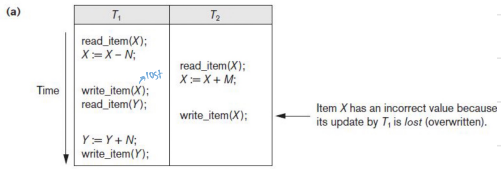


Figure 20.3 Some problems that occur when concurrent execution is uncontrolled (a) The lost update problem

2. Dirty Read Problem → reading uncommited data

↳ T_1 updates a X and the transaction fails → for some reason

↳ meanwile updated transaction is read by T2 before its changed back to its og value

roll back

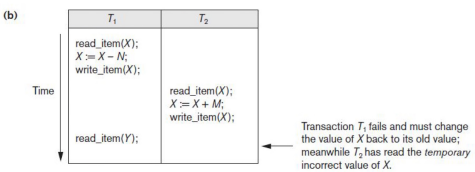


Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (b) The temporary update problem

3. Incorrect Summary Problem → invalid result on aggregating data

↳ T_1 calculating an aggregate summary function on multiple DB items

↳ while other transactions are updating some of these items

↳ the aggregate function may calculate some values before and some values after they are updated

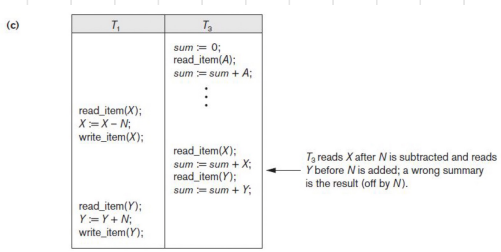


Figure 20.3 (cont'd.) Some problems that occur when concurrent execution is uncontrolled (c) The incorrect summary problem

4. Unrepeatable Read Problems → diff value for same data set variable

↳ T_1 reads same DB item twice

↳ while T_2 changed its value before T_1 read it the second time

↳ so T_1 receive 2 diff values of same item

WHY IS RECOVERY NEEDED

↳ To make sure

1. Commited Transactions: effects are permanentally recorded in DB
2. Aborted Transactions: dont affect the DB

Types of Transaction Failures

→ during transaction execution

* failure 1, 4 occurs system must keep sufficient info to recover quickly

common

1. Computer failure, system crash → hardware/software/network error
2. Transaction/system error → integer overflow, division by zero, logical precondions error
3. local errors → datas item not found — or exception conditions → insufficient account balance in banking may cause withdraw fund be cancelled
4. concurrency control enforcement → aborts transactions when ↳serializability violations ↳deadlocks → restarted automatically later

long recovery time

5. Disk failure → block disks lost their data due to readlwrite head crash
6. Physical Problems, catastrophes → power, air conditioning failure, fire, theft

Transaction States

→ for recovery purposes to keep track

↳ BEGIN_TRANSACTION

↳ READ/WRITE

↳ END_TRANSACTION

↳ COMMIT_TRANSACTION

↳ ROLLBACK/ABORT

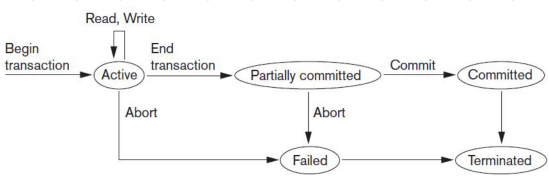


Figure 20.4 State transition diagram illustrating the states for transaction execution

System log

→ to be able to recover from system failures

↳ keeps track of transaction operations

↳ its a sequential, append only file

↳ not affected by failure → except disk/castrophic

↳ periodically backed up

↳ allows undo/redo operations based on log

log buffer

↳ main memory buffers

↳ when full, appends to the end of system log file

Commit Point of Transaction

↳ when all operations completed sucessfully

↳ and effect of operation stored in log

- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded start_transaction but no commit record
- Force-writing the log buffer to disk
  - Writing log buffer to disk before transaction reaches commit point

DBMS Specific Buffer Replacement Policy

* DBMS cache: hold disk pages that are currently being processed in main memory buffers

* Domains: DBMS cache divided into domains

- Page replacement policy
  - Selects particular buffers to be replaced when all are full
- Domain separation (DS) method → LRU page replacement
  - Each domain handles one type of disk pages (DBMS cache divided into domains)
    - Index pages
    - Data file pages
    - Log file pages

    Page replacements for each domain handled via LRU
  - Number of available buffers for each domain is predetermined
- Hot set method → page replacement algo
  - Useful in queries that scan a set of pages repeatedly → join operation in nested loops
  - Does not replace the set in the buffers until processing is completed → disk pages that will be accesses repeatedly
- The DBMIN method → page replacement policy
  - Predetermines the pattern of page references for each algorithm for a particular type of database operation
    - Calculates locality set using query locality set model (QLSM)

ACID PROPERTIES: Transactions Several Properties

→ enforced by concurrency control, recovery methods

- Atomicity
  - Transaction performed in its entirety or not at all
- Consistency preservation
  - A transaction ↑ Takes database from one consistent state to another, if there's no interference from any other transaction
- Isolation
  - A transaction must ↑ Not interfered with/by other transactions
- Durability or permanency
  - Changes must persist in the database
    of commit transaction.
    These changes must not be lost

- Levels of isolation
  - Level 0 isolation does not overwrite the dirty reads of higher-level transactions
  - Level 1 isolation has no lost updates
  - Level 2 isolation has no lost updates and no dirty reads
  - Level 3 (true) isolation has repeatable reads (no un repeatable)
    - In addition to level 2 properties
  - Snapshot isolation → another type of isolation
    → no phantom records

schedule / History

- Schedule or history of n transactions
  - Order of operations of transactions
  - Operations from different transactions can be interleaved in the schedule
- Total ordering of operations in a schedule
  - For any two operations in the schedule, one must occur before the other

conflicting operations in a schedule

- Two conflicting operations in a schedule are in conflict, if the satisfy these conditions
  1. Operations belong to different transactions
  2. Operations access the same item X
  3. At least one of the operations is a write_item(X)
- Two operations conflict if changing their order results in a different outcome
- Read-write conflict
- Write-write conflict

Characterizing Schedules Based on Recoverability

Recoverable schedules → no abort

↳ a commited T should never be rolled back

↳ recoverable

T_1	T_2
r(x)	
	r(x)
w(x)	
r(y)	
	w(x)
	commit
w(y)	
commit	

Non recoverable schedules → abort used

↳ a commited T is rolled back

↳ should not be permitted by DBMS

T_1	T_2
r(x)	
w(x)	
	r(x)
r(y)	
	w(x)
	commit
abort	

↓ so now value of x is no longer valid

T_1	T_2
r(x)	
w(x)	
	r(x)
r(y)	
	w(x)
	commit
abort	

Cascading rollback

↳ uncommited T, may need to be rolled back if dirty read

↳ This is time consuming ↴

SOLUTION

Casadless schedules → recoverable

↳ every T only reads items written by commited T

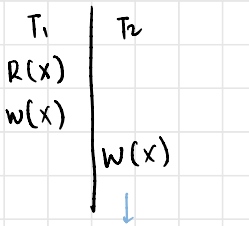
↳ read needs to be commited

Strict schedules → recoverable, cascadeless by default

↳ every T only reads, writes items written by commited T

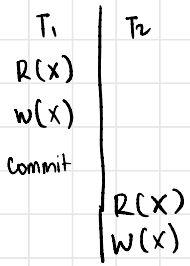
↳ read, write commited

↳ simpler recovery process → restore the before image



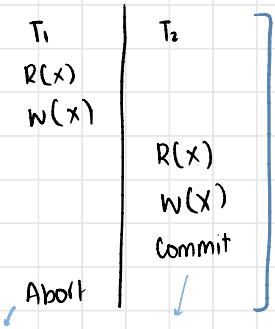
cascadless → no read in T_2

not strict → no commit



cascadeless

strict



Recoverable

Problem: Dirty read

1. commit
2. Abort

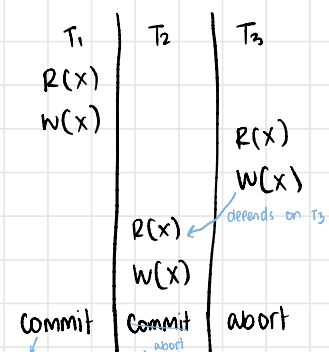
2 possibilities

↳ no write w/o read

cascading abort

↳ if T_1 aborts so do the rest of the transaction

↳ concurrency failure transaction



→ irrecoverable

depends on T_3

doesn't matter if commit late

abort

so this should be abort to make it recoverable

Q1)

T_1	T_2	T_3
$R(x)$		
	$R(z)$	
$R(z)$		
		$R(x)$
		$R(y)$
$W(x)$		
commit		
		$W(y)$
		commit
	$R(y)$	
	$W(z)$	
	$W(y)$	
	commit	

no issue as only read

cascadeless
strict
Recoverable

Q2)

T_1	T_2	T_3
$R(x)$		
	$R(z)$	
$R(z)$		
		$R(x)$
		$R(y)$
$W(x)$		
		$W(y)$
	$R(y)$	
	$W(z)$	
	$W(y)$	
commit		
	commit	
		commit

only read

$R(y)$ → is committed read

$W(z)$ → is committed write

last transaction and committed in the end so its committed

cascadeless
strict
Recoverable

Q3)

T_1	T_2	T_3
$R(x)$		
	$R(z)$	
		$R(x)$
$R(z)$		
	$R(y)$	
		$R(y)$
$W(x)$		
commit		
	$W(z)$	
		$W(y)$
	$W(y)$	
	commit	
		commit

cascadeless
strict
Recoverable

T_1	T_2	T_3
$R(x)$		
$W(x)$		
	$R(x)$	
	$W(x)$	
abort		
		$W(x)$
	~~commit~~ abort	
		~~commit~~ abort → now its recoverable

irrecoverable

CHARACTERIZING SCHEDULES BASED ON SERIALIZABILITY

Serializable schedules

↳ always correct when concurrent T are executing

↳ places simultaneous T in series → T T_1 before T_2 / T_2 before T_1

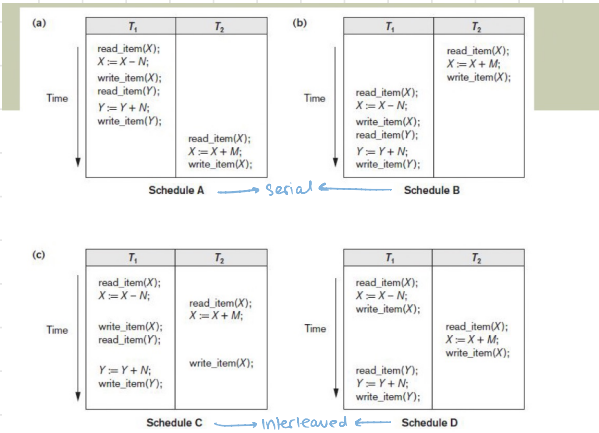


Figure 20.5 Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c) Two nonserial schedules C and D with interleaving of operations

Serial Schedule

↳ T_1 complete then T_2 start

↳ no concurrency

interleaved schedule

↳ T_1, T_2 concurrency

- Problem with serial schedules
  - Limit concurrency by prohibiting interleaving of operations
  - Unacceptable in practice
  - Solution: determine which schedules are equivalent to a serial schedule and allow those to occur

Serializability

↳ interleaved schedule is equivalent to serial schedule

↳ interleaved, serial both produce same final state

Conflicting Operations

↳ W(x), R(x) no R(x), R(x)

↳ W(x), W(x)

↳ R(x), W(x)

Conflict equivalence

↳ relative order of conflicting operations is same in both schedules

Check Serializability

If conflict equivalance

make precedence graph

↳ make nodes of all transactions

↳ check for conflicting

↳ $W_1(x), R_2(x)$ $T_1 \rightarrow T_2$

↳ $R_1(x), W_2(x)$ $T_1 \rightarrow T_2$

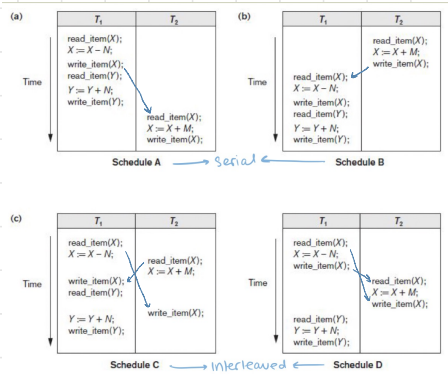
↳ $W_1(x), W_2(x)$ $T_1 \rightarrow T_2$

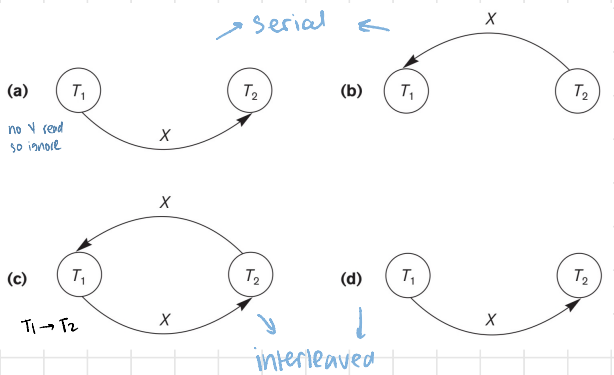
*min incoming edges runs first

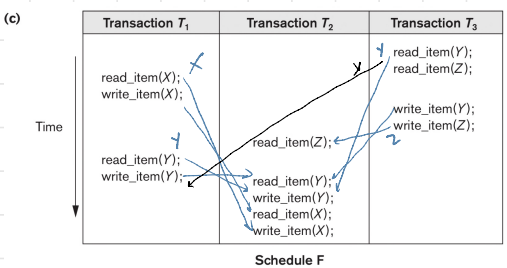
cycle → not equivalent to serial

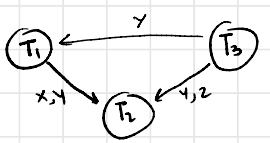
no cycle → equivalent to serial

Precedence graph



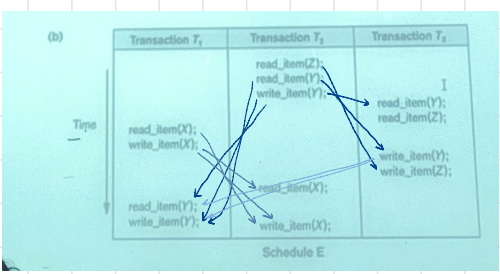


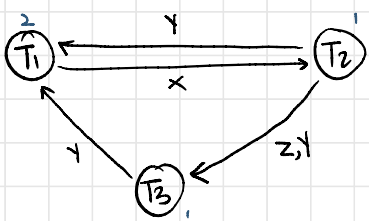




equivalent serial s

$T_3 \rightarrow T_1 \rightarrow T_2$





(b)

Transaction T_1	Transaction T_2	Transaction T_3

	read_item(Z);
	read_item(Y);
	write_item(Y);	

		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(X);	
read_item(Y); write_item(Y);		
	write_item(X);	

Time

Schedule E

equivalent serial schedule

None

as cycle $T_1 \rightarrow T_2$

Serializable schedules

conflict serializable

→ by default view resizable

- ↳ has write operation
- ↳ both operations on same data item
- ↳ both operations are by diff T

check

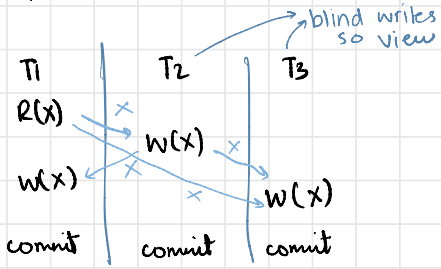
- ↳ draw precedence graph
  - ↳ if no cycles then ✓
  - ↳ if cycles then ✗

view serializable

- ↳ if conflict resizable then ✓
- ↳ if
  - blind operation ✓ (write without a previous read operation)
  - no blind operation ✗ → write

constrained write assumption

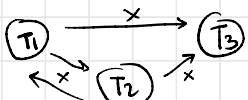
- ↳ read operation, then write operation only
- ↳ no blind writes



view resizable
not conflict resizable

unconstrained write assumption

- ↳ value written by an operation can be independent of its old value



↓ cycle
so not conflict

How serializability is used for concurrency control

serilizable schedule
↳ diff from being serial as (no concurrency)
↳ it gives benift of concurrent execution without giving up any correctness

- Difficult to test for serializability in practice
  - Factors such as system load, time of transaction submission, and process priority affect ordering of operations

- DBMS enforces protocols
  - Set of rules to ensure serializability

View equivalance of 2 schedules

1. same set of T and operations of those T
2. read operation value reads result of same write operation in both schedule
   write operation produces same results in both schedule
3. last operation same in both schedules

OTHER TYPES OF EQUIVALANCE SCHEDULES

Debit card transactions

↳ can use non serilizable schedules → as order doesnt matter

TRANSACTION SUPPORT IN SQL

- No explicit Begin_Transaction statement
- Every transaction must have an explicit end statement
  - COMMIT
  - ROLLBACK
- Access mode is READ ONLY or READ WRITE
- Diagnostic area size option
  - Integer value indicating number of conditions held simultaneously in the diagnostic area

- Isolation level option
  - Dirty read
  - Nonrepeatable read
  - Phantoms

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Table 20.1 Possible violations based on isolation levels as defined in SQL

Phantoms

↳ T_1 read a set of rows

↳ while T_2 inserts a row r

↳ r is a phantom record as
it wasnt there when T_1 starts
but is there when T_1 ends

SOLUTION

Snapshot Isolation

↳ T reads only commited value of data items

↳ ensures Phantom record wont occur

CHP 21

Concurrency control protocols
↳ set of rules to guarantee serializability

uses →

Timestamp
↳ unique identifier for each T

Two Phase locking protocol (2PL)
↳ locks data items to prevent multiple T from acessing items concurrently
↳ has high overhead

Multiversion concurrency protocols
↳ uses multiple versions of a data item

Optimistic protocols
↳ validation of a T
↳ certification of a T
↳ they also assume multiple versions of a data item

LOWER OVERHEAD

TWO PHASED LOCKING TECHINIQUES FOR CONCURRENCY CONTROL

lock: status for operation to be applied

high overhead

Binary lock

too restrictive for DB items

- **LOCKED**
  - item cant be accessed
  - lock = 1
- **UNLOCKED**
  - item can be accessed
  - lock = 0

shared/exclusive locks

aka read/write locks

1. read_lock()
   - shared
   - many T can read
2. write_lock()
   - exclusive
   - only 1 T can write
3. unlock()

- Transaction requests access by issuing a lock_item(X) operation

```
lock_item(X):
B:  if LOCK(X) = 0              (*item is unlocked*)
        then LOCK(X) ←1         (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 21.1 Lock and unlock operations for binary locks

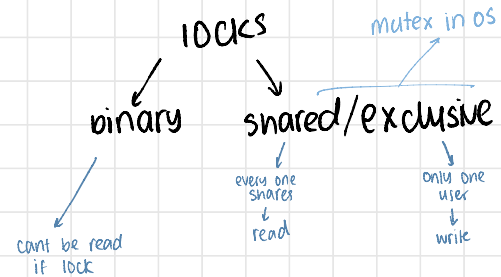
```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

lock Table

- specifies items that have locks

lock manager

- tracks and controls access to locks
- rules enforced by lock manager



```
read_lock(x):
    B:
        If  lock(x) == 'unlock'
            lock(x) ← 'read_lock"
            no_of_locks(x) ← 1
        else if  lock(x) == "read lock"
            no_of_reads(x) = no_of_reads(x) + 1
        else Begin

            ask   req_to_use(x)
            If   ASK == 'NO'
                wait ( until  lock(x) == "unlock" )
                  //
                go  to  B
    END

write_lock(x)
    B:
        If  lock(x) == "unlocked"
            lock(x) ← "write-locked"
        else Begin
            wait(until  lock(x) == 'unlocked')
            GO TO B
    END:

unlock(x):
    If  lock(x) == 'write-lock"
        lock(x) ← 'unlocked'
    else If  lock(x) = "read-lock"
        no_of_reads(x) = no_of_reads(x) - 1
        If  no_of_reads(x) = 0
            lock(x) ← 'unlocked"
```

lock conversion

↳ a T already holding a lock can convert lock to another state

upgrade

read_lock(x) → write_lock(x)

downgrade

write_lock(x) → read_lock(x)

Two faced locking Protocol (TPL)

↳ guarantees serializibility
↳ it limits the amount of concurrency

↳ once T releases its first lock (unlock), it can no longer aquire new locks
↳ There are 2 Phases

Growing Phase (Phase 1) — aka expanding phase

↳ new locks can be aquired
↳ no locks can be released
↳ lock conversion upgrades
↳ lock

Shrinking Phase (Phase 2)

↳ no new locks can be aquired
↳ locks can be released
↳ lock conversion downgrades
↳ unlock

* once gone shrink, cant go back to growing phase

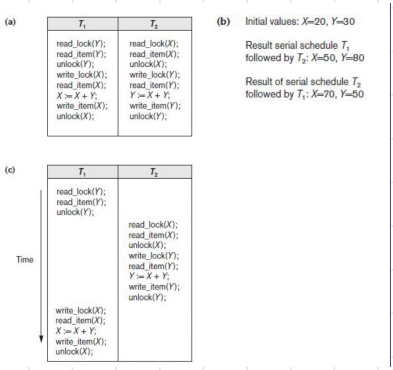


Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions T1 and T2 (b) Results of possible serial schedules of T1 and T2 (c) A nonserializable schedule S that uses locks

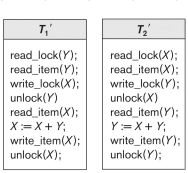


Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

variations of Two Phased locking

Basic 2PL
↳ as done above

Conservative 2PL
aka static

↳ T locks all items to be acessed before the T begins

↳ no deadlocks → in shrinking phase when it starts

STrict 2PL
↳ until T commits/aborts

T dosent release exclusive locks

write

hence no other T can read/write an item that is written by T unless T is committed

Rigious 2PL
↳ until T commits/aborts

T dosent release any locks → in expanding phase till it ends

read, write

concurrency control system
↳ generates read_lock, write_lock reavests

Deadlock
↳ set of locked T, each holding a data item and waiting to aquire a data item held by another T

↳ process stuck in circular waiting for the resources

Starvation
↳ when a T waits because it requires a data item to run, that is never allocated to this T

↳

deadlock implies starvation BUT starvation does not impy deadlock

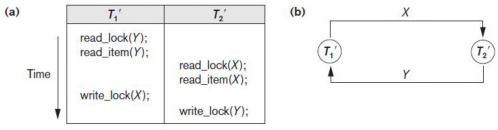


Figure 21.5 Illustrating the deadlock problem (a) A partial schedule of *T1'* and *T2'* that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

Deadlock Prevention Protocols

1
- ↳ conservative 2PL
- ↳ lock all data items it needs in advance

2
- ↳ Order all items in DB
- ↳ T that needs several items will lock them in that order

both limit concurrency

Time stamp Protocols
Ty = younger T To = older T

wait die
- ↳ if Ty wants to acess item locked by To
  Ty is aborted and restarted with same priority
- ↳ if To wants to acess item locked by Ty
  To will have to wait

wound wait
- ↳ if Ty wants to acess item locked by To
  Ty is has to wait
- ↳ if To wants to acess item locked by Ty
  To will have acess

avoids starvation

NO waiting algo (NW)
- ↳ if T cant obtain a lock
  it is aborted and restarted later
  without checking if deadlock occurs or not

needlessly aborts, restarts T

solution

Cautious waiting algo (CW)
- ↳ If T_1 is not blocked (waiting for a data item)
  then T_2 blocked and allowed to wait
- ↳ If T_1 is blocked
  abort T_1

Deadlock Detection
- ↳ system checks if deadlock state exists
  by making a wait for graph

Victim selection: which T to abort due to deadlock

Timeouts: if a T waits longer than a predefined time, abort the T

Starvation: when T cant proceed for a indefinite time, while other continue normally
- ↳ solution: first come first serve queue

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Timestamp
  - Unique identifier assigned by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction start time
- Concurrency control techniques based on timestamps do not use locks
  - Deadlocks cannot occur

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock
- General approach
  - Enforce equivalent serial order on the transactions based on their timestamps

- Timestamp ordering (TO)
  - Allows interleaving of transaction operations
  - Must ensure timestamp order is followed for each pair of conflicting operations
- Each database item assigned two timestamp values
  - read_TS(X)
  - write_TS(X)

- Basic TO algorithm
  - If conflicting operations detected, later operation rejected by aborting transaction that issued it
  - Schedules produced guaranteed to be conflict serializable
  - Starvation may occur
- Strict TO algorithm
  - Ensures schedules are both strict and conflict serializable

- Thomas's write rule
  - Modification of basic TO algorithm
  - Does not enforce conflict serializability
  - Rejects fewer write operations by modifying checks for write_item(X) operation

DATABAS RECOVERY TECHNIQUES CHP 22

- Recovery process restores database to most recent consistent state before time of failure
- Information kept in system log

Typical recovery strategies

1. Restore backed up copy of DB → when extensive damage
2. Identify any changes that may cause inconsistency → when noncatasphoric failure
   ↳ some operations may require redo

3. Deffered Update Technique

↳ when T commits
then physically update DB
↳ no undo needed
↳ redo might be needed from log
↳ no undo/redo algo
no steal

- Can only be used for short transactions and transactions that change few items
  - Buffer space an issue with longer transactions

Transaction does not reach its commit point until all its REDO-type log entries are recorded in log and log buffer is force-written to disk

4. Immediate Update Technique

↳ DB might be updated by some operations before T commits
↳ operations recorded in log
↳ then operation changes applied on DB
↳ undo / redo algo
steal / no force strategy

if it crashes after recording some changes undo operation done in db from log

Undo and redo operations

↳ idempotent → executing operation multiple times = executing just once
↳ entire recovery process should be idempotent

Caching (buffering) of disk blocks

DBMS cache: a collection of in-memory buffers
cache directory: tracks which DB items are in buffers
flush: cache buffers flushed to make space for new items
if dirty bit = 1 → modified buffer
↳ contents written back to disk before flush
If Pin-unpin bit = 1 → page cant be written back to disk yet

Main strategies when flushing a modified buffer back to disk

→ dirty bit = 1

Inplace updating

↳ writes in same DB disk location

↳ overwrites the changed values

Shadowing

↳ writes in diff disk location

to maintain multiple versions of data items

log for recovery in order to undo/redo

- Before-image: old value of data item → BFIM
- After-image: new value of data item → AFIM

Write ahead logging

↳ ensure before image is recorded in log and

↳ log entry is flushed to disk before AFIM overwrites

undo type log entries → old value → BFIM → before flush image memory

redo type log entries → new value → AFIM → after flush image memory

Steal/no steal, Force/no force

↳ rules for when a page from DB cache can be written to disk

NO steal approach

↳ T commits, only then

↳ write updated buffer to disk

Steal approach

↳ write updated buffer to disk

↳ when T commits

NO Force approach

↳ T commits, only then

↳ write all updated buffer to disk immediately

Force approach

↳ write all updated buffer to disk immediately

↳ when T commits

↳ eliminates need for redo

- Typical database systems employ a steal/no-force strategy
  - Avoids need for very large buffer space → steal
  - Reduces disk I/O operations for heavily updated pages → no force

- Write-ahead logging protocol for recovery algorithm requiring both UNDO and REDO
  - BFIM of an item cannot be overwritten by its after image until all UNDO-type log entries have been force-written to disk
  - Commit operation of a transaction cannot be completed until all REDO-type and UNDO-type log records for that transaction have been force-written to disk

check Point

is another type of entry in log

- Taking a checkpoint
  - Suspend execution of all transactions temporarily
  - Force-write all main memory buffers that have been modified to disk
  - Write a checkpoint record to the log, and force-write the log to the disk
  - Resume executing transactions
- DBMS recovery manager decides on checkpoint interval

time needed to force write may delay T

solution

- Fuzzy checkpointing
  - System can resume transaction processing after a begin_checkpoint record is written to the log
  - Previous checkpoint record maintained until end_checkpoint record is written

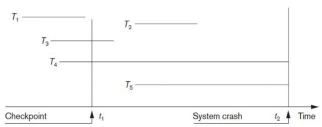


Figure 22.2 An example of a recovery timeline to illustrate the effect of checkpointing

Slide 22-20

Transaction Rollback

- Transaction failure after update but before commit
  - Necessary to roll back the transaction
  - Old data values restored using undo-type log entries
- Cascading rollback
  - If transaction T is rolled back, any transaction S that has read value of item written by T must also be rolled back
  - Almost all recovery mechanisms designed to avoid this

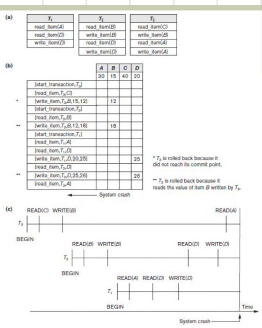
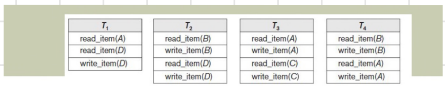


Figure 22.1 Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules) (a) The read and write operations of three transactions (b) System log at point of crash (c) Operations before the crash

Slide 22- 1

Transactions that Do Not Affect the Database

- Example actions: generating and printing messages and reports
- If transaction fails before completion, may not want user to get these reports
  - Reports should be generated only after transaction reaches commit point
- Commands that generate reports issued as batch jobs executed only after transaction reaches commit point
  - Batch jobs canceled if transaction fails



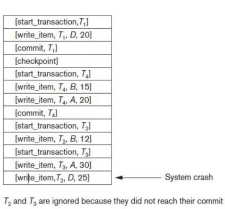


Figure 22.3 An example of recovery using deferred update with concurrent transactions (a) The READ and WRITE operations of four transactions (b) System log at the point of crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

ishma hafeez notes