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

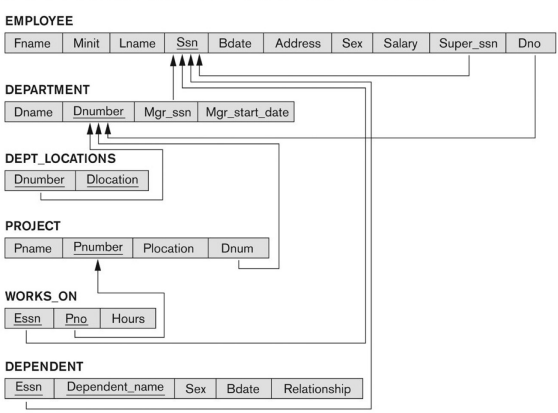
SUM, MAX, MIN, AVG, COUNT

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

a) 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

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

   $finalresult \leftarrow \pi_{fname}(DEP5\_EMP)$

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

↳ Combine R and S

↳ 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
  - $R \cup S = S \cup R$, and $R \cap S = S \cap R$
- Both union and intersection can be treated as n-ary operations applicable to any number of relations as both are *associative* operations; that is
  - $R \cup (S \cup T) = (R \cup S) \cup T$
  - $(R \cap S) \cap T = R \cap (S \cap T)$
- The minus operation is not commutative; that is, in general
  - $R - S \neq S - R$

2. Intersection (∩) $R \cap S$

↳ 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

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):
  - $FEMALE\_EMPS \leftarrow \sigma_{SEX='F'}(EMPLOYEE)$
  - $EMPNAMES \leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE\_EMPS)$
  - $EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$
- 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):
  - $FEMALE\_EMPS \leftarrow \sigma_{SEX='F'}(EMPLOYEE)$
  - $EMPNAMES \leftarrow \pi_{FNAME, LNAME, SSN}(FEMALE\_EMPS)$
  - $EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$
  - $ACTUAL\_DEPS \leftarrow \sigma_{SSN=ESSN}(EMP\_DEPENDENTS)$
  - $RESULT \leftarrow \pi_{FNAME, LNAME, DEPENDENT\_NAME}(ACTUAL\_DEPS)$
- 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 ⋈ R ⋈<join condition> S

- Example: Suppose that we want to retrieve the 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 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)

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

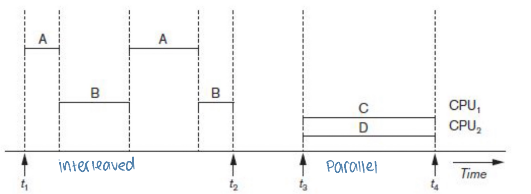


Figure 20.1 Interleaved processing versus parallel processing of concurrent transactions

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

(a)

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X); (lost)	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Time ↓

← Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

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

2. Temporary Update / Dirty Read Problem → reading uncommited data

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

↳ Meanwile updated transaction is read by T_2 before its changed back to its og value

roll back

(b)

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	

Time ↓

← Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

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

(c)

T_1	T_3
	$sum := 0$;
	read_item(A);
	$sum := sum + A$;
	⋮
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

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

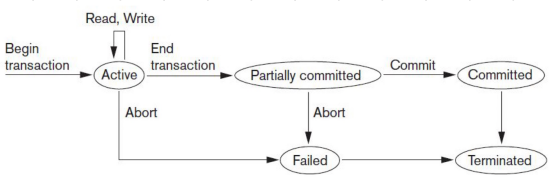


Figure 20.4 State transition diagram illustrating the states for transaction execution

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 (DBMS cache divided into domains) handles one type of disk pages
    - Index pages
    - Data file pages
    - Log file pages
  - Number of available buffers for each domain is predetermined
- Hot set method → page replacement also
  - Useful in queries that scan a set of pages → join operation in nested loops repeatedly
  - Does not replace the set (disk pages that will be accesses repeatedly) in the buffers until processing is completed
- 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
    - In addition to level 2 properties
  - Snapshot isolation → another type of isolation

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

↳ This is time consuming

SOLUTION

Casadless schedules → recoverable

↳ every T only reads items written by commited T

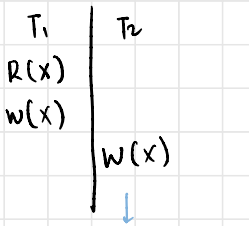
↳ read needs to be commited

Strict schedules → recoverable, cascadeless by default

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

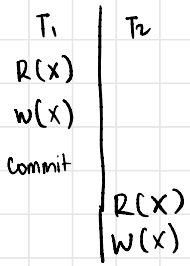
↳ read, write commited

↳ simpler recovery process → restore the before image



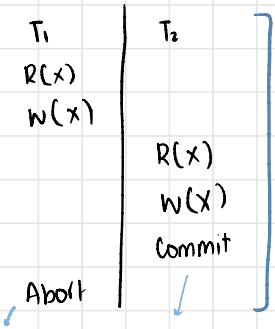
cascadless → no read in T_2

not strict → no commit



cascadeless

strict



Recoverable

Problem: Dirty read

1. commit
2. Abort

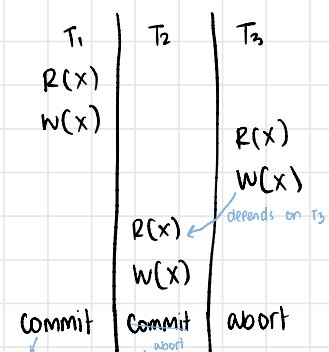
2 possibilities

↳ 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

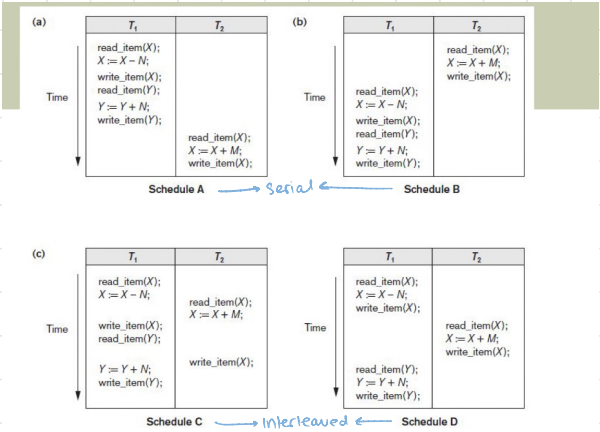


Figure 20.5 Examples of serial and nonserial schedules involving transactions *T*1 and *T*2 (a) Serial schedule A: *T*1 followed by *T*2 (b) Serial schedule B: *T*2 followed by *T*1 (c) Two nonserial schedules C and D with interleaving of operations

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 make same transation changes as serial

↳ 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 nodes of all transactions

↳ check for conflicting

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

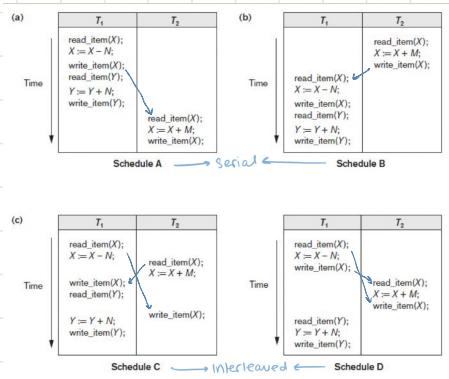
↳ $R_1(X)$, $W_2(X)$ $T_1 \rightarrow T_2$

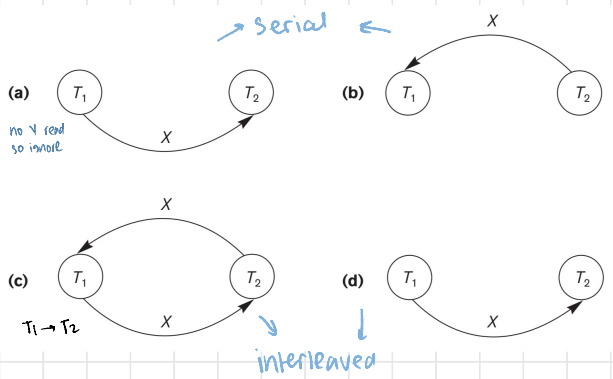
↳ $W_1(X)$, $W_2(X)$ $T_1 \rightarrow T_2$

* min incoming edges runs first

cycle → not equivalent to serial

no cycle → equivalent to serial





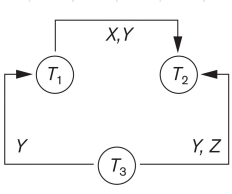
(c)

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

Time

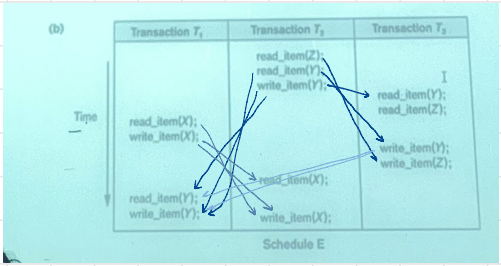
Schedule F

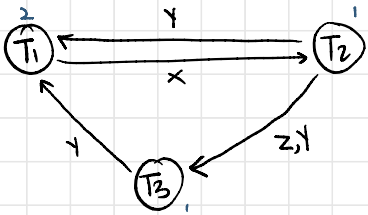
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$





$T_3 \rightarrow T_2 \rightarrow T_1$

OR

$T_2 \rightarrow T_3 \rightarrow T_1$

(b)

Time

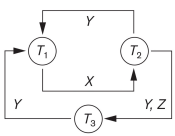
Transaction T_1	Transaction T_2	Transaction T_3

	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);	

Schedule E

(d)



Equivalent serial schedules

None

Reason

Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$
Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

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

View Resizability schedule

↳ view equivalent to a serial schedule

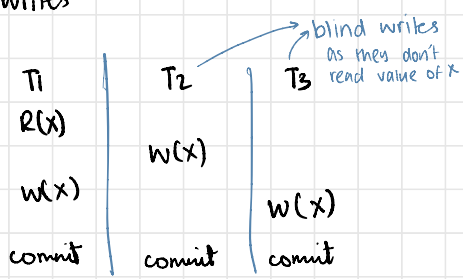
conflict Resizability

(by default view resizable)

↳ blind writes allowed
↳ write without a preceding read operation

constrained write assumption

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



Unconstrained write assumption

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

View Resizable
not conflict Resizable → as no conflict equivalance

?

OTHER TYPES OF EQUIVALANCE SCHEDULES

Debit card transactions

↳ can use non serilizable schedules → as order doesn't 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

again, T_1 will see a different value.

3. **Phantoms.** A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T_2 inserts a new row r that also satisfies the WHERE-clause condition used in T_1, into the table used by T_1. The record r is called a **phantom record** because it was not there when T_1 starts but is there when T_1 ends. T_1 may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is T_1 followed by T_2, then the record r should not be seen; but if it is T_2 followed by T_1,then the phantom record should be in the result given to T_1. If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

- 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

- Snapshot isolation
  - Used in some commercial DBMSs
  - Transaction sees data items that it reads based on the committed values of the items in the database snapshot when transaction starts
  - Ensures phantom record problem will not occur

CHP 21

Concurrency control protocols
↳ set of rules to guarantee serializability

uses →

Timestamp
↳ unique identifier for each T

Two Phase locking protocol
↳ 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

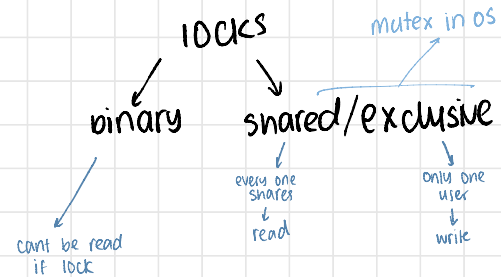
```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and 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



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

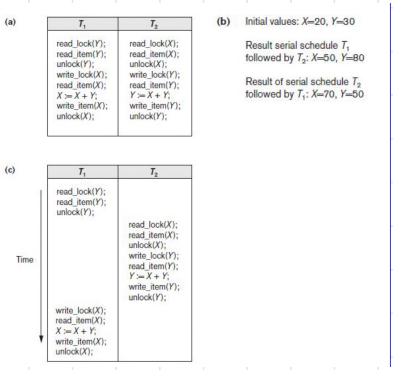


Figure 21.3 Transactions that do not obey two-phase locking (a) Two transactions T1 and T2 (b) Results of possible serial schedules of T1 and T2 (c) A nonserializable schedule S that uses locks

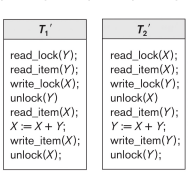


Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

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)

Rigious 2PL
↳ until T commits/aborts

T dosent release any locks (read, write) → in expanding phase till it ends

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

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

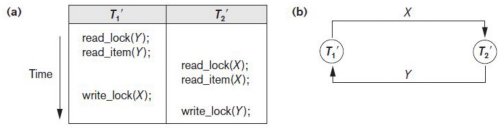


Figure 21.5 Illustrating the deadlock problem (a) A partial schedule of *T1'* and *T2'* that is in a state of deadlock (b) A wait-for graph for the partial schedule in (a)

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

avoids starvation

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

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

4. Immediate Update Technique

- ↳ DB might be updated by some operations before T commits
- ↳ operations recorded in log
- ↳ undo/redo algo

steal / no force strategy

- 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

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)
	write_item(D)	write_item(C)	write_item(A)

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

[start_transaction, T_1]
[write_item, T_1, D, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, B, 15]
[write_item, T_4, A, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, B, 12]
[start_transaction, T_3]
[write_item, T_3, A, 30]
[write_item, T_2, D, 25] ← System 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.

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 disc 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 do disk befor AFIM overwrites
undo type log enteries → old value → BFIM
redo type log enteries → new value → AFIM

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 updated buffer to disk

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

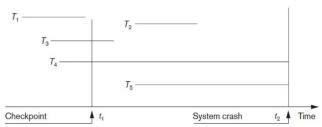


Figure 22.2 An example of a recovery timeline to illustrate the effect of checkpointing

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

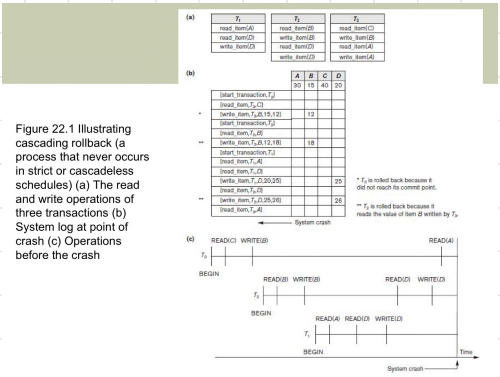


Figure 22.1 Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules) (a) The read and write operations of three transactions (b) System log at point of crash (c) Operations before the crash

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

ishma hafeez notes