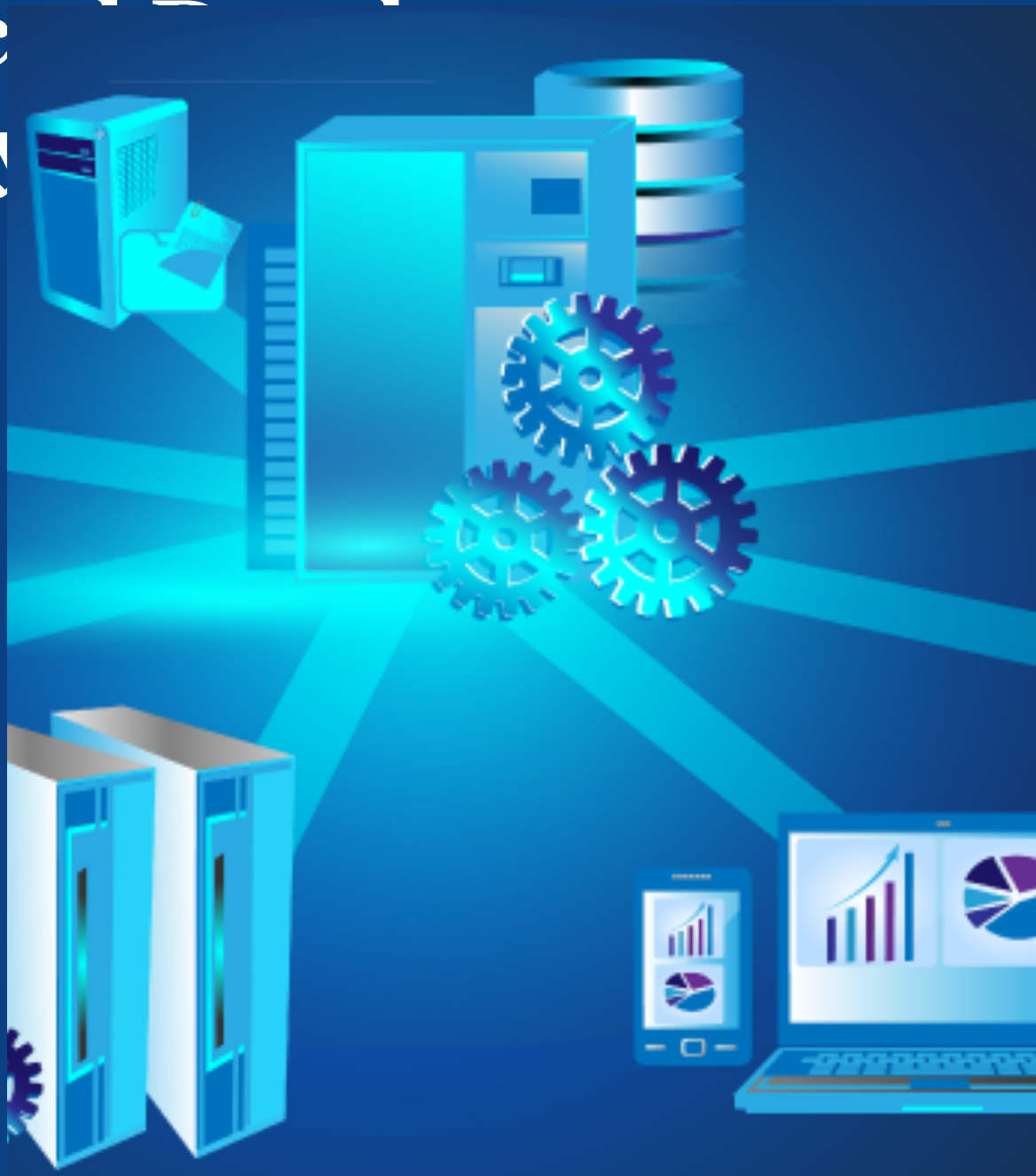
COMP90050: Advanc… Sy…

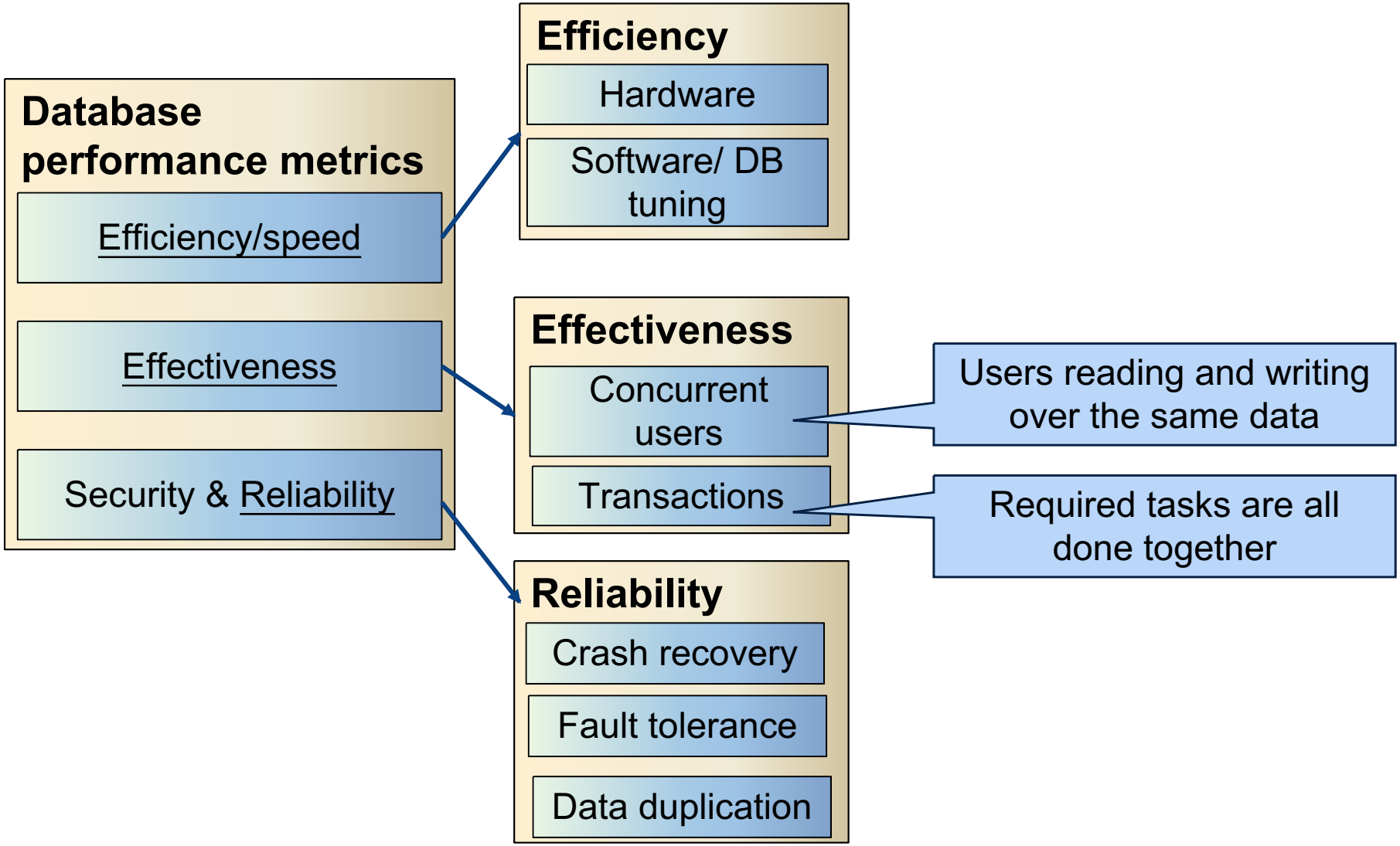


Lecturer: Farhana Choudhury (PhD)

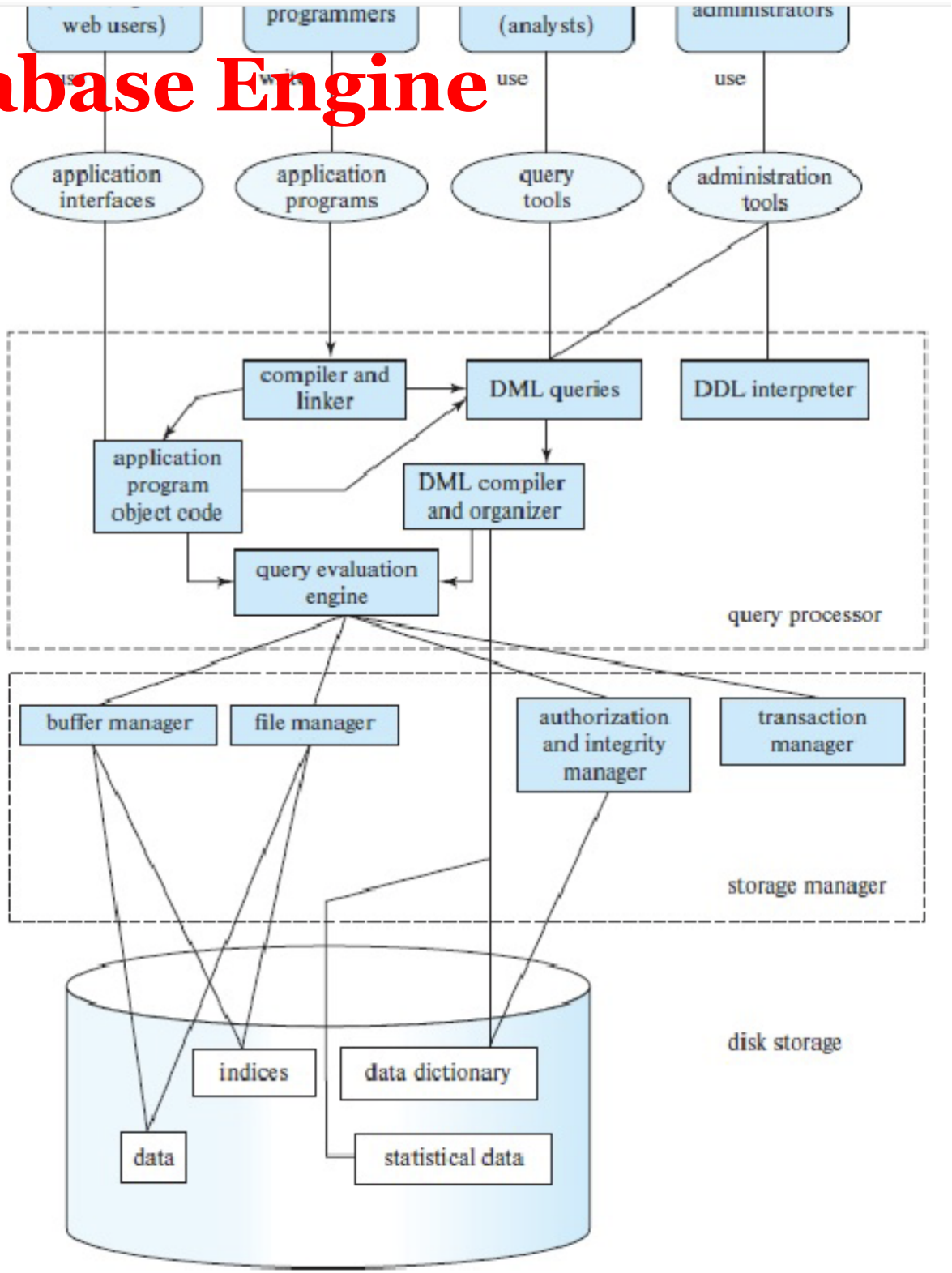
Transaction concepts and TPM

Week 4

Core Concepts of Database management system



Database Engine



Topics

- Transaction concepts
- Different types of transactions
- Concurrent use of data and its issues
- Isolation concepts - ensuring that concurrent transactions leaves the DB in the same state as if they were executed separately
- How to achieve isolation while keeping good efficiency

Database Transactions

Transaction - A unit of work in a database

- A transaction can have any number and type of operations in it
- Either happens as a whole or not
- Transactions ideally have four properties, commonly known as ACID properties

Transaction models

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Atomicity - All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

Example – A transaction that (i) subtracts $100 if balance >100 (ii) deposits $100 to another account ☆(若第二项 不发生，第一项发生，钱就没了)

(both actions with either happen together or none will happen) ☆ ☆

why bank prefer to ~~Transfer~~ ACID.

Transaction models

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Consistency - Data is in a 'consistent' state when a transaction starts and when it ends – in other words, any data written to the database must be valid according to all defined rules (e.g., no duplicate student ID, no negative fund transfer, etc.)

- What is 'consistent' - depends on the application and context constraints
- It is not easily computable in general
- Only restricted type of consistency can be guaranteed, e.g. serializable transactions which will be discussed later.

Transaction models ...

ACID (Atomicity, Consistency, Isolation, Durability) properties:

Isolation- transaction are executed as if it is the only one in the system.

- For example, in an application that transfers funds from one account to another, the isolation ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability- the system should tolerate system failures and any committed updates should not be lost.

Transaction models (cont.)

Types of Actions

- **Unprotected actions** - no ACID property
- **Protected actions -** these actions are not externalised before they are completely done. These actions are controlled and can be rolled back if required. These have ACID property.
- **Real actions -** these are real physical actions once performed cannot be undone. In many situations, atomicity is not possible with real actions (e.g., firing two rockets as a single atomic action)

Let’s look at a simple example that will be used to understand different types of transactions …

Embedded SQL example in C

(Open Database Connectivity)

```
int main()
{       exec sql INCLUDE SQLCA; /*SQL Communication Area*/
        exec sql BEGIN DECLARE SECTION;
          /* The following variables are used for communicating
                between SQL and C */
          int OrderID; /* Employee ID (from user) */
          int CustID; /* Retrieved customer ID */
          char SalesPerson[10] /* Retrieved salesperson name */
          char Status[6] /* Retrieved order status */
        exec sql  END DECLARE SECTION;
/* Set up error processing */
        exec sql WHENEVER SQLERROR GOTO query_error;
        exec sql WHENEVER NOT FOUND GOTO bad_number;
```

```
 /* Prompt the user for order number */
            printf ("Enter order number: ");
            scanf_s("%d", &OrderID);
/* Execute the SQL query */
            exec sql SELECT CustID, SalesPerson, Status
                 FROM Orders
          WHERE OrderID = :OrderID // ":" indicates to refer to  C variable
          INTO :CustID, :SalesPerson, :Status;
/* Display the results */
            printf ("Customer number: %d\n", CustID);
            printf ("Salesperson: %s\n", SalesPerson);
            printf ("Status: %s\n", Status);
       exit();
 query_error:
             printf ("SQL error: %ld\n", sqlca->sqlcode); exit();
 bad_number:
             printf ("Invalid order number.\n"); exit(); }
```

Some concepts…

Host Variables

Declared in a section enclosed by the BEGIN DECLARE SECTION and END DECLARE SECTION. While accessing these variables, they are prefixed by a colon ":". The colon is essential to distinguish between host variables and database objects (for example tables and columns).

Data Types

The data types supported by a DBMS and a host language can be quite different. Host variables play a dual role:

- Host variables are program variables, declared and manipulated by host language statements, and
- they are used in embedded SQL to retrieve database data.

If there is no host language type corresponding to a DBMS data type, DBMS automatically converts the data. So, the host variable types must be chosen carefully.

Error Handling

The DBMS reports run-time errors to the applications program through an SQL Communications Area (SQLCA) by INCLUDE SQLCA. The WHENEVER...GOTO statement tells the pre-processor to generate error-handling code to process errors returned by the DBMS.

Singleton SELECT

The statement used to return the data is a singleton SELECT statement; that is, *it returns only a single row of data.* Therefore, the code example does not declare or use *cursors*.

Reference:

http://msdn.microsoft.com/enus/library/ms714570(VS.85).aspx

Flat Transaction

```
exec sql CREATE Table accounts (
   AccId                    NUMERIC(9),
   BranchId        NUMERIC(9), FOREIGN KEY REFERENCES branches,
   AccBalance    NUMERIC(10),
   PRIMARY KEY(AccId));
**********************************
```

Everything inside BEGIN WORK and COMMIT WORK is at the same level; that is, the transaction will either survive together with everything else (commit), or it will be rolled back with everything else (abort)

everything survive together or rolled back ~~together~~ together.

Flat Transaction ...

```
exec sql BEGIN DECLARE SECTION;
        long AccId, BranchId, TellerId, delta, AccBalance;
exec sql END DELCLARATION;


/* Debit/Credit Transaction*/
DCApplication()
{read input msg;
exec sql BEGIN WORK;
AccBalance = DodebitCredit(BranchId, TellerId, AccId, delta);
send output msg;
exec sql COMMIT WORK;
}
```


```
/* Withdraw money -- bank debits;  Deposit money – bank credits */
Long DoDebitCredit(long BranchId,
            long TellerId, long AccId, long AccBalance, long delta){
   exec sql UPDATE accounts
   SET AccBalance =AccBalance + :delta
   WHERE AccId = :AccId;
   exec sql SELECT AccBalance INTO  :AccBalance
   FROM accounts WHERE AccId = :AccId;

   exec sql UPDATE tellers
   SET TellerBalance = TellerBalance + :delta
   WHERE TellerId = :TellerId;

   exec sql UPDATE branches
   SET BranchBalance = BranchBalance + :delta
   WHERE BranchId = :BranchId;

   Exec sql INSERT INTO history(TellerId, BranchId, AccId, delta, time)
   VALUES( :TellerId, :BranchId, :AccId, :delta, CURRENT);
return(AccBalance);
}
```

Let’s include a check on the account balance and refusing any debit that overdraws the account.

```
…
DCApplication(){
        read input msg;
                exec sql BEGIN WORK;
                AccBalance = DodebitCredit(BranchId, TellerId, AccId, delta);
                if (AccBalance < 0 && delta < 0){
                        exec sql ROLLBACK WORK;
                }
                else{
                        send output msg;
                        exec sql COMMIT WORK;
                }
}
```

Limitations of Flat Transactions

Flat transactions do not model many real applications.

E.g. airline booking

BEGIN WORK

S1: book flight from Melbourne to Singapore

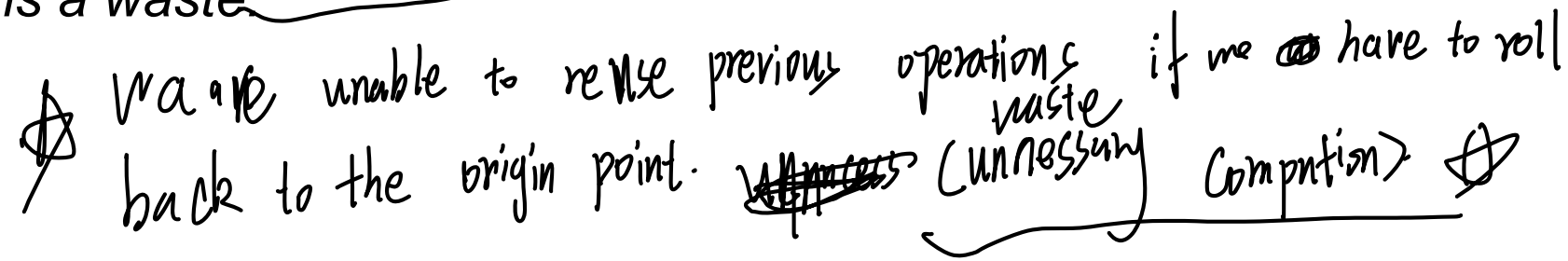
S2: book flight from Singapore to London

S3: book flight from London to Dublin

END WORK

Problem: from Dublin if we cannot reach our final destination instead we wish to fly to Paris from Singapore and then reach our final destination.

If we roll back we need to re do the booking from Melbourne to Singapore *which is a waste*

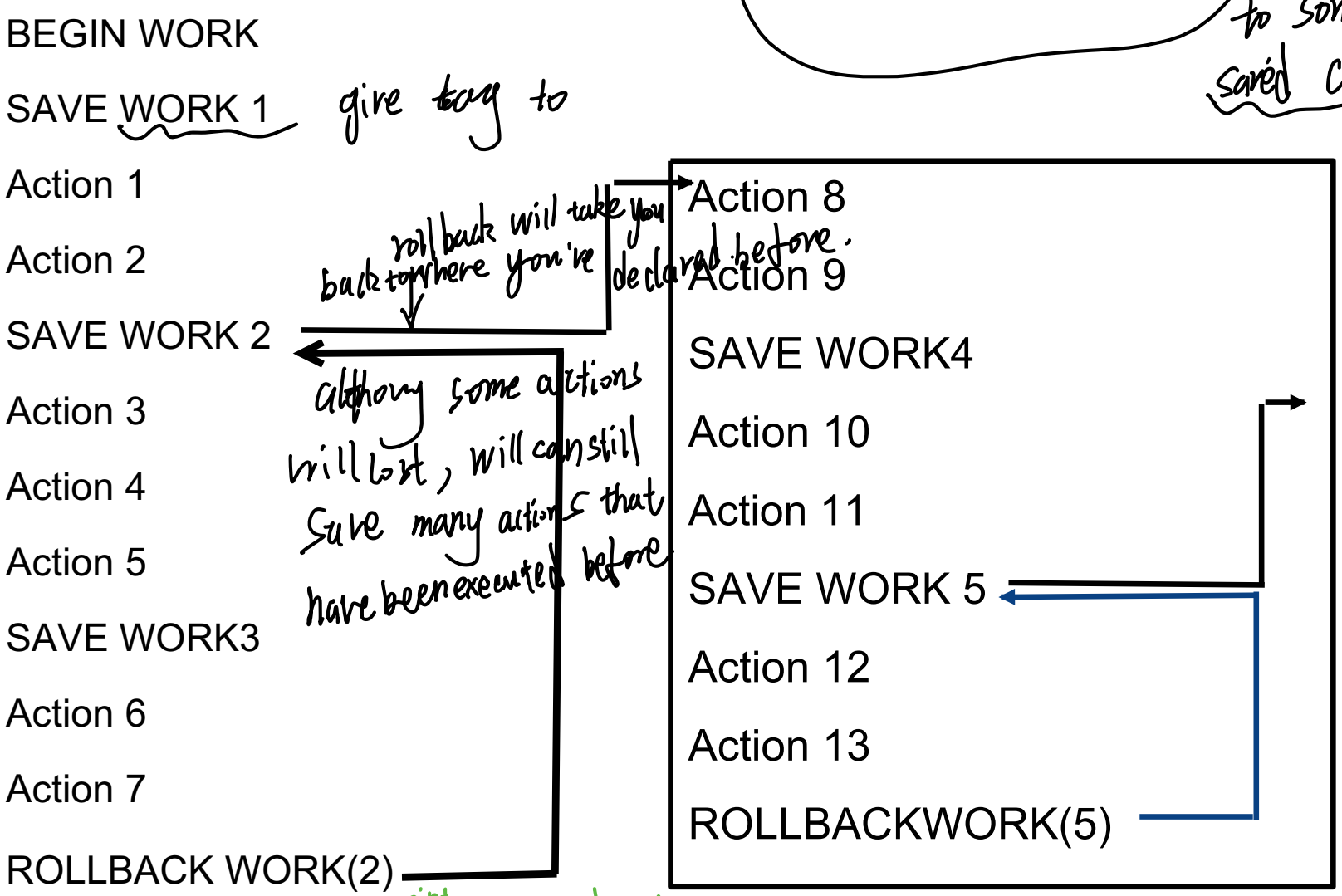


Limitations of Flat Transactions ...

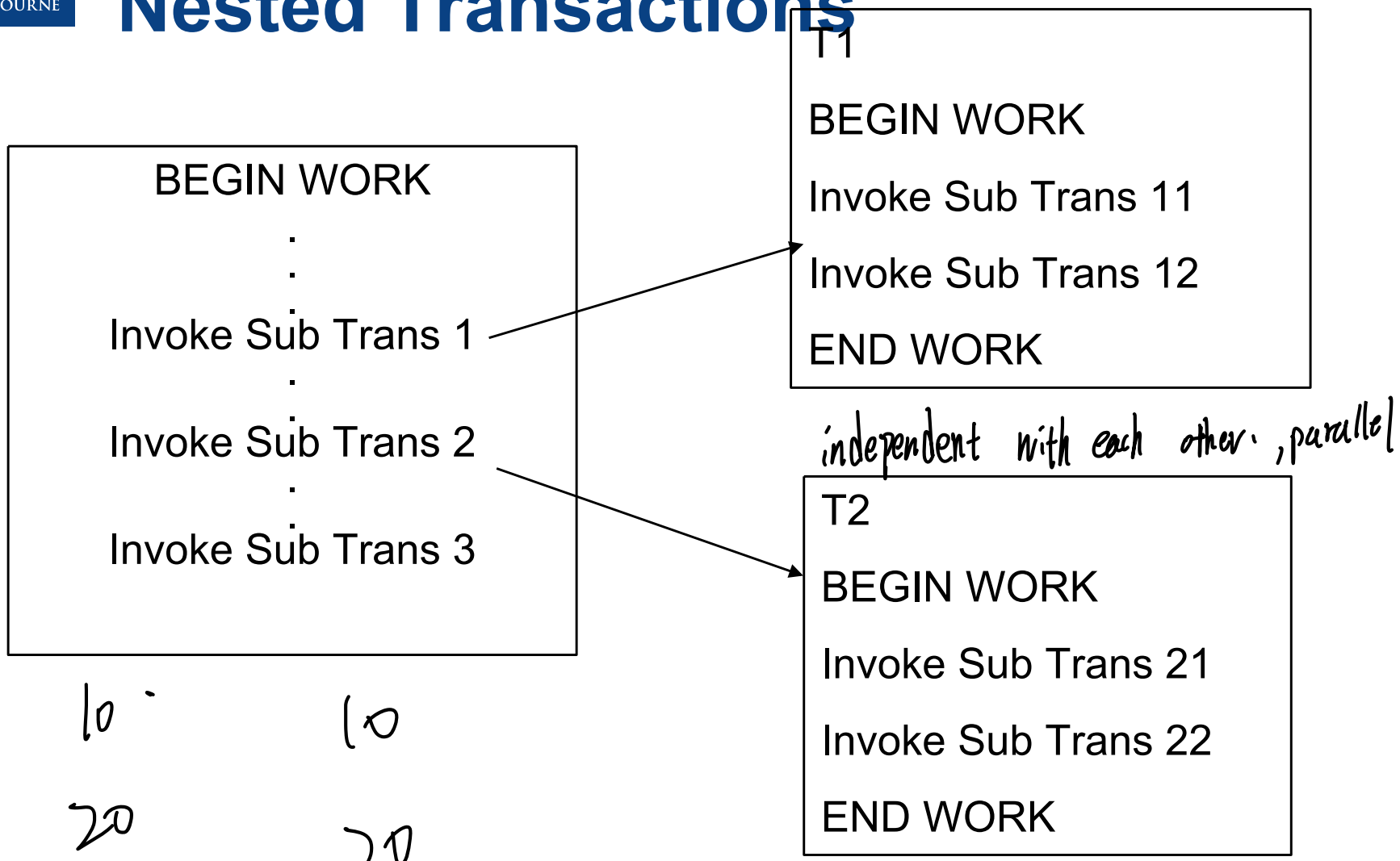
```
IncreaseSalary()
{       real percentRaise;
        receive(percentRaise);
        exec SQL BEGIN WORK;
                exec SQL UPDATE employee
                set salary = salary*(1+ :percentRaise)
                send(done);
        exec sql COMMIT WORK;
        return
}
```

This can be a very long running transaction. ***Any failure of transaction requires lot of unnecessary computation.***

Flat transaction with save points



Nested Transactions



10 10

20 20

25 25

30 30

Nested Transaction Rules

Commit rule

- A subtransaction can either commit or abort, however, **commit cannot take place unless the parent itself commits.**

- Subtransactions have A, C, and I properties but not D property unless all its ancestors commit.

- Commit of a sub transaction makes its results available only to its parents.

Roll back Rules

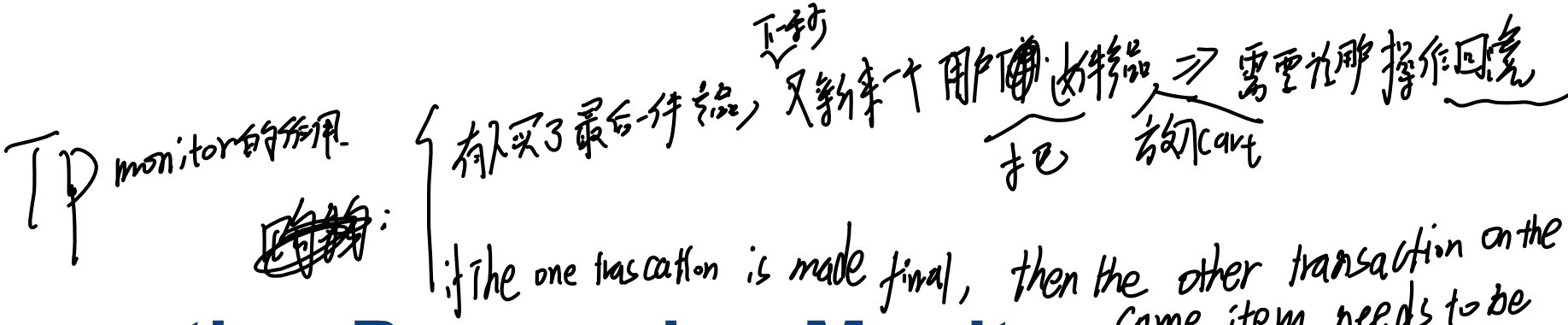
If a subtransaction rolls back, all its children are forced to roll back.

Visibility Rules

Changes made by a subtransaction are visible to the parent only when the subtransaction commits. All objects of parent are visible to its children. Implication of this is that the **parent should not modify objects while children are accessing them.** This is not a problem as parent does not run in parallel with its children.

Reading – Open-nested transactions

TP monitor的作用：



Transaction Processing Monitor

The main function of a TP monitor is to *integrate* other system components and manage resources.

- TP monitors manage the transfer of data between clients and servers
- breaks down applications or code into transactions and ensures that all databases are updated properly
- It also takes appropriate actions if any error occurs

TP monitor services

- **Heterogeneity**: If the application needs access to different DB systems, local ACID properties of individual DB systems is not sufficient. Local TP monitor needs to interact with other TP monitors to ensure the overall ACID property. A form of 2 phase commit protocol must be employed for this purpose (will be discussed later).
- **Control communication**: If the application communicates with other remote processes, the local TP monitor should maintain the communication status among the processes to be able to recover from a crash.

TP Services ...

Terminal management: Since many terminals run client software, the TP monitor should provide appropriate ACID property between the client and the server processes.

Presentation service: this is similar to terminal management in the sense it has to deal with different presentation (user interface) software -- e.g. X-windows

Context management: E.g. maintaining the sessions etc.

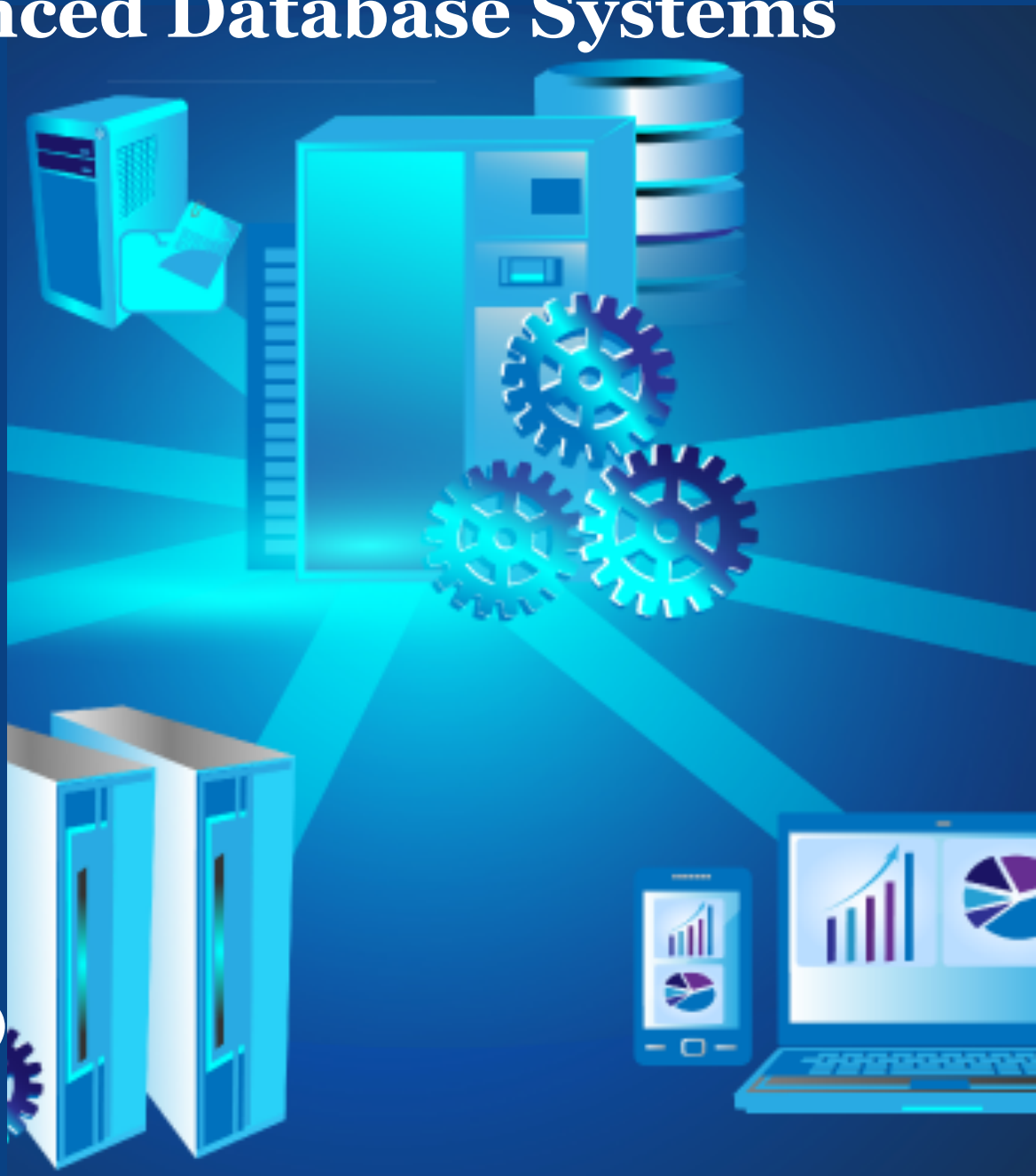
Start/Restart: There is no difference between start and restart in TP based system.

☆ session 管理、维护☆

☆ e.g: 在我的session → 找个 student records → if make a request for changing just one particular students record ⇒ so the ~~controller~~ send ok just a single retriever, 这个 update 可以被 easily efficient made ⇒ without the needing of refreshing whole tuple. row

THE UNIVERSITY OF MELBOURNE

COMP90050 Advanced Database Systems



Semester 2, 2024

Lecturer: Farhana Choudhury (PhD)

Live lecture – Week 4

Continuation from last week's live lecture

Effect of indexes – do they always improve performance?

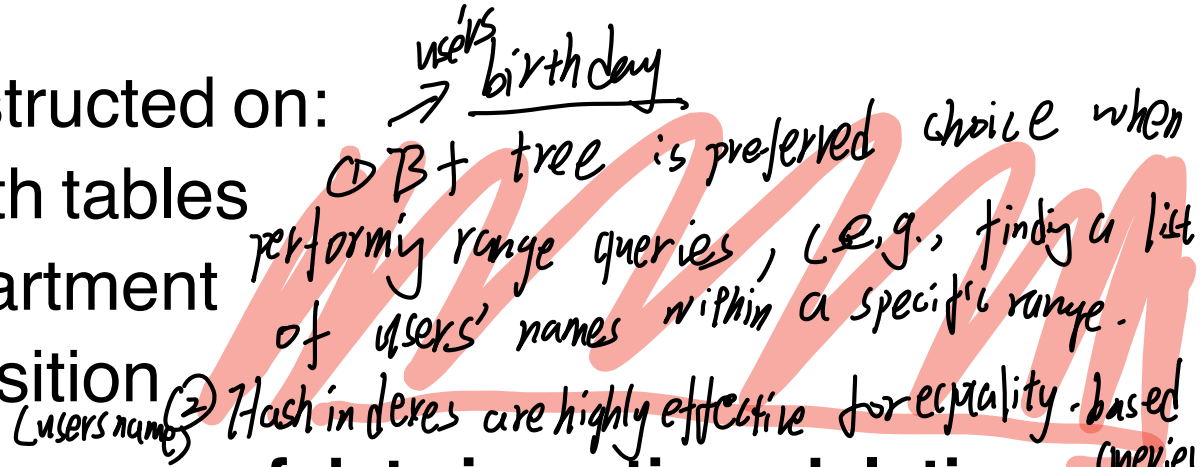
Group activity - https://tinyurl.com/49asf2rc

ID	Name	Department
1	Jane	Comp. Sci
2	John	Biology

ID	Position	Salary
1	Lecturer	75,000
1	Research assistant	40,000
2	Senior lecturer	82,000

There are indexes constructed on:

- B+tree on IDs of both tables
- Hash index on Department
- Bitmap index on position



What will be the performance of data insertion, deletion, and updates, compared to no indexing (or just on IDs)

① B+ tree on IDs of both tables ② Hash index on department ③ bitmap index on position.

Query 1: Inserting records for a new lecturer (不用 V.S 用)

Inserting records means that we need to update those indexes for Id: 用 B+tree 的优劣对比

容易 straightforward (有 empty space 的情况), 但有时候要 restructure/reorganize the index. 很快.

不止① (别的 index), 其他 index 也一样, 也会被更新 (index 本身 updated as well)

∴ all four indexes need to be updated ⇒ because those corresponding info needs to be inserted ⇒ it is extra overhead

But without using index ⇒ we just insert directly into the table row no extra index overhead.

(☆No index faster☆)

Query 2: updating the salary of specific instructor with ID 2 用 VS 不用:

⇒ we just need to go to that 'id' position in the right table ⇒ we don't have to go to other table ⇒ just going to the table for a particular ID ⇒ simply use B+tree on the ID much much faster than non-index. if we don't have B+tree we have to go through those all records even if the Ids were sorted ⇒ we still have to do binary search (more time-consuming)

Query 3: Find the total number of research assistants 用 V.S 不用

用 (there just 1 and 0 to match with corresponding columns value ⇒ finding the RA is just equivalent to count the number of ones on a particular position on a bitmap ⇒ very fast.. (bit sequence)

Query 4: Finding the name of all instructors who are in 'Biology' department. 用 V.S 不用

Hash index will just go to bucket ⇒ Then we can quickly find the instructor in Biology department (in the same bucket) we don't have to go through other instructors who work in computer science or ECE. (faster to focus on exactly those records that meet this criteria. quickly filter based on the department name)

Query 5: Find: person for which the salary is greater than 30000 · 用区不用.
at least one
和any join 后用right table, 不用ID.

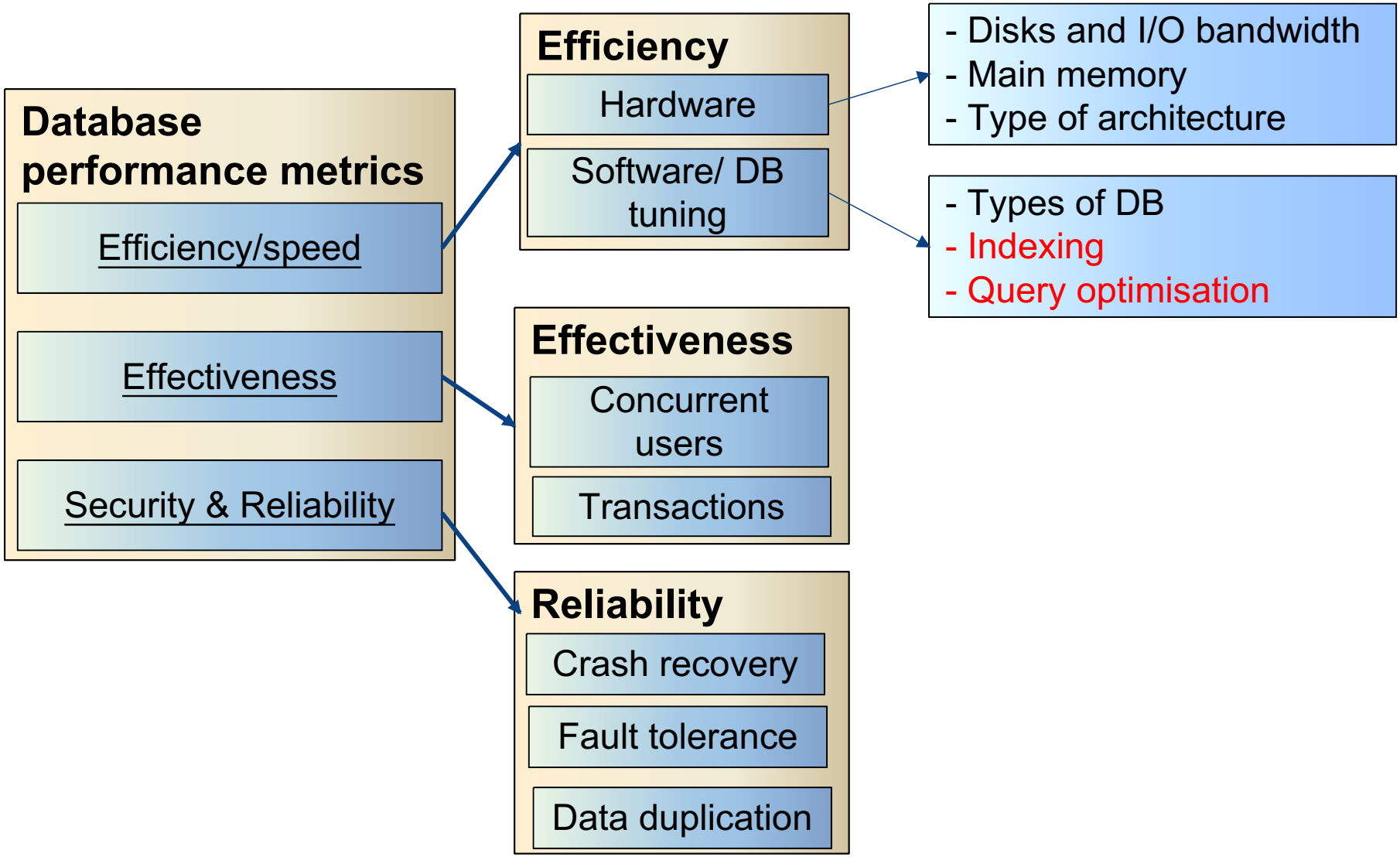
Main Cost is search through all the records in this table to make comparison one by one ⇒ hardly will those four index be not helpful for compurison.
Such huge volume of

Python cheatsheet

What you need to do now?

- Given a data set, when uploading to the DBMS
  - Find the potential query types
  - Research what indices that particular DBMS would have for that data type
  - Research for what queries you would better do on what index
  - Create index if you have large data
  - Monitor performance
  - Tune or create other indices

Summary of last week



Discussion on this week's topics

- Some terminologies on transactions
- Types of transactions
  - Flat transactions
  - Flat transactions with save points
  - Nested transactions
- Transaction processing monitor

Embedded SQL example in C

(Open Database Connectivity)

```
int main()
{       exec sql INCLUDE SQLCA; /*SQL Communication Area*/
        exec sql BEGIN DECLARE SECTION;
          /* The following variables are used for communicating
                between SQL and C */

          int OrderID; /* Employee ID (from user) */

          int CustID; /* Retrieved customer ID */

          char SalesPerson[10] /* Retrieved salesperson name */

          char Status[6] /* Retrieved order status */

        exec sql  END DECLARE SECTION;

/* Set up error processing */
        exec sql WHENEVER SQLERROR GOTO query_error;
        exec sql WHENEVER NOT FOUND GOTO bad_number;
```

Proper error handling is important!

What can go wrong in this example?

(Open Database Connectivity)

```
int main()
{       …../code to get data from database
        Printf("%d", (get_salary(EMPID)/800000);

/* Set up error processing */
        exec sql WHENEVER DIVIDE_BY_ZERO GOTO inf_error;
        exec sql WHENEVER NOT_PERMITTED GOTO permission_err;
        exec sql WHENEVER NOT_FOUND GOTO bad_number;
```

按序check error (order 也很重要)

Time for a poll - Pollev.com/farhanachoud585

Proper error handling is important!

a = get_salary(EMPID)/80000

if a=1 then do 0/0 # a=1时只有员工 salary是80000时

这个就会报错 (divide_by_zero).

我可以用这样run所有员工看谁掉30000

Transaction Processing Monitor

used in large scale business

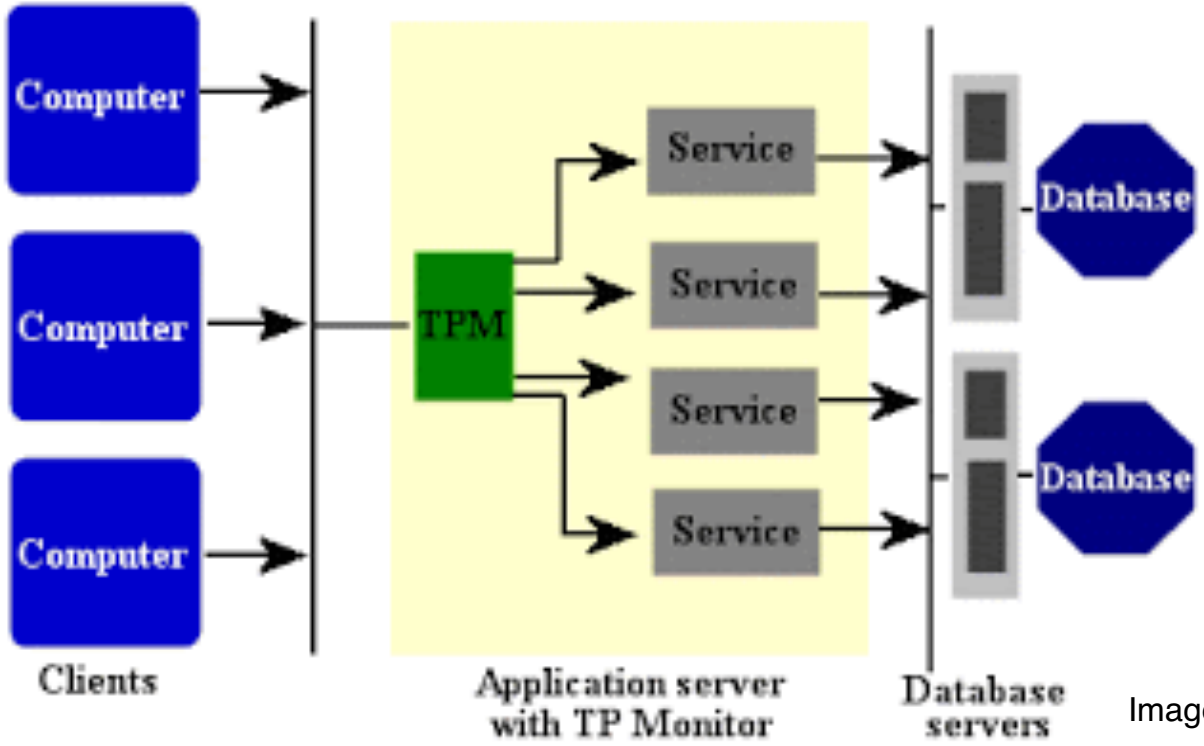


Image source: http://3.bp.blogspot.com

I*ntegrates* other system components and manage resources.

- Manages the transfer of data between clients and servers
- breaks down applications or code into transactions and ensures that all databases are updated properly
- It also takes appropriate actions if any error occurs

Flat Transaction

Everything inside BEGIN WORK and COMMIT WORK is at the same level; that is, the transaction will either survive together with everything else (commit), or it will be rolled back with everything else (abort)

```
exec sql BEGIN WORK;
      AccBalance = DodebitCredit(BranchId, TellerId, AccId, delta);
      send output msg;
exec sql COMMIT WORK;
```

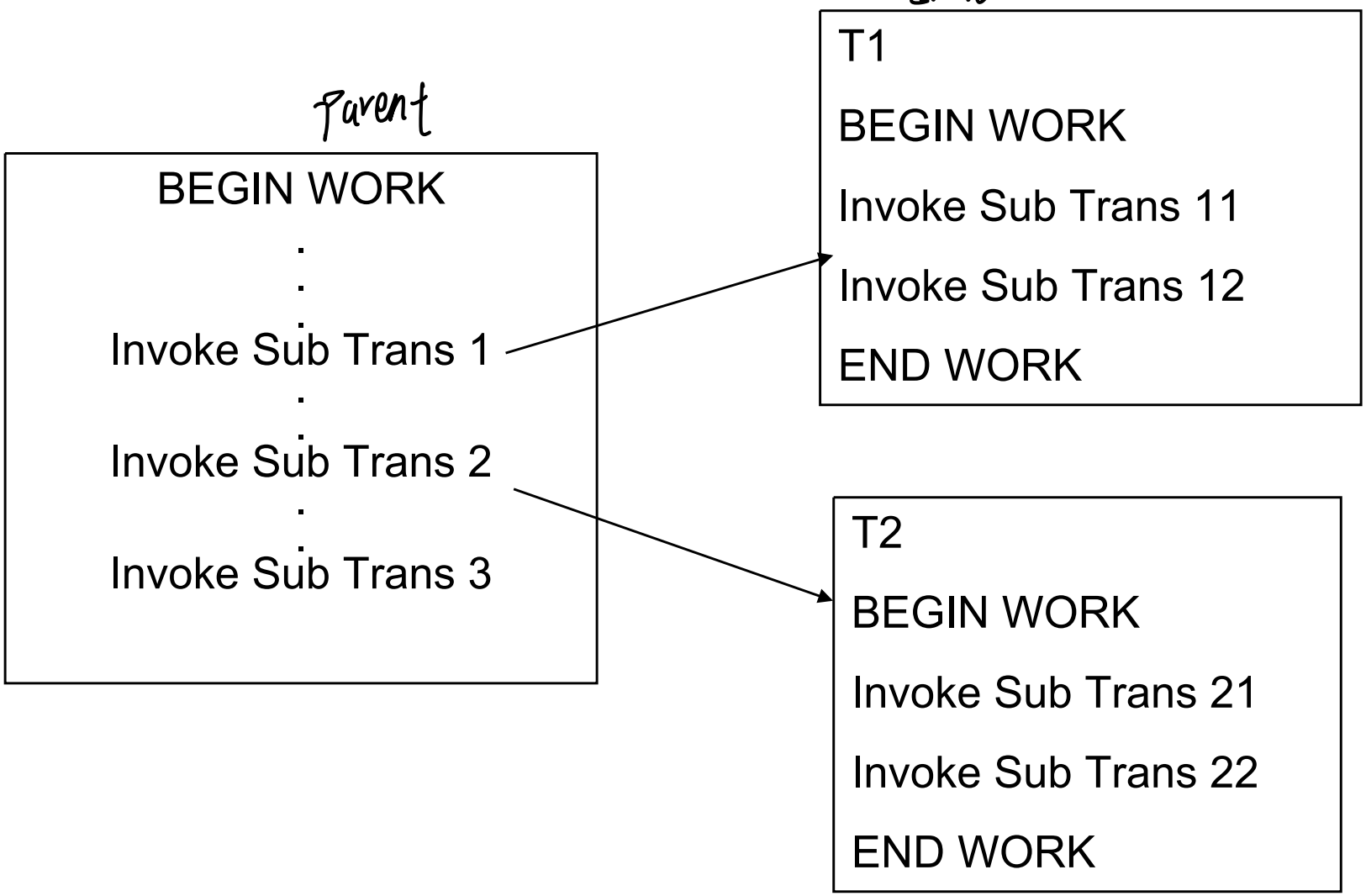
Can be a very long running transaction with many operations

Limitations of Flat Transactions?

Time for a poll - Pollev.com/farhanachoud585

If something happen in the middle of a long runnig transaction and it rolls back, we have to start again, all the things were done will be wasted (waste time, computation power, etc.)

Nested Transactions



Nested Transactions

Commit rule

- A subtransaction can either commit or abort, however, **commit cannot take place unless the parent itself commits.**

- Subtransactions have A, C, and I properties but not D property unless all its ancestors commit.

- Commit of a sub transaction makes its results available only to its parents.

Roll back Rules

If a subtransaction rolls back, all its children are forced to roll back.

Visibility Rules

Changes made by a subtransaction are visible to the parent only when the subtransaction commits. All objects of parent are visible to its children. Implication of this is that the **parent should not modify objects while children are accessing them.** This is not a problem as parent does not run in parallel with its children.

What are the advantages?

pros: Unlike flat transaction. everything gets rolled back.

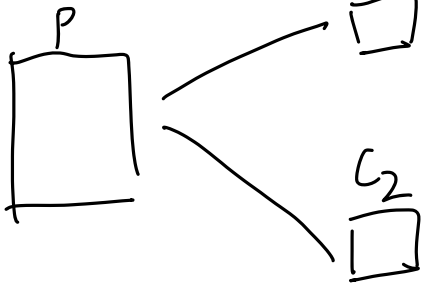
child

① subtransaction can run in parallel if there are no association between them, making the whole transaction much faster. (divide into several task chunks for example)

② if some transaction rolled back, it won't affect others. ⇒ enhance modularity.

③ Sometime ⇒ we want to roll back purposefully ⇒ sometimes we want users to input something ⇒ account balance, amount of cash the person wants to withdraw from their account ⇒ which might make account balance negative ⇒ we just can withdraw that part of the transaction ⇒ (maybe before that the users made some other changes. which were perfectly fine, and won't affect those changes.)

So for some specific input or under some specific condition ⇒ we just want to roll back part of transaction without impacting others ⇒ nested transaction allows you to design such functionality.



if C_1 has satisfied, we don't need to execute this part

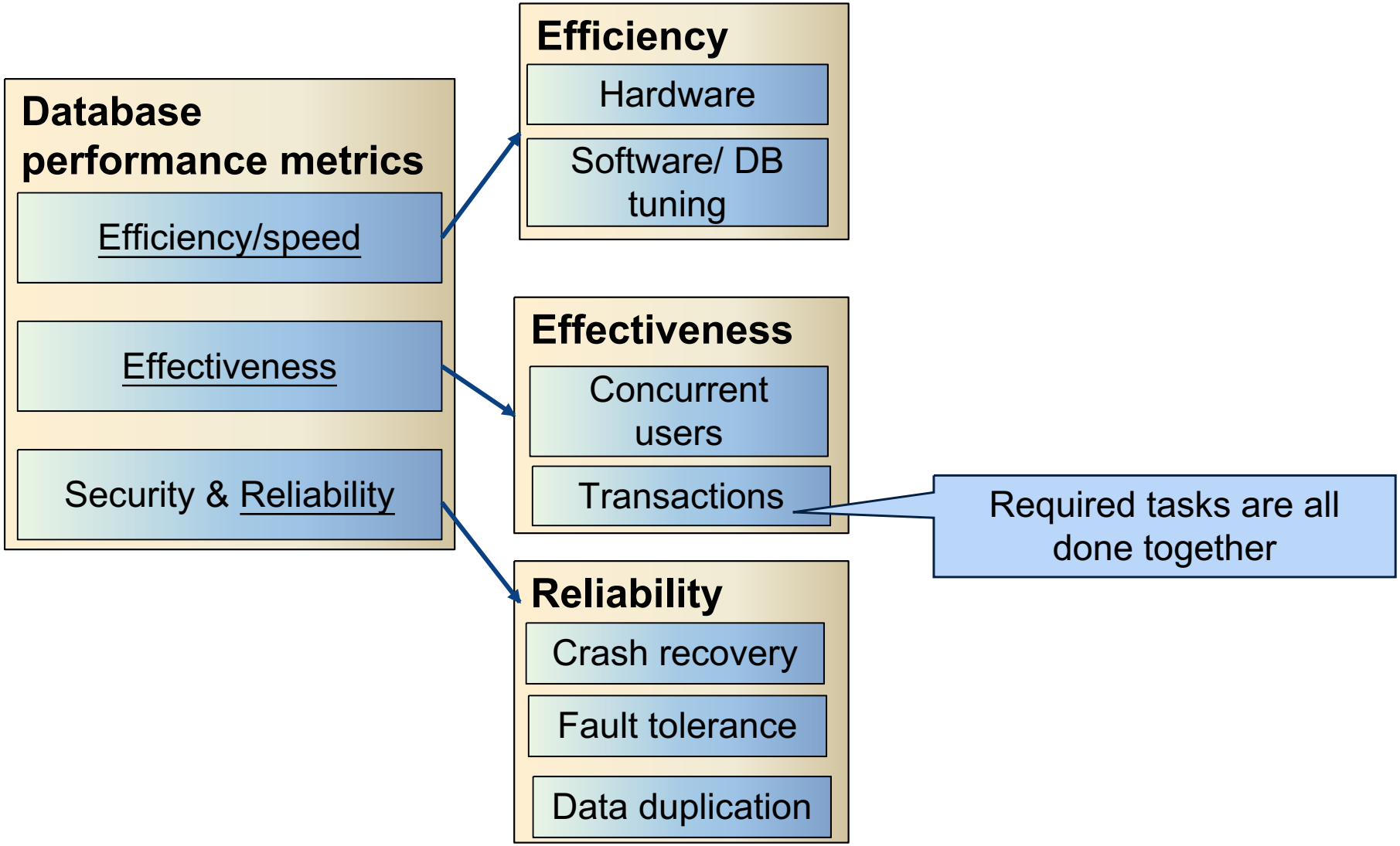
if C_2 is wrong, we can just let C_2 roll back

such sort of functionality is possible for nested transaction.

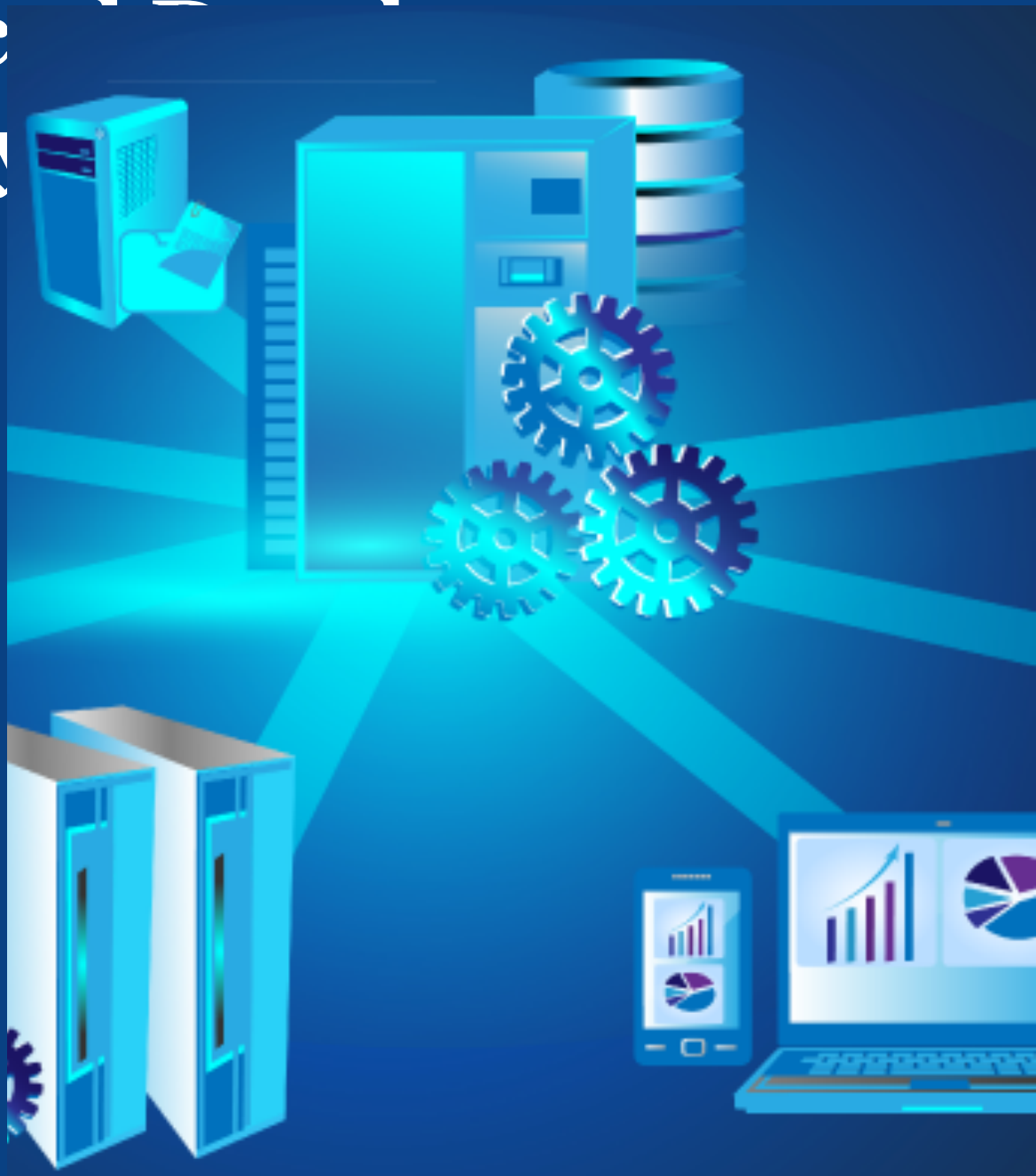
fine tuning roll back is possible.

roll back a specific part of transaction while keeping others unimpacted.

Core Concepts of Database management system



COMP90050: Advanc… Sy…

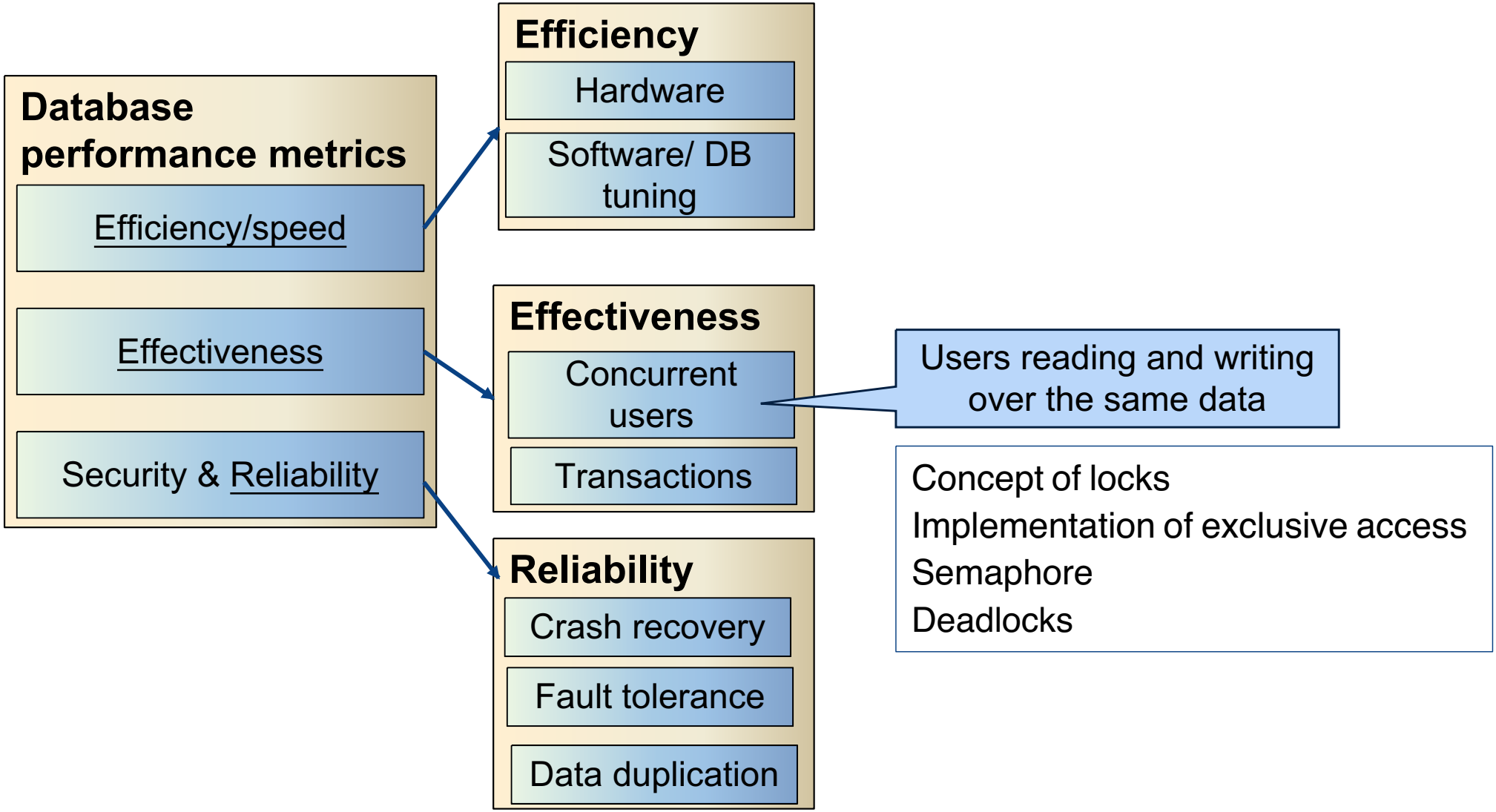


Lecturer: Farhana Choudhury (PhD)

Concurrency control

Week 5

Core Concepts of Database management system



Concurrency Problems

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 are running concurrently. What are the possible values of counter after the end of Task1 and Task2?

a) counter == 110;

b) counter == 130;

c) counter == 140;

For correct execution we need to impose exclusive access to the shared variable counter by Task1 and Task2.

Concurrency Problems

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently. What are the possible values of counter after the end of Task1 and Task2?

Note: == means equals.

a) counter == 110

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T2: Writes counter == 100+30

T1: Writes counter == 100+10

b) counter == 130

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T1: Writes counter == 100+10

T2: Writes counter == 100+30

c) counter == 140;

Sequence of actions

T1: Reads counter == 100

T1: Writes counter == 100+10

T2: Reads counter == 110

T2: Writes counter == 110+30

different versions

Time

Concurrency Control

- To resolve conflicts
- To preserve database consistency

Different ways for concurrency control

- **Dekker's algorithm (using code)** - needs almost no hardware support, but the code is very complicated to implement for more than two transactions/processes
- **OS supported primitives (through interruption call)** - expensive, independent of number of processes, machine independent
- **Spin locks (using atomic lock/unlock instructions)** – most commonly used

Concurrency control: Implementation of exclusive access

Dekker's algorithm

```
int c1, c2, turn = 1; /* global variable*/
```

T1

```
{ some code T1}
/* T1 wants exclusive access to the resource
     and we assume initially c1 = = 0*/
c1 = 1; turn = 2;
repeat until { c2 == 0 or turn == 1}
/* Start of exclusive access to the
     shared resource (successfully
     changed variables) */
use the resource
counter = counter+1;
/* release the resource */
c1 = 0;
{some other code of T1}
```

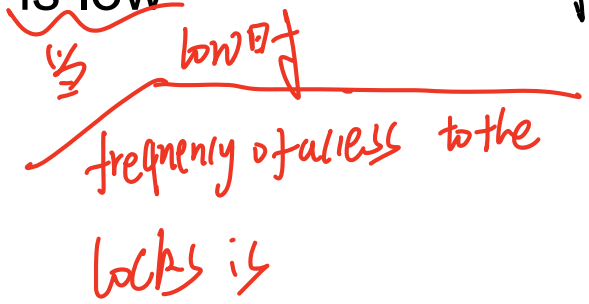
T2

```
{ some code T2}
/* T2 wants exclusive access to the resource
     and we assume initially c2 = = 0 */
c2 = 1; turn = 1;
repeat until { c1 == 0 or turn == 2}
/* Start of exclusive access to the
     shared resource */
use the resource
counter = counter+1;
/* release the resource */
c2 = 0 ;
          {some other code of T2}
```

Implementation of exclusive access

- Dekker's algorithm
  - needs almost no hardware support although it needs atomic reads and writes to main memory, That is exclusive access of one time cycle of memory access time!
  - **the code is very complicated to implement if more than two transactions/process are involved**
  - harder to understand the algorithm for more than two process
  - takes lot of storage space
  - uses busy waiting
  - efficient if the lock contention (that is frequency of access to the locks) is low

waiting for the resources to be released by the other transaction



if the resource has been by other transaction → T_1 cannot do anything but just to waity → keep repeating this loop and just wait → busy waity → one transaction is just waity in a loop for the resource to be released ⇒ it cannot do anything at that time

hampers the overall performance of the transaction.

Implementation of exclusive access

- OS supported primitives such as lock and unlock
  - through an interrupt call, the lock request is passed to the OS
  - need no special hardware
  - are very expensive (several hundreds to thousands of instructions need to be executed to save context of the requesting process.)
  - do not use busy waiting and therefore more effective

- All modern processors do support some form of spin locks.

lock: a request to access to have exclusive access to a shared resources.
after using that resource the process or transaction is released
⇒ e.g. shared database table or rows

Implementation of exclusive access

Spin Locks

Executed using atomic machine instructions such as test and set or swap

- need hardware support – should be able to lock bus (communication channel between CPU and memory + any other devices) for two memory cycles (one for reading and one for writing). During this time no other devices' access is allowed to this memory location.
- use busy waiting (if shared resources is currently being used by another transaction, it has to wait in the loop and wait until that resource is released by the other process)
- algorithm does not depend on number of processes
- are very efficient for low lock contentions – all DB systems use them

Span locks:
- Once get exclusive lock on the resource, only that process of the transaction will be able to read and write on that resource ⇒ no other device or transaction will be able to access that particular memory location that is being currently read or written.

Implementation of Atomic operations: test and set

```
testAndSet(int *lock)
{ /* the following is executed atomically, memory bus can be locked for up
     to two cycles (one for read and for writing*/
        if (*lock == 1){ * lock = 0; return (true)}
        else return (false);
}
```

Using test and set in spin lock for exclusive access

int lock = 1; % initial value

T1	T2
/*acquire lock*/	/*acquire lock*/
while (!testAndSet(&lock);	while (!testAndSet(&lock);
/*Xlock granted*/	/*Xlock granted*/
//exclusive access for T1;	//exclusive access for T2;
counter = counter+1;	counter = counter+1;
/* release lock*/	/* release lock*/
lock = 1;	lock = 1;

Implementation of Atomic operation: compare and swap

The following code may have lost values

```
temp = counter +1; //unsafe to increment a shared counter
counter = temp;  //this assignment may suffer a lost update
```

a) counter == 110

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T2: Writes counter == 100+30

T1: Writes counter == 100+10

第一行由于同时有人修改了counter的值

⇒ counter值无效了(不知道)

(无法第一次读到的是什么)

为了防止这种现象,保证atomicity.

⇒ compare and swap用上了.

(也是用于 exclusive access)

Using compare and swap in spin lock for exclusive access

Implementation of Atomic operation: compare and swap

The following code may have lost values

```
temp = counter +1; //unsafe to increment a shared counter

counter = temp;  //this assignment may suffer a lost update
```

Instead, we can use the atomic operation of compare and swap instruction

```
boolean cs(int *cell, int *old, int *new)
{/* the following is executed atomically*/
if (*cell == *old)      { *cell = *new; return TRUE;}
else {*old = *cell; return FALSE;}
}
```

```
temp = counter;
do
        new = temp+1;
while(!cs(&counter,&temp,&new);
```

Using compare and swap in spin lock for exclusive access

Semaphores

Semaphores derive from the corresponding mechanism used for trains: a train may proceed through a section of track only if the semaphore is clear. Once the train passes, the semaphore is set until the train exits that section of track.

Try to Get(track), wait if track not clear

If Get(track) successful, use it (no other train will be able to use it now)

So it will improve the performance ~~tha~~ => if multiple transactions are able to run concurrently. => the overall number of operations that are being executed in the system improve. <=> run more operations together

Compare and swap

for writing part

just check if it is changed by someone => whether it's safe to change now, just doing this check and not preventing other transaction from reading

=> then this compare and swap is ~~more~~ more suitable option.

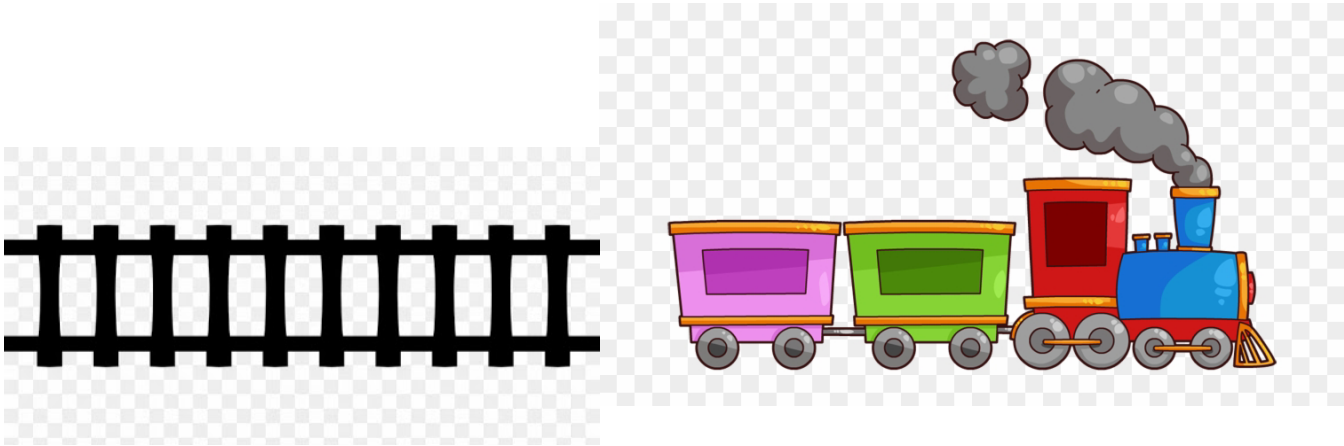
当访问资源的transaction 打衡时 => 有什么流程?

Semaphores

Semaphores derive from the corresponding mechanism used for trains: a train may proceed through a section of track only if the semaphore is clear. Once the train passes, the semaphore is set until the train exits that section of track.

release after using

Release(track) after using
(so that others can use it)



Semaphores

Computer semaphores have a get() routine that acquires the semaphore (perhaps waiting until it is free) and a dual give() routine that returns the semaphore to the free state, perhaps signalling (waking up) a waiting process.

Semaphores are very simple locks; indeed, they are used to implement general-purpose locks.

Implementation of Exclusive mode Semaphore

Pointer to a queue of processes

If the semaphore is busy but there are no waiters, the pointer is the address of the process that owns the semaphore.

If some processes are waiting, the semaphore points to a linked list of waiting processes. The process owning the semaphore is at the end of this list.

After usage, the owner process wakes up the oldest process in the queue (first in, first out scheduler)

Implementation of Exclusive mode Semaphore

```
type long PID

type struct Process{

        PID pid; /*process ID*/

        PCB * sem_wait; /* waiting process are put in the queue */

} PCB;

PID MyPID (void); /* returns the caller's process ID */

PCB * MyPCB(void)

/* returns pointer to caller's process descriptor */

void wait (void); /* suspends  calling process */

void wakeup( PCB * him) wakes him.
```

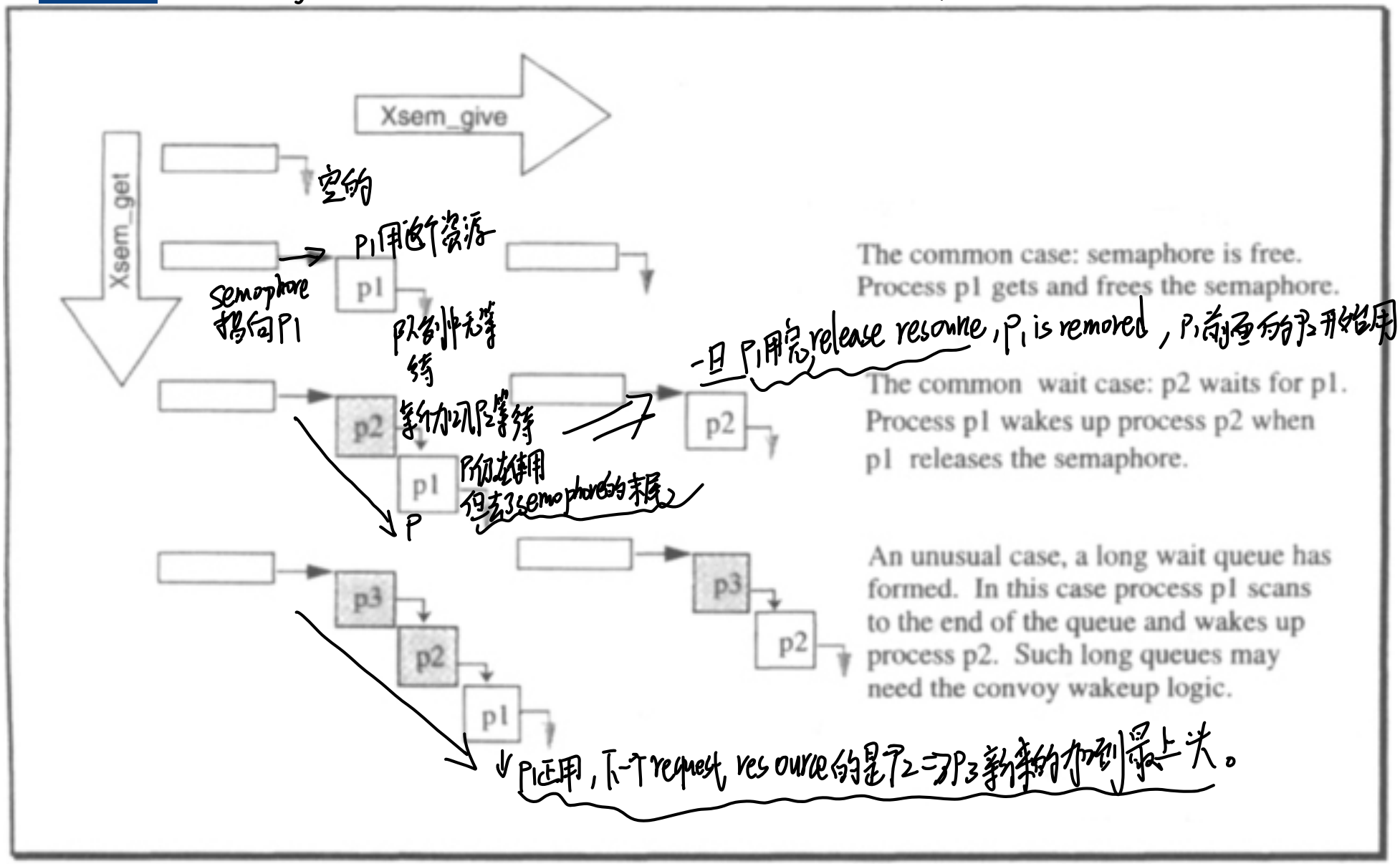
Implementation of exclusive lock semaphore operations

```
type struct Process{
        PID pid;
        PCB * sem_wait;
}PCB;
typedef PCB *Xsemaphore
Void initialise( Xsemaphore *sem)
{*sem = NULL; return;}
```

```
void lock(Xsemaphore *sem){
   PCB * new = myPCB();
   PCB * old   = NULL;
 /*put the process in the semWait queue*/
   do{
       new ->semWait = old;
   } while(! cs(sem, &old, &new));
/*if queue not null then must wait for a
wakeup from the semaphore owner*/
   if (old != NULL) wait() % OS call
   return;
}
```

```
void unlock(Xsemphore *sem){
   PCB * new = NULL;
   PCB * old   = myPCB();
    if(cs(sem, &old, &new)) return;
          while(old->semWait
     !=myPCB()){
        old = old->semWait
          }
          old->semWait = NULL;
     /*wake up the oldest process (first in,
     first out scheduler)*/
          wakeup(old); % OS Call
}
```

每次只允许一个访问，其他的放进队列等待，一个用完，下个用.



Convoy avoiding semaphore

The previous implementation may result a long list of waiting processes – called convoy

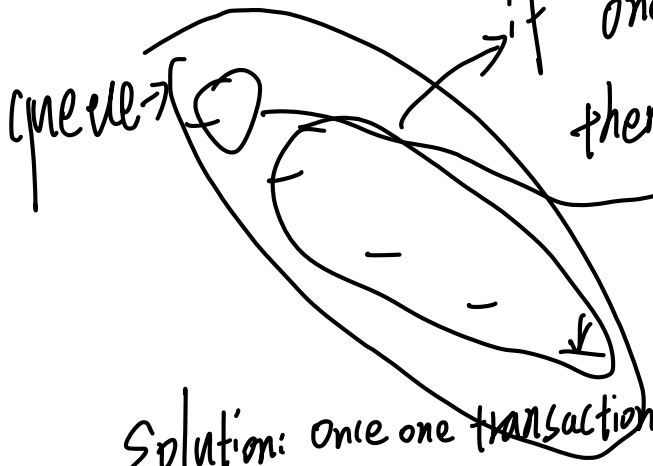
To avoid convoys, a process may simply free the semaphore (set the queue to null) and then wake up every process in the list after usage.

In that case, each of those processes will have to re-execute the routine for acquiring semaphore.

problems: long list of waiting processors ⇒ convoy

if one of them or couple of them are long-running processors,

then the one that make request later has to wait for longer time in the queue.



Solution: once one transaction is done ⇒ locks released ⇒ every process in the waiting list is awaken up ⇒ recall the resources again ⇒ order of the waiting in next queue will not.

Convoy avoiding semaphore

```
void lock(Xsemaphore *sem){
   PCB * new = myPCB();
   PCB * old   = NULL;
   do{
       new ->semWait = old;
    }while( ! cs(sem, &old, &new));
   if(old != NULL) {
            wait();  lock(sem);
    }
   return;
}
```

```
void unlock(Xsemphore *sem){
    PCB * new = NULL;
        PCB * old   = myPCB()
        while(!cs(sem, &old,
&new));
        while(old->semWait !=
myPCB()){
                new = old-
>semWait;
                old->sem_wait =
NULL;
        wakeup(old);
                old = new;
    return;
}
```

ensure long waiting queue do not occur and this loop is responsible for that.

A quick summary

Concurrency problems – why we need concurrency control

Implementation of exclusive access –

- Dekker's algo
- OS primitives
- **Spin locks -** Atomic operations to get and release locks

different ways of how the exclusive access can be granted.

Semaphores - get lock, release lock, maintain queue of processes

一旦锁被release => if other processors that are also requesty for that lock the semaphore will maintain those processors.

有问题的地方 reshuffle.

Avoiding long queues in semaphores

Concurrency problems: When multiple processors or transactions are running concurrently and they are working on the same shared resources or the same shared data.

问题在哪: The order of their execution can effect the final output of that shared resource

which is not desirable. To have consistency on the value of those shared resources the processors or transactions need some exclusive access on those resources

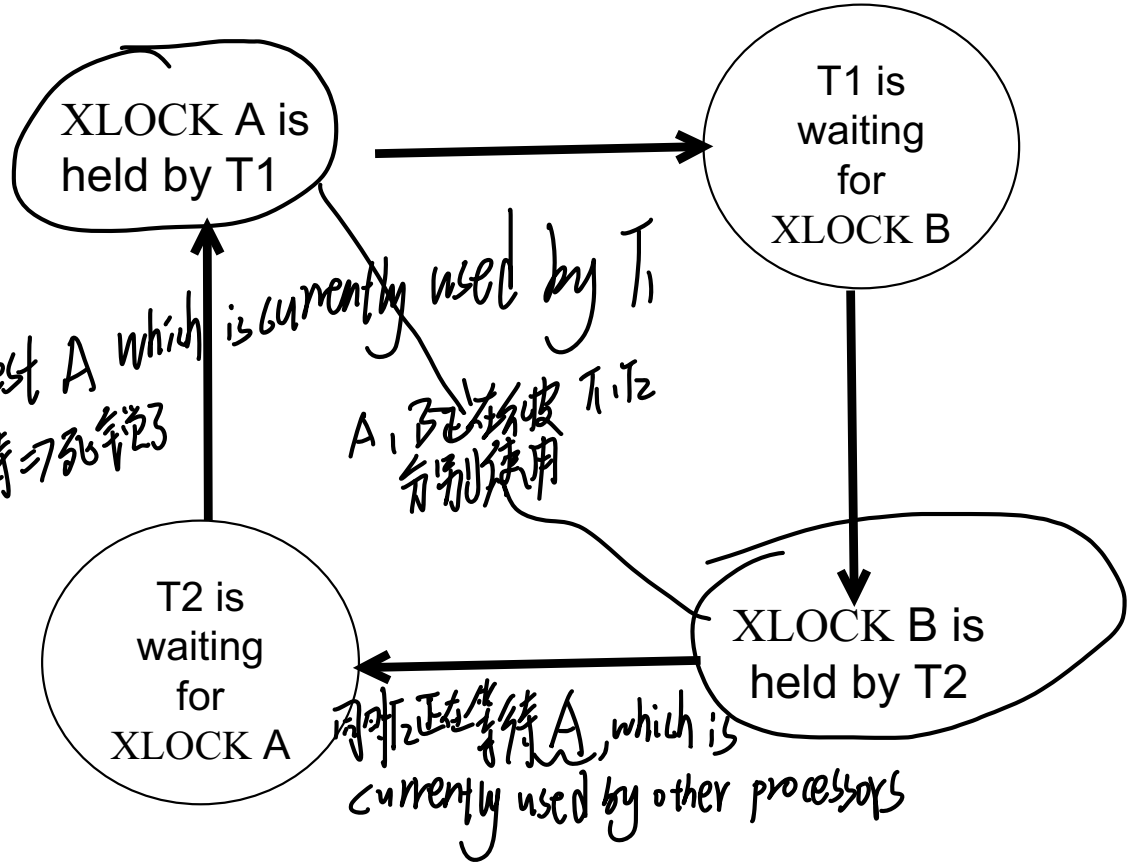
Deadlocks

In a deadlock, each process in the deadlock is waiting for another member to release the resources it wants.

A very simple Deadlock

T1	T2
Begin	Begin
XLOCK(A)	XLOCK(B)
Write to A	Write to B
XLOCK(B)	XLOCK(A)
Write B	Write A
Unlock (A)	Unlock (A)
Unlock (B)	Unlock (B)
end	end

Resource Dependency Graph



这种loop：两个transaction都等 等对方释放资源，同时他们held的 the lock of the resources which are currelly requested by other transactions

Deadlocks

So that no process can proceed to the next set of operations ⇒ the operation will not be executed ⇒ because each of processors are just waiting

Solutions:

- Have enough resources so that no waiting occurs – not practical
- Do not allow a process to wait, simply rollback after a certain time. This can create live locks which are worse than deadlocks.
- Linearly order the resources and request of resources should follow this order, i.e., a transaction after requesting i^{th} resource can request j^{th} resource if $j > i$ .This type of allocation guarantees no cyclic dependencies among the transactions.

① → 不允许循环步 trans去同时请求资源，资源数量总是有限的.

② 判断：一般等待一段时间仍未proceed ⇒ 判断 deadlock ⇒ 释放资源并roll back
roll back

③ N个资源 ⇒ put linear order on these resources ⇒ if a transaction requesty for a particular resource i ⇒ then it can only request another resource that is in a later position in that linear order resource j [(j>i)]

Deadlocks

Pa: Holds resources at level ***i*** and request resource at level ***j*** which are held by Pb. ***j > i***

Pb: Holds resources at level ***j*** and request resource at level ***k*** which are held by Pc. ***k > j***

Pc: Holds resources at level ***k*** and request resource at level ***l*** which are held by Pd ***l > k***

Pd: Holds resources at level ***l*** and and is currently running.

Pq: Holds resources at level ***g*** and request resource at level ***l*** which are held by Pd. ***l > g***

l > k > j > i and l > g.

We cannot have loops. The dependency graph can be a tree or a linear chain and hence cannot have cycles.

Deadlock avoidance/mitigation

- Pre-declare all necessary resources and allocate in a single request.
- Periodically check the resource dependency graph for cycles. If a cycle exists - rollback (i.e., terminate) one or more transaction to eliminate cycles (deadlocks). The chosen transactions should be cheap (e.g., they have not consumed too many resources).
- Allow waiting for a maximum time on a lock then force Rollback. Many successful systems (IBM, Tandem) have chosen this approach.
- Many distributed database systems maintain only local dependency graphs and use time outs for global deadlocks.

Deadlocks ...

Deadlocks are rare, however, they do occur and the database has to deal with them when they occur

What is the probability of a deadlock occurrence? - tutorial

Summary

Concurrency problems – why we need concurrency control

Implementation of exclusive access – Dekker's algo, OS primitives, spin locks

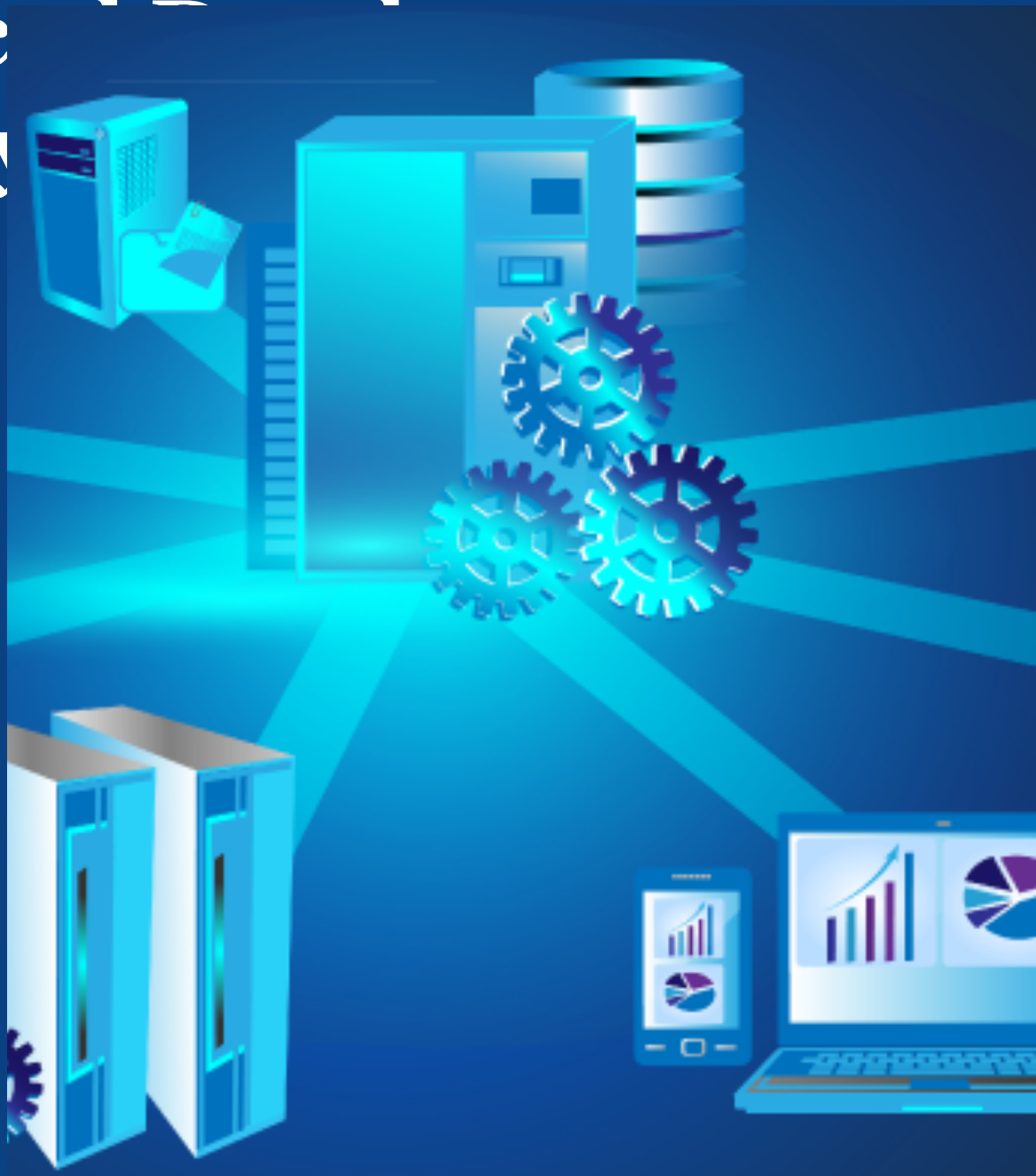
Semaphores - get lock, release lock, maintain queue of processes

Atomic operations to get and release locks

Semaphore queue, avoiding long queues

Deadlocks

COMP90050: Advanc… Sy…



Lecturer: Farhana Choudhury (PhD)

Isolation concepts

Week 6

What we have seen in previous lecture

- Problems may arise from multiple concurrent transactions
- Concurrency control is needed
  - Take exclusive access to shared resources to handle concurrency problems

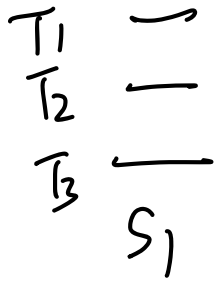
In this lecture we will see the concurrency control more formally and in more details

Isolation Concepts – I in ACID

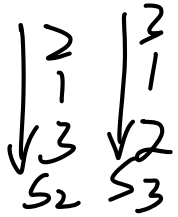
Isolation ensures that concurrent transactions leaves the database in the same state as if the transactions were executed separately.

Isolation guarantees consistency, provided each transaction itself is consistent.

isolation property: if there are concurrent transactions ⇒ those concurrent transactions should leave the data base in the same state as if these concurrent transaction were executed separatly.



run them one by one → seperate execution order



3个-起跑的state 在依次 S1 S2 S3 中的一个 match

The current should not have different state than a

Isolation – expected output example

Shared counter = 100;

Task1/Trans/Process/Thread	Task2/Trans/Process/Thread
counter = counter +10;	counter = counter +30;

Task1 and Task2 run concurrently.

a) counter == 110

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T2: Writes counter == 100+30

T1: Writes counter == 100+10

b) counter == 130

Sequence of actions

T1: Reads counter == 100

T2: Reads counter == 100

T1: Writes counter == 100+10

T2: Writes counter == 100+30

c) counter == 140;

Sequence of actions

T1: Reads counter == 100

T1: Writes counter == 100+10

T2: Reads counter == 110

T2: Writes counter == 110+30

Alternatively, T2 executing before T1 will also have the same state

Isolation Concepts ...

We can achieve isolation by sequentially processing each transaction - generally not efficient and provides poor response times.

We need to run transactions concurrently with the following goals:

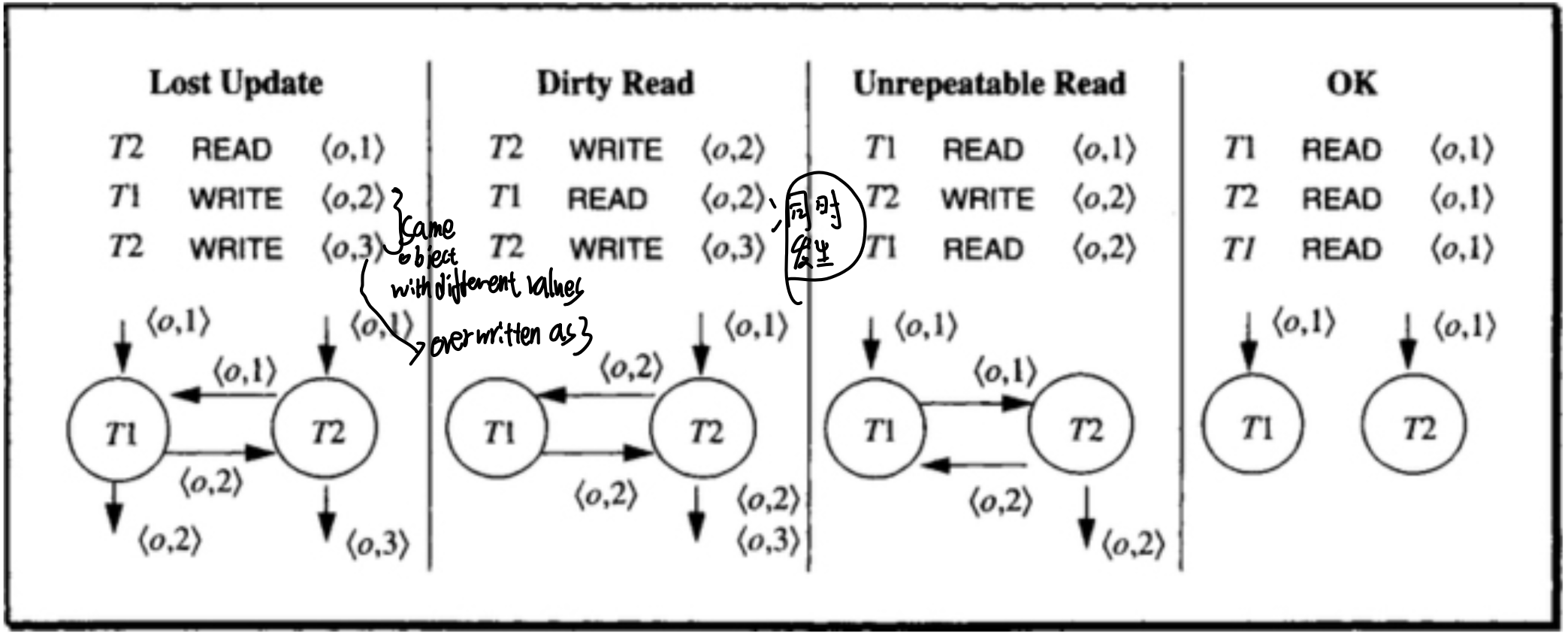
- concurrent execution should not cause application programs (transactions) to malfunction.
- Concurrent execution should not have lower throughput or bad response times than serial execution.

execute sequentially => no concurrecy but low efficiency => 要等时间长 poor response time.

To achieve isolation we need to understand dependency of operations

State of runny transactions concurrently = state of runny transactions seperately.

Possible dependencies



How can we find the dependencies?

Given a set of transactions, how can we determine which transaction depends on which other transaction?

Dependency model

I_i: set of inputs (objects that are read) of a transaction T_i

O_i : set of outputs (objects that are modified) of a transaction T_i

Note O_j *and* I_j are not necessarily disjoint that is $(O_j \cap I_j \neq \emptyset)$

Given a set of transactions , Transaction T_j has no dependency on any transaction T_i in if -

$$O_i \cap (I_j \cup O_j) = empty\ for\ all\ i \neq j$$

This approach cannot be planed ahead as in many situation inputs and outputs may be state dependant/not known in prior.

the set of objects · whether it is ready or writty ⇒ it will never be modified by any other transaction T_i.

rather T_j is operating

output: modified

$$O_i \cap (I_j \cup O_j) = \text{empty for all } i \neq j$$

Input ready

if they are not disjoint ⇒ $I_j \cup O_i$ not hold

	T_1	T_2	T_3	T_4
Trans In ∪ Out $(I_i \cup O_i)$	$O_i \cap (I_1 \cup O_1)$	$O_i \cap (I_2 \cup O_2)$	$O_i \cap (I_3 \cup O_3)$	$O_i \cap (I_4 \cup O_4)$
T_1	O_1	∅	∅	∅
T_2	∅	O_2	∅	∅
T_3	∅	∅	O_3	∅
T_4	∅	∅	∅	O_4

4 trans run concurrently

$T_1, \ldots, T_4$ 没有关联 those not have any relationship with any other transactions.

O : output modification

I : input ready

If the inputs and outputs of the concurrent transactions are not disjoint, the following dependencies can occurs –

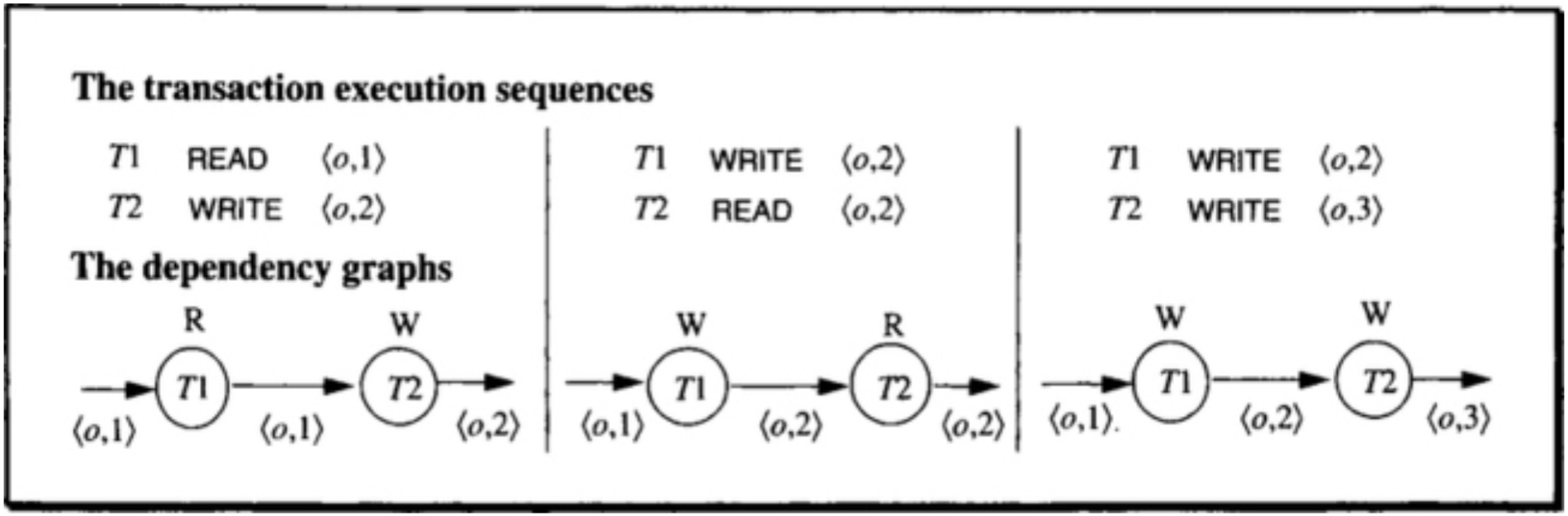
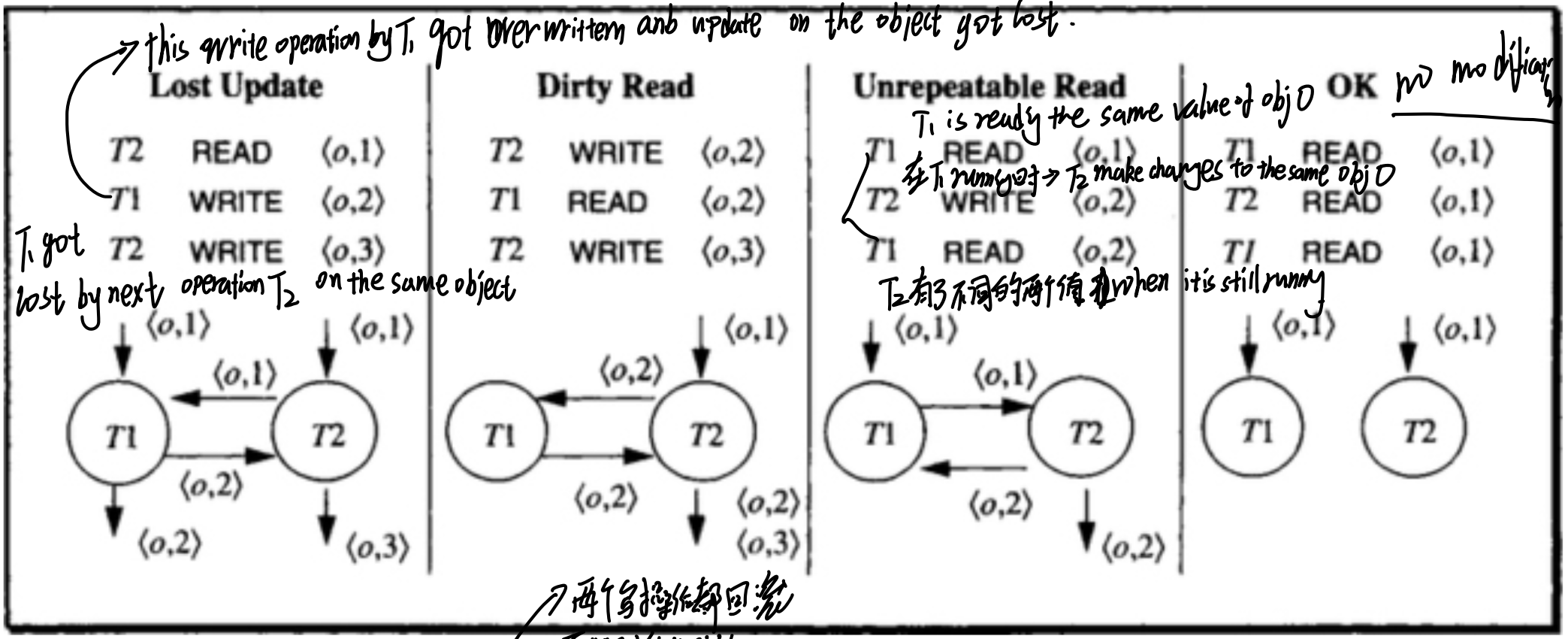


Fig 7.1 in the main reference book

Read-Read dependency do not affect isolation

Dependencies

When dependency graph has cycles then there is a violation of isolation and a possibility of inconsistency.



% T2 did not see the update of T1

What if T2 aborts? T1's read will be invalid (Inconsistency)

The value of o changes by another transaction T2 while T1 is still running (Inconsistency) if T2 aborts ⇒ last read 无效了.

Formal definition of dependency

Let H is a history sequence of tuples of the form (T, action, object).

Let T1 and T2 are transactions in H. If T1 performs an action on an object O, then T2 performs an action on the same O, and there is no write action in between by another transaction on O – then T2 depends on T1.

Formally, the dependency of T2 on T1 (T1, O, T2) exists in history H if there are indexes i and j such that i < j, H[i] involves action a1 on O by T1, (i.e., H[i] = (T1,a1,O)) and H[j] involves action a2 on O by T2 (i.e., H(j) = (T2, a2,O)) and there are no other H[k] = (T',WRITE,O) for i < k < j

Dependency graph : Transactions are nodes, and object labels the edges from the node Ti to Tj if (Ti, O, Tj) is in DEP(H).

Dependency relations

We focus on the dependency in three scenarios

- a1 = WRITE & a2 = WRITE;
- a1 = WRITE & a2 = READ;
- a1 = READ & a2 = WRITE (dependency as T1 may read again after a2).

Dependency relations - equivalence

DEP(H) = { (Ti, O, Tj) | Tj depends on Ti }.

Given two histories H1 and H2 contain the same tuples, H1 and H2 are equivalent if DEP(H1) = DEP(H2)

This implies that a given database will end up in exactly the same final state by executing either of the sequence of operations in H1 or H2

E.g., H1 = <(T1,R,O1), (T2, W, O5), (T1,W,O3), (T3,W,O1), (T5,R,O3), (T3,W,O2), (T5,R,O4), (T4,R,O2), (T6,W,O4)>

DEP(H1) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

H2 = <(T1,R,O1), (T3,W,O1), (T3,W,O2),(T4,R,O2),(T1,W,O3), (T2, W, O5), (T5,R,O3), (T5,R,O4), (T6,W,O4)>

DEP(H2) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

DEP(H1) = DEP(H2)

H =< (T1,R,O1), (T3,W,O3), (T3,W,O1), (T2,R,O3), (T1,R,O1), (T2,W,O2), (T1,R,O2), (T1,R,O3), (T2,R,O2) >

$DEP(H1) = \{\langle T_1, O_1, T_3 \rangle, \langle T_3, O_3, T_2 \rangle, \langle T_2, O_2, T_1 \rangle\}$.

E.g.,

H1 = <(T1,R,O1), (T2, W, O5), (T1,W,O3), (T3,W,O1), (T5,R,O3), (T3,W,O2), (T5,R,O4), (T4,R,O2), (T6,W,O4)>

$DEP(H1) = \{\langle T_1, O_1, T_3 \rangle, \langle T_1, O_3, T_5 \rangle, \langle T_3, O_2, T_4 \rangle, \langle T_5, O_4, T_6 \rangle\}$

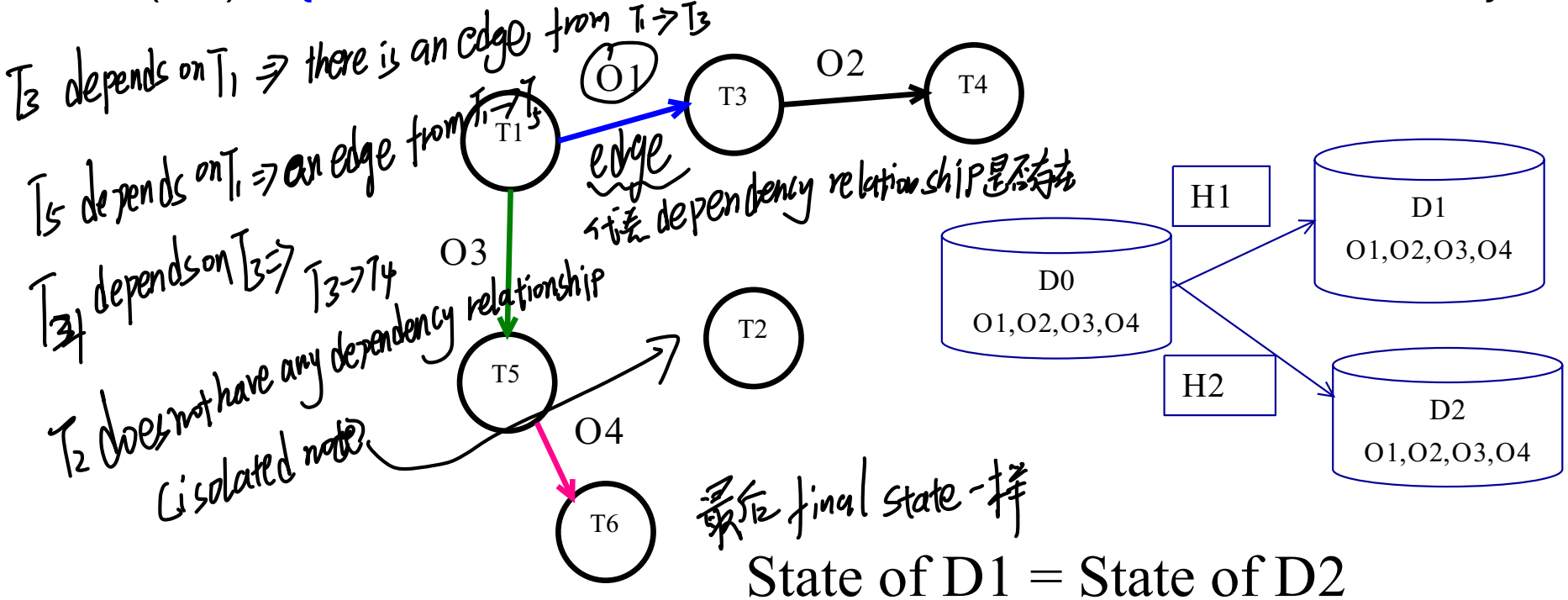
Dependency relations - equivalence

H1 = <(T1,R,O1), (T2, W, O5), (T1,W,O3), (T3,W,O1), (T5,R,O3), (T3,W,O2), (T5,R,O4), (T4,R,O2), (T6,W,O4)>

H2 = <(T1,R,O1), (T3,W,O1), (T3,W,O2),(T4,R,O2),(T1,W,O3), (T2, W, O5), (T5,R,O3), (T5,R,O4), (T6,W,O4)>

DEP(H1) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

DEP(H2) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }



Isolated history

A history is said to be isolated if it is equivalent to a serial history (as if all transactions are executed serially/sequentially)

A serial history is history that is resulted as a consequence of running transactions sequentially one by one. N transactions can result in a maximum of N! serial histories.

If T1 precedes T2,

it is written as T1 << T2.

Before(T) = {T' | T' << T}

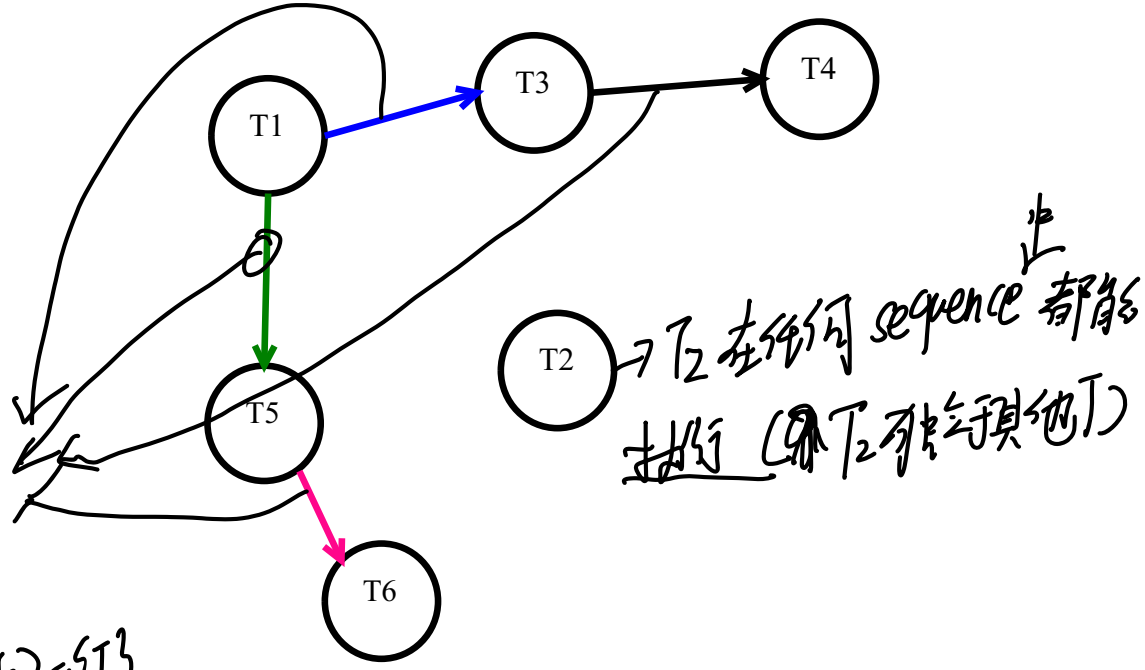
After(T) = {T'| T << T'}

E.g. After(T1) = {T5, T6, T3, T4}

After(T3) = {T4}

After (T5) = {T6}

After(T3) = {T4}



Isolation Concepts ...

A transaction T' is called a wormhole transaction if

$$T' \in Before(T) \bigcap After(T)$$

That is T << T' << T. This implies there is a cycle in the dependency graph of the history. Presence of a wormhole transaction implies it is not isolated (=> not a serial schedule).

A history is serial if it runs one transaction at a time sequentially, or equivalent to a serial history. A serial history is an **isolated** history.

Wormhole theorem: A history is isolated if and only if it has no wormholes.

serializability ≠ isolated history ⇔ no wormhole.

We will now introduce a new type of lock -

SLOCK (shared lock) that allows other transactions to read, but not write/modify the shared resource

The wormhole transaction concept will be useful in a later topic!

To grant lock or not to…

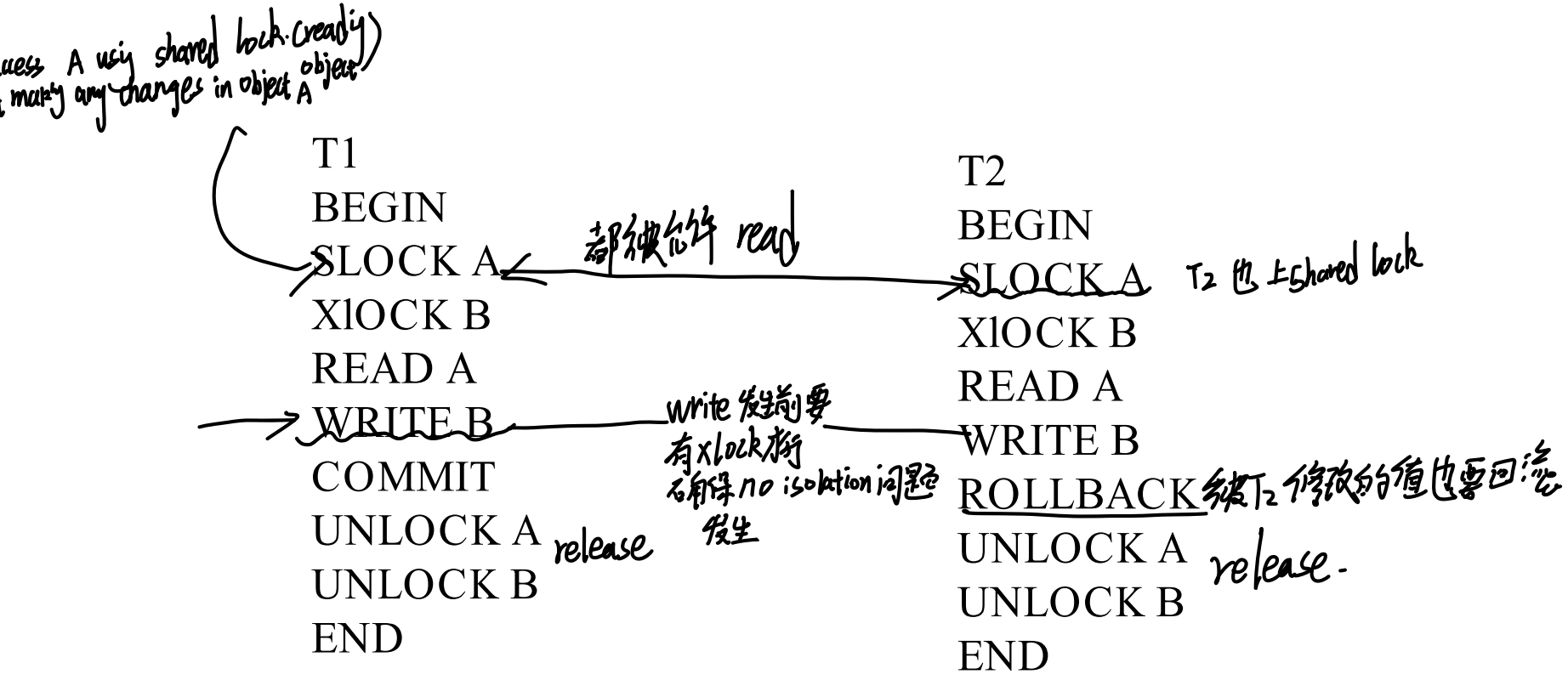
A lock on an object should not be granted to a transaction while that object is locked by another transaction in an **incompatible mode**.

Lock Compatibility Matrix

	Mode of Lock		
Current Mode	Free	Shared	Exclusive
Shared request (SLOCK) Used to block others writing/modifying	Compatible Request granted immediately Changes Mode from Free to Shared	Compatible Request granted immediately Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive
Exclusive request (XLOCK) Used to block others reading or writing/modifying	Compatible Request granted immediately Changes Mode from Free to Exclusive	Conflict Request delayed until the state becomes compatible Mode stays Shared	Conflict Request delayed until the state becomes compatible Mode stays Exclusive

When to use what type of lock

Actions in Transactions are: READ, WRITE, XLOCK, SLOCK, UNLOCK, BEGIN, COMMIT, ROLLBACK



Isolation Concepts ...

BEGIN, END , SLOCK, XLOCK can be ignored as they can be automatically inserted in terms of the corresponding operations

E.g. if a transaction ends with a COMMIT, it is replaced with:

{UNLOCK A if SLOCK A or XLOCK A appears in T for any object A}.

(That is to simply release all locks)

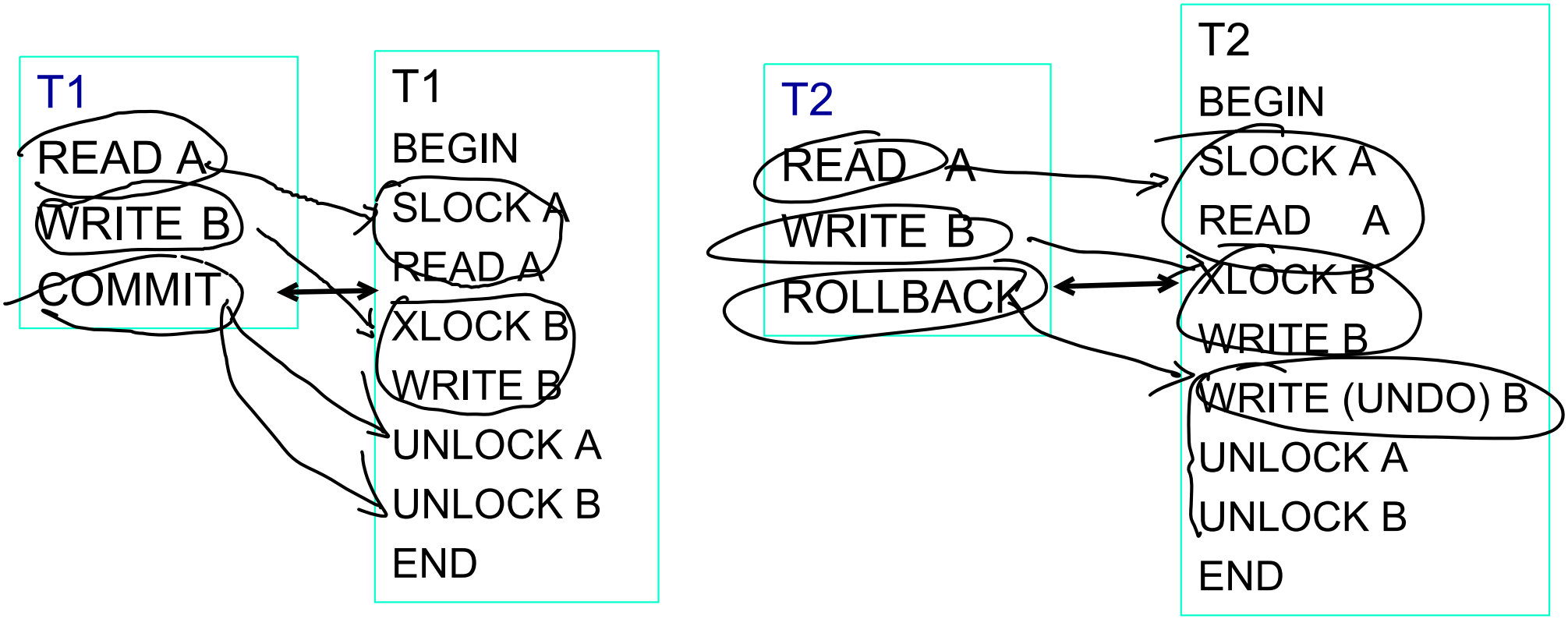
Similarly ROLLBACK can be replaced by

{WRITE(UNDO) A if WRITE A appears in T for any object A}

{ UNLOCK A if SLOCK A or XLOCK A appears in T for any object A}.

Isolation Concepts ...

We can replace previous transaction sequences by:



Isolation Concepts ...

well-formed transaction: ①R, Read之前有Slock ②R或write之前有Xlock. ③且Unlock前在lock.

Well-formed transactions: A transaction is well formed if all READ, WRITE and UNLOCK operations are covered by appropriate LOCK operations

Two phase transactions: A transaction is two phased if all LOCK operations precede all its UNLOCK operations.

all lock要在all unlock前

所有lock操作要在unlock前

objA — lock
objB — lock

√ ⇒

objA — lock
objA — unlock
objB — lock
objB — unlock.

unlock在lock前3(不对)

X

not a two phase transaction

objA — unlock
objB — unlock

Isolation Theorems

Summary:

A transactions is a sequence of READ, WRITE, SLOCK, XLOCK actions on objects ending with COMMIT or ROLLBACK.

A transaction is well formed if each READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation.

A history is legal if does not grant conflicting grants.

A transaction is two phase if its all lock operations precede its unlock operations.

Isolation Theorems

Locking theorem: If all transactions are well formed (READ, WRITE and UNLOCK operation is covered earlier by a corresponding lock operation) and two-phased (locks are released only at the end), then any legal (does not grant conflicting grants) history will be isolated.

Locking theorem (Converse): If a transaction is not well formed or is not two-phase, then it is possible to write another transaction such that it is a wormhole.

Rollback theorem: An update transaction that does an UNLOCK and then does a ROLLBACK is not two phase.

When a transaction unlocks (releases) a resource before making a final decision (commit or roll back), it leaves the possibility open for other transactions to access and modify that object. This disrupts the isolation property because other transactions could see intermediate potentially inconsistent result of data, which goes against the ACID properties.

Degrees of Isolation

Degree 3: A Three degree isolated Transaction has no lost updates, and has repeatable reads. This is "true" isolation.

Lock protocol is two phase and well formed.

It is sensitive to the following conflicts:

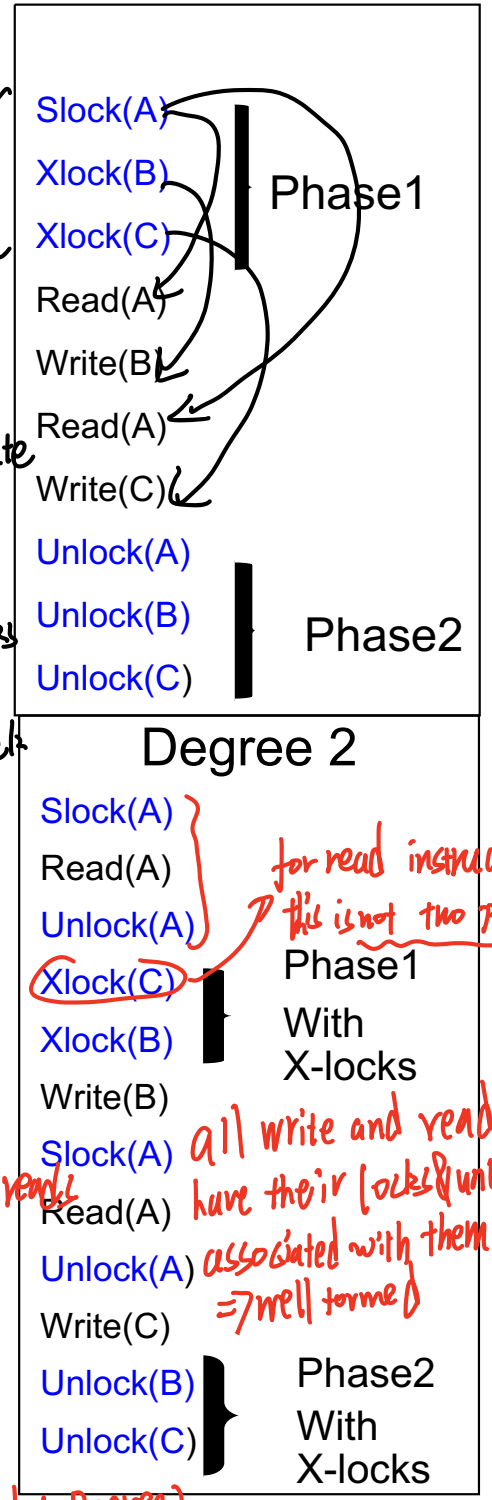
write->write; write ->read; read->write

Degree 2: A Two degree isolated transaction has no lost updates and no dirty reads.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to Reads and writes. (May have Non repeatable reads.)

It is sensitive to the following conflicts:

write->write; write ->read;



Degree 1: A One degree isolation has no lost updates.

Lock protocol is two phase with respect to exclusive locks and well formed with respect to writes.

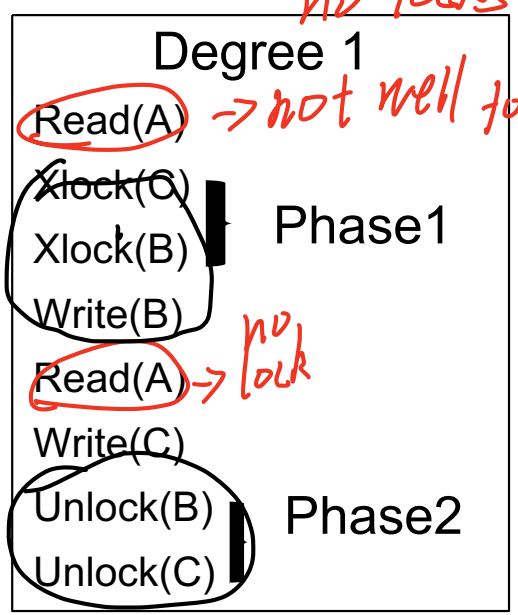
It is sensitive the following conflicts:

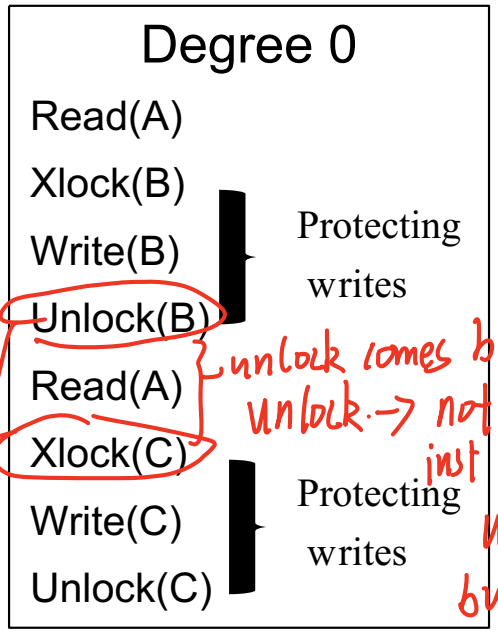
write->write;

Degree 0 : A Zero degree transaction does not overwrite another transactions dirty data if the other transaction is at least One degree.

Lock protocol is well-formed with respect to writes.

It ignores all conflicts.???





read-write ? Read(A) → Xlock(A) write(A) unlock(A) → 冲突

write-read ? Xlock(A) write(A) unlock(A) → read(A) → 冲突

write-write ? Xlock(A) write. = Xlock(A)

unlock