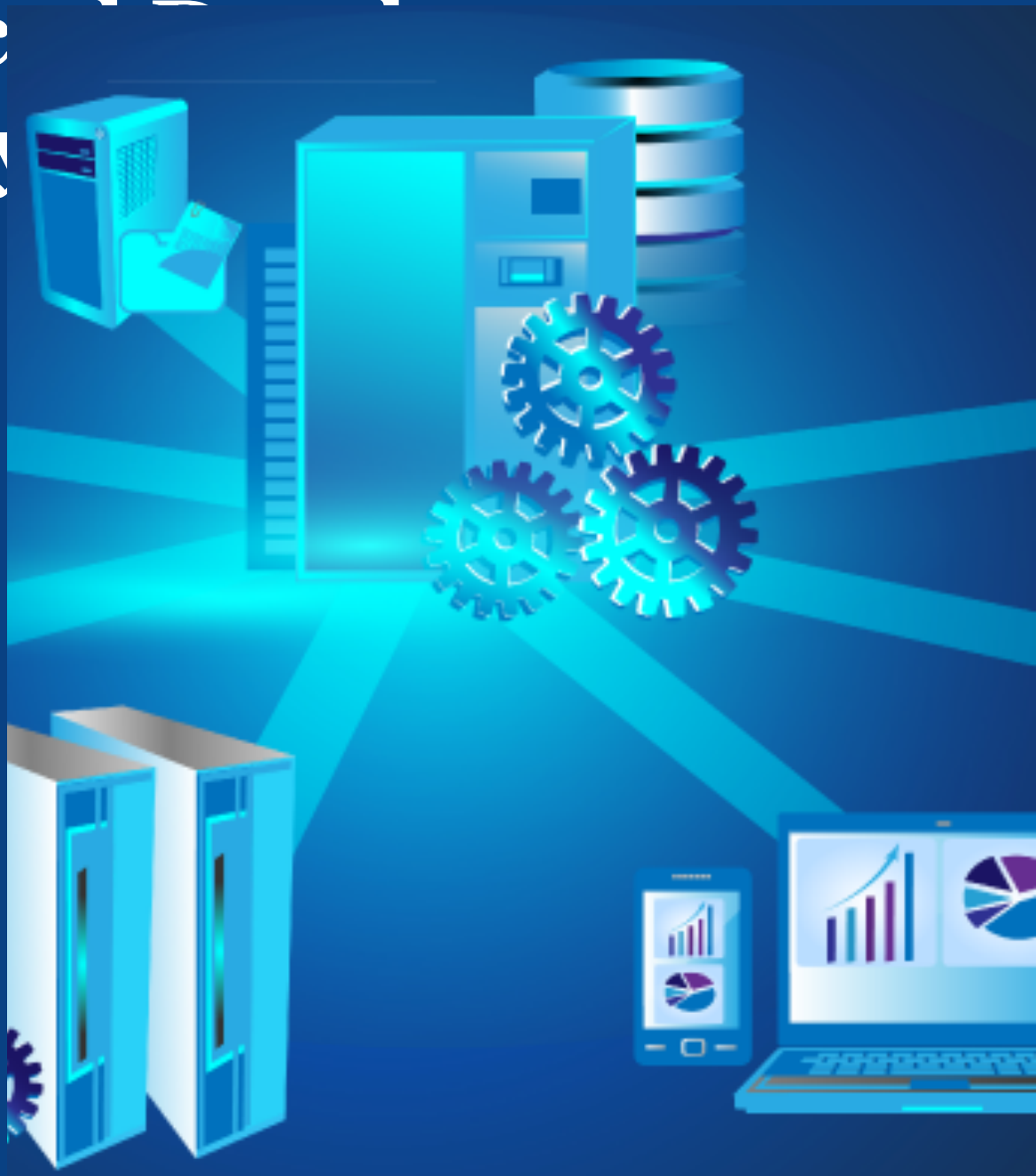
COMP90050: Advanc… Sy…



Lecturer: Farhana Choudhury (PhD)

Other locks

Week 7

Concurrent transactions – Conflicts and performance issues

Multiple concurrently running transactions may cause conflicts

- Still we try to allow concurrent runs as much as possible for a better performance, while avoiding conflicts as much as possible

A new solution:

Use granular locks - we need to build some hierarchy, then locks can be taken at any level, which will automatically grant the locks on its descendants.

Granularity Of Locks

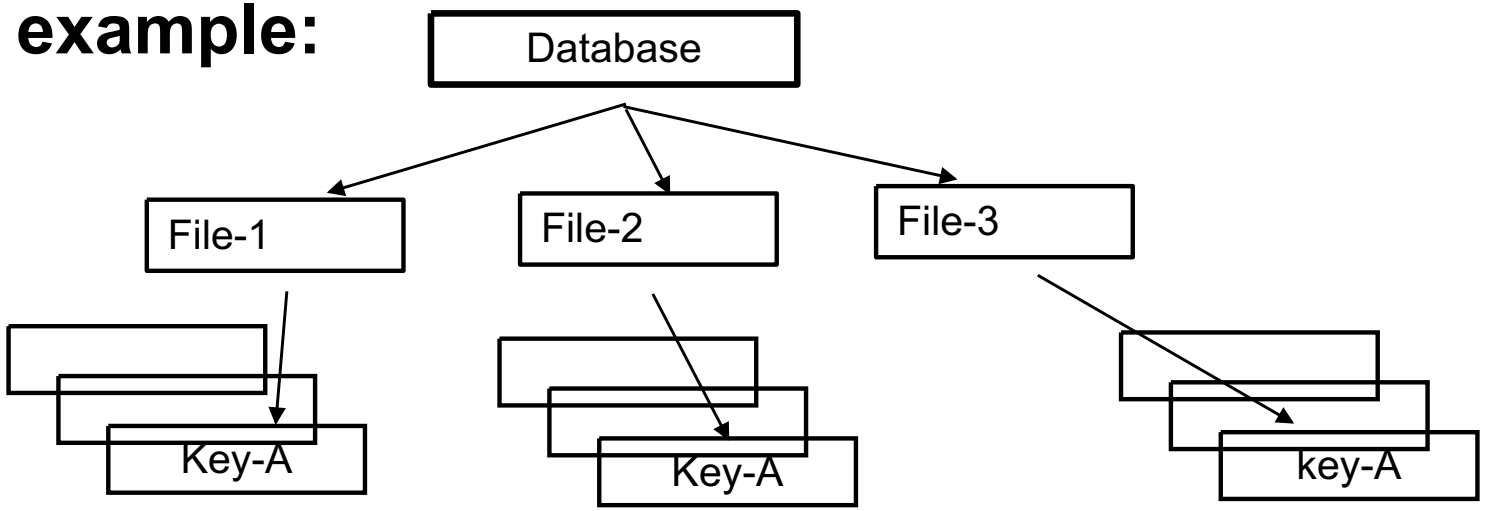
Idea:

Pick a set of column values (predicates).

They form a graph/tree structure.

Lock the nodes in this graph/tree

Simple example:



It allows locking of whole DB, whole file, or just one key value.

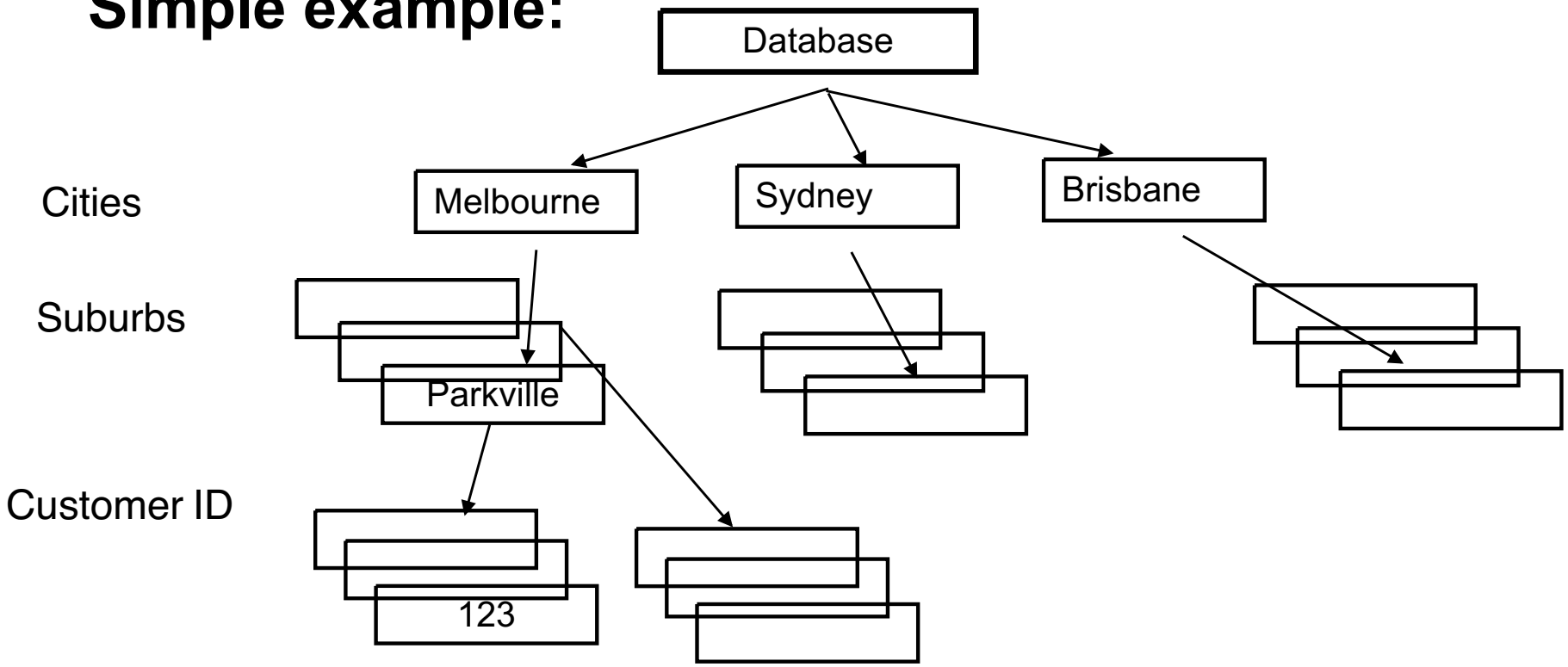
Granularity Of Locks

Example of predicates

Pick a set of column values (predicates).

They form a graph/tree structure.

Simple example:



Granularity of locks

Lock the whole DB – less conflicts, but poor performance

Lock at individual records level – more locks, better performance

Granularity of locks

Lock the whole DB – less conflicts, but poor performance

Lock at individual records level – more locks, better performance

How can we allow both granularities?

Intention mode locks on coarse granules.

+ granted

– delayed

Compatibility Matrix

Mode	Free	I (Intent)	S (Share)	X (Exclusive)
I	+ (I)	+ (I)	- (S)	- (X)
S	+ (S)	- (I)	+ (S)	- (X)
X	+ (X)	- (I)	- (S)	- (X)

Actual granular locks in practice

X eXclusive lock

S Shared lock

U Update lock -- Intention to update in the future

IS Intent to set Shared locks at finer granularity

IX Intent to set shared or eXclusive locks at finer granularity

SIX a coarse granularity Shared lock with an Intent to set finer granularity eXclusive locks

Source: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15

Isolation concepts ...

Acquire locks from root to leaf. 锁自上而下

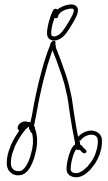
Release locks from leaf to root. 放锁自下而上

IS: intend to set finer S locks

IX: intend to set finer S or X locks

SIX: S + IX

intention to modify later



To acquire an S mode or IS mode lock on a non-root node, one parent must be held in IS mode or higher (one of {IS,IX,S,SIX,U,X}).

某一节点来一个lock，父节点必须有IS或更高级的mode

To acquire an X, U, SIX, or IX mode lock on a non-root node, all parents must be held in IX mode or higher (one of {IX,SIX,U,X}).

Isolation concepts ...
Tree locking and Intent Lock Modes

None : no lock is taken all requests are granted.

IS (intention to have shared lock at finer level) allows IS and S mode locks at finer granularity and prevents others from holding X on this node.

IX (intention to have exclusive lock at finer level) allows to set IS, IX, S, SIX, U and X mode locks at finer granularity and prevents others holding S, SIX, X, U on this node.

S (shared) allows read authority to the node and its descendants at a finer granularity and prevents others holding IX, X, SIX on this node.

一个节点 拿35 后我能持有 IX、X、SIX

Isolation concepts ...

SIX (share and intension exclusive) allows reads to the node and its descendants as in IS and prevents others holding X, U, IX, SIX, S on this node or its descendants but allows the holder IX, U, and X mode locks at finer granularity. SIX = S + IX

U (Update lock) allows read to the node and its descendants and prevents others holding X, U, SIX, IX and IS locks on this node or its descendants.

prevent to hold any other typers of locks

X (exclusive lock) allows writes to the node and prevents others holding X, U, S, SIX, IX locks on this node and all its descendants.

Compatibility Mode of Granular Locks

Current	None	IS	IX	S	SIX	U	X
Request	+|- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)

if there's no lock then the shared lock request will be granted

if it's currently holding a intend to shared lock it will also be granted.

it's ok for another transaction to read only

shared mode lock → another transaction is also allowed to read.

shared request from these mode cannot be granted.

现在是 None → 然后request X → can be granted，但是如果现在是其他的话，都不行。

Update mode Locks – why necessary

```
T1:
SLock A
Read A
If (A== 3)
{
% Upgrading Slock to Xlock
        Xlock A
        Write A
}
Unlock A
```

```
T2:
SLock A
Read A
If (A==3)
{
% Upgrading Slock to Xlock
        Xlock A
        Write A
}
Unlock A
```

```
T3:
SLock A
Read A
Unlock A
```

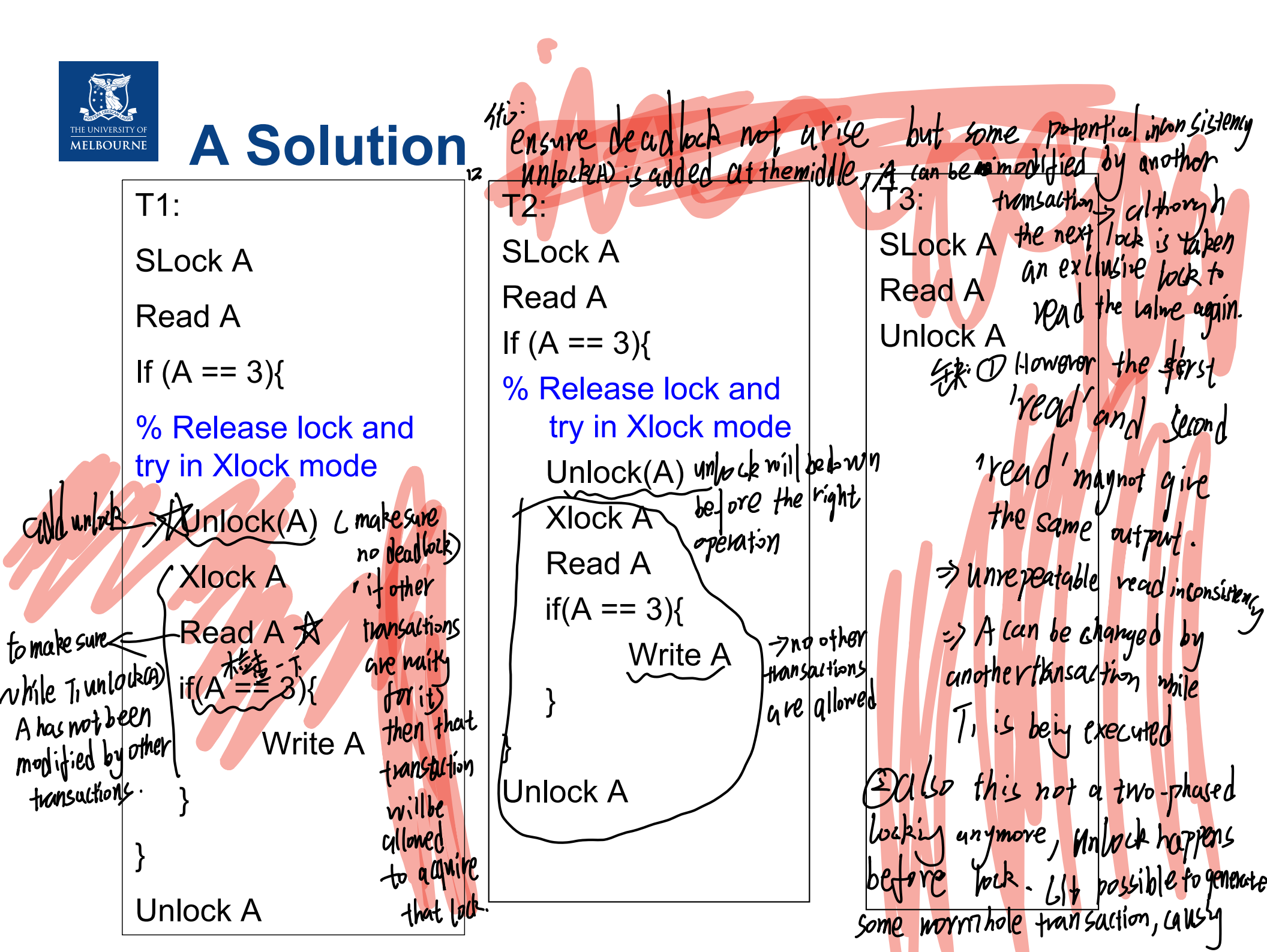
This can cause very simple deadlock. As per Jim Gray virtually all deadlocks in System R were found to be of this form!

A Solution

```
T1:
SLock A
Read A
If (A == 3){
% Release lock and
try in Xlock mode
Unlock(A)
Xlock A
Read A
if(A == 3){
Write A
}
}
Unlock A
```

```
T2:
SLock A
Read A
If (A == 3){
% Release lock and
    try in Xlock mode
Unlock(A)
Xlock A
Read A
if(A == 3){
Write A
}
}
Unlock A
```

```
T3:
SLock A
Read A
Unlock A
```



Update mode Locks

T1:
Ulock A
Read A
If (A== 3){
Xlock A
Write A
}
Unlock A

T2:
Ulock A
Read A
If (A==3){
Xlock A
Write A
}
Unlock A

T3:
SLock A
Read A
Unlock A

Granting the first Ulock excludes granting any other subsequent locks, and thus eliminates very simple dead locks in addition also reduces starvation caused by subsequent shared lock requests by not granting them immediately.

does not occur.

request can be granted concurrently. For T_3 gets shared lock first, then at the same time, T_1 or T_2 can be granted an update lock on the same object A while the update is granted (T_1或T_2), then still T_3 is allowed to read the same object A.

Also at the same time, if there is another transaction, which just needs to read object A. That transaction gets shared lock is granted first. Still that Transaction (T_3) can run concurrently, while later another transaction gets (T_3已经拿到) and update lock on the same object.

☆update mode locks更新用☆.

Optimistic locking

When conflicts are rare, transactions can execute operations without managing locks and without waiting for locks - higher throughput

- Use data without locks
- Before committing, each transaction verifies that no other transaction has modified the data (by taking appropriate locks) – **duration of locks are very short**
- If any conflict found, the transaction repeats the attempt
- If no conflict, make changes and commit

Optimistic locking

```
Read A into A1
Read B into B1
Read C into C1
Loop: Compute new values based on A1 and B1
% Start taking locks
      Slock A; Read A into A2
      Slock B; Read B into B2
      Xlock C; Read C into C2
      if (A1 == A2 & B1 == B2 & C1 == C2)
            Write  new value into C
            commit
            Unlock A, B and C
      else   % read data is changed
            A1 = A2
            B1 = B2
            C1 = C2
            unlock A ,B and C
            goto Loop
end
```

read phase

validation phase

two phases

write phase

Once the condition is true – it is effectively 2 phase locking but duration of locking is very short but can force many repeated attempts due to failure of the condition.

if conflicts are rare, this technique is very efficient and performance

However, if some values always be changed by many transactions, the conflicts will arise, in that situation many repeated attemps

will better compare with deadlock — in the loop may need to be required

Snapshot Isolation

```
Loop:   Read C into C1
        Read D into D1
        Read A into A1
        Read B into B1
Compute new values based on A1 and B1
% Start taking locks on records that need modification.
        Let new value for C is C3 and for D is D3
                Xlock C
                Xlock D
                Read C into C2
                Read D into D2
                if (C1 = = C2 & D1 = = D2)
        %    first writer commits
                        write C3 to C
                        write D3 to D
                        commit
                        unlock(C and D)
                else    % not first modifier
                        C1 = C2
                        D1 = D2
                        unlock(C and D)
                        goto Loop
        end
```

Snapshot Isolation method is used in Oracle but it will not guarantee Serializability. However, its transaction throughput is very high compared to two phase locking scheme.

CD要被修改 AB不改.

read without taking any locks

修改的上锁，不修改的只读不锁

判断是否修改，不修改的AB不判断.

若一致则

提交

解

若被修改了，说明发生冲突，用新值再run一遍loop

Cons: it does not guarantee consistency (Serializability)
If execute transactions in a serial order, final output can be different if run concurrently

Pros { higher throughput: less # of locks, no locks on the unchanged value A, B
duration is very short: taken right before the 'commit' takenplace.

Two phase locking Transaction

Integrity constraint A+B >= 0; A = 100; B = 100;

T1:

Lock(X,A)

Lock(S,B)

Read A to A1;

Read B to B1;

A1 = A1 -200;

if (A1+ B1 >= 0)

  Write A1 to A

  Commit

else abort

end

Unlock (all locks)

T2:

Lock(S,A)

Lock(X,B)

Read A to A1;

Read B to B1;

B1 = B1 -200;

If (A1+ B1 >= 0)

  Write B1 to B

  Commit

else abort

end

Unlock (all locks)

Only one transaction can commit.

Snapshot Isolation Transaction

Integrity constraint A+B >= 0; A = 100; B = 100;

T1:

```
Loop: Read A to A1;
      Read B to B1;
      A3 = A1 -200;
      Lock(X, A)
      Read A to A2
      if (A1 != A2)
          Unlock(A)
          goto Loop
      elseif (A3+B1 >= 0)
          Write A3 to A
          Commit
      else abort
      Unlock (all locks)
```

T2:

```
Loop: Read A to A1;
      Read B to B1;
      B3 = B1 -200;
      Lock(X, B)
      Read B to B2
      if (B1 != B2)
          Unlock(B)
          goto Loop
      elseif (A1+ B3 >= 0)
          Write B3 to B
          Commit
      else abort
      Unlock (all locks)
```

One or both transactions can commit but when both are committed, it is not serializable as only one should be able to commit.

it's possible that both transactions commit but when both transaction commit, the final output of database will not be the same as a serial execution of this two transaction. (在Snapshot下, A=-100, B=-100).
(可以一个一个来, 但 but cannot do both in balance)

Time stamping → no lock involved.

each action has an associate time stamp

These are a special case of optimistic concurrency control. At commit, time stamps are examined. If time stamp is more recent than the transaction read time the transaction is aborted. 检查时间戳, 看有没有更新的修改 这个技术就提交.

Time Domain Versioning

Data is never overwritten a new version is created on update.

<o,<V1, [t1, t2)>, <V2, [t2,t3)>, <V3,[t3,*)>

T1 T2 T3 时间段 last update → 持续至现在

At the commit time, the system validates all the transaction's updates and writes updates to durable media. This model of computation unifies concurrency, recovery and time domain addressing.

crash recovery, query for transaction (检查什么时候干了什么)

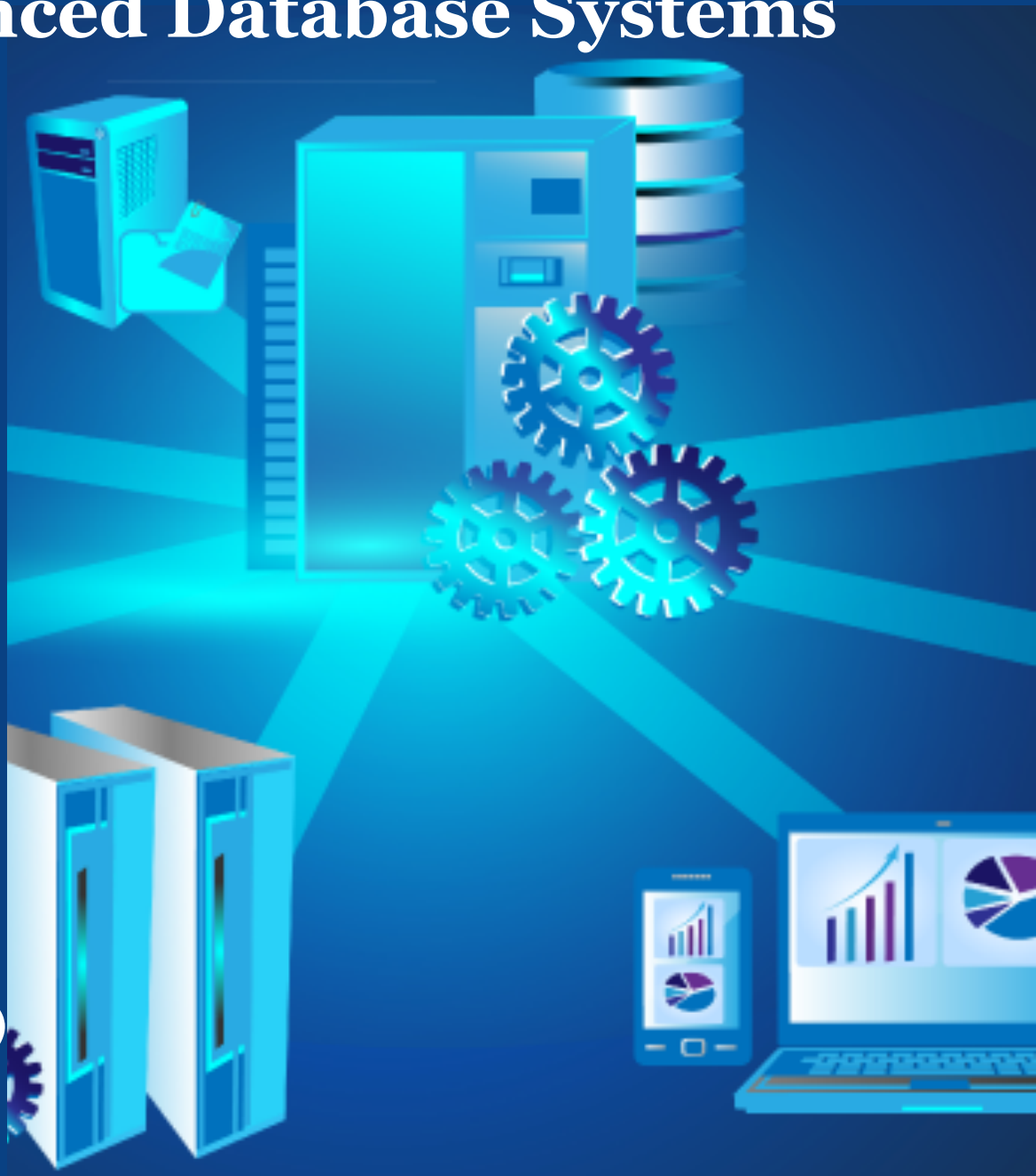
Time stamping

T1	T2
select average (salary)	update employee
from employee	set salary =
	salary*1.1
	where salary < $40000

If transaction T1 commences first and holds a read lock on a employee record with salary < $40000, T2 will be delayed until T1 finishes. But with time stamps T2 does not have to wait for T1 to finish!

THE UNIVERSITY OF MELBOURNE

COMP90050 Advanced Database Systems



Semester 2, 2024

Lecturer: Farhana Choudhury (PhD)

Live lecture – Week 7

Concurrent transactions – Conflicts and performance issues

Multiple concurrently running transactions may cause conflicts

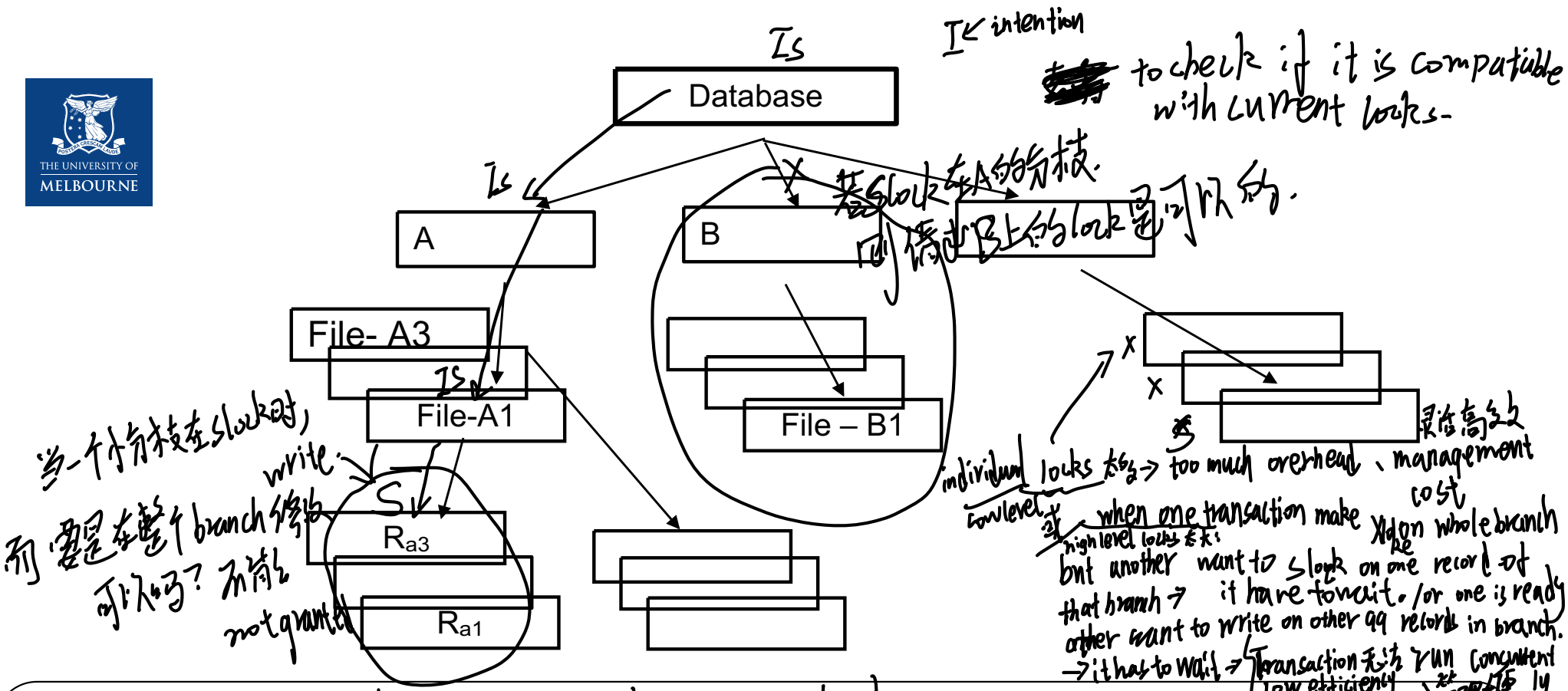
- Still we try to allow concurrent runs as much as possible for a better performance, while avoiding conflicts as much as possible

Solutions:

1. Use granular locks

idea: have some hierachy in database → transaction take lock only in part of relation, does not interfere with other transaction taky locks on other parts.

Even multiple transactions can take locks on the same hierachy, as long ~~there~~ they locks are compatible.



Rules:

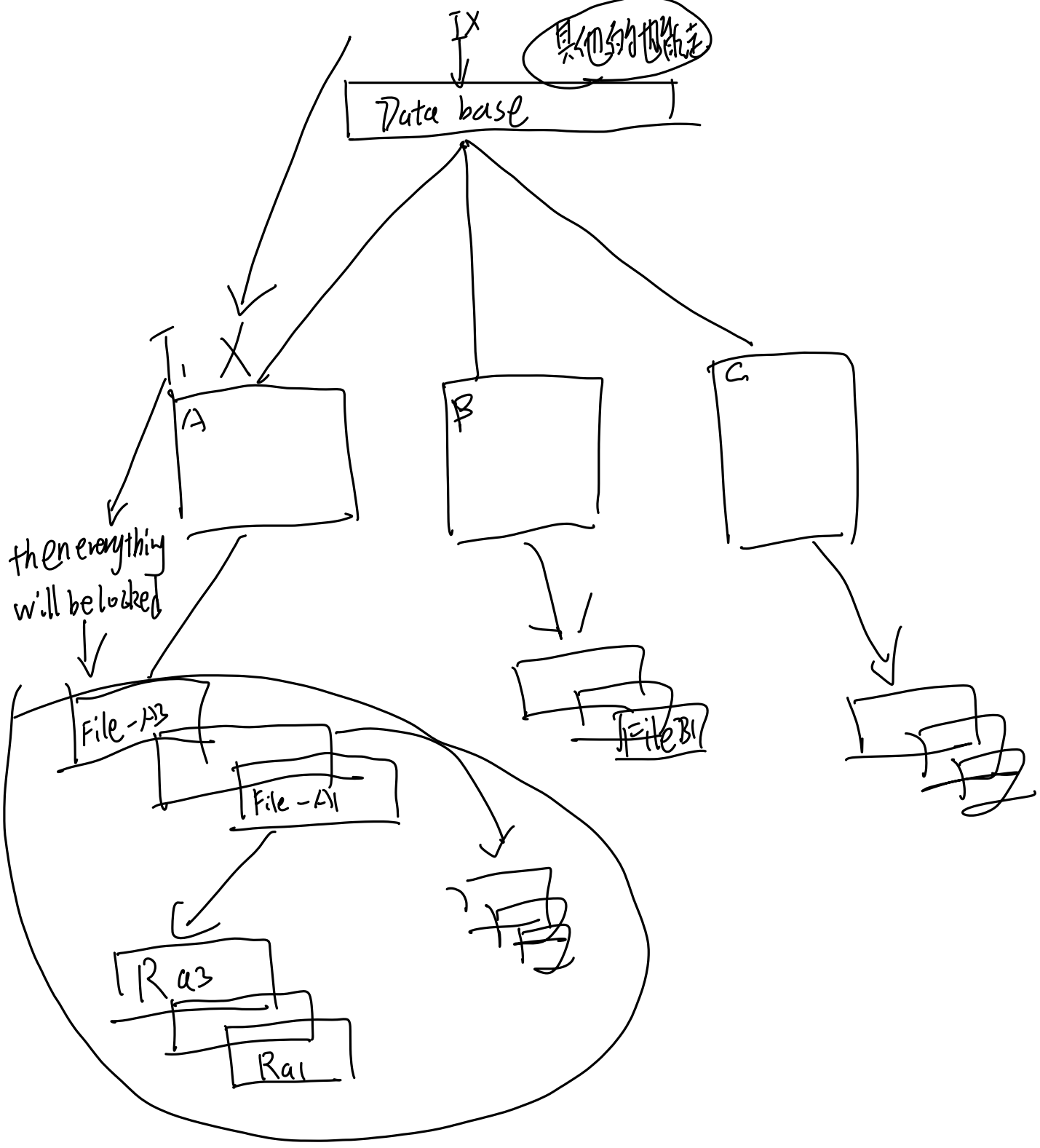
Lock root to leaf

To set X or S below, get IX or IS above respectively (or higher mode)

If no conflict/locks are compatible, multiple transactions run concurrently

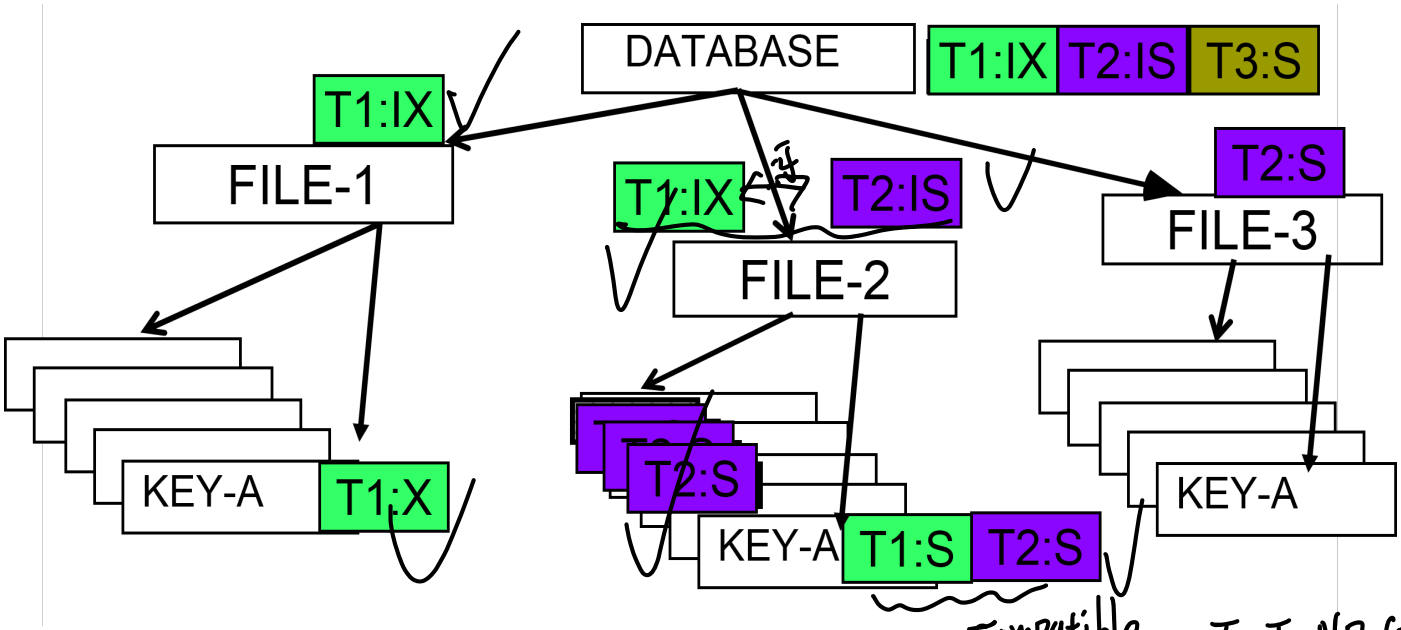
If T1 reads record R_{a1} then T1 needs to lock the database, node A , and File – A1 in IS mode (or higher mode). Finally, it needs to lock R_{a1} in S mode.

If T2 modifies record R_{a3} then it can do so after locking the database, node A, and File – A1 in IX mode. Finally, it needs to lock the R_{a3} in X mode.


IX
Data base
T1 X
A
B
C
then everything will be locked
File-A3
File-A1
Ra3
Ra1
FileB1

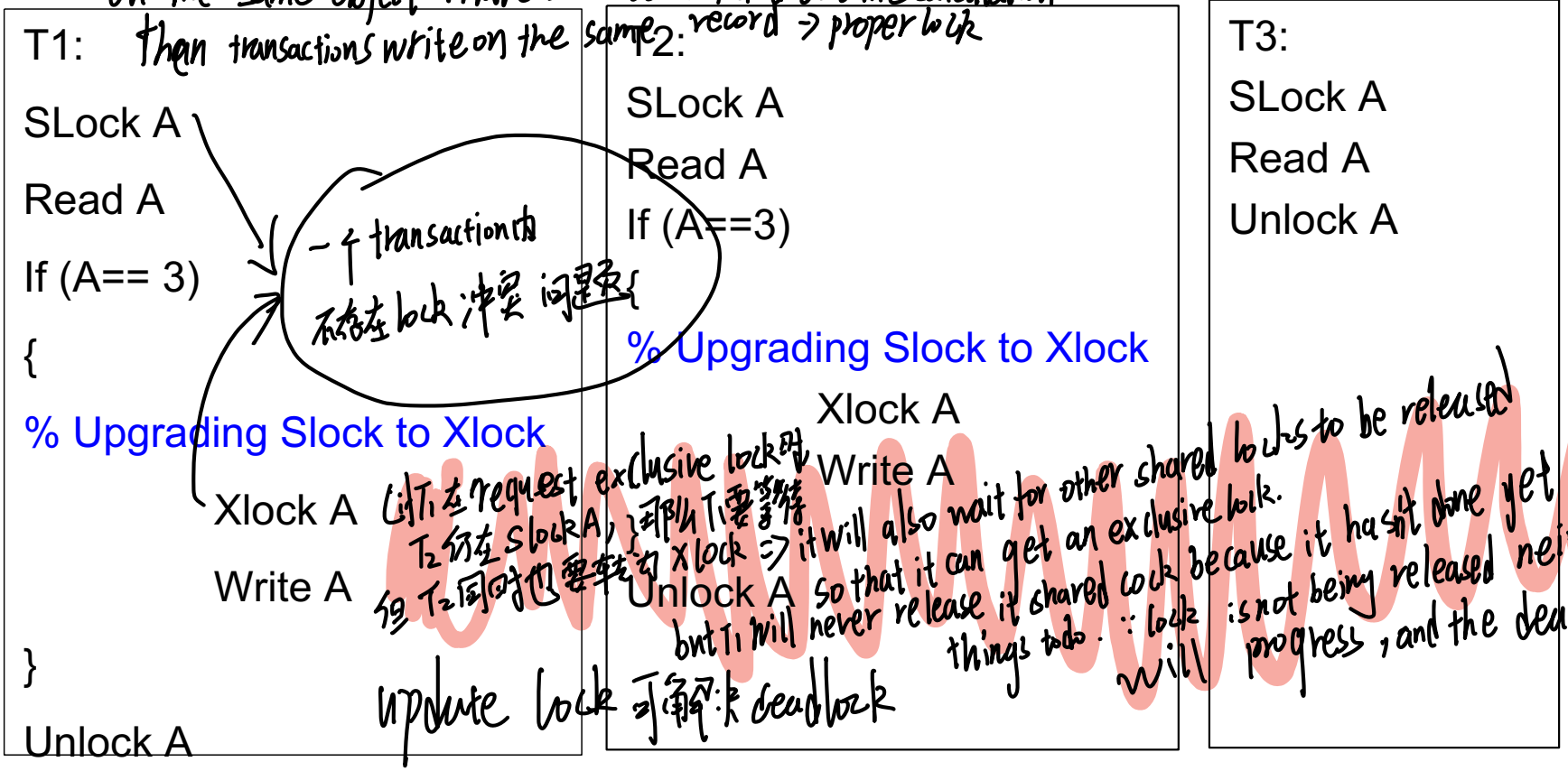
Source: https://docs.microsoft.com/en-us/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15

Compatibility Mode of Granular Locks							
Current	None	IS	IX	S	SIX	U	X
Request	+|- (Next mode) + granted / - delayed						
IS	+(IS)	+(IS)	+(IX)	+(S)	+(SIX)	-(U)	-(X)
IX	+(IX)	+(IX)	+(IX)	-(S)	-(SIX)	-(U)	-(X)
S	+(S)	+(S)	-(IX)	+(S)	-(SIX)	-(U)	-(X)
SIX	+(SIX)	+(SIX)	-(IX)	-(S)	-(SIX)	-(U)	-(X)
U	+(U)	+(U)	-(IX)	+(U)	-(SIX)	-(U)	-(X)
X	+(X)	-(IS)	-(IX)	-(S)	-(SIX)	-(U)	-(X)



Tutorial: Which transactions can run concurrently and which transactions need to wait if the transactions arrive in the order T1-T2-T3?

Update mode Locks – why necessary

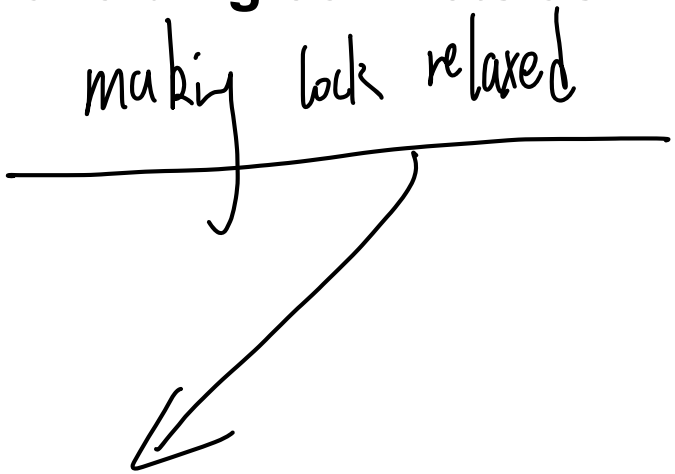


This can cause very simple deadlock. As per Jim Gray virtually all deadlocks in System R were found to be of this form!

Concurrent transactions – Conflicts and performance issues

Multiple concurrently running transactions may cause conflicts

- Still we try to allow concurrent runs as much as possible for a better performance, while avoiding conflicts as much as possible



Solutions:

1. Use granular locks
2. Optimistic and snapshot isolation

Optimistic locking

take read locks before commit

When conflicts are rare, transactions can execute operations without managing locks and without waiting for locks - higher throughput

- Use data without locks

read 时无lock.

- Before committing, each transaction verifies that no other transaction has modified any data that it needs (by taking appropriate locks) – **duration of locks are very short**

- If any conflict found, the transaction repeats the attempt

- If no conflict, make changes and commit

because locks are taken just before commit to do some quick checks, and do necessary write operations, the durations of locks are very short. So sometimes if transaction needs to do calculate or make decisions... those operations will be done without any locks and it will allow other transactions to read write

or run concurrently.

Snapshot Isolation

```
                Read C into C1
                Read D into D1
Loop:
                Read A into A1
                Read B into B1
Compute new values based on A1 and B1
% Start taking locks on records that need modification.
                Let new value for C is C3 and for D is D3
                        Xlock C
                        Xlock D
                        Read C into C2
                        Read D into D2
                        if (C1 = = C2 & D1 = = D2)
                %       first writer commits
                                write C3 to C
                                write D3 to D
                                commit
                                unlock(C and D)
                        else    % not first modifier
                                C1 = C2
                        D1 = D2
                                unlock(C and D)
                                goto Loop
        end
```

Snapshot Isolation method is used in Oracle but it will not guarantee Serializability. However, its transaction throughput is very high compared to two phase locking scheme.

Isolation Concepts ...

SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED |

READ COMMITTED| REPEATABLE READ | SERIALIZABLE}

Slight difference with the four degrees of isolation

- SERIALIZABLE – degree 3
- REPEATABLE READ – like degree 3, but other transactions can insert new rows
- READ COMMITTED – prevents dirty reads like degree 2*
- READ UNCOMMITTED – Like degree 0

*Options can also be paired with SNAPSHOT on/off

Source: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15

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

$T_1 O_1 T_3 \quad T_3 O_1 T_1$

$T_3 O_3 T_2 \quad T_2 O_2 T_1 \quad T_1 O_2 T_2$

Isolation Concepts ...

SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | READ COMMITTED| REPEATABLE READ | SERIALIZABLE}

READ_COMMITTED_SNAPSHOT can be set on or off

If on – Shared locks are not used for reading

- Read committed with READ_COMMITTED_SNAPSHOT off – degree 2

- Read committed with READ_COMMITTED_SNAPSHOT on – *like degree 1

T_1 01 T_3 T_3 01 T_1

T_3 03 T_2

Isolation Concepts ...

What kind of applications use relaxed isolation?

- Have you experienced any inconsistency in reading values as users?

Time for a poll - Pollev.com/farhanachoud585

Other types of locks

Phantoms: They commonly occur when a transaction lock records/table but another transaction adds new records in that table

So reading the same table twice by one transaction presents two different sets of records as the other has inserted some in the meantime

Predicate locks solve this problem. Rather than locking records, lock based on condition

Select * from Employee where salary > 80000

Another transaction will be able to insert new records in Employee table as long as the salary is not within this predicate range

Page Locks can solve this problem too. In this case, the storage should be organized based on values of attributes.

phantom: e.g when subject coordinator wants to send emails to all students who are in the enroll list, when she is going to compose the email and send to the list, some more student just finish the enrollment (In this case, I miss these new student if the system weren't phantom lock, and I have to manually check all list to find newly added students.) if I use phantom, it can prevent any other transaction from inserts records

What kind of applications use relaxed isolation? and achieve current consistency

Work on a certain set, if the set changes (add new records into it) ⇒ it's not desirable of our database.

- Have you experienced any inconsistency in reading values as users?

condition.

websit ⇒ if one user is using ⇒ the web is locked ⇒ others have not been able to use it
not good experience

e.g. Adding something to shopping cart, everybody is able to add this item into cart).
But only one of them will be available.

what type of locks it might be? ☆☆

It is 1, item available; when 0 unavailable.

① while others putting this item into their cart, when you go to actual shopping cart payment.
multiple users can add items into their shopping cart concurrently ⇒ but cannot pay at the end
if others have bought that previously ⇒ then it (system) informs you this item is not available
and shopping cart gets updated.

② even sometimes ⇒ paid for it already ⇒ get email saying sorry we have to return money.

ready is the one result

① ② have inconsistency ⇒ when we actually write on it ⇒ it shows us the value has been changed by someone.
but (want to)

Tickets, hotel booking ....

these scenarios have relaxed locking ⇒ inconsistency exists

Concurrency and Isolation summary

- Why concurrency control is needed
- Concepts of isolation
  - What causes conflicts
  - Conflict types
- How to avoid conflicts (by locking)
  - Different types of locks
  - Issues caused by locks (deadlocks, convoy)

Core Concepts of Database management system

Database performance metrics

- Efficiency/speed
- Effectiveness
- Security & Reliability

Efficiency

- Hardware
  - Disks and I/O bandwidth
  - Main memory
  - Type of architecture
- Software/ DB tuning
  - Types of DB
  - Indexing
  - Query optimisation

Effectiveness

- Concurrent users
  - Users reading and writing over the same data
- Transactions
  - Required tasks are all done together

Reliability

- Crash recovery
- Fault tolerance
- Data duplication