COMP90050
Advanced Database Systems

THE UNIVERSITY OF
MELBOURNE

Winter Semester, 2023

Lecturer: Farhana Choudhury (PhD)

Live lecture – week 3

Quizzes

Quiz 3 and Quiz 4 are done!

Time to discuss the solutions

Part 1-2 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!

Embedded SQL example in C

(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;
```

Proper error handling is important!

Transaction Processing Monitor

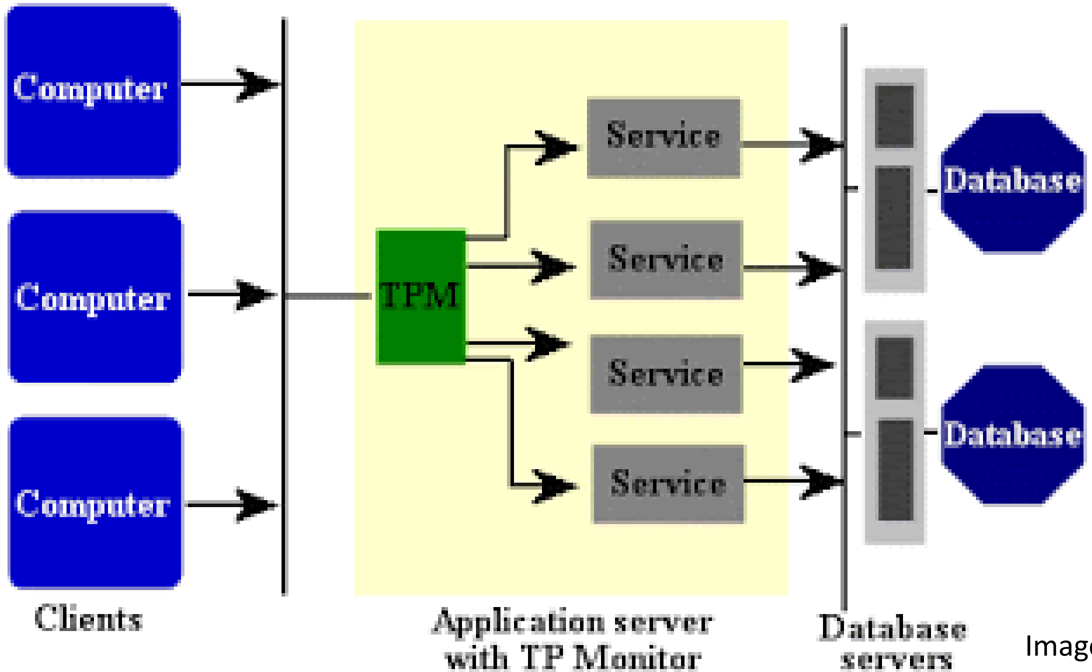


Image source: http://3.bp.blogspot.com

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?

PollEv.com/farhanachoud585

Respond at **PollEv.com/farhanachoud585**

Text **FARHANACHOUD585** to **+61 427 541 357** once to join, then text your message

Nested Transactions

BEGIN WORK
.
.
.
Invoke Sub Trans 1
.
.
Invoke Sub Trans 2
.
.
Invoke Sub Trans 3

T1
BEGIN WORK
Invoke Sub Trans 11
Invoke Sub Trans 12
END WORK

T2
BEGIN WORK
Invoke Sub Trans 21
Invoke Sub Trans 22
END WORK

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?

Part 3 topics

Concurrency problems – why we need concurrency control

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

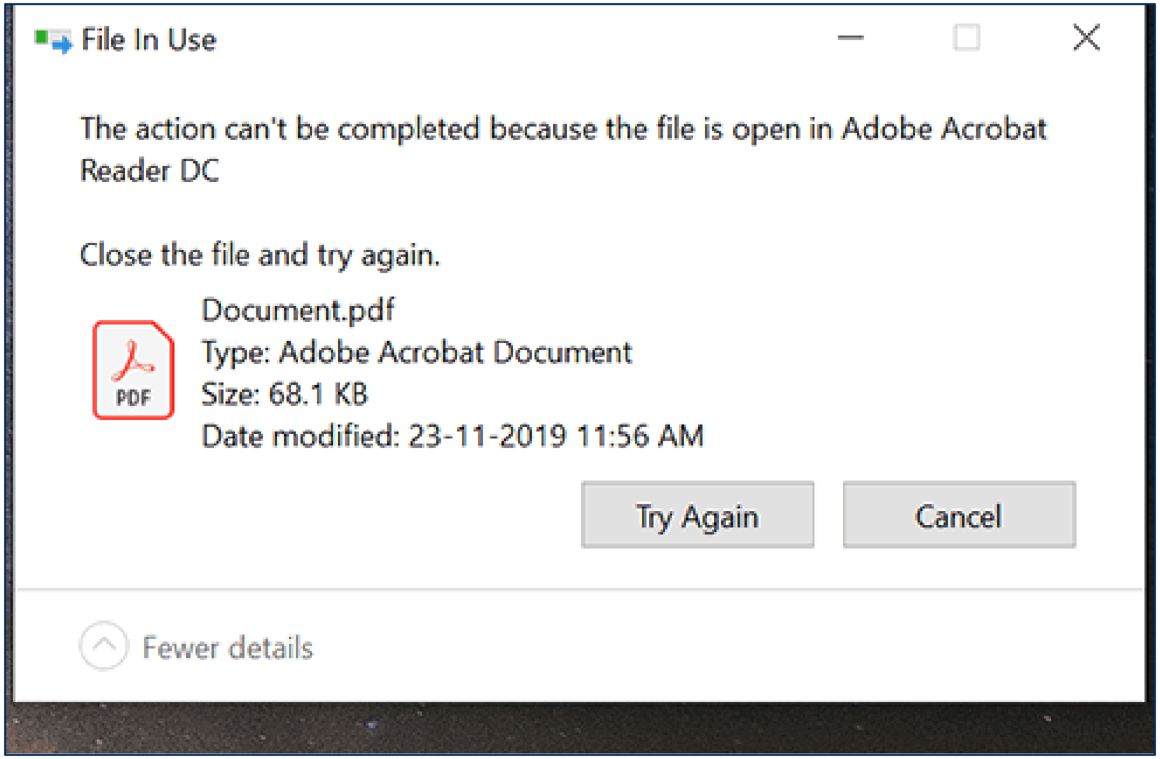
Semaphores - get lock, release lock, maintain queue of processes

Atomic operations to get and release locks

Semaphore queue, avoiding long queues

Deadlocks

Concurrency problem



Concurrent transactions can cause issues in Database – need concurrency control

Concurrency Problems

Transaction 1	Transaction 2
balance = balance + 50;	balance = balance - 110;

Account balance = 100;

Both transactions are running concurrently. What are the possible outcomes if no concurrency control in place?

Write by a transaction gets lost/overwritten

a) balance == 150

Sequence of actions

T1: Reads balance == 100

T2: Reads balance == 100

T2: Writes balance == 100-110

T1: Writes balance == 100+50

b) balance == -10

Sequence of actions

T1: Reads balance == 100

T2: Reads balance == 100

T1: Writes balance == 100+50

T2: Writes balance == 100-110
(not enough balance message)

c) balance == 40;

Sequence of actions

T1: Reads balance == 100

T1: Writes balance == 100+50

T2: Reads balance == 150

T2: Writes balance == 150-110

d) balance == 150;

Sequence of actions

T2: Reads balance == 100

T2: Writes balance == 100-110

(not enough balance message)

T1: Reads balance == 100

T1: Writes balance == 100+50

Try withdrawing money after some 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

General purpose locks - semaphores

Part 3 topics

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

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

Probability of deadlock happening increases with $O(r^4)$ with respect to the number of locks taken and $O(n^2)$ with the number of concurrent transactions and inversely proportional to $O(R^2)$ with the number of records in the database.

Concurrency problem

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

What we need to know –

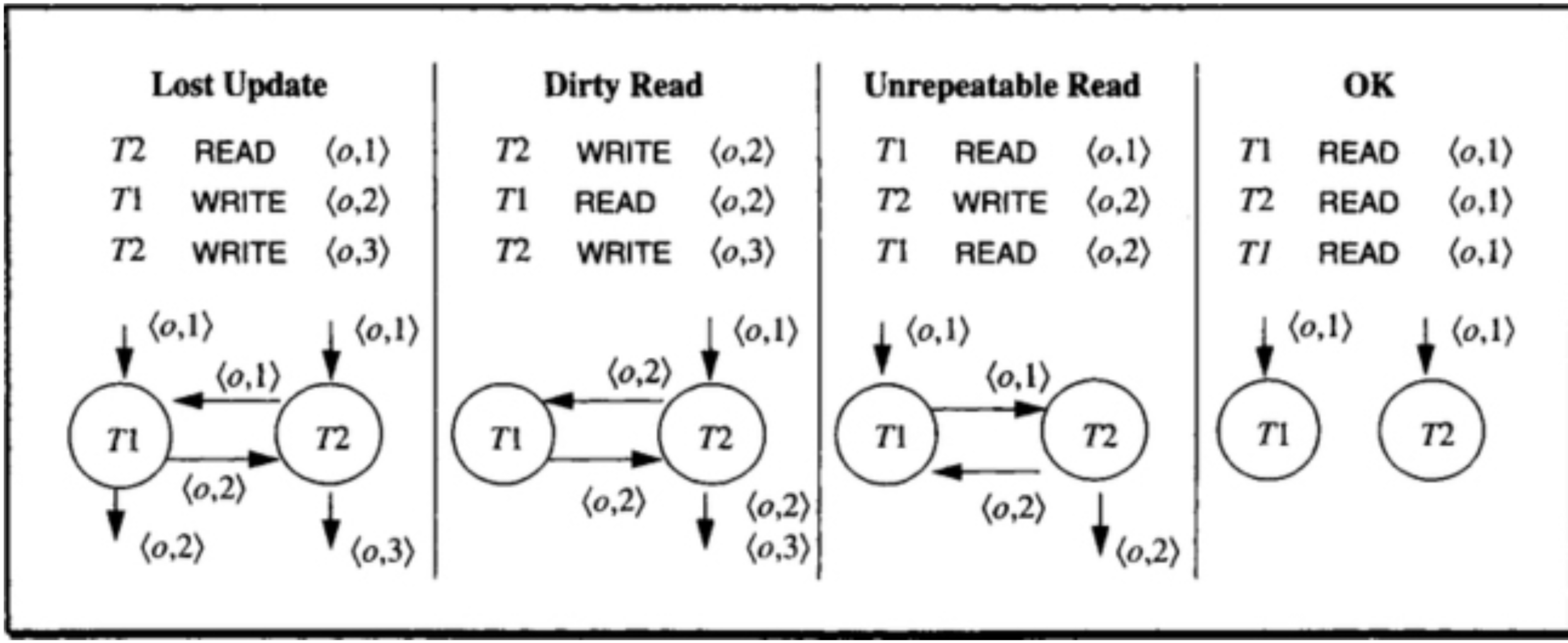
- What are the possible conflicts/dependencies

- Given a set of concurrent transactions, can we determine whether there will be any conflict or not?

- Is there any way to re-order the execution of transactions to avoid conflicts (without making any change to the intended final output/final state of the database?

Dependencies

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



Some activities !

PollEv.com/farhanachoud585

Respond at **PollEv.com/farhanachoud585**

Text **FARHANACHOUD585** to **+61 427 541 357** once to join, then text your message

Dependency relations - equivalence

The operations of concurrently running transactions can occur in any order.

Given two different order of executions, can we have some insight on the final output/state of the database?

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

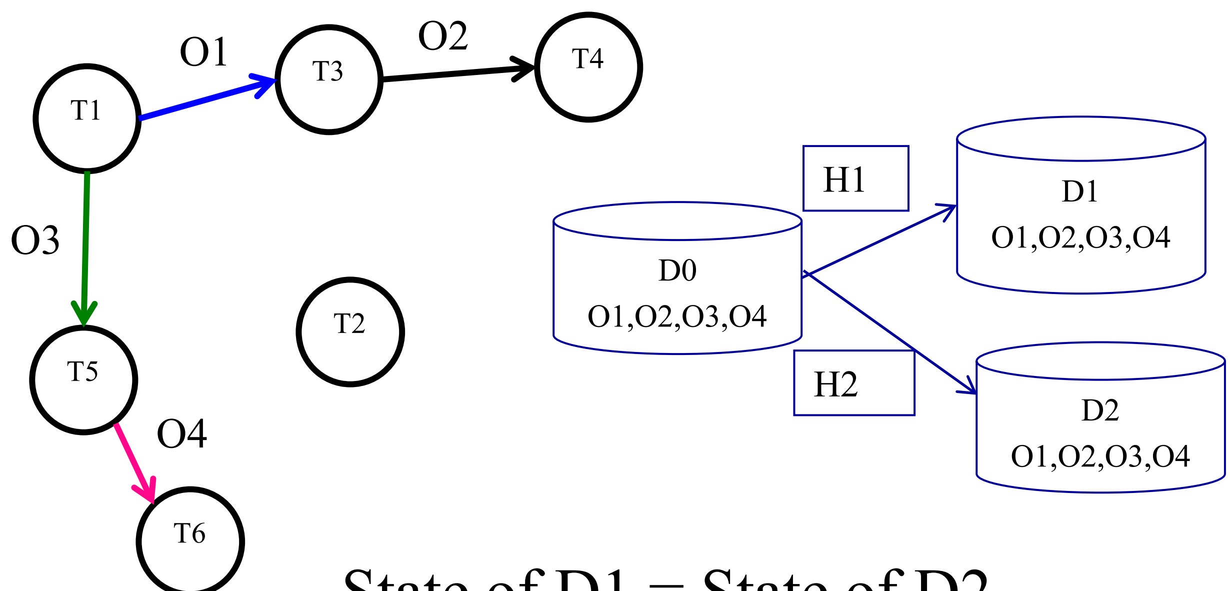
DEP(H2) = {<T1, O1,T3>, <T1,O3,T5>, <T3,O2,T4>, <T5,O4,T6> }

DEP(H1) = DEP(H2)

Dependency relations - equivalence

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> }



State of D1 = State of D2

Dependency relations - equivalence

Tutorial tasks

5. What are the dependencies in the following history (a sequence of tuples in the form (T i, Oi, T j))? Draw the dependency graph mapping to this dependency set as well.

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

Isolated history

If the transactions are running sequentially one after another (serial history) – there won't be any conflict

Can we run transactions concurrently, but still have the same final output/state of the database as if the transactions are serially executed?

A history is called isolated if it is equivalent to a serial history.

Given a history, how can we determine whether it is equivalent to a serial history? There are maximum N! possible serial executions.

Isolated history

Given a history, how can we determine whether it is equivalent to a serial history? There are maximum N! possible serial executions.

– We try to find a cycle.

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

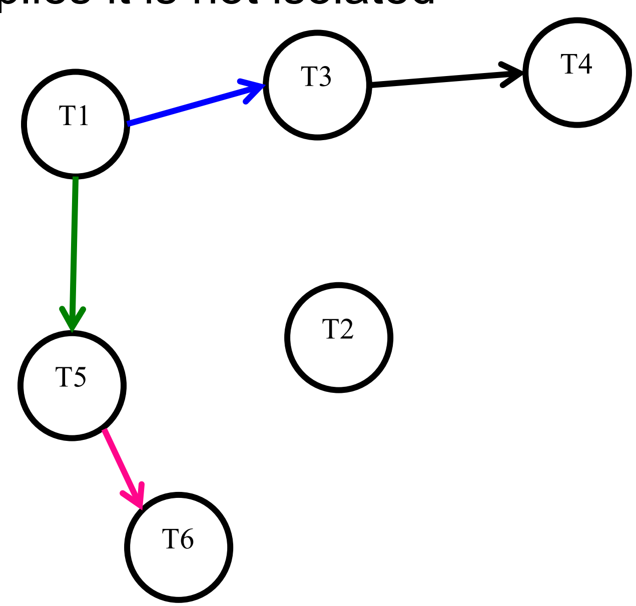
If T1 precedes T2, it is written as T1 << T2.

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

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

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

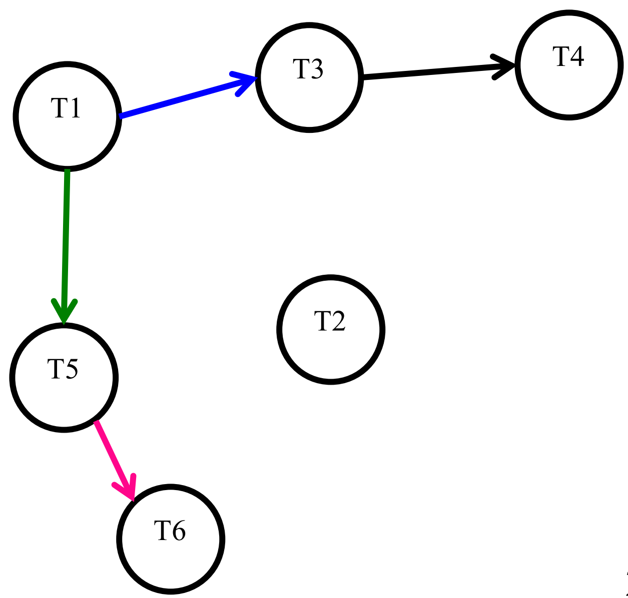
After(T3) = {T4}



Isolation Concepts ...

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.

Is there any way to re-order the execution of transactions to avoid conflicts (without making any change to the intended final output/final state of the database?



Some activities !

PollEv.com/farhanachoud585

Respond at **PollEv.com/farhanachoud585**

Text **FARHANACHOUD585** to **+61 427 541 357** once to join, then text your message

One more activity!

PollEv.com/farhanachoud585

Respond at **PollEv.com/farhanachoud585**

Text **FARHANACHOUD585** to **+61 427 541 357** once to join, then text your message

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;

Slock(A)
Xlock(B)
Xlock(C)
Phase1
Read(A)
Write(B)
Read(A)
Write(C)
Unlock(A)
Unlock(B)
Unlock(C)
Phase2

Degree 2

Slock(A)
Read(A)
Unlock(A)
Xlock(C)
Xlock(B)
Phase1 With X-locks
Write(B)
Slock(A)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)
Phase2 With X-locks

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.

Degree 1

Read(A)
Xlock(C)
Xlock(B) — Phase1
Write(B)
Read(A)
Write(C)
Unlock(B)
Unlock(C) — Phase2

Degree 0

Read(A)
Xlock(B)
Write(B) — Protecting writes
Unlock(B)
Read(A)
Xlock(C)
Write(C) — Protecting writes
Unlock(C)

Tutorial exercise

8. What degree of isolation does the following transaction provide?

Slock(A)
Xlock(B)
Read(A)
Write(B)
Read(C)
Unlock(A)
Unlock(B)

Isolation

Multiple concurrently running transactions may cause conflicts

- **Different types of conflicts**
- **Avoiding conflicts – using locks**

Now we will see –

- More types of locks
- Relaxed isolation – for better performance

Isolation Concepts ...

In SQL2 one can declare isolation level as follows:

```
SET TRANSACTION ISOLATION LEVEL

{READ UNCOMMITED

| READ COMMITED

| REPEATABLE READ

| SERIALIZABLE}
```

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

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

Isolation Concepts ...

What kind of applications use relaxed isolation?

- Have you experienced any dirty reads as users?

PollEv.com/farhanachoud585

Respond at **PollEv.com/farhanachoud585**

Text **FARHANACHOUD585** to **+61 427 541 357** once to join, then text your message

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

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}

READ_COMMITTED_SNAPSHOT can be set on or off

If on – Shared locks are <u>not</u> used for reading

- Read committed with READ_COMMITTED_SNAPSHOT off – degree 2

- Read committed with READ_COMMITTED_SNAPSHOT on – degree 1

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 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.

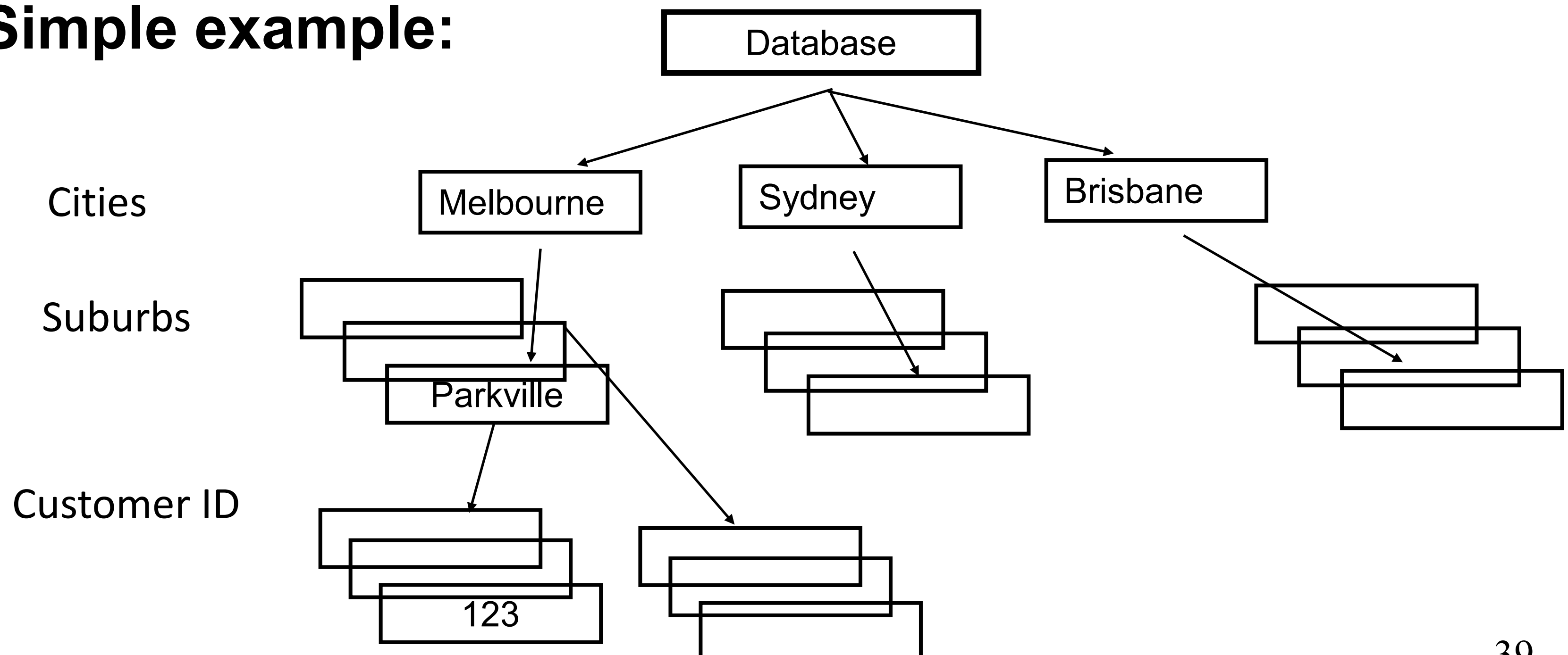
Granularity Of Locks

Pick a set of column values (predicates).

They form a graph/tree structure.

Lock the nodes in this graph/tree

Simple example:



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

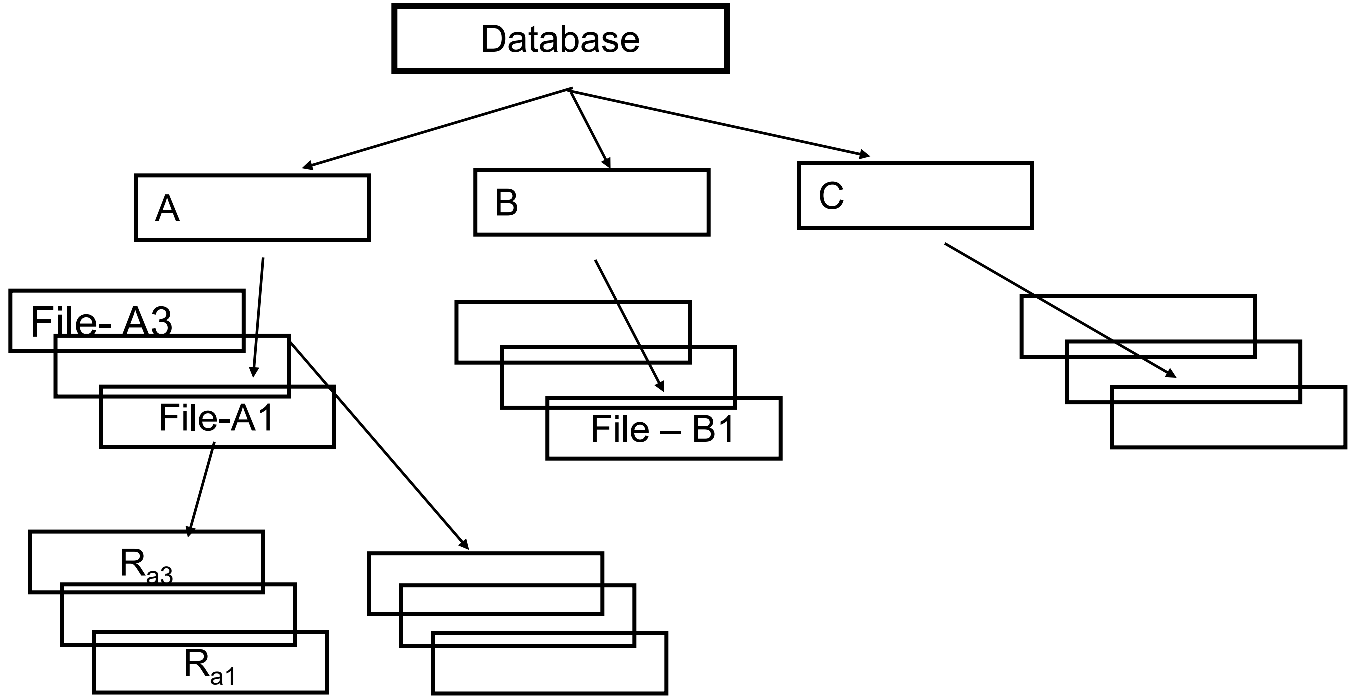
IX Intent to set shared or eXclusive locks at finer granularity

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

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)

Isolation concepts ...



Rules:
Lock root to leaf
To set X or S below, get IX or IS above respectively (or higher mode)

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.

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