COMP90050
Advanced Database Systems

THE UNIVERSITY OF MELBOURNE

Winter Semester, 2023

Lecturer: Farhana Choudhury (PhD)

Week 3 part 5

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

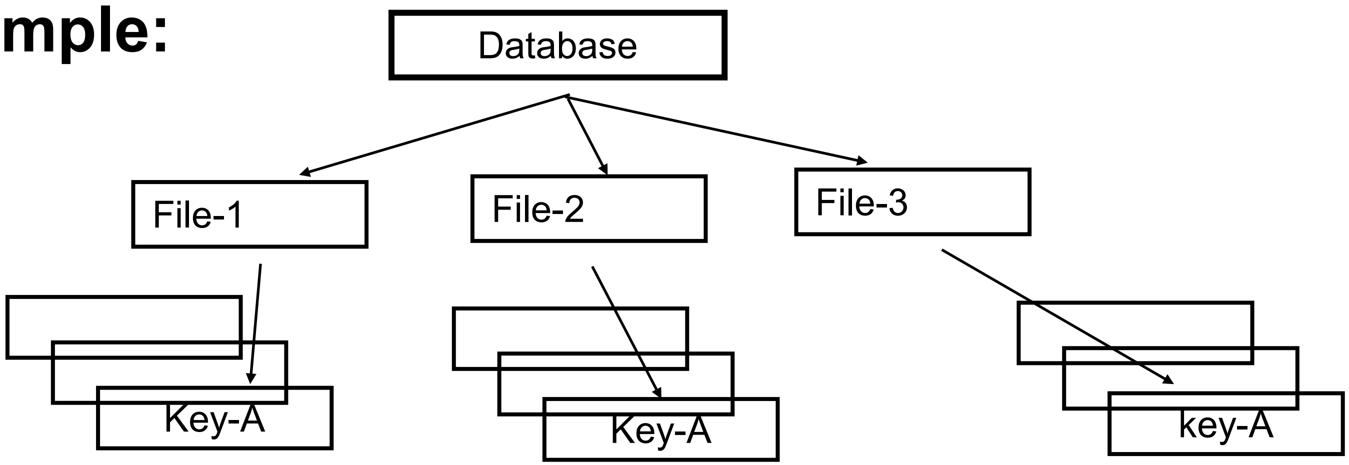
Idea:

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.

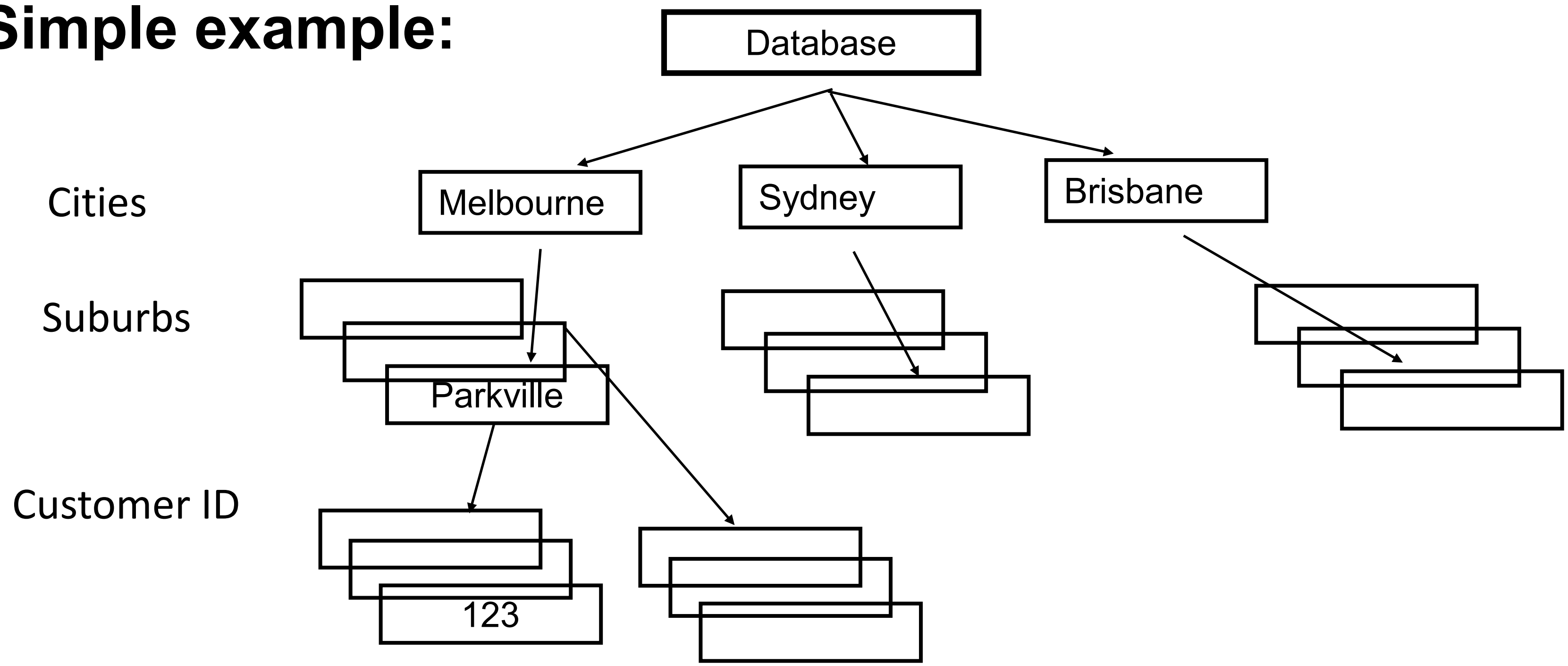
Granularity Of Locks

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.

Release locks from leaf to root.

IS: intend to set finer S locks

IX: intend to set finer S or X locks

SIX: S + IX

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}).

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.

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.

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)

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.

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
```

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.

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.

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.

Time stamping

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.

Time Domain Versioning

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

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

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.

Time stamping

T1

select average (salary)

from employee

T2

update 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!