Concurrency

Overview

- Three concurrency problems
- Locking
- Three concurrency problems – Revisited
- Deadlock
- Serializability
- Recovery – Schedules
- Isolation Levels
- Intent Locking

Concurrency

- Concurrency – DBMS allow many transactions to access the same database at the same time.
- A kind of control is needed to ensure that concurrent transactions do not interfere with each other.

Three concurrency problems

- The lost update problem
- The uncommitted dependency problem
- The inconsistent analysis problem

The Lost Update Problem

- Transaction A's update is lost at time t4.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
RETRIEVE t	t1	--
--		--
--	t2	RETRIEVE t
--		--
UPDATE t	t3	--
--		--
--	t4	UPDATE t
--		--

Dirty write

The Uncommitted Dependency Problem

- Transaction A sees an uncommitted update at time t2.
- Transaction A produces an incorrect result.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t
--		--
RETRIEVE t	t2	--
--		--
--	t3	ROLLBACK

Dirty read

The Uncommitted Dependency Problem

- Transaction A become dependent on an uncommitted change at time t2.
- Loses an update at time t3.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t
--		--
UPDATE t	t2	--
--		--
--	t3	ROLLBACK

The Inconsistent Analysis Problem

ACC1 = 40

ACC2 = 50

ACC3 = 30

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
RETRIEVE ACC 1:	t1	--
sum = 40		
--		--
RETRIEVE ACC 2:	t2	--
sum = 90		
--		--
--	t3	RETRIEVE ACC 3
--		--
--	t4	UPDATE ACC 3: 30 --> 20
--		--
--	t5	RETRIEVE ACC 1
--		--
--	t6	UPDATE ACC 1: 40 --> 50
--		--
--	t7	COMMIT
--		--
RETRIEVE ACC 3:	t8	
sum = 110, not 120		

Conflict Operations

- From a concurrency point of view, the primary focus on database retrievals and updates --> *reads / writes*
- If A and B are concurrent transactions, problems occur if A and B want to read or write the same database object, say tuple t.
  - RR: A and B both wants to read t. Reads can not interfere with each other – No Issue.
  - RW: A reads t and then B wants to write t. Inconsistent analysis problem – RW conflicts.
  - WR: A writes t and then B wants to read t. Uncommitted dependency problem – WR conflicts.
  - WW: A writes t and then B wants to write t. Lost update problem – WW conflicts.

Locking

- Concurrency problems can be solved by concurrency control mechanism called **locking**.
- Assume system supports two kinds of locks: **exclusive locks** (X locks) and **shared locks** (S locks).
  - X and S locks are sometimes called write locks and read locks.
- If transaction A holds an exclusive (X) lock on tuple *t*, then a request from a transaction B for a lock of either type on *t cannot* be immediately granted.
- If transaction A holds a shared (S) lock on tuple *t*, then:
  - A request from transaction B for an X lock on *t* cannot be immediately granted.
  - A request from transaction B for an S lock on *t* can be granted.

Locking

- The following code performs the *read* operation:

```
B: if LOCK (X) = “unlocked” then
        begin LOCK (X) ← “read-locked”;
              no_of_reads (X) ← 1;
        end
else if LOCK (X) ← “read-locked” then
        no_of_reads (X) ← no_of_reads (X) +1
else begin wait (until LOCK (X) = “unlocked” and
        the lock manager wakes up the transaction);
        go to B
     end;
```

Locking

- The following code performs the *write* operation:

```
  B: if LOCK (X) = "unlocked" then
then LOCK (X) ← "write-locked";
else begin
wait (until LOCK (X) = "unlocked" and
the lock manager wakes up the transaction);
        go to B
end;
```

Lock Type Compatibility

- N – conflict, Y – compatibility

	Read	Write
Read	Y	N
Write	N	N

- Retrieve t – Transaction must acquire S lock.
- Update t – Transaction must acquire X lock. If already holds S lock on that tuple, then *upgrade* the lock.
- If lock request from transaction B conflicts with A, B goes into a *wait state.*
- X locks are released at end-of-transaction. S locks are normally released at that time.

Three Concurrency Problems Revisted

- The Lost Update Problem
- The Uncommitted Dependency Problem
- The Inconsistent Analysis Problem

The Lost Update Problem

- Under strict two-phase locking protocol, the lost update problem is resolved.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
RETRIEVE t [acquire S lock on t]	t1	--
--		--
--	t2	RETRIEVE t [acquire S lock on t]
--		--
UPDATE t [request X lock on t]	t3	--
wait		--
wait	t4	UPDATE t [request X lock on t]
wait		
wait		wait

The Uncommitted Dependency

- A is no longer dependent on an uncommitted update.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t
--		[acquire X lock on t] --
RETRIEVE t	t2	--
wait [request S lock on t]		--
wait	t3	ROLLBACK
wait		[release X lock on t] --
resume: RETRIEVE t [acquire S lock on t]	t4	

The Uncommitted Dependency

- A is no longer dependent on an uncommitted update.

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t [acquire X lock on t]
--		--
UPDATE t [request X lock on t]	t2	--
wait		--
wait	t3	ROLLBACK [release X lock on t]
wait		
resume: UPDATE t [acquire X lock on t]	t4	

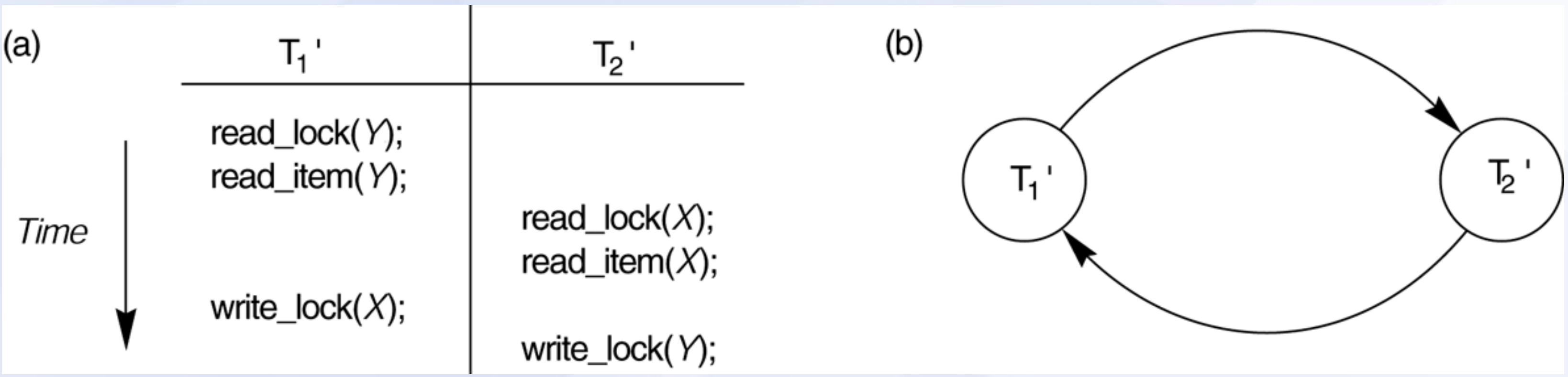
The Inconsistent Analysis Problem

TRANSACTION A	Time	TRANSACTION B
--		--
RETRIEVE ACC 1:	t1	--
[acquire S lock on ACC1]		
sum = 40		--
--		--
RETRIEVE ACC 2:	t2	--
[acquire S lock on ACC2]		
sum = 90		--
--		--
--	t3	RETRIEVE ACC 3
		[acquire S lock on ACC3]
--		--
--	t4	UPDATE ACC 3:
		[acquire X lock on ACC3]
--		--
--	t5	RETRIEVE ACC 1
		[acquire S lock on ACC1]
--		--
--	t6	UPDATE ACC 1:
		[request X lock on ACC1]
--		
RETRIEVE ACC 3:	t7	wait
[request S lock on ACC 3]		wait
wait		wait
wait		wait

Deadlock

- Locking introduce the problem of **deadlock**.
- Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each of them waiting for one of the others to release a lock.
- Detecting the deadlock involves detecting a cycle in the Wait-For Graph.
- Breaking the deadlock involves choose one of the transaction in the cycle as the *victim* and roll back.

Dealing with Deadlock



Wait-for graph to detect deadlock

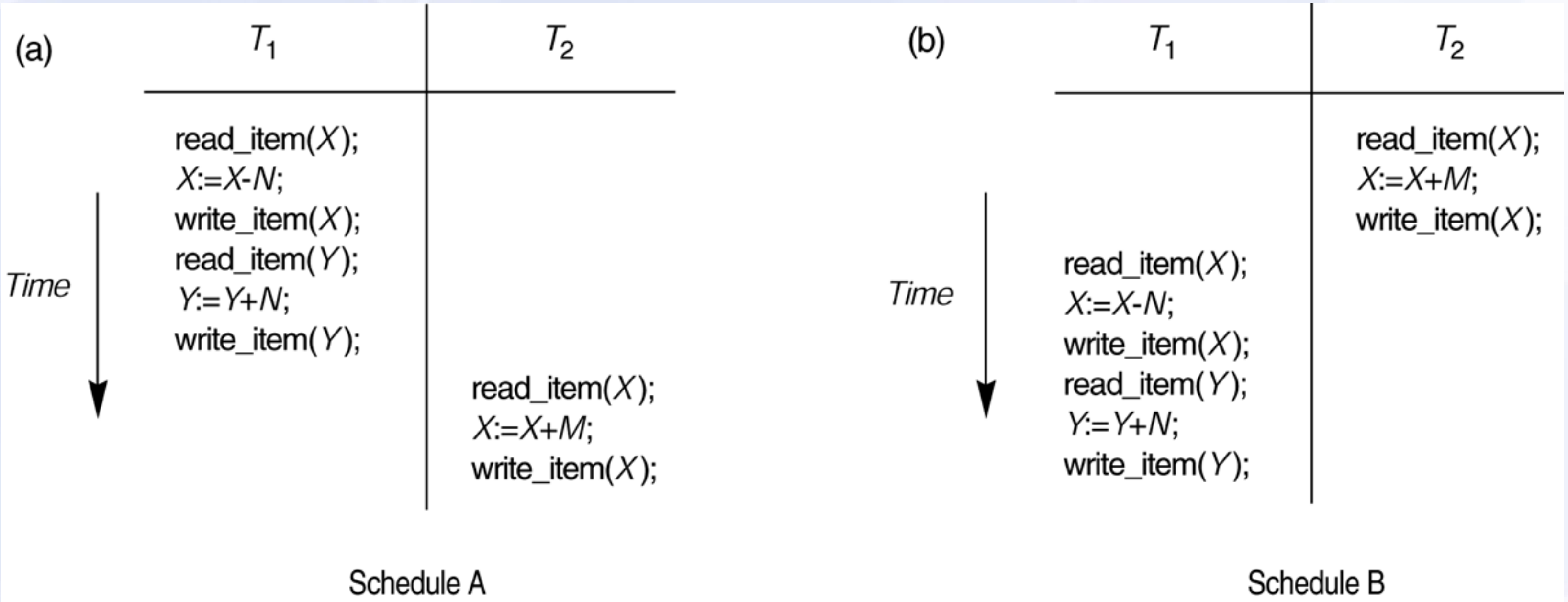
Deadlock Avoidance

- It would be possible to avoid deadlock by modifying the locking protocol.
- Every transaction is timestamped with its start time.
- When transaction A requests a lock on a tuple that is already locked by transaction B, then:
  - **Wait-Die**: A waits if it is older than B; otherwise [A dies], it is rolled back and restarted.
  - **Wound-Wait**: A waits if it is younger than B; otherwise [A wounds B], B is rolled back and restarted.
- If a transaction has to be restarted, it retains its original timestamp.

Serializability

- Given a set of transactions, any execution of those transactions, interleaved or otherwise, is called a schedule.
- Consider two transactions T1 and T2 which is submitted at the same time. If no interleaving is permitted then there are two possible ways:
  - Execute all the operations of T1 and then T2
  - Execute all the operations of T2 and then T1
- Executing the transactions one at a time, with no interleaving is called a serial schedule.
- A schedule that is not serial is an *interleaved* or a nonserial schedule

Serial Schedule



(a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1.

Non-Serial Schedule

(c)

Time

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

Schedule C

T_1	T_2

read_item(X);
X:=X-N;
write_item(X);	

	read_item(X);
	X:=X+M;
	write_item(X);

read_item(Y);
Y:=Y+N;
write_item(Y);	

Schedule D

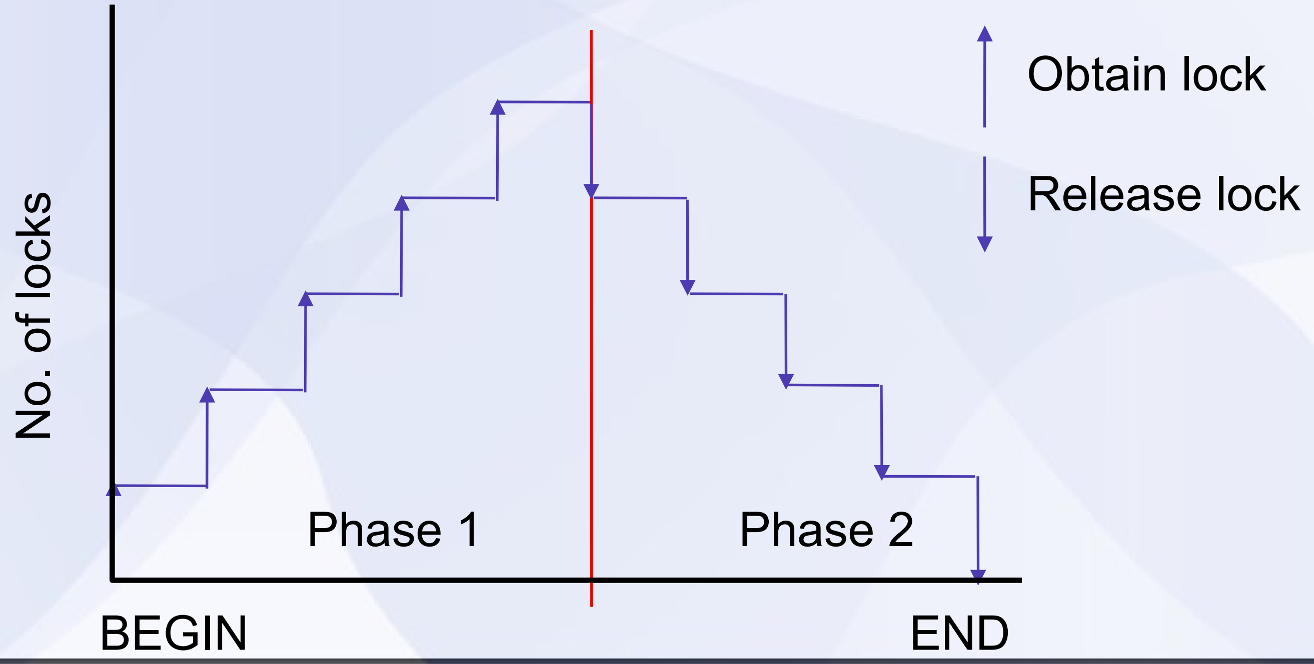
Two nonserial schedules C and D with interleaving of operations

Serializability

- Individual transactions are assumed to be correct – transform a correct state of the database into another correct state.
- Every <u>serial schedule is considered correct</u>, because the transaction do not depend on one another.
- A given execution of a given set of transactions is **serializable** – if and <u>only if it is equivalent</u> to some serial execution of the same transactions.
- If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.

Two-phase Locking Protocol – 2PL

- Before operating on any object (e.g., a tuple), a transaction must acquire a lock on that object. [lock acquisiton – **growing** phase]
- After releasing a lock, a transaction must never go on to acquire any more locks. [lock release – **shrinking** phase]



Two-phase Locking Protocol – 2PL

- The strict two-phase locking protocol is to force serializability.
- The following interleaved execution is equivalent to B-then-A

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t
--		-- [acquire X lock on t]
RETRIEVE t	t2	--
wait [request S lock on t]		--
wait	t3	ROLLBACK
wait		-- [release X lock on t]
resume: RETRIEVE t	t4	
[acquire S lock on t]		

Serializability

- Let I be an interleaved schedule involving some set of transactions T1,T2,...,Tn.
- If there exists some serial schedule S involving T1,T2,...,Tn such that

  I is *equivalent* to S, then I is said to be **serializable**.

For more on serializability refer slides based on Elmasri & Abraham

Recovery – Schedules

- A schedule to be recoverable is as follows:
  - If A sees any of B's updates, then A must not commit before B terminates.
  - w2(t); r1(t); c1; a2 → unrecoverable schedule

TRANSACTION A	Time	TRANSACTION B
--		--
--		--
--	t1	UPDATE t
--		--
RETRIEVE t	t2	--
--		--
COMMIT	t3	--
		--
	t4	ROLLBACK

An unrecoverable schedule

Recovery – Schedules

- A schedule to be cascade-free is as follows:
  - If A sees any of B's updates, then A must not do so before B terminates.

TRANSACTION A	Time	TRANSACTION B
--		--
--	t1	UPDATE t [acquire X lock on t]
--		--
RETRIEVE t [request S lock on t]	t2	--
wait		--
wait	t3	release lock on t
wait		--
resume:RETRIEVE t [acquire S lock on t]	t4	--
--		--
forced rollback	t5	ROLLBACK

A schedule involving cascaded rollback

Isolation Levels

- In practice, systems usually support a variety of isolation levels.
- **Isolation level** to a given transaction is defined as the *degree of interference* the transaction in question is prepared to tolerate on the part of concurrent transactions.
- If serializability to be guaranteed --> amount of interfernce that can be tolerated is none --> Isolation level should be maximum
- Higher the isolation level, less the interference --> lower concurrency
- Lower the isolation level, more the interference --> higher concurrency

Isolation Levels

- If a transaction executes at a lower isolation level following three violations may occur:
  - Dirty Read
  - Non-repeatable Read
  - Phantom

Dirty Read

- A transaction T1 may read the update of a transaction T2, which has not committed yet.
- If T2 fails and is aborted ,then T1 would have read a value that does not exist and is incorrect.

What happens	Transaction T1	Transaction T2
User 1 starts a transaction T1	`BEGIN WORK;`	
User 1 reads and updates a record: Changes the salary of a given employee	`UPDATE employee SET salary = 50000 WHERE empno = 100;`	
Another User 2 starts a new transaction T2		`BEGIN WORK;`
User 2 reads all the employee records: T2 sees data updated by T1 that has not been committed yet. T1 could still rollback (as in this case, and the value previously read by T2 would be invalid)		`SELECT * FROM employee;`
	`ROLLBACK;`	

time

Non-repeatable Read

- A *Nonrepeatable Read* occurs if transaction T1 retrieves a different result from each read.
- Transaction T1 reads an item
- Transaction T2 reads and updates the same item
- Transaction T1 reads the same item again, but now it has a new, modified value

Non-repeatable Read

What happens	Transaction T1	Transaction T2
User 1 starts a transaction T1	`BEGIN WORK;`	
User 1 reads an item	`SELECT * FROM employee WHERE empno = 100;`	
Another User 2 starts a new transaction T2		`BEGIN WORK;`
Transaction T2 reads and updates the same item. If T1 issues the same SELECT statement again (as in this case), the results will be different		`UPDATE employee SET salary = 50000 WHERE empno = 100`
	`SELECT * FROM employee WHERE empno = 100;`	

time

Phantom Read

- A *Phantom Read* occurs if transaction T1 obtains a different result from each Select for the same criteria
- Transaction T1 executes search on certain criteria and retrieve m items from a table
- Transaction T2 inserts another item that would match the search criteria
- Transaction T1 again executes search and now retrieves m+1 items from the table

Phantom Read

What happens	Transaction T1	Transaction T2
User 1 starts a transaction T1	`BEGIN WORK;`	
User 1 issues a query to search records based on a criteria	`SELECT * FROM employee WHERE salary > 30000 ;`	
Another User 2 starts a new transaction T2		`BEGIN WORK;`
User 2 inserts another item that would match or satisfy the selection criteria of T1 query		`INSERT INTO employee (empno, firstnme, lastname, job, salary) VALUES (125, 'ROBERT', 'BROWN','IT Consultant',35000)`
If T1 runs the same query within the same transaction. The results will differ and this time it will get N+1 records satisfying the search criteria (a new phantom record)	`SELECT * FROM employee WHERE salary > 30000 ;`	

time

Isolation Levels @ DB2

- **REPEATABLE READ:** Protects against Dirty Reads, Nonrepeatable Reads, and Phantoms
- **READ STABILITY:** Protects against Dirty Reads, and Nonrepeatable Reads. Read stability does not protect against Phantoms.
- **CURSOR STABILITY:** Protects against Dirty Reads, but does not protect against Nonrepeatable Reads and Phantoms.
- **UNCOMMITED READ:** Uncommitted Read does not protect against Phantoms, Dirty Reads, and Nonrepeatable Reads.

Repeatable Read

- This isolation level is the most restrictive of the four – this ensures that the schedules are serializable.
- It makes sure that none of the phenomenona will occur.
- It is also the most restrictive when it comes to database concurrency.
- When a transaction with isolation level REAPETABLE READ (RR) executes:
  - It locks every row it references.
  - The lock is held for every row, not only rows that are updated, but also those that are merely selected

Repeatable Read – Example

- Transaction A with Repeatable Read isolation level selects all rows on table A.
- Transaction B attempts to update any of the rows selected by Transaction A and fails because all rows selected by Transaction A are locked.

Read Stability

- Read Stability is not as restrictive as the Repeatable Read isolation level.
- In Read Stability, only rows that are retrieved or modified are locked, whereas in Repeatable Read, all rows that are being referenced are locked.

Read Stability – Example

- Transaction A with isolation level Read Stability selects row 1.
- Transaction B cannot update row 1 while Transaction A is holding a lock on row 1.
- This prevents against Lost Updates, Dirty Reads, and Nonrepeatable Reads.
- Transaction B can insert a new row that can show up in a result set of Transaction A. This allows for Phantoms.

Cursor Stability

- Cursor Stability only locks one row at a time — the row that is currently being referenced by a cursor.
- As soon as the cursor moves to the next row, the lock on the previous row is released.
- This provides for much more concurrency because other transactions can update rows before and after a row that is being referenced by a cursor with cursor stability.
- There are two exceptions to this rule:
  - If the cursor with cursor stability retrieves rows using an index, now rows can be modified or inserted into the cursor's result set.
  - No transaction can modify or delete a row that has been updated by the Cursor Stability Cursor until the owning transaction is terminated.

Cursor Stability – Example

- Transaction A opens cursor A with Cursor Stability and starts reading rows.
- While cursor A is executing and referencing row 2, transaction B can not update or delete row 2, but can update and insert all other rows.
- Transaction B can update row 2 and commit the change as soon as cursor A releases row 2 given that it has not modified it.
- If transaction A re-runs cursor A, there is no guarantee that resultset will be the same – a nonrepeatable read

Cursor stability

- We can infer that:

If isolation level is less than the maximum, then no guarantee that transaction T running concurrently with other transaction will transform a correct state of the database into another correct state.

- Another special problem that can occur if transaction operate at less than the maximum isolation lelvel is – phantom problem.

Uncommitted Read

- Uncommitted read is the least restrictive of all isolation levels and provides the most database concurrency.
- Allows an application to access uncommitted changes of other transactions.
- This isolation level is mostly used to retrieve read-only data.
- An Uncommitted Read transaction locks only those rows that it modifies or if another transaction attempts to alter or drop the table the rows are being retrieved from.
- Uncommitted Read does not protect against dirty reads, nonrepeatable reads, and phantoms.

Uncommitted Read

- Transaction A, running with isolation level Uncommitted Read ,is updating row 1 and 2.
- While transaction A is running, transaction B (not running under Uncommitted Read) cannot read rows 1 and 2 because these rows are locked byTransaction A—lost updates phenomena cannot occur.
- While transaction A is running, transaction C (running under Uncommitted Read) can read rows 1 and 2—a dirty read can occur.

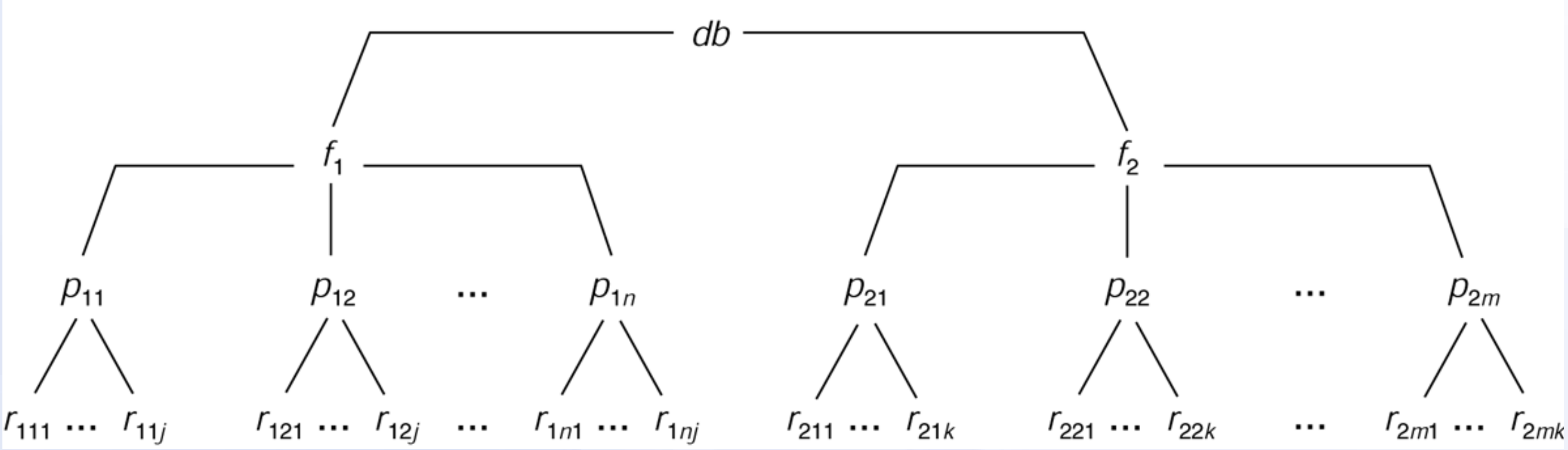
Summary

Isolation Level	Dirty Read	Non-repeatable	Phantom
Repeatable Read (RR)	No	No	No
Read Stability (RS)	No	No	Possible
Cursor Stability (CS)	No	Possible	Possible
Uncommitted Read (UR)	Possible	Possible	Possible

Intent Locking

- Locking granularity – the finer the granularity, the greater the concurrency.
- If a transaction has X lock on an entire relvar, no need to set X locks on individual tuples within that relvar.
- *Fine granularity* refers to small item sizes, *coarse granularity* refers to large item sizes.
- Given the above tradeoffs : *What is the best item size?*
  It depends on the types of transactions involved.

Intent Locking



A granularity hierarchy – multiple granularity level locking

Intent Locking

- To make multiple granularity level locking practical, additional types of locks, called intent locks are needed.
- The idea behind intention lock is for a transaction to indicate, along the path from the root to the desired node, what type of lock is required from one of the node's descendants.

1. *Intention-shared (IS)* – indicates that a shared lock(s) will be requested on some descendants node(s).

2. *Intention-exclusive (IX)* – indicates that an exclusive lock(s) will be requested on some descendants node(s).

3. *Shared-intention-exclusive (SIX)* – indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendants node(s).

Intent Locking

- Lock compatibility matrix

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Intent Locking

- The multiple granularity locking or intent locking protocol consists of following rules:

1. The lock compatibility must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by T in S or IS only if the parent N is already locked by T in either IS or IX.
4. A node N can be locked by T in X, IX or SIX only if the parent N is already locked by T in either IX or SIX.
5. A transaction T can lock a node only if it has not unlocked any node.
6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

References

- *Chapter 16: Concurrency*
  An introduction to database systems, *CJ. Date*