Transactions

Controlling Concurrent Behavior

But what’s a transaction?

- A **process** that reads or modifies the DB is called a **transaction**.
  - Unit of execution of database operations.

```
begin transaction;

delete from product
where model in (
    select model
    from pc
    where hd<500
    );

delete from pc
where hd < 500;

commit;
```

SQL statements: COMMIT and ROLLBACK

- **COMMIT** causes a transaction to *complete*.
  - Its database modifications are now **permanent** in the database.

- **ROLLBACK** causes a transaction to end, but by *aborting*.
  - **No effects** on the database.

Failures like division by 0 or a constraint violation can also **cause rollback**, even if the programmer does not request it.

Busy, busy, busy...

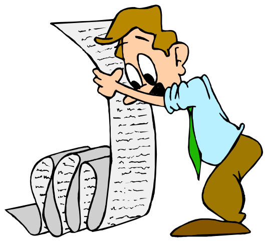
- In production environments there could many users.
- Consequently, it’s possible for multiple transactions to be submitted at approximately the same time.

Busy, busy, busy...

- If all transactions were small, we could just execute them on a first-come-first-served basis.

- However, many transactions are complex and time consuming.
  - Executing those would make other queries wait a long time for a chance to execute.

So, in practice, DBMS may be running many different transactions at about the same time.



Concurrent Transactions

- DMBS needs to keep processes from **troublesome interactions**.

- Even when there is **no "failure," several** transactions can **interact** to turn a

  consistent state

  into an

  inconsistent state.

Transactions and Schedules

- A transaction can be considered at a low level to be a list of actions:
  - read, write, commit, abort

- A schedule is a **list of actions** from a **set** of transactions. E.g.

T1	T2
r(A)	
w(A)	
	r(B)
	w(B)
	commit
r(C)	
w(C)	
commit	

Anomalies: Reading Uncommitted Data

T1	T2
r(A)	
w(A)	
	r(A)
	w(A)
	r(B)
	w(B)
	commit
r(B)	
w(B)	
commit	

Of course there would be no problem if we executed T1, then T2, or vice versa. (**Serial execution**)

- T1 transfers $100 from A to B
- T2 increments both A and B by 1% (e.g. daily interest)

- Problem with schedule above is that the bank didn't pay interest on the $100 that was being transferred.

Anomalies: Unrepeatable Reads

- Suppose A is the **number of copies** available for a book.

- T1 and T2 both **place an order** for this book. First they **check availability** of the book.

- Consider now the following scenario:

1. T1 checks whether A is greater than 1.

   Suppose T1 sees (reads) value 1.
2. T2 also reads A and sees 1.
3. T2 decrements A to 0.
4. T2 commits.
5. T1 tries to decrement A, which is now 0, and gets an error because some check constraint doesn’t allow it.

- This situation can never arise in a **serial execution** of T1 and T2.

Anomalies: Overwriting uncommitted data

- Suppose Larry and Harry are two employees, and their salaries must be kept the equal.
- T1 sets their salaries to $2000 and
- T2 sets their salaries to $2100.
- Now consider the following schedule:

T1	T2
r(Larry)	
w(Larry)	
	r(Harry)
	w(Harry)
r(Harry)	
w(Harry)	
	r(Larry)
	w(Larry)
	commit
commit	

Of course there would be no problem if we executed T1, then T2, or vice versa. (**Serial execution**)

Unfortunately, Larry will be paid more than Harry.

ACID Transactions

- *ACID transactions* are:
  - *Atomic* : Whole transaction or none is done.
  - *Consistent* : Database consistency preserved.
  - *Isolated* : Appears to the user as if only their process executes.
  - *Durable* : Effects of a process survive a crash.

- Optional: weaker forms of transactions are often supported as well.

Isolated:
That is, even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after another in some **serial order**.

Summarizing the Terminology

- A **transaction** (model) is a *sequence* of *r* and *w* actions on database elements.

- A **schedule** is a *sequence* of read/write actions performed by a collection of transactions.

- **Serial Schedule** = All actions for each transaction are consecutive.

  r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B); …

- **Serializable Schedule**: A schedule whose “**effect**” is equivalent to that of some serial schedule.

We will introduce a **sufficient condition** for serializability.

Swaps and Conflicts

Definition

- A schedule is ***conflict-serializable*** if it can be **converted into** a **serial schedule** with the **same effect** by a series of **non-conflicting swaps** of **adjacent elements**

There is a **conflict** if one of these two conditions hold.

1. A read and a write of the same X, or
2. Two writes of the same X

- Such actions **cannot** *be swapped in order*.
- All other actions **can** be swapped
  - without changing the **effect** of the schedule (on the DB).

Example

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); \underline{\mathbf{w_2(A)}}; \underline{\mathbf{r_1(B)}}; w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); \underline{\mathbf{r_2(A)}}; \underline{\mathbf{r_1(B)}}; w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); \underline{\mathbf{w_2(A)}}; \underline{\mathbf{w_1(B)}}; r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); \underline{\mathbf{r_2(A)}}; \underline{\mathbf{w_1(B)}}; w_2(A); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A)w_2(A); r_2(B); w_2(B)$

The operations in bold can be safely swapped.

Precedence graphs

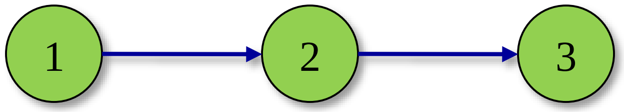
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Note the following:

- $w_1(B) <_S r_2(B)$
- $r_2(A) <_S w_3(A)$

➢ These are conflicts since they contain a read/write on the same element

➢ They cannot be swapped. Therefore $T_1 < T_2 < T_3$



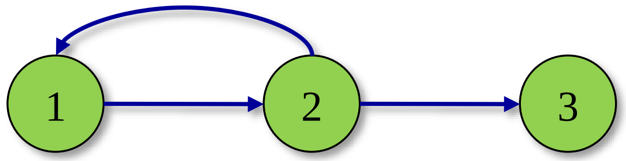
Conflict serializable

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Note the following:

- $r_2(B) <_S w_1(B)$
- $w_2(A) <_S w_3(A)$
- $r_1(B) <_S w_2(B)$

➢ Here, we have $T_1 < T_2 < T_3$, but we also have $T_2 < T_1$



Not conflict serializable

- **If there is a cycle in the graph**
  - **There doesn't exist** a serial schedule that is **conflict-equivalent** to S.
    - **Why?** Each arc represents a requirement on the order of transactions in a conflict equivalent *serial schedule*.

- **If there is no cycle in the graph**
  - Any **topological order** of the graph gives a conflict-equivalent serial schedule.

Topological order of a DAG:
Each node comes before all nodes to which it has outbound edges.

Why the Precedence-Graph Test Works?

Idea: if the precedence graph is acyclic, then we can swap actions to form a serial schedule.

Given that the **precedence graph is acyclic**, there exists T_i in S such that there is no T_j in S that T_i depends on.

- We can swap all actions of T_i to the front (of S).
- **(Actions of T_i)(Actions of the other *n*-1 transactions)**
- The **tail** is a precedence graph that is the same as the original without T_i, i.e. it has *n*-1 nodes.
- Repeat for the tail.

Lock Actions

- Before reading or writing an element X, a transaction T_i **requests a lock** on X from the scheduler.

- The scheduler can either **grant the lock** to T_i or make T_i **wait for the lock**.

- If granted, T_i should eventually **unlock** (release the lock on) X.

- Shorthands:
  - $l_i(X)$ = "transaction T_i requests a lock on X"
  - $u_i(X)$ = "T_i unlocks/releases the lock on X"

Legal Schedule Doesn't Mean Serializable

T_1	T_2	A	B
		25	25
$l_1(A); r_1(A)$			
A = A + 100			
$w_1(A);u_1(A)$		125	
	$l_2(A);r_2(A)$		
	A = A * 2		
	$w_2(A);u_2(A)$	250	
	$l_2(B);r_2(B)$		
	B = B * 2		
	$w_2(B);u_2(B)$		50
$l_1(B);r_1(B)$			
B = B + 100			
$w_1(B);u_1(B)$			150

Consistency constraint assumed for this example: A=B

Two Phase Locking

There is a simple condition, which guarantees conflict-serializability:
In every transaction, all lock requests (phase 1) precede all unlock requests (phase 2).

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$			
A = A + 100			
$w_1(A)$; $l_1(B)$; $u_1(A)$		125	
	$l_2(A)$; $r_2(A)$		
	A = A * 2		
	$w_2(A)$	250	
	$l_2(B)$ **Denied**		
$r_1(B)$			
B = B + 100			125
$w_1(B)$; $u_1(B)$			
	$l_2(B)$; $u_2(A)$; $r_2(B)$		
	B = B * 2		
	$w_2(B)$;$u_2(B)$		250

Why 2PL Works?

- **Theorem.** A legal schedule S of 2PL transactions is conflict-serializable.

- Proof is an induction on ***n***, the number of transactions.

Why 2PL Works (Cont'd)

- **Basis:** if n=1, then S={T_1}, and hence S is conflict-serializable.

- **Induction:** S={T_1,…,T_n}. Find the **first** transaction, say T_i, to perform an **unlock** action, say $u_i(X)$.
- We show that the r/w actions of T_i can be moved to the front of the other transactions without conflict.
- Consider some action such as **$w_i(Y)$.** Can it be preceded by some conflicting action **$w_j(Y)$** or **$r_j(Y)$**? In such a case we cannot swap them.
- If so, then $u_j(Y)$ and $l_i(Y)$ must intervene
  $w_j(Y)...u_j(Y)...l_i(Y)...w_i(Y)$
- Since T_i is the first to unlock, $u_i(X)$ appears before $u_j(Y)$
  $w_j(Y)...\ u_i(X)…u_j(Y)...l_i(Y)...w_i(Y)$
- But then $l_i(Y)$ appears after $u_i(X)$, contradicting 2PL.

- **Conclusion:** **$w_i(Y)$** can slide forward in the schedule without conflict.

What should we lock?

- Whole table or just single rows?
- What's database object?
- What should be the locking granularity?

```
SELECT min(year)
FROM Movies;
```

- What happens if we insert a new tuple with the smallest year?
- **Phantom** problem: A transaction retrieves a collection of tuples twice and sees different results, even though it doesn't modify those tuples itself.

Transaction support in SQL

- SET TRANSACTION ISOLATION LEVEL **X**
  - Where X can be
    - SERIALIZABLE (Default)
    - REPEATABLE READ
    - READ COMMITED
    - READ UNCOMMITED

With a scheduler based on locks:

- A SERIALIZABLE transaction obtains locks before reading and writing objects, including locks on sets (e.g. table) of objects that it requires to be unchangeable and holds them until the end, according to 2PL.

- A REPEATABLE READ transaction sets the same locks as a SERIALIZABLE transaction, except that it doesn’t lock sets of objects, but only individual objects.

Transaction support in SQL

- A READ COMMITED transaction T
  - obtains locks before writing objects and keeps them until the end.
  - obtains locks before reading values, then immediately releases them;

  their effect is to ensure that the transaction that last modified the values is complete.
- Thus,
  1. No value written by T is changed by any other transaction until T is completed.
  2. T reads only the changes made by committed transactions.
  3. However, a value read by T may well be modified by another transaction (which eventually commits) while T is still in progress.

- A READ UNCOMMITED transaction doesn't obtain any lock at all. So, it can read data that is being modified. Such transactions are allowed to be READ ONLY only. So, such transaction doesn't ask for any lock at all.

In Summary

Level	Reading Uncommited Data (Dirty Read)	Unrepeatable Read	Phantom
READ UNCOMMITED			
READ COMMITTED			
REPEATABLE READ			
SERIALIZABLE			

In Summary

Level	Reading Uncommited Data (Dirty Read)	Unrepeatable Read	Phantom
READ UNCOMMITED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READ	No	No	Maybe
SERIALIZABLE	No	No	No