9. Transaction Processing Concepts

Goals: Understand the basic properties of a transaction and learn the concepts underlying transaction processing as well as the concurrent executions of transactions.

A *transaction* is a unit of a program execution that accesses and possibly modifies various data objects (tuples, relations).

DBMS has to maintain the following properties of transactions:

- **A**tomicity: A transaction is an atomic unit of processing, and it either has to be performed in its entirety or not at all.
- **C**onsistency: A successful execution of a transaction must take a consistent database state to a (new) consistent database state. ($\leadsto$ integrity constraints)
- **I**solation: A transaction must not make its modifications visible to other transactions until it is committed, i.e., each transaction is unaware of other transactions executing concurrently in the system. ($\leadsto$ concurrency control)
- **D**urability: Once a transaction has committed its changes, these changes must never get lost due to subsequent (system) failures. ($\leadsto$ recovery)

Model used for representing database modifications of a transaction:

- **read**(A,x): assign value of database object A to variable x;
- **write**(x,A): write value of variable x to database object A

Example of a Transaction T

read(A,x)
x := x - 200
write(x,A)
read(B,y)
y := y + 100
write(y,B)

Transaction Schedule reflects chronological order of operations

Main focus here: Maintaining isolation in the presence of multiple, concurrent user transactions

Goal: "Synchronization" of transactions; allowing concurrency (instead of insisting on a strict serial transaction execution, i.e., process complete T_1, then T_2, then T_3 etc.)

$\leadsto$ increase the throughput of the system,
$\leadsto$ minimize response time for each transaction

Problems that can occur for certain transaction schedules without appropriate concurrency control mechanisms:

Lost Update

Time	Transaction T_1	Transaction T_2
1	**read**(A,x)	
2	x:=x+200	
3		**read**(A,y)
4		y:=y+100
5	**write**(x,A)	
6		**write**(y,A)
7		**commit**
8	**commit**	

The update performed by T_1 gets lost; possible solution: T_1 locks/unlocks database object A
$\Longrightarrow$ T_2 cannot read A while A is modified by T_1

Dirty Read

Time	Transaction T_1	Transaction T_2
1	**read**(A,x)	
2	x:=x+100	
3	**write**(x,A)	
4		**read**(A,y)
5		**write**(y,B)
6	**rollback**	

T_1 modifies db object, and then the transactionT_1 fails for some reason. Meanwhile the modified db object, however, has been accessed by another transaction T_2. Thus T_2 has read data that "never existed".

Inconsistent Analysis (Incorrect Summary Problem)

Time	Transaction T_1	Transaction T_2
1	**read**(A,y1)	
2		**read**(A,x1)
3		x1 := x1 - 100
4		**write**(x1, A)
5		**read**(C,x2)
6		x2 := x2+x1
7		**write**(x2,C)
8		**commit**
9	**read**(B,y2)	
10	**read**(C,y3)	
11	sum := y1 + y2 + y3	
12	**commit**	

In this schedule, the total computed by T_1 is wrong (off by 100).
$\Longrightarrow$ T_1 must lock/unlock several db objects

Serializability

DBMS must control concurrent execution of transactions to ensure read consistency, i.e., to avoid dirty reads etc.

$\leadsto$ A (possibly concurrent) schedule S is *serializable* if it is equivalent to a serial schedule S', i.e., S has the same result database state as S'.

How to ensure serializability of concurrent transactions?

Conflicts between operations of two transactions:

T_i	T_j
read(A,x)	
	read(A,y)

(order does not matter)

T_i	T_j
read(A,x)	
	write(y,A)

(order matters)

T_i	T_j
write(x,A)	
	read(A,y)

(order matters)

T_i	T_j
write(x,A)	
	write(y,A)

(order matters)

A schedule S is *serializable* with regard to the above conflicts **iff** S can be transformed into a serial schedule S' by a series of swaps of non-conflicting operations.

Checks for serializability are based on *precedence graph* that describes dependencies among concurrent transactions; if the graph has no cycle, then the transactions are serializable.

$\rightsquigarrow$ they can be executed concurrently without affecting each others transaction result.

Concurrency Control: Lock-Based Protocols

- One way to ensure serializability is to require that accesses to data objects must be done in a mutually exclusive manner.
- Allow transaction to access data object only if it is currently holding a **lock** on that object.
- Serializability can be guaranteed using locks in a certain fashion $\Longrightarrow$ Tests for serializability are redundant !

Types of locks that can be used in a transaction T:

- **slock**(X): shared-lock (read-lock); no other transaction than T can write data object X, but they can read X
- **xlock**(X): exclusive-lock; T can read/write data object X; no other transaction can read/write X, and
- **unlock**(X): unlock data object X

Lock-Compatibility Matrix:

requested lock	existing lock	
	slock	**xlock**
slock	OK	No
xlock	No	No

E.g., **xlock**(A) has to wait until all **slock**(A) have been released.

Using locks in a transaction (lock requirements, LR):

- before each **read**(X) there is either a **xlock**(X) or a **slock**(X) and no **unlock**(X) in between
- before each **write**(X) there is a **xlock**(X) and no **unlock**(X) in between
- a **slock**(X) can be tightened using a **xlock**(X)
- after a **xlock**(X) or a **slock**(X) sometime an **unlock**(X) must occur

But: "Simply setting locks/unlocks is not sufficient"

replace each **read**(X) → **slock**(X); **read**(X); **unlock**(X), and
write(X) → **xlock**(X); **write**(X); **unlock**(X)

Two-Phase Locking Protocol (TPLP)

A transaction T satisfies the TPLP iff

- after the first **unlock**(X) no locks **xlock**(X) or **slock**(X) occur
- That is, first T obtains locks, but may not release any lock (growing phase)
  and then T may release locks, but may not obtain new locks (shrinking phase)

Strict Two-Phase Locking Protocol:

All unlocks at the end of the transaction T $\Longrightarrow$ no dirty reads are possible, i.e., no other transaction can write the (modified) data objects in case of a rollback of T.

Concurrency Control in PostgreSQL

In PostgreSQL (or Oracle) the user can specify the following locks on relations and tuples using the command

lock table in <mode> **mode**;

mode	$\hat{=}$	tuple level	relation level
row share	$\hat{=}$	**slock**	**intended slock**
row exclusive	$\hat{=}$	**xlock**	**intended xlock**
share	$\hat{=}$	—	**slock**
share row exclusive	$\hat{=}$	—	**sixlock**
exclusive	$\hat{=}$	—	**xlock**

The following locks are performed automatically by the scheduler:

select	$\longrightarrow$	no lock
insert/update/delete	$\longrightarrow$	**xlock** /row exclusive
select . . . for update	$\longrightarrow$	**slock** /row share
commit	$\longrightarrow$	releases all locks

PostgreSQL (and Oracle) furthermore provide *isolation levels* that can be specified before a transaction by using the command

set transaction isolation level <level>;

- read committed (default): each query executed by a transaction sees the data that was committed before the query (not the transaction!)

  ($\rightsquigarrow$ statement level read consistency)

T_1	T_2
select A from R	
$\longrightarrow$ old value	
	update R set A = new
select A from R	
$\longrightarrow$ old value	
	commit
select A from R	
$\longrightarrow$ new value	

  Non-repeatable reads (same select statement in TA gives different results at different times) possible; dirty-reads are not possible

- serializable: serializable TAs see only those changes that were committed at the time the TA began, plus own changes.

  PostgreSQL generates an error when such a transaction tries to update or delete data modified by a transaction that commits after the serializable transaction began.

T_1	T_2
set transaction isolation level serializable	
	set transaction . . .
	update R set A = new where B = 1
	commit
update R set A = new where B = 1	
$\longrightarrow$ ERROR	

  Dirty-reads and non-repeatable reads are not possible. Furthermore, this mode guarantees serializability (but does not provide much parallelism).