X. Database Transactions

Database Transactions
ACID Properties and Anomalies
Schedules and Serializability
View- and Conflict Serializability
Two-Phase Locking
Deadlocks

Database Transactions

- Databases are a global resource within an organization.
- Organizations could not afford to have this resource serve its users on a one-at-a-time basis!
- Accordingly, concurrent access of a database by multiple (possibly a large number of) users has been a requirement since the early days of database technology.
- Database transaction mechanisms were developed in response to this requirement.

What is a Transaction?

- An atomic unit of work performed by an application, with specific constraints with respect to correctness, robustness and isolation
- Each transaction is delimited by the commands
  - ✓ **`begin transaction`** (**`bot`**)
  - ✓ **`end transaction`** (**`eot`**)
- Within a transaction, exactly one the commands **`commit work`** (commit), **`rollback work`** (abort)
- ***A transactional system*** is a system which offers facilities for defining and executing transactions on behalf of multiple, concurrent applications.

(Simple) Example

- A transaction is ***well-formed*** if it starts with **`bot`**, ends with **`eot`**, and executes exactly once `commit` or `abort`. No database operations can execute after the execution of `commit` or `abort`.
- Here is an example

```
bot
x := x – 10
y := y + 10
commit
eot
```

ACID Properties

- In order to cope with the concurrent execution, transactions need to have four properties.
- ***ACID*** is an acronym for:
  - ✓ ***A****tomicity* -- transactions are atomic;
  - ✓ ***C****onsistency* -- preserve database constraints
  - ✓ ***I****solation* -- transactions execute independently of each other;
  - ✓ ***D****urability* -- the effects of a transaction are persistent.

Transactions -- 5

Atomicity

- A transaction is an ***atomic unit of computation*** on a database; can't leave database in an intermediate state:
  - ✓ a fault or error prior to commit causes an ***undo*** of all work done earlier by the transaction;
  - ✓ A fault or error after the commit may require a ***redo*** of the work made earlier, if its effect on the database state are not guaranteed.
- Possible behaviours for a transaction:
  - ✓ Commit -- normal behaviour (99.9% of the time)
  - ✓ Rollback requested by the application -- ***suicide!***
  - ✓ Rollback requested by the system -- ***murder!!***

Transactions -- 6

Consistency

- Consistency amounts to requiring that the transaction does not violate any integrity constraints.
- Integrity constraint verification can be:
  - ✓ ***Immediate***: during the transaction (the operation causing the violation is rejected)
  - ✓ ***Deferred***: at the end of the transaction (if some integrity constraint is violated, the entire transaction is rejected)

Isolation

- Isolation requires that a transaction executes independently of the execution of all other concurrent transactions.
- This means that the concurrent execution of a collection of transactions yields the same result as an arbitrary sequential execution of the same transactions]

Durability (Persistence)

- Durability requires that the effect of a transaction that has successfully committed will not be lost (i.e., the effect will "last forever")

Transactions and System Modules

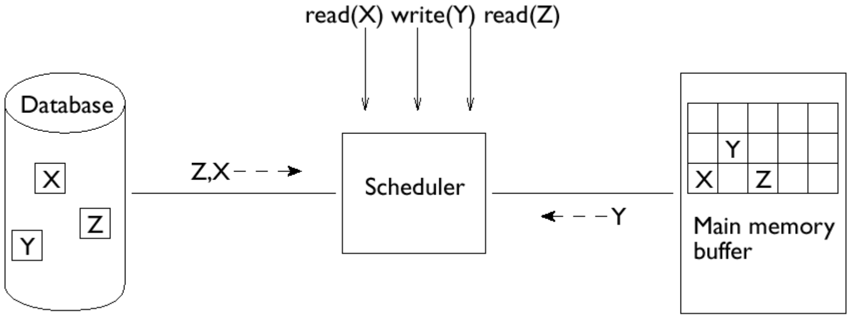
- Atomicity and durability are guaranteed by the ***Reliability Control System*.**
- Isolation is guaranteed by the ***Concurrency Control System.***
- Consistency is managed during the normal query execution by the ***DBMS System*** (verification activities are generated by the DDL Compilers and executed during query processing)

Concurrency Control

- Concurrency is measured in *tps* (transactions per second), with values that can go up to 10^5-10^6 tps.
- Typical examples of large transaction systems include banking, airline reservation systems, etc.
- We assume that transactions are input-output operations on abstract objects *x*, *y*, *z* (tuples, relations, objects,...)
- Each input-output operation reads secondary memory blocks into buffer pages or writes buffer pages into secondary memory blocks.
- For simplicity, we assume that there is a one-to-one mapping from disk blocks to memory pages.
- The main problem we have to cope with involves anomalies due to concurrent execution.

The Scheduler

- Traditionally, disk blocks are copied into pages as they are loaded into memory



Anomaly I: Update Loss

- Consider two identical transactions:
  $t\,1$: $r(x)$, $x = x + 1$, $w(x)$, $t\,2$: $r(x)$, $x = x + 1$, $w(x)$,
  and assume initially x=2; after serial execution x=4
- Consider the concurrent execution:
  - ✓ Transaction t_1 | Transaction t_2

Transaction t_1	Transaction t_2
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
commit	
	$w_2(x)$
	commit

- One update is lost, final value is x=3 instead of x=4!

Anomaly II: Dirty Read

- Consider the same two transactions, and the following execution (note that the first transaction fails):
  - ✓ Transaction t_1 — Transaction t_2

Transaction t_1	Transaction t_2
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
abort	
	$w_2(x)$
	commit

- t_2 reads from an intermediate state of t_1 (dirty read)

Anomaly III: Inconsistent Read

- t_1 repeats two reads:
  - ✓ Transaction t_1 — Transaction t_2

Transaction t_1	Transaction t_2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- t_1 reads different values for x

Anomaly IV: Ghost Update

- Assume the integrity constraint $x + y + z = 1000$;
  - ✓ Transaction t_1 / Transaction t_2

Transaction t_1	Transaction t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	`commit`
$r_1(z)$	
$s = x + y + z$	
`commit`	

- $s = 1100$, because t_1 sees a ghost update.

Concurrency Control Theory

- A transaction is a sequence of read or write actions.
- Each transaction has a unique, system-assigned transaction identifier.
- Each transactions is initiated by the **`bot`** command and terminated by **`eot`**, we won't show these in the following examples.
- For example,

$$t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)$$

- This model ignores the manipulation operations performed on the data by the transaction.

Schedules

- A schedule represents the sequence of input/output operations requested by concurrently executing transactions.
- For example,

$$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$$

- To keep things simple, we assume that transactions don't include abort statements (***commit-projection*** assumption).
- This assumption is not acceptable in practice, application-requested aborts are useful.

Concurrency Control

- We want to disallow schedules that cause any of the anomalies presented earlier.
- A ***scheduler*** is a system component that accepts or rejects the operations requested by transactions
- We are interested in ***serial schedules,*** i.e., ones where the actions to be executed by each transaction appear in sequence, e.g.,

$$S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$$
$$S_3 : r_0(x)\ r_0(y)\ w_0(x)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)\ r_1(y)\ r_1(x)\ w_1(y)$$
$$S_4 : r_1(y)\ r_1(x)\ w_1(y)\ r_0(x)\ r_0(y)\ w_0(x)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$$
$$S_5 : r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)\ r_0(x)\ r_0(y)\ w_0(x)$$
$$S_6 : r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)\ r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)$$
$$S_7 : r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)\ r_1(y)\ r_1(x)\ w_1(y)\ r_0(x)\ r_0(y)\ w_0(x)$$

Serializability

- A schedule is ***serializable*** if it produces the same result as some serial schedule S_j of the same transactions.
- We need to define what "same result" means, i.e., a notion of equivalence between schedules.
- We are going to define several notions of equivalence: **view-equivalence**, ***conflict-equivalence***, ***two-phase locking***, ***timestamp-based.***
- We want to design schedulers which allow the identification of a broad class of acceptable schedules without having to test each schedule for equivalence.

View-Serializability

- Some definitions:
  - ✓ $r_i(x)$ ***reads-from*** $w_j(x)$ in a schedule S when $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
  - ✓ $w_i(x)$ in a schedule S is a ***final write*** if it is the last write of the object x to appear in S
- Two schedules S_i and S_j are **v*iew-equivalent*** ($S_i \approx_V S_j$) if they possess the same sets of *reads-from* relations and *final writes.*
- A schedule is called ***view-serializable*** if it is view-equivalent to some serial schedule.
- The set of view-serializable schedules is called ***VSR.***

View-Serializability

- View-serializability is computationally expensive:
  - ✓ Deciding on the view-equivalence of two given schedules can be done in polynomial time;
  - ✓ Deciding on the view serializability of a generic schedule is an NP-complete problem.
- Makes sense to adopt a more limited definition of equivalence, which does not cover all cases of view-equivalence between schedules, but is computationally more tractable.

Examples of View Serializability

- S_3 : $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  S_4 : $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  S_5 : $w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
  S_6 : $w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$
- S_3 is view-equivalent to the serial schedule S_4 (and therefore, it is view-serializable)
- S_5 is not view-equivalent to S_4, but it is view-equivalent to the serial schedule S_6, and it is therefore also view-serializable
- S_7 : $r_1(x)\ r_2(x)\ w_2(x)\ w_1(x)$
  S_8 : $r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$
  S_9 : $r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$
- S_7 includes an update loss, S_8 an inconsistent read, and S_9 a ghost update; they are not view-serializable.

Conflict-Serializability

- Two actions a_i, a_j ($i \neq j$), are in ***conflict*** if both operate on the same object and at least one of them is a write.
- There can be ***read-write*** conflicts (*rw* or *wr*), and ***write-write*** conflicts (*ww*).
- Two schedules are ***conflict-equivalent*** ($S_i \approx_C S_j$) if they include the same operations and every pair of operations in conflict is in the same order in both schedules.
- A schedule is ***conflict-serializable*** if there is a conflict-equivalent serial schedule.
- The set of conflict-serializable schedules is called **CSR.**
- CSR is a proper subset of VSR.

Transactions -- 23

Testing Conflict-Serializability

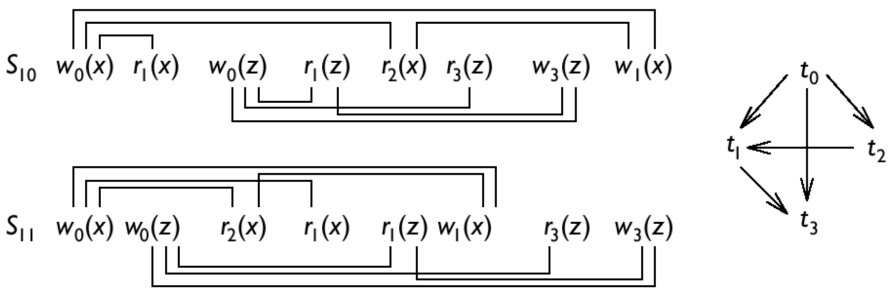
- Can be done in terms of a ***conflict graph*** which has;
  - ✓ a node for each transaction t_i;
  - ✓ an edge from t_i to t_j if there is at least one conflict between an action a_i and an action a_j such that a_i precedes a_j;
- A schedule is conflict-serializable if and only if its conflict graph is acyclic.
- Testing for cyclicity of a graph has a linear complexity with respect to the size of the graph.
- Conflict serializability is still too laborious in practice, especially for distributed databases.

<u>This is not a practical alternative either!</u>

Transactions -- 24

Example of Conflict-Serializability

- S_{10} is conflict-equivalent to the serial schedule S_{11}



Two-Phase Locking

- Used by almost all commercial DBMSs
- Basic idea: Every read is preceded by ***r_lock*** (shared lock) and followed by an ***unlock***; Every write operation is preceded by ***w_lock*** (exclusive lock), followed by *unlock*.
- A transaction execution following these rules is ***well formed wrt locking.***
- When a transaction first reads and then writes an object it can either use a write lock, or move from a shared lock to an exclusive one (***lock escalation***.)
- The ***lock manager*** receives lock/unlock requests and grants resources according to a conflict table:
  - ✓ When the lock is granted, the resource is acquired;
  - ✓ At unlock, the resource is released.

The Lock manager

Conflict table

Request	Resource state		
	free	*r_locked*	*w_locked*
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

- A counter keeps track of the number of readers for each resource and release it when counter = 0.
- If a lock request is not granted, requesting transaction is put in a waiting state; waiting ends when the resource is unlocked and becomes available.
- The locks already granted are stored in a ***lock table***, managed by the lock manager

The Two-Phase Locking Protocol

- The ***two-phase locking*** (2PL) protocol does not allow a transaction to acquire new locks after it has released a lock.
- The two phases refer to the fact that the set of locks of a transaction grows, then shrinks monotonically.
- If a scheduler uses well-formed transactions, conflict-based lock granting, and 2PL, then it produces the class of 2PL schedules.
- 2PL schedules are serializable and strictly included in CSR
- Example of a schedule that is in CSR and not in 2PL:

  S_{12} : $r_1(x)$ $w_1(x)$ $r_2(x)$ $w_2(x)$ $r_3(y)$ $w_1(y)$

Strict 2PL

- Strict 2PL imposes an additional constraint: ***The locks on a transaction can be released only after the transaction has committed/aborted.***
- This version is used by commercial DBMSs. It obviously has the disadvantage that some locks may be maintained longer that they are needed, but it does eliminate the dirty read anomaly.

Concurrency Control Based on Timestamps

- A ***timestamp*** is an identifier which defines a total ordering of temporal events within a system.
- Every transaction is assigned a timestamp *ts* that represents the time at which the transaction begins
- A schedule is accepted only if it reflects the serial ordering of the transactions based on the value of the timestamp of each transaction.

Basic Timestamp Mechanism

- The scheduler maintains a counter RTM(x) and WTM(x) for each object, and receives timestamped read and write requests for objects:
  - ✓ *read*(x,*ts*): if *ts* < WTM(x) then the request is rejected and the transaction is aborted, otherwise the request is accepted and RTM(x) is set equal to the greater of RTM(x) and *ts*
  - ✓ *write*(x,*ts*): if *ts* < WTM(x) or *ts* < RTM(x) then the request is rejected and the transaction is killed, otherwise the request is accepted and WTM(x) is set equal to *ts*
- This method causes the forced abort of a large number of transactions.

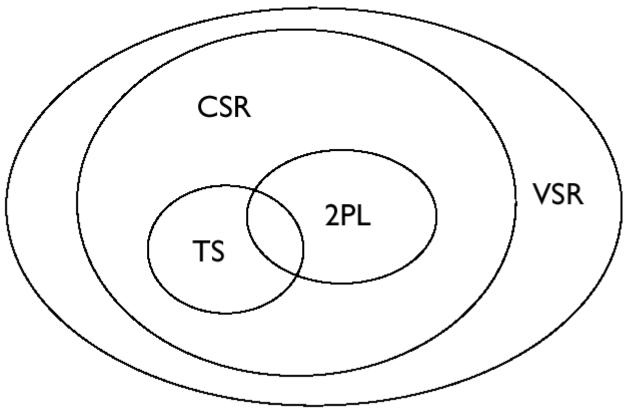
Example

Request	Response	New value
read(x,6)	ok	
read(x,8)	ok	RTM(x) = 8
read(x,9)	ok	RTM(x) = 9
write(x,8)	no	t_8 killed
write(x,11)	ok	WTM(x) = 11
read(x,10)	no	t_{10} killed

Multiversion Concurrency Control

- Writes generate new copies each with a WTM. At any time, $N \geq 1$ copies of each object x are active, with $WTM_N(x)$. There is only one global $RTM(x)$.
- Rules that make this method work:
  - ✓ *read*(x,ts) is always accepted -- select copy x_k such that if $ts > WTM_N(x)$, then $k = N$, otherwise select k such that $WTM_k(x) < ts < WTM_{k+1}(x)$
  - ✓ *write*(x,ts) -- if $ts < RTM(x)$, request is refused, else a new version of the item is added *(N* increased by one) with $WTM_N(x) = ts$
- Old copies are discarded when there are no read transactions interested in their values.

Taxonomy of VSR, CSR, 2PL and TS



2PL vs TS

- In 2PL, transactions that want an object used by another transaction are put in waiting; in TS they are killed and then restarted.
- The serialization order in 2PL is imposed by conflicts, while in TS it is imposed by the timestamps
- The necessity of waiting for the commit of the transaction causes strict 2PL and buffering of writes in TS
- 2PL can give rise to deadlocks (discussed next.)
- Restarting costs higher than waiting ones: 2PL wins!

Lock Management

- Lock manager supports three operations:
  - ✓ `r_lock(T,x,errcode,timeout)`
  - ✓ `w_lock(T,x,errcode,timeout)`
  - ✓ `unlock(T,x)`

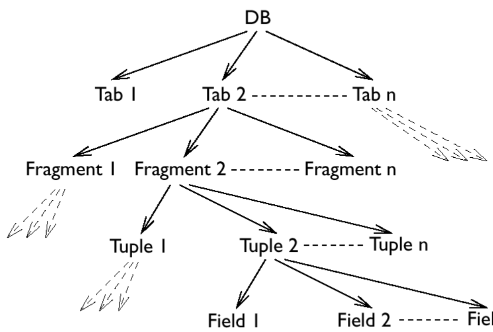
    `T`: transaction identifier
    `X`: data element
    `timeout`: max wait in queue
- If timeout expires, `errcode` signals an error, typically the transaction rolls back and restarts.

Hierarchical Locking

- In many real systems, locks can be specified at different levels of granularity, e.g. tables, fragments, tuples, fields. These are organized in a hierarchy, possibly a directed acyclic graph.
- There are five locking modes in hierarchical locking:
  - ✓ ˚Two are shared and exclusive, renamed as XL, SL;
  - ✓Three modes are new:
    - ✓ISL: intention-shared lock;
    - ✓IXL: intention-exclusive lock;
    - ✓SIXL: shared-intention-exclusive lock.
- The choice of lock granularity is left to application designers; coarse granularity means that many resources are blocked, fine means that many locks are requested.

Transactions -- 37

Hierarchica Locking Example



If we wish to place a write lock on a tuple of the table, then we must first request an IXL on the database level. When the request is satisfied, we can request an IXL for the relation and fragment in which the desired tuple lies. When these locks are granted, we can request an XL for the particular tuple. Then when the transaction is ended, it will have to release the locks in reverse order to that in which they were granted, ascending the hierarchy one step at a time.

Transactions -- 38

Hierarchical Locking Protocol

1. Locks are requested from the root to descendents in a hierarchy
2. Locks are released starting at the node locked and moving up the tree
3. In order to request an SL or ISL on a node, a transaction must already hold an ISL or IXL lock on the parent node.
4. In order to request an IXL, XL, or SIXL on a node, a transaction must already hold an SIXL or IXL lock on the parent node.

Conflicts in Hierarchical Locking

The new conflict table has as follows:

	Resource state				
Request	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Lock Options in SQL2

- Some transactions are defined as **`read-only`** (they can't request exclusive locks)
- The level of isolation can be set for each transaction.
- **`Serializable`** guarantees max isolation: keeps predicate locks so as not to change the content even of aggregate functions evaluated on data sets
- **`Repeatable read`** is equal to strict 2PL (note: repeated reading of values are the same, but repeated readings of aggregates over data are not)
- **`Committed read`** excludes the reading of intermediate states (uncommitted data)
- **`Uncommitted read`** does no concurrency control at all on read

Deadlocks

- Deadlocks are created by concurrent transactions, each of which holds and waits for resources held by others
- Example:
  - ✓ t_1: *read*(x), *write*(y), t_2: *read*(y), *write*(x)
  - ✓ Schedule: $r\_lock_1(x)$, $r\_lock_2(y)$, $read_1(x)$, $read_2(y)$ $w\_lock_1(y)$, $w\_lock_2(x)$
- This is a deadlock! t_1 is waiting for y which is blocked by t_2, and t_2 is waiting for x which is blocked by t_1.
- Deadlock probability grows linearly with number of transactions and quadratically with the number of lock requests by each transaction (under suitable uniformity assumptions)

Deadlock Resolution Techniques

- A deadlock is a cycle in the *wait-for* graph which indicates wait conditions between transactions (where nodes represent transactions, arcs wait conditions).
- There are three techniques for deadlock detection:
  - ✓ ***Timeout*** -- abort transactions that waited too long;
  - ✓ ***Deadlock detection*** -- performs the search for cycles in the wait-for graph;
  - ✓ ***Deadlock prevention*** -- kill the transactions that could cause a cycle (an overkill!).