TRANSACTION THEORY

A transaction is a logical unit of database access.

A correct transaction is
- Atomic
- Consistency Preserving
- Isolated
- Durable

Transaction theory defines allowable concurrency between sets of transactions.

From the point of view of transaction processing, a transaction is a sequence of read and write operations on data items in the database.

Theory classifies possible interleavings (schedules) of operations from a set of transactions in terms of
- recoverability and
- correctness.

In theory, we can talk about modifying the schedules to gain desired characteristics.
In reality, the schedule is determined by the real-time order in which the operations arrive. The only “rescheduling” that is possible is delaying certain operations.

Nevertheless, the theory tells us how to recognize “good/bad” schedules as they occur, so we can define policies to prevent bad schedules.

The mechanism that implements the chosen policies is called concurrency control (Chap 18)

Recoverability classes:

recoverable: if transactions conflict, we can recover by aborting one or more

cascading-abort avoiding: recoverable and guaranteed that aborting one transaction will not cause others to abort

strict: avoids cascading aborts, recoverable and recovery from an abort can be accomplished by restoring the values that existed before the aborted transaction began

serial: transaction are not interleaved, no conflict is possible

Correctness classes:

view serializable: guaranteed to be correct, but no practical test exists

conflict serializable: guaranteed to be correct, because equivalent to some serial schedule.
Concurrency Control Protocols

Two Phase Locking:
- prevents unwanted schedules by delaying conflicting operations
+ guarantees equivalence to some serial schedule
+ never requires transaction aborts due to conflict
- reduces concurrency
- may cause deadlock, livelock or starvation

Timestamping:
- rejects transactions that request operations that are out of order
order is determined by unique timestamps assigned to each transaction
+ guarantees equivalence to a particular serial schedule
+ cannot cause deadlock
- may cause (cascading) aborts due to conflict
- may cause starvation

Multiversion:
- keeps multiple versions of modified data items and selects the appropriate versions that each transaction sees.
+ reads can proceed concurrently with conflicting writes
+ avoids cascading aborts due to conflict
- requires additional storage space and maintenance
- may cause deadlock
- transaction commit may be delayed

Optimistic:
- No checking is done while a transaction is executing.
All operations are performed on local copies of data items.
Validity of the transaction is checked at commit time and invalid transactions are aborted.
+ maximal concurrency
+ no possibility of deadlock
- may cause aborts due to conflict (conflict can be tested using precedence graphs)
- determination of validity is delayed until latest possible time

Multistate:
- keep multiple values for everything for each transaction
merge resulting states by resolving conflict
+ useful for applications with long transaction: computer-aided design tools
+ some applications never merge states (temporal databases)
- conflict resolution may require user intervention
**Data Item Granularity**

When we say r1(X) or w1(X), what does X mean?

At a minimum, it means a particular data value, i.e. the number of reserved seats on an airplane, or the balance of a bank account.

However, it takes a lot of bookkeeping to keep track of all the reads and writes of each individual value.

Granularity is the size data items for which reads and writes are tracked.

Possible granularities:

- **primitive value**: an attribute of a tuple, an attribute of an object
- **record**: a whole tuple, a whole object
- **file**: a relation, a class extent
- **disk block**: some subset of tuples in a relation or some subset of objects in a class
- **entire database**: all tuples in all relations, all tuples in all classes

The tradeoff:

- finer granularity is more expensive
- courser granularity reduces concurrency

Whatever granularity is chosen the transaction theory and concurrency control policies are valid.

**Locking**

Locking protocols for concurrency control are based on the idea that transactions must request permission before accessing a data item.

If/when permission is granted, then that transaction has a lock on that data item.

The state of locks held by other transactions is used to determine if/when a request for a lock can be granted.

The locking protocol guarantees the resulting schedule will be in the desired class (of recoverability and serializability) by delaying the execution of conflicting operations until they don’t violate the scheduling criteria.

Example:

- Suppose we want to guarantee cascading-abort avoidance.
- This means that no transaction can read something written by an uncommitted transaction.
- Locking policy: If a transaction tries to read a data item written by an uncommitted transaction, delay the read until the other transaction commits.

T1: r1(X), w1(X) ...
T2: r2(Y), w2(Z), ... r1(X) ..... (delayed)

The operation r2(X) (and any T2 operations following it) will be delayed until T1 commits.

This reduces concurrency in favor of avoiding cascading abords
**Locks**

**Binary locks:**
- Holding a lock gives a transaction exclusive access to a data item.
- No other transaction can read or write that data item until the lock is released.
- + easy to implement
- - too restrictive, assumes two read operations conflict

**Shared and Exclusive Locks:**
- $s(X)$ = request a shared lock on $X$ = intent to read $X$
- $x(X)$ = request an exclusive lock on $X$ = intent to write (or read) $X$
- $u(X)$ = release the lock on $X$ (unlock)

- Before reading a data item a transaction must request a shared lock.
- Before writing a data item a transaction must request an exclusive lock.
- A request for a shared lock will be delayed as long as any other transaction holds an exclusive lock.
- A request for an exclusive lock will be delayed as long as any other transaction holds a shared lock or an exclusive lock.

<table>
<thead>
<tr>
<th>requested lock</th>
<th>$s(X)$</th>
<th>$x(X)$</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>grant delay</td>
<td>delay</td>
<td>delay</td>
<td>grant</td>
</tr>
</tbody>
</table>

Locking does not guarantee serializability or recoverability.

This requires additional restrictions on when locks are requested and released.

**Two Phase Locking (2PL)**

Every transaction runs in two phases:

- Phase 1 (expanding): locks can be requested, but not released
- Phase 2 (shrinking): locks can be released, but not requested

- basic 2PL: no further restrictions
- conservative 2PL: all locks must be acquired at start of transaction
- strict 2PL: no locks can be released until the end of the transaction

- basic 2PL: guarantees serializability
- conservative 2PL: never deadlocks
- strict 2PL: guarantees strict schedules
2PL Locking Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1(X)</td>
<td>s2(X)</td>
</tr>
<tr>
<td>r1(X)</td>
<td>x2(Y)</td>
</tr>
<tr>
<td>X = X-2</td>
<td>r2(X)</td>
</tr>
<tr>
<td>x1(X)</td>
<td>u2(X)</td>
</tr>
<tr>
<td>w1(X)</td>
<td>Y = X+3</td>
</tr>
<tr>
<td></td>
<td>&lt;delay waiting for user input&gt;</td>
</tr>
<tr>
<td></td>
<td>w2(Y)</td>
</tr>
<tr>
<td></td>
<td>u2(Y)</td>
</tr>
</tbody>
</table>

The resulting schedule is guaranteed to be serializable, since improper ordering of conflicting operations is not allowed.

The cost is less concurrency: Some transactions will be delayed waiting on other transactions.

**Conservative 2PL**

In conservative 2PL, a transaction has to acquire all locks that it will need before it does anything.

Locks can then be released at any time.

Deadlock is impossible. A transaction cannot be holding one lock while waiting on another lock.

**Strict 2PL**

In strict 2PL, a transaction can acquire locks at any time, but must release all locks simultaneously at the end of the transaction.

Deadlock is possible.

The resulting schedules are strict, meaning that recovery from aborts is easy.

Definition of a strict schedule: No transaction can read or write anything written by an uncompleted transaction.
Deadlock

Basic 2PL and strict 2PL may result in deadlock.

Deadlock is when two or more transactions are waiting for locks held by the others and no one can proceed.

The only solution is to abort one or more transactions.

Example:

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1(Y)</td>
<td>s2(X)</td>
</tr>
<tr>
<td>r1(Y)</td>
<td>r2(X)</td>
</tr>
<tr>
<td>cannot be granted until T2 unlocks X</td>
<td>x2(Y)</td>
</tr>
<tr>
<td>x1(X)</td>
<td></td>
</tr>
</tbody>
</table>

Either T1 or T2 will have to be aborted.

Note: if we are not enforcing cascading-abort avoidance, this may cause other transactions to abort.

Dealing with Deadlock

Deadlock Prevention:

- conservative 2PL: prevents deadlock by limiting locking schedules
- preemptive techniques: wait-die, wound-wait, cautious waiting: these protocols prevent deadlock by aborting transactions that might cause it.

Deadlock Detection:

- Periodically check for deadlocked transaction.
- If two or more transactions are deadlocked, chose one to abort.
- Useful if conflict between transactions is known to be rare.

In any case, aborted transactions must be restarted.

Livelock and Starvation

Livelock: A transaction waits indefinitely while other transactions proceed normally.

Starvation: A transaction is repeatedly aborted and never allowed to complete.

Livelock and starvation are caused by unfair concurrency control and deadlock prevention/detection protocols.
DATABASE RECOVERY

Recovery is the process of reestablishing a correct database state after system failure.

Recovery ensures:
atomicity: undoes the effects of transactions that do not commit
durability: makes sure committed transactions survive

Types of failure:
1: Computer failure: hardware malfunctions, software errors or program aborts
2: Transaction or system error: math error, bad parameters, logic errors, user abort
3: Local errors detected by the transaction: domain, key or integrity constraint violations
4: Concurrency control enforcement: transaction aborted due to non-serializability or deadlock
5: Disk failure: data lost due to media failure
6: Catastrophic problems: computer system damaged, disks or tapes destroyed (lost, overwritten, stolen)

DBMS Support for Recovery

Backups: A backup is a complete copy of the database at some specific time. Backups are stored external to the DBMS, preferably at a different physical location.

Archives: An archive is a complete backup of the database and log at some point in time. An archive exists on storage isolated from the database itself (another disk, a tape, etc.) Ideally, the archive is stored in a different physical location than the database.

Checkpoints: A checkpoint is an operation that forces the database on disk to be consistent with all committed transactions.

Simple checkpoint processing:
1) stop accepting new transactions.
2) wait for all active transactions to commit or abort (and log the commit or abort).
3) flush the log to the disk.
4) write a <CKPT> in the log and flush to disk.
5) resume accepting transactions.

non-quiescent checkpointing: Checkpointing while transactions are active requires processing and logging until transactions active during the checkpoint are completed.

Shadowing: Shadowing (mirroring) involves keeping an active copy of the database on another disk. All transactions are written to both the primary database and the shadow database. This allows for quick recovery in the case of failure to the primary disk.

The System Log: The system log keeps a record of the operations executed by all transactions.
- start point
- reads
- writes
- commit/abort point

This information is independent of the actual database state.

The exact information that is stored in the log depends on
- the concurrency control protocol
- the recovery protocol

A system log is also referred to as a transaction log, a trail or a journal.
Techniques for Recovery

Reprocessing

1) Restore database from latest backup.
2) Reprocess all transactions that were committed since the backup.

Reprocessing is generally not feasibly if transactions involved human inputs, or inputs from external systems.

Roll Forward

1) Restore database from latest backup.
2) Reapply all changes since the backup by extracting them from the system log.

For true durability, system log must be kept external to the DBMS.

Roll Back

1) Start with current database state
2) Undo all transactions that have corrupted the database, using the system log.

Only effective for corruption due to incorrect transactions.

Utilizes same processing used for transaction aborts (in concurrency control).

May require roll-forward if correct transactions were removed due to cascading aborts.

General algorithm for recovery from a system crash

1) Identify last checkpoint
2) Use Roll-Forward to redo all transactions committed since that checkpoint
3) Use Roll-Back to undo any transactions still active at the time of the crash. (May induce cascade aborts of committed transactions.)
**Recovery from Crash: Example**

**Initial state:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

**Checkpoint:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>27</td>
<td>35</td>
<td>43</td>
</tr>
</tbody>
</table>

**CRASH**

(end of log)

**Recovery:**

1) Undo T4: not committed at crash
2) Undo T1: read B after T4 wrote B
3) Redo T2: committed after checkpoint
4) T3: no problem, committed before checkpoint

**Recovered state:**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>22</td>
<td>35</td>
<td>47</td>
</tr>
</tbody>
</table>

(this example assumes immediate update)