Transactions

Problems Caused by Failures

- Update all account balances at a bank branch.
  Accounts(Anum, CId, BranchId, Balance)

  Update Accounts
  Set Balance = Balance * 1.05
  Where BranchId = 12345

Partial Updates - Lack of Atomicity

If the system crashes while processing this update, some, but not all, tuples with BranchId = 12345 may have been updated.
Another Failure-Related Problem

- transfer money between accounts:
  
  Update Accounts
  Set Balance = Balance - 100
  Where Anum = 8888

  Update Accounts
  Set Balance = Balance + 100
  Where Anum = 9999

Partial Updates - Lack of Atomicity

If the system fails between these updates, money may be withdrawn but not redeposited
Problems Caused by Concurrency

• Application 1:
  Update Accounts
  Set Balance = Balance - 100
  Where Anum = 8888

  Update Accounts
  Set Balance = Balance + 100
  Where Anum = 9999

• Application 2:
  Select Sum(Balance)
  From Accounts

Lack of Isolation

If the applications run concurrently, the total balance returned to application 2 may be inaccurate.
Another Concurrency Problem

• Application 1:
  Select balance into :balance
  From Accounts
  Where Anum = 8888

  compute :newbalance using :balance

  Update Accounts
  Set Balance = :newbalance
  Where Anum = 8888

• Application 2: same as Application 1

Lost Updates

If the applications run concurrently, one of the updates may be "lost".
Transactions

Transaction Properties

- Transactions are *durable*, *atomic* application-specified units of work.
  - **Atomic**: indivisible, all-or-nothing.
  - **Durable**: effects survive failures.

The “ACID” Properties

<table>
<thead>
<tr>
<th>A</th>
<th>Atomic: a transaction occurs entirely, or not at all</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Consistent</td>
</tr>
<tr>
<td>I</td>
<td>Isolated: a transaction’s unfinished changes are not visible to others</td>
</tr>
<tr>
<td>D</td>
<td>Durable: once it is complete, a transaction’s changes are permanent</td>
</tr>
</tbody>
</table>
Abort and Commit

Commit:
- When a transaction commits, any updates it made become durable, and they become visible to other transactions.
- A commit is the “all” in “all-or-nothing” execution.
- SQL: commit work

Abort:
- When a transaction aborts any updates it may have made are undone (erased), as if the transaction never ran at all.
- An abort is the “nothing” in “all-or-nothing” execution.
- SQL: rollback work
- the DBMS may unilaterally abort a running transaction
Serializability (informal)

- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order.
- If $T_i$ and $T_j$ are concurrent transactions, then either:
  - $T_i$ will appear to precede $T_j$, meaning that $T_j$ will "see" any updates made by $T_i$, and $T_i$ will not see any updates made by $T_j$, or
  - $T_i$ will appear to follow $T_j$, meaning that $T_i$ will see $T_j$’s updates and $T_j$ will not see $T_i$’s.
Concurrency Control

- Serializability can be guaranteed by executing transactions serially, but this may result in poor performance.
- Alternative: allow transactions to execute concurrently, but use a concurrency control protocol is used to ensure that their execution is serializable.
- Some tools used by concurrency control protocols:
  - block, or delay, operations
  - abort transactions
  - multi-versioning
- Many concurrency control protocols have been proposed, based on:
  - locking, or
  - timestamps, or
  - conflict analysis
Two-Phase Locking

• The rules
  1. Before a transaction may read or write an object, it must have a lock on that object.
     • a *shared lock* is required to read an object
     • an *exclusive lock* is required to write an object
  2. Two or more transactions may not hold conflicting locks on the same object at the same time
     • exclusive locks conflict with exclusive and shared locks
     • shared locks do not conflict with other shared locks
  3. Once a transaction has released (unlocked) any object, it may not obtain any new locks.
     • Phase 1: acquiring locks, Phase 2: releasing locks
     • In *strict* 2PL, all locks are held until commit or abort.

If all transactions use two-phase locking, the execution history is guaranteed to be serializable.
Transaction Blocking

- Consider the following sequence of events:
  - $T_1$ acquires a shared lock on $x$ and reads $x$
  - $T_2$ attempts to acquire an exclusive lock on $x$ (so that it can write $x$)

- The two-phase locking rules prevent $T_2$ from acquiring its exclusive lock - this is called a lock conflict.

- Lock conflicts can be resolved in one of two ways:
  1. $T_2$ can be blocked - forced to wait until $T_1$ releases its lock
  2. $T_1$ can be pre-empted - forced to abort and give up its locks
Transactions

Deadlocks

- transaction blocking can result in *deadlocks*
- for example:
  - $T_1$ reads object $x$
  - $T_2$ reads object $y$
  - $T_2$ attempts to write object $x$ (it is blocked)
  - $T_1$ attempts to write object $y$ (it is blocked)

A deadlock can be resolved only by forcing one of the transactions involved in the deadlock to abort.
Serializability Theory: A Brief Detour

- A transaction is a sequence of read and write operations.
- An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.
- Two operations conflict if:
  - they belong to different transactions
  - they operate on the same object
  - at least one of the operations is a write
Serial and Serializable Histories

- \( T_1 = w_1[x] w_1[y] \), \( T_2 = r_2[x] r_2[y] \)
- An interleaved execution of \( T_1 \) and \( T_2 \):
  \[ H_a = w_1[x] r_2[x] w_1[y] r_2[y] \]
- An equivalent serial execution of \( T_1 \) and \( T_2 \):
  \[ H_b = w_1[x] w_1[y] r_2[x] r_2[y] \]
- An interleaved execution of \( T_1 \) and \( T_2 \) with no equivalent serial execution:
  \[ H_c = w_1[x] r_2[x] r_2[y] w_1[y] \]

\( H_a \) is serializable because it is equivalent to \( H_b \), a serial schedule. \( H_c \) is not serializable.
Testing for Serializability

A history is serializable iff its serialization graph is acyclic.
Transactions

Serialization Graphs

\[
\begin{align*}
  r_1[x] & \quad r_3[x] & \quad w_4[y] & \quad r_2[u] & \quad w_4[z] & \quad r_1[y] & \quad r_3[u] & \quad r_2[z] & \quad w_2[z] & \quad r_3[z] & \quad r_1[z] & \quad w_3[y] \\
\end{align*}
\]

The history above is equivalent to

\[
\begin{align*}
  w_4[y] & \quad w_4[z] & \quad r_2[u] & \quad r_2[z] & \quad w_2[z] & \quad r_1[x] & \quad r_1[y] & \quad r_1[z] & \quad r_3[x] & \quad r_3[u] & \quad r_3[z] & \quad w_3[y] \\
\end{align*}
\]

That is, it is equivalent to executing $T_4$ followed by $T_2$ followed by $T_1$ followed by $T_3$. 
Two-Phase Locking Revisited

- How does 2PL ensure serializability?
- Consider again non-serializable $H_c$:
  \[ H_c = w_1[x] \ r_2[x] \ r_2[y] \ w_1[y] \]
- 2PL prevents non-serializable histories by blocking (and hence reordering) operations that might lead to non-serializability
- 2PL can be too conservative - it may also prevent some serializable histories from occurring
  \[ H_\alpha = w_1[x] \ r_2[x] \ w_1[y] \ r_2[y] \]
Transactions

Phantoms

Transaction 1:
Insert Into Employee
Values ('123', 'Shel', 'Jetstream', 'D11', 52000)

Transaction 2:
Select *
From Employee
Where WorkDept = 'D11'

Select *
From Employee
Where Salary > 50000

• Transaction 2 may observe a phantom tuple, not possible in a serial execution.
• 2PL of database records (or pages) may not be enough to ensure serializability in the presence of insertions or deletions.
Insertions, Deletions and Serializability

- Queries must conflict with (and hence “lock”) all tuples that satisfy the query predicates, including previously-deleted or to-be-inserted tuples that are not currently in the database.
- One solution: relation-level locks
- Another solution: index key-range locking
  - consider query with predicate $\text{WorkDept} = 'D11'$
  - suppose there is an index on $\text{WorkDept}$
  - query locks index key range covering ‘D11’, preventing insertions in that range
Multi-Granularity Locking

- allow transactions to lock entire relation, or individual records in the relation, as appropriate
- before locking smaller granules (e.g., records), set intention-mode locks on the containing larger granule (e.g., relation)
- lock conflict table, including intention-mode locks

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>S</th>
<th>IX</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>S</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>IX</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IS</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

- IX = intention exclusive, IS = intention shared
Snapshot Isolation (SI)

- every transaction $T$ “sees” a snapshot of the database
  - $T$’s snapshot includes updates made by all transactions that committed before $T$ starts
  - $T$’s snapshot does not include any updates made by concurrent transactions
- each transaction sees its own updates
- concurrent transactions may not perform conflicting updates

<table>
<thead>
<tr>
<th>Si vs. serializability</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pro:</strong> read-only transactions never block</td>
</tr>
<tr>
<td><strong>Con:</strong> potential anomalies from non-serializable behavior</td>
</tr>
</tbody>
</table>
Implementing Snapshot Isolation

- multi-versioning
  - if transaction $T$ updates object (e.g., page) $p$, concurrent transactions must continue to see pre-update version of $p$

- detecting write-write conflicts
  - can be done using write locks (no need for read locks)
  - can be done using commit-time validation
    - each transaction $T$ maintains a list of updated objects (e.g., pages)
    - when $T$ tries to commit, ensure no already-committed concurrent transactions updated any of the same objects. If no conflicts, commit $T$ else abort $T$.
    - implements a first committer wins rule for concurrent conflicting updates
Recovery Management

Recovery management means:

- implementing voluntary or involuntary rollback of individual transactions
- implementing recovery from system failures so that transaction ACID properties are guaranteed
- system failure means:
  - the database server is halted
  - processing of in-progress SQL command(s) is halted
  - connections to application programs (clients) are broken.
  - contents of memory buffers are lost
To ensure that transactions are atomic, every transaction that is active when a system failure occurs must either be

- restarted after the failure from the point it which it left off, or
- rolled back after the failure

It is difficult to restart applications after a system failure, so the recovery manager does the following:

- abort transactions that were active at the time of the failure
- ensure that changes made by transactions that committed before the failure are not lost
Buffers and Persistent Storage

![Diagram of database system components]

- **Client lib**
- **DBMS**
- **buffer cache**
- **DB**

Cheriton School of Computer Science

CS448 Database Systems Implementation (Winter 2012)
Committing a Transaction

Suppose that a running transaction results in the following sequence of events:
- update page \( P_a \)
- update page \( P_b \)
- update page \( P_c \)
- update page \( P_d \)
- commit

How should this transaction be committed?
Transactions Logging

- log is an **append-only** list of log records
- log tail (most recent entries) is in memory
- log entries are flushed from log buffer to disk in order
- log entries are flushed in batches when possible
Committing a Transaction Using Logging

Events for transaction $T$:
- update page $P_a$
- update page $P_b$
- update page $P_c$
- update page $P_d$
- commit

Logging for $T$:
- log updated state of $P_a$
- log updated state of $P_b$
- log updated state of $P_c$
- log updated state of $P_d$
- log commit record for $T$

- ensure the commit record is on disk before acknowledging commit to application
- buffer manager need not flush updated pages to disk
- use log records to restore effects of $T$ after a failure
UNDO Logging and WAL

- the previous example illustrated REDO logging
  - each log entry describes how to re-apply an update that has been lost
  - REDO logging is used to ensure durability of committed updates

- UNDO logging is also useful
  - an UNDO log entry describes how to erase or undo an update that has already been applied
  - UNDO logging is used to ensure that the effects of aborted transactions are erased.

Write-Ahead Logging (WAL) Protocol

The WAL protocol requires that the log record describing an update be on persistent storage before the update itself is on persistent storage.
Log Example

log head  →  $T_0$, begin
(oldest part of the log)  $T_0$, $P_d, d_0, d_1$
$T_1$, begin
$T_1$, $P_b, b_0, b_1$
$T_2$, begin
$T_2$, $P_c, c_0, c_1$
$T_1$, $P_a, a_0, a_1$
$T_1$, commit
$T_3$, begin
$T_2$, abort

on disk

in memory

(newest part of the log)

log tail  →  $T_3$, commit
$T_2$, abort
$T_3$, $P_b, b_1, b_2$
$T_4$, begin
$T_4$, $P_a, a_1, a_2$
Log-Based Recovery

- on recovery from a failure, use the log to ensure that the database
  - contains the effects of all committed transactions
  - does not contain any effects of aborted transactions before allowing new transactions to begin executing
- simple two-pass log-based recovery
  Pass 1 (tail to head): identify losers and winners and rollback the losers
  Pass 2 (head to tail): redo the winners
- this simple log and recovery algorithm assumes
  - page-level locking (why??)
  - idempotent log records

Transaction Commit

A transaction is atomically and durably committed when its commit log record is flushed to the log disk.
Transactions

Checkpoints

- As the log grows, the time required to recover from a system failure also grows.
- **checkpoints** are used to reduce the amount of log data that must be scanned after a system failure.
- simple checkpointing algorithm
  1. prevent new transactions from starting, and wait for active transactions to finish
  2. flush all modified pages from the buffer pool to the disk
  3. truncate the log
- simple algorithm is effective but expensive
- other checkpointing algorithms try to achieve a similar effect with less impact on performance
Buffer Management and Transactions

- Force vs. No Force Buffer Management
  - **Force**: All of a transaction’s changes are present on the disk by the time the transaction commits.
  - **No Force**: Some of a transaction’s changes may not be on disk by the time the transaction commits.

- Steal vs. No Steal Buffer Management
  - **Steal**: Uncommitted changes may be present on the disk.
  - **No Steal**: Uncommitted changes are never present on the disk.
Buffer Management and Transactions (cont’d)

Force, No Steal: no logging required, but impractical (why?)

Force, Steal: UNDO logging required, transaction commits are expensive

No Force, No Steal: REDO logging required, No Steal constrains buffer manager

No Force, Steal: REDO and UNDO logging required, only constraint on buffer management is WAL
Enforcing WAL

- each log entry is assigned a log sequence number (LSN)
- log manager tracks safeLSN, the largest LSN among log entries that have been flushed to disk
- for each page \( p \), buffer manager tracks \( \text{pageLSN}(p) \), the LSN of the most recent update applied to \( p \)
- buffer manager may not flush a dirty page \( p \) to disk unless \( \text{pageLSN}(p) \leq \text{safeLSN} \)
Introduction to ARIES

- ARIES has the same goal (durable, failure-atomic transactions) as the simple logging technique already described, but tries to achieve it with less impact on transaction performance during normal (non-failure) operation.

- Some differences between ARIES and simple technique:
  - ARIES allows fine-grained (i.e., record-level) locking
  - ARIES allows operational logging
  - ARIES supports non-idempotent log operations
  - ARIES supports fuzzy checkpointing
  - ARIES recovery uses three log passes, not two
Physical State Logging

\[ Op_1 \]
\[ s_0 \rightarrow s_1 \]

- REDO log info for \( Op_1 \) consists \( s_1 \), the after-image of the affected object (e.g., page)
- UNDO log info for \( Op_1 \) consists of \( s_0 \), the before-image of the affected object
- the object (page) must remain locked until \( Op_1 \)'s transaction commits, to avoid lost updates
- log entries are idempotent: REDOing \( Op_1 \) multiple times has the same effect as REDOing it one time. Same for UNDO.
(Page-level) Operational Logging

$Op_1$

$s_0 \rightarrow s_1$

- REDO log info for $Op_1$ consists of a description of $Op_1$
- UNDO log info for $Op_1$ consists of a description of a compensating operation for $Op_1$.
- the affected object (page) need not remain locked until $Op_1$’s transaction commits
- log entries may not be idempotent
Eventually Exactly Once Execution

- recall that ARIES tracks $\text{pageLSN}(p)$ for each page $p$
- suppose the ARIES is REDOing a logged operation, with $\text{LSN} = n$, on page $p$
- ARIES will only REDO the operation if $n > \text{pageLSN}(p)$, otherwise the page already reflects the effects of the operation.
Rolling Back Transactions in ARIES

- each compensation is logged and gets a LSN
- CLR = compensation log record
- compensations are never undone
Checkpointing in ARIES

- ARIES checkpoints periodically during normal operation
- Checkpointing involves logging checkpoint information including:
  - List of active transactions
  - LSN of most recent log record for each active transaction
  - List of dirty buffered pages, including their recLSNs.
- recLSN(p) is the LSN of the update that made p dirty.
- There is no need to quiesce transactions.
- There is no need to flush any pages to the disk - the buffer manager can do this any time (as long as WAL is observed), asynchronously.
- Called fuzzy checkpointing: no sharp log boundary.
Recovery in ARIES

Analysis Pass:  
- start at most recent complete checkpoint  
- redoLSN is minimum recLSN(p) of pages in the checkpointed dirty page list  
- scan forward to identify losers and the most recent LSN for each

REDO Pass:  
- start at redoLSN  
- scan forwards, redoing all updates, including compensations and loser updates

UNDO Pass:  
- scan backwards, perform compensations for uncompensated updates of losers
ARIES Log Example

log head → 1, $T_0, P_d$, update, prev=NULL
2, $T_1, P_b$, update, prev=NULL
3, $T_2, P_c$, update, prev=NULL

checkpoint: active=$T_0(1), T_1(2), T_2(3)$,
dirty=$P_c(3), P_d(1)$

4, $T_1, P_a$, update, prev=2
5, $T_1$, commit
6, $T_2, P_a$, update, prev=3
7, $T_2$, CLR for LSN=6, prev=3
8, $T_3, P_b$, update, prev=NULL
9, $T_3, P_c$, update, prev=8
10, $T_2$, CLR for LSN=3, prev=NULL
11, $T_2$, abort
12, $T_4, P_a$, update, prev=NULL

log tail → 13, $T_3$, commit