Transactions

- Many enterprises use databases to store information about their state
  - *E.g.*, balances of all depositors

- *When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way*
  - *E.g.*, balance must be updated when you deposit

- Such a program is called a **transaction**
ACID Properties of Transactions

- Transaction execution must maintain the **correctness** of the database model
- Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs
  - Atomicity
  - Consistency
  - Isolation
  - Durability

ACID Properties

- **Atomic** - Transaction should either **complete or no effect** at all
  - Responsibility of transaction processing system

- **Consistent** - Transaction should **correctly transform** the database state to reflect the effect of a real world event
  - Responsibility of transaction designer

- **Isolation** - The effect of concurrently executing a set of transactions is the same as if they had executed serially *(serializable)*
  - Responsibility of transaction processing system

- **Durable** - The effect of a transaction on the database state **should not be lost** once the transaction has committed
  - Responsibility of transaction processing system
Isolation

- Serial execution of a set of (consistent) transactions is correct, but performance might be inadequate.

- Concurrent (interleaved) execution of a set of transactions offers performance benefits, but might not be correct.

Interleaved Execution
Serializable Schedules

- The concurrent schedule \( S: r_1(x) w_2(z) w_1(y) \) is equivalent to the serial schedules of \( T1 \) and \( T2 \) in either order:
  - \( T1, T2: r_1(x) w_1(y) w_2(z) \) and
  - \( T2, T1: w_2(z) r_1(x) w_1(y) \)

  since operations of distinct transactions on different data items commute. Hence, \( S \) is a serializable schedule.

- Is \( S \) equivalent to \( T2, T1? \)

  - No! Since read and write operations (or two write operations) of distinct transactions on the same data item do not commute.
Non-Serializable Schedule

- Example: course registration; \textit{cur\_reg} is the number of current registrants

\begin{align*}
T1: \text{r}(\text{cur\_reg} : 29) & \quad \text{w}(\text{cur\_reg} : 30) \\
T2: & \quad \text{r}(\text{cur\_reg} : 29) \quad \text{w}(\text{cur\_reg} : 30)
\end{align*}

- Schedule not equivalent to \textit{T1}, \textit{T2} or \textit{T2, T1}
- Database state no longer corresponds to real-world state, integrity constraint violated

Commutativity

- Two operations commute if, when executed in either order:
  - The values returned by both are the same and
  - The database is left in the same final state
- Two schedules are “equivalent” if one can be derived from the other by a series of simple interchanges of commutative operations
- A schedule is serializable if it is equivalent to a serial schedule
Concurrent Control

- Performance requirements might not be achievable if schedules are serializable
- In addition to serializable, DBMSs implement less stringent isolation levels
  - Serializable schedules correct for all applications
  - Less stringent levels do not guarantee correctness for all applications, but are correct for some
- The concurrency control of a DBMS is responsible for implementing isolation levels
- Application programmer is responsible for choosing appropriate level

Implementing Serializability: Two-Phase Locking

- Locks are associated with each data item
- A transaction must acquire a read (shared) or write (exclusive) lock on an item in order to read or write it
  - A write lock on an item conflicts with all other locks on the item; a read lock conflicts with a write lock
  - If T1 requests a lock on x and T2 holds a conflicting lock on x, T1 must wait
Lock Release

Two-Phase locking: All locks are acquired before any lock is released

Strict: Transaction holds all locks until completion

Correctness of Strict Two-Phase Locking

- Intuition: Active transactions cannot have executed operations that do not commute (since locks required for non-commutative operations conflict)
- Hence, a schedule produced by a two-phase locking concurrency control is serializable since operations of concurrent transactions can always be reordered to produce a serial schedule
Non-Strict Concurrency Controls

- Non-strict controls: locks can be released before completion
- Problem: (Bank account example)
  \[ w_1(Bal) \ u_2(Bal) \ r_2(Bal) \ w_2(Cred-Lim) \ commit_2 \ abort_1 \]
  - Although \( abort_1 \) rolls \( Bal \) back, the new value of \( Cred-Lim \) might have been affected
    - The new credit limit might have been based on a deposit that never happened
    - \( T1 \) has an effect even though it is aborted
    - Hence, atomicity is violated

Anomalies in Non-Serializable Schedules

- Dirty read (previous example – write lock given up early)
  \[ w_1(x) \ r_2(x) \ abort_1 \]
- Non-Repeatable Read (read lock given up early)
  \[ r_1(x) \ w_2(x) \ commit_2 \ r_1(x) \]
- Lost Update (result of non-repeatable read – read lock given up early)
  - Two transactions trying to deposit in the same bank account – the deposit of transaction 2 is lost
  \[ r_1(Bal) \ r_2(Bal) \ w_2(Bal) \ commit_2 \ w_1(Bal) \ commit_1 \]
Deadlock

- When a transaction can hold locks and request another lock (e.g., in two-phase locking), a cycle of waiting transactions can result:
  - Suppose two transactions are both trying to update the value of \( x \) (for example to deposit in the same bank account)
    \( r_1(x) \) \( r_2(x) \) \( request_w_1(x) \) \( request_w_2(x) \)
  - A transaction in the cycle must be aborted by DBMS (since transactions will wait forever)
  - DBMS uses deadlock detection algorithms or timeout to deal with this

Locking in Relational Databases

```
SELECT *
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Semester = 'F2000'
```

- Locking entire table restricts concurrency
- Locking only rows returned yields new anomaly

T1: execute SELECT
T2: insert a new row satisfying WHERE clause
T1: execute SELECT again

- Inserted row is called a phantom
- Strict two-phase row locking does not prevent phantoms
ANSI Standard Isolation Levels

- Defined in terms of anomalies
  - Anomaly prohibited at one level is also prohibited at all higher levels
  - READ UNCOMMITTED: all anomalies possible
  - READ COMMITTED: dirty read prohibited
  - REPEATABLE READ: reads of individual tuples are repeatable (but phantoms are possible)
  - SERIALIZABLE: phantoms prohibited; transaction execution is serializable

Locks in Relational Databases

- DBMS guarantees that each SQL statement is isolated
- Early (non-strict) lock release used to implement levels
  - Short-term locks - held for duration of single statement
  - Long-term locks - held until transaction completes (strict)
- At all levels, transactions obtain long-term write locks
Locking Implementation of Isolation Levels

- **READ UNCOMMITTED** - no read locks (dirty reads possible since transaction can read a write-locked item)
- **READ COMMITTED** - short-term read locks on rows (non-repeatable reads possible since transaction releases read lock after reading)
- **REPEATABLE READ** - long-term read locks on rows (phantoms possible)
- **SERIALIZABLE** - combination of table, row, and index locks

Atomicity and Durability

- **Atomicity** deals with failure:
  - User aborts transaction (e.g., cancel button)
  - System aborts transaction (e.g., deadlock)
  - Transaction aborts itself (e.g., unexpected db state)
  - System crashes
- **Durability** deals with failure:
  - Media failure
- Mechanism for dealing with failures is the log
Log

- Log:
  - Append-only sequence of records used to restore database to a consistent state after a failure.
  - Stored on non-volatile device distinct from mass storage device that contains database
    - Survives processor crash and media failure
- Update record:
  - Appended to log when a transaction updates an item
  - Contains "before image": value of item prior to update
  - Used to restore item when transaction is aborted

Aborting a Transaction

- Transaction abort:
  - Scan log backward; apply "before image" in each of the transaction's update records to database items to restore them to their original state. Begin Record terminates scan
Log

End of scan when T2 is aborted

End of log when T2 is aborted

B - begin
U - update

Recovery From Crash

- Crash:
  - **Active transactions must be identified and aborted when system recovers**
  - *Commit and Abort Records* identify completed transactions. If, during a backward log scan, the first record encountered for T is an update record, T was active at time of crash and must be rolled back
Log

<table>
<thead>
<tr>
<th>B</th>
<th>U</th>
<th>B</th>
<th>U</th>
<th>U</th>
<th>B</th>
<th>A</th>
<th>U</th>
<th>C</th>
<th>B</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T1</td>
<td>T2</td>
<td>T1</td>
<td>T2</td>
<td>T1</td>
<td>T3</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
<td>T4</td>
</tr>
</tbody>
</table>

End of scan for crash recovery

End of log when crash occurs, roll back T1 and T4 on recovery

B - begin
U - update
C - commit
A - abort

Commit

- Transaction is not committed until its commit record is in the log
- A crash at any time before that causes transaction to be rolled back
Write-Ahead Log

- Both the log and database must be updated when a transaction modifies an item. If a crash occurs between updates, abort the transaction
  - Database updated first - On recovery, item is in the new state but there is no before image to roll it back. Transaction cannot be aborted.
  - Log updated first - On recovery, item in old state and before image in log. Use of before image has no effect, but transaction can be aborted
- Update record in log must be written-ahead of update to item in database

Practical Database Systems

- Two writes to mass store for each database update implies intolerable performance
- Real world:
  - DBMS maintains cache of recently accessed pages in memory. Most accesses are to cache. Pages which have been updated eventually written to disk
  - DBMS maintains log buffer in memory. Records appended to buffer until it fills; then buffer written to log
  - Maintaining write-ahead feature more complex when buffers taken into account
Media Failure

- Durability requires that database be stored redundantly
- Log can be used as second copy if:
  - Update records contain *after image* (as well as before image): new value of item
  - A snapshot, or dump, of the database is periodically stored in non-volatile memory
- Recovery: Starting with most recent dump, play the log forward, update database using after images appended subsequent to dump
Distributed Transactions

• Information supporting a large enterprise is generally stored on multiple computer systems scattered through the enterprise
  • E.g., Bank has a database at each branch recording local branch data and a database at the main office recording aggregate data
  • Each DBMS is independent, supporting local transactions at a site
  • With increased enterprise integration and automation, *global*, or *distributed*, transactions, involving multiple sites must also be supported

Distributed Transaction

• A transaction that invokes local transactions
  • E.g., Bank transfer: invoke withdraw at one site and deposit at another
  • A system that supports distributed transactions is often referred to as a *multidatabase system*
  • In addition to the local integrity constraints that apply at each site, *global integrity constraints* might also exist
    • E.g., Aggregate bank assets at central site = sum of assets at each branch
ACIDity of Distributed Transaction

- Although a distributed transaction is consistent, maintaining isolation in a multi-database is an important issue
- Even if local sites are serializable, sub-transactions of two distributed transactions might be serialized in different orders at different sites
  - At site A, T_{1A} is serialized before T_{2A}
  - At site B, T_{2B} is serialized before T_{1B}

ACIDity of Distributed Transaction

- Although a distributed transaction is consistent, maintaining atomicity in a multidatabase is an important issue
- Guaranteeing that sub-transactions of a distributed transaction either all commit or all abort in spite of failures (e.g., message loss, site crash) requires the use of a two-phase commit protocol
Two-Phase Commit Protocol

- Implemented as an exchange of messages between the coordinator and the cohorts
  - The cohorts are the individual subtransactions that participated in the transaction
  - The coordinator polls the cohorts to see if they want to commit

Two-Phase Commit Protocol

- *Prepare message* (coordinator to cohort):
  - If cohort wants to abort, it aborts
  - If cohort wants to commit, it moves all update log records to non-volatile store and forces a prepared record to its log
  - Cohort sends a (ready or aborting) *vote message* to coordinator
Two-Phase Commit Protocol

- **Vote message** (cohort to coordinator): Cohort indicates ready to commit or aborting.
  - If any are aborting, coordinator decides abort
  - If all are ready, coordinator decides commit and forces commit record to its log
  - Coordinator sends commit/abort message to all cohorts that voted ready

Two-Phase Commit Protocol

- **Commit/abort message** (coordinator to cohort):  
  - Cohort commits locally by forcing a commit record to its log. Or, if abort message, it aborts
  - Cohort sends done message to coordinator

- **Done message** (cohort to coordinator):  
  - When coordinator receives done message from all cohorts, it writes a complete record to its log
Global Serializability

- **Theorem:** If all sites use a two-phase locking protocol and a two-phase commit protocol is used, transactions are globally serializable
  - Transactions are serialized in the same order at every site – the order in which the transactions committed

ACIDity of Distributed Transactions

- Global deadlock can be another result of implementing two-phase locking and two-phase commit protocols
  - At site A, T_{1A} is waiting for a lock held by T_{2A}
  - At site B, T_{2B} is waiting for a lock held by T_{1B}
- System uses deadlock detection algorithms or timeout to deal with this
Replication

- Information is often replicated in a distributed system
  - Performance enhancement possible: access to a local replica replaces network communication
  - Availability improved: if a site containing a data item is unavailable, access a replica at a different site
- Major implementation problem: how do you keep the replicas synchronized when a replicated data item is updated?

Implementation of Replication

- DBMSs provide *replica control modules* to make replication invisible to the application
- Typical implementation: *read one/write all*
  - When application requests to read an item, replica control fetches the nearest copy
  - When application requests to write an item, replica control updates all copies
- As compared with non-replicated systems, performance of read better, of write worse
Implementation of Replication

- **Synchronous update systems**: all replicas updated as part of transaction. Supports serializability, but performance bad, deadlocks frequent, and cannot handle disconnected sites.

- **Asynchronous update systems**: one replica updated as part of transaction. Others updated after transaction commits. Performance better, deadlocks less frequent, and disconnected sites can be supported, but serializability sacrificed.

- Practical systems are generally asynchronous.

Correctness

- Application designers must be aware of the fact that real-world systems do not always support ACID executions even if all transactions are consistent.
  - Isolation levels lower than **SERIALIZABLE** might be used.
  - Two-phase commit protocol might not be used.
  - Replication might use asynchronous update.