Transaction Management

❖ Motivation and Basics
  The ACID Properties
❖ Transaction Schedules
  Conflicts and Other Issues
❖ Concurrency Control
  SQL/RDBMS Features
❖ Basics of Recovery
Motivation

❖ An RDBMS does not exist in a vacuum!
  ❖ It manages structured data at scale for an application
  ❖ Banking, insurance, finance, retail, telecom, etc.
  ❖ Workload could involve a mix of reads and writes
❖ Application determines schema of databases in RDBMS, integrity constraints, instance content, physical design, etc.

Application-independence is a key benefit of RDBMSs!

Q: But at what level does an application deal with an RDBMS?
Motivation

- An SQL query is often too fine-grained for an application
- In most real-world applications, a “logical unit of work” could be coarser / high-level:
  - May need multiple DDL + DML queries together
  - May involve reading and/or writing (update/delete/insert)
  - May need to check for integrity constraints
- A “transaction” is a sequence of operations on the database that captures one logical unit of work for an application
  (NB: Not really specific to the relational data model!)
Motivating Example

❖ A logical unit of work in a bank’s DBMS:

“Transfer $5000 from Checking to Savings account”

❖ A sequence of fine-grained operations on the database:

Read (C)  
Assert (C >= 5000)  
Write (C ← C – 5000)  
Read (S)  
Write (S ← S + 5000)

C is Checking balance
S is Savings balance
One logical unit of work
Transaction Management Overview

❖ **Transaction:**

DBMS abstraction to capture a sequence of database operations that form one logical unit of work

❖ **Transaction Manager:**

Software module in a DBMS that manages transactions

❖ **Benefits of having the “transaction” abstraction:**

Enables us to reason about concurrent queries (e.g., DMA, multi-core parallelism)

Enables us to reason about recovery from crashes
Q: What operations does a transaction (txn) contain?

- Txn Manager understands only “Read” (R) and “Write” (W)
- **Granularity** can vary: item, tuple, relation, etc.
- All SQL queries mapped to a sequence of R/W!
- Three additional special operations:
  - “Begin”, “Commit”, “Abort”
Motivating Example

Application’s unit of work

- Read (C)
- Assert (C >= 5000)
- Write (C ← C – 5000)
- Read (S)
- Write (S ← S + 5000)

The abstract transaction

- R (C)
- W (C)
- R (S)
- W (S)

Disk

C 8000
S 2000

RAM

C 8000
S 2000
Seems to work fine. What is the point of a “transaction” abstraction then?
What if this happens?

Application’s unit of work

| Read (C) | Assert (C >= 5000) | Write (C ← C − 5000) | Read (S) | Write (S ← S + 5000) |

The abstract transaction

| R (C) | W (C) | R (S) | W (S) |

OS crashes!

Your $5000 has vanished! 😞
Transaction Commits

Application’s unit of work

Read (C)
Assert (C >= 5000)
Write (C ← C – 5000)
Read (S)
Write (S ← S + 5000)

The abstract transaction

Begin
R (C)
W (C)
R (S)
W (S)
Commit

Tells the DBMS that the transaction has finished successfully and all of its changes need to be persisted (on disk)!

C 8000
S 2000
C 8000
S 2000
Disk
RAM
Transaction Aborts

Application’s unit of work

- Read (C)
- Write (C ← C – 5000)
- Assert (C ≥ 0)
- Read (S)
- Write (S ← S + 5000)

The abstract transaction

Begin
- R (C)
- W (C)
- Abort

Disk

RAM

C 4000
S 2000

Tells the DBMS that the transaction failed (for some reason) and it should be terminated without persisting any of its changes!
Moral of the Story

❖ A transaction must satisfy “all or nothing” property

Application’s “logical unit of work” is indivisible
Either all operations get done or none of them get done

❖ The database state must remain consistent

Application’s invariants define what is consistent
A transaction is assumed to get the database from one consistent state to another (inconsistency ok in between)

C=8000  ✔️
S=2000  ✔️

X C=3000, S=2000
X C=8000, S=7000
X C=3000
S=7000
Transaction Management

- Motivation and Basics
  - The ACID Properties
- Transaction Schedules
  - Conflicts and Other Issues
- Concurrency Control
  - SQL/RDBMS Features
- Basics of Recovery
The ACID Properties

- **Transaction management should ensure 4 key properties**
- **Atomicity**
  - Techniques: **Logging** and **Recovery**
  - A transaction should be *indivisible*/*“all or nothing”*
- **Consistency**
  - App semantics; Techniques: **Schedules**
  - Database should not become *inconsistent* in the end
- **Isolation**
  - Techniques: **Concurrency Control**
  - A transaction should not worry about or interact with other *concurrent* transactions on the DBMS
- **Durability**
  - Techniques: **Logging** and **Recovery**
  - All changes of a “Committed” transaction must *persist*
Transaction Management

- Motivation and Basics
  - The ACID Properties
  - Transaction Schedules
  - Conflicts and Other Issues
- Concurrency Control
- SQL/RDBMS Features
- Basics of Recovery
## Transaction Schedules

- Tells us how to *interleave* concurrent transactions
- Benefits of interleaving:
  - Lets us exploit disk-CPU and multi-core *parallelism*
  - Helps avoid "starvation" of shorter transactions

Temporal order of ops in a transaction must be preserved!

### A “schedule”
```plaintext

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>T1</td>
<td>R$_{T1}(A)$</td>
<td>R$_{T2}(A)$</td>
</tr>
<tr>
<td></td>
<td>W$_{T1}(A)$</td>
<td>W$_{T2}(A)$</td>
</tr>
<tr>
<td>T1</td>
<td>R$_{T1}(B)$</td>
<td>R$_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td>W$_{T1}(B)$</td>
<td>W$_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td>T1</td>
<td>R$_{T2}(B)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W$_{T2}(B)$</td>
<td>Commit</td>
</tr>
</tbody>
</table>
```

Transaction Schedules

❖ **Complete** Schedule
Each transaction ends with either a Commit or an Abort

❖ **Serial** Schedule
No interleaving of ops from different transactions

Any serial schedule is considered "acceptable", even if they end up with different database states!
**Q:** What is wrong with this schedule?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>$R_{T1}(A)$</td>
<td>$R_{T2}(A)$</td>
</tr>
<tr>
<td></td>
<td>$W_{T2}(A)$</td>
</tr>
<tr>
<td>$W_{T1}(A)$</td>
<td></td>
</tr>
<tr>
<td>$R_{T1}(B)$</td>
<td></td>
</tr>
<tr>
<td>$W_{T1}(B)$</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td>$R_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td>$W_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>

The update of A by T2 is lost!

“Bad” schedules like this could lead to an **inconsistent** state!

**Q:** What is a “**acceptable**” schedule?
Equivalence of Schedules:

Two schedules are equivalent iff they lead the database to the same end state irrespective of its start state.

Serializable Schedule:

A schedule that is equivalent to some complete serial schedule (for now, assume only Committed transactions; Aborts will be discussed later)

Only a serializable schedule is an “acceptable” schedule!
Transaction Schedules

Q: Is this schedule serializable?

Yes! It is equivalent to the serial schedule T1 → T2 (why?)
Transaction Schedules

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>R_{T1}(A)</td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td>W_{T1}(A)</td>
<td>R_{T2}(A)</td>
</tr>
<tr>
<td></td>
<td>R_{T1}(B)</td>
<td>W_{T2}(A)</td>
</tr>
<tr>
<td></td>
<td>W_{T1}(B)</td>
<td>W_{T2}(B)</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

Q: Is this schedule serializable?

Yes! It is also equivalent to the serial schedule T1 → T2 (why?)
Transaction Schedules

Q: What about this one from before?

No, it is not equivalent to any possible serial schedule!
T1 → T2 or T2 → T1 (why?)
The update of A by T2 is lost!

This is called a “conflict”
Transaction Management

- Motivation and Basics
  - The ACID Properties
- Transaction Schedules
  - Conflicts and Other Issues
- Concurrency Control
  - SQL/RDBMS Features
- Basics of Recovery
Kinds of Transaction Conflicts

❖ **WW** Conflict (Overwriting Uncommitted Data)

❖ **WR** Conflict (Reading Uncommitted Data aka “Dirty” Read)

❖ **RW** Conflict (Unrepeatable Reads)
## WW Conflict

(Overwriting Uncommitted Data)

T1 overwrites T2’s update without reading new A

This schedule is not serializable!

If a transaction writes an item without reading it, it is called a “**Blind Write**”

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$R_{T2}(A)$</td>
</tr>
<tr>
<td>$W_{T1}(A)$</td>
<td></td>
<td>$W_{T2}(A)$</td>
</tr>
<tr>
<td>$R_{T1}(B)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_{T1}(B)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>$R_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$W_{T2}(B)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commit</td>
</tr>
</tbody>
</table>
**WR Conflict**

(Reading Uncommitted Data / Dirty Read)

T1’s writes of A and B may be related; T2 may read inconsistent database!

This schedule is not serializable!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>$R_{T_1}(A)$</td>
<td>$R_{T_2}(A)$</td>
</tr>
<tr>
<td>$W_{T_1}(A)$</td>
<td>$W_{T_2}(A)$</td>
</tr>
<tr>
<td>$R_{T_2}(B)$</td>
<td>$R_{T_2}(B)$</td>
</tr>
<tr>
<td>$W_{T_2}(B)$</td>
<td>$W_{T_2}(B)$</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
<tr>
<td>$R_{T_1}(B)$</td>
<td></td>
</tr>
<tr>
<td>$W_{T_1}(B)$</td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
</tr>
</tbody>
</table>
**RW(R) Conflict**

(Unrepeatable Reads)

The two reads of A by T1 may yield different values

This schedule is not serializable!

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>$R_{T1}(A)$</td>
<td></td>
<td>$R_{T2}(A)$</td>
</tr>
<tr>
<td>$W_{T1}(C)$</td>
<td></td>
<td>$W_{T2}(A)$</td>
</tr>
<tr>
<td>$R_{T2}(B)$</td>
<td></td>
<td>$R_{T2}(B)$</td>
</tr>
<tr>
<td>$W_{T2}(B)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>$R_{T1}(A)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_{T1}(D)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Q:** Is this a serializable schedule?

**Q:** Does it have any conflicts?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td><strong>R_{T1}(A)</strong></td>
<td><strong>W_{T1}(A)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>R_{T2}(A)</strong></td>
<td><strong>R_{T1}(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>W_{T2}(A)</strong></td>
<td><strong>W_{T1}(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>W_{T2}(A)</strong></td>
<td><strong>R_{T2}(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>W_{T2}(B)</strong></td>
<td><strong>W_{T2}(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Commit</strong></td>
<td><strong>Commit</strong></td>
</tr>
</tbody>
</table>
Okay, what about Aborted transactions?
Transactions with Aborts

❖ Serializability: only worry about Commit transactions and pretend as if Aborted transactions did not even happen!
❖ To make the above “illusion” possible, 2 new issues to deal with if Aborted transactions present:
❖ How to “undo” the effects of an Aborted transaction?
  All changes made by it should be undone
  Use Logging and Recovery (Later)
❖ What if some other transactions got “affected” by it?
  Must undo all affected transactions as well!
Cascading Aborts

Abort of T1 leads to Abort of T2!

T2’s Commit will be replaced with an Abort by the Txn Manager!

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>( R_{T1}(A) )</td>
<td>( R_{T2}(A) )</td>
</tr>
<tr>
<td>( W_{T1}(A) )</td>
<td>( W_{T2}(A) )</td>
</tr>
<tr>
<td>Abort</td>
<td>Abort</td>
</tr>
</tbody>
</table>
Cascading Aborts

**Q:** What about this case?

T2 has already Committed!
Not allowed to Abort it now (why?)

Known as an **"Unrecoverable"** schedule

- **Recoverable** schedule:
  All transactions Commit (if at all)
  only after all others that supply dirty data Commit/Abort

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>$R_{T1}(A)$</td>
<td></td>
</tr>
<tr>
<td>$W_{T1}(A)$</td>
<td>$R_{T2}(A)$</td>
</tr>
<tr>
<td></td>
<td>$W_{T2}(A)$</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td></td>
<td>Abort</td>
</tr>
</tbody>
</table>
Q: Is it possible to avoid cascading aborts? Yes!

Avoid-Cascading-Aborts (ACA) schedule:

No txn is allowed to read dirty data, i.e., all txns read changes of Committed txns only

Guarantees that cascading aborts will not arise

Also guaranteed to be a Recoverable schedule

A similar issue arises if a transaction overwrites dirty data!

Handled using Concurrency Control
Transaction Management

- Motivation and Basics
  - The ACID Properties
- Transaction Schedules
  - Conflicts and Other Issues
- Concurrency Control
- SQL/RDBMS Features
- Introduction to Recovery
Introduction to Locking

❖ To ensure serializability and recoverability, DBMSs use concurrency control (CC); most common way: “locking”

❖ Lock: small bookkeeping object associated with a “data item” (tuple, attribute, table, etc.); managed by Lock Manager

❖ Simplified view:

Lock is a 3-tuple: (TxnID, RecID, Mode)

<table>
<thead>
<tr>
<th></th>
<th>--</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>S</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

Shared (S) or Exclusive (X)

**Q:** *When are locks acquired?*
During the request to BufMgr!

**Q:** *What if a txn cannot get a lock?*
Suspended and put on a “wait queue”
Locking Protocols

- Determines when locks should be acquired/released by txns
- **Strict 2-Phase Locking (Strict 2PL):**
  1. Request S (resp. X) lock before R (resp. W) on item
  2. Release all locks only when txn finishes
- **(Non-strict) 2PL:**
  Relax 2: if any lock released, cannot acquire new locks!

*Both protocols guarantee **serializability**!*  
*Strict 2PL also guarantees **ACA schedules** and thus, **recoverability**!*  

Note: Lock acquire/release done automatically by DBMS
### Locking Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>Begin</td>
</tr>
<tr>
<td>(R_T(A))</td>
<td>(R_T(A))</td>
</tr>
<tr>
<td>(W_T(A))</td>
<td>(W_T(A))</td>
</tr>
<tr>
<td>(R_T(B))</td>
<td>(W_T(B))</td>
</tr>
<tr>
<td>Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

#### Questions:

**Q:** Is this schedule serializable?  
**Q:** Is this schedule recoverable?  

Suppose we use **Strict 2PL**

DBMS makes \(T_1\) acquire X lock on A  

\(T_2\) forced to wait before \(R(A)\)!

**Q:** When will \(T_1\) release X lock on A?  

**Q:** Is this new schedule serializable?  

**Q:** What if we use (non-strict) 2PL? \(T_2\) resumed
Deadlocks

- Deadlocks arise when two txns wait on each other
  System made to wait and do nothing forever!
  \[ X_{T1}(B), X_{T2}(A), S_{T1}(A), S_{T2}(B), \text{Unlocks…} \]
  **Q:** Does this satisfy 2PL? Strict 2PL?

- Deadlock prevention
  Avoid schedules that could cause deadlocks

- Deadlock detection and breaking
  A naive “solution”: timeout and abort txns
  More sophisticated solutions exist (not in syllabus)
Granularity of Locking

❖ What is the precise “data item” a transaction needs to lock? Tuple? Attribute? Whole table? Index?

❖ Affects both efficiency (throughput) and correctness

UPDATE Students SET Grade = “A” WHERE StudentID=123;

*Locking whole table might be an overkill!*

❖ **Thrashing**: too many txns asking X lock on same data item; causes throughput (number txns done/sec) to plummet

❖ Some ways to avoid thrashing: lock smallest granularity “needed” for txn, and reduce the time a txn holds a lock
Granularity of Locking

❖ What is the precise “data item” a transaction needs to lock? Tuple? Attribute? Whole table? Index?

❖ Affects both efficiency (throughput) and correctness

❖ DBMSs allow for “predicate locks” too. Bizarre correctness issues can arise on interleaving granularity-mode combos!

T1: S lock on table, X lock on predicate; read table, update only tuples matching predicate

T2 comes in to insert new tuple, but it satisfies predicate

T1 reads again: sees hitherto unseen tuple!

“Phantom Problem”
Transaction Management

❖ Motivation and Basics
  The ACID Properties
❖ Transaction Schedules
  Conflicts and Other Issues
❖ Concurrency Control
❖ SQL/RDBMS Features
❖ Introduction to Recovery
SQL/RDBMS Features

❖ **Keywords**: Begin is implicit; COMMIT; ROLLBACK (Abort); to save intermediate work of long-running txns: SAVEPOINT

❖ **Access modes**: READ ONLY vs. READ WRITE

❖ **Isolation Levels**: determines precise locking protocol

  - **READ UNCOMMITTED** (Long X locks only; no S locks!)
    Vulnerable to inconsistency; WR and RWR might arise!
  - **READ COMMITTED** (Long X locks; short S locks)
    WR conflicts do not arise; RWR might
  - **REPEATABLE READ** (Long X and S locks on real objects)
    Neither WR nor RWR arise; but phantom problem might!
  - **SERIALIZABLE** (Long X and S locks on phantoms too)
    No conflicts, no phantom. Perfect! Default in most RDBMSs!
Transaction Management

❖ Motivation and Basics
  The ACID Properties
❖ Transaction Schedules
  Conflicts and Other Issues
❖ Concurrency Control
❖ SQL/RDBMS Features
❖ Introduction to Recovery
Introduction to Recovery

- Recovery helps persist Committed txns changes and undo effects of Aborted txns (ensures Durability and Atomicity)
- **Log**: A file in which any changes to DBMS are recorded
  Precise entry depends on kind of change
- **Write-Ahead Logging (WAL) Protocol**: Ensure change is written to Log first before actual data!
  Upon Commit, force all Log records to disk first
  During Recovery, Log tells what to undo and/or redo
- Two mechanisms crucial to enable WAL:
  
  - “Stealing” Frames
  - “Forcing” Pages
Introduction to Recovery

- **Stealing Frames:**
  - Allow stealing buffer frames from uncommitted txns
  - Helps improve throughout, but challenge for Atomicity

- **Forcing Pages**
  - Every page write is sent to disk immediately
  - Hurts latency, but nice for Durability

- **Ideal: Steal + No Force**
  - To steal frame, write “summary” to Log; helps undo
  - To avoid forcing, write “summary” to Log; helps redo
Transaction Management

❖ Motivation and Basics
   The ACID Properties
❖ Transaction Schedules
   Conflicts and Other Issues
❖ Concurrency Control
❖ SQL/RDBMS Features
❖ Introduction to Recovery