CS460: Intro to Database Systems

Class 23: Transactional Management Overview

Instructor: Manos Athanassoulis

https://midas.bu.edu/classes/CS460/
Administrativia – what lies ahead

PA2 – Row-store vs Column-store & Query Opt. (deadline 12/2)
   will upload today

Hands-on-SQL test (bonus) on 12/4 (last few minutes of class)
   more details on Piazza very soon

WA4 – on transaction management (deadline 12/8)

PA3 (last PA) – on Key-Value Stores (deadline 12/15)

Final: last day of class on 12/11
Transaction Management

Overview of ACID

Readings: Chapter 16.1

Concurrency control

Logging and recovery
Components of a DBMS

DBMS: a set of cooperating software modules
Problem Statement

Goal: concurrent execution of independent transactions
– utilization/throughput ("hide" waiting for I/Os)
– response time
– fairness

Example:

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0: tmp1 := read(X)</td>
<td>tmp2 := read(X)</td>
</tr>
<tr>
<td>t1:</td>
<td>tmp2 := tmp2 + 10</td>
</tr>
<tr>
<td>t2: tmp1 := tmp1 - 20</td>
<td></td>
</tr>
<tr>
<td>t3:</td>
<td></td>
</tr>
<tr>
<td>t4: write tmp1 into X</td>
<td></td>
</tr>
</tbody>
</table>

Arbitrary interleaving can lead to inconsistencies
Definitions

A program may carry out many operations on the data retrieved from the database

The DBMS is only concerned about what data is read/written from/to the database

**database**
a fixed set of named data objects \((A, B, C, \ldots)\)

**transaction**
a sequence of read and write operations \((\text{read}(A), \text{write}(B), \ldots)\)
Correctness: The **ACID** properties

**Atomicity**: All actions in the transaction happen, or none happen

**Consistency**: If each transaction is consistent, and the DB starts consistent, it ends up consistent

**Isolation**: Execution of one transaction is isolated from that of other transactions

**Durability**: If a transaction commits, its effects persist
Transaction Management

Overview of ACID

Concurrency control

Logging and recovery

Readings: Chapter 16.2-16.6
Transaction Consistency

Consistency - data in DBMS is accurate in modeling real world and follows integrity constraints

User must ensure that transaction is consistent

Key point:

consistent database S1 \xrightarrow{\text{transaction } T} \text{consistent database } S2
Recall: Integrity constraints
  – must be true for DB to be considered consistent
  – Examples:
    1. FOREIGN KEY R.sid REFERENCES S
    2. ACCT-BAL \geq 0

System checks integrity constraints and if they fail, the transaction rolls back (i.e., is aborted)
  – Beyond this, DBMS does not understand data semantics
  – e.g., how interest on a bank account is computed
Isolation of Transactions

Users submit transactions, and

Each xact executes *as if* it was running *by itself*

- Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.

Techniques for achieving isolation:

- Pessimistic – don’t let problems arise in the first place
- Optimistic – assume conflicts are rare, deal with them *after* they happen.
Example

Consider two transactions:

\[
\begin{align*}
T1: & \quad \text{BEGIN} \quad A = A+100, \quad B = B-100 \quad \text{END} \\
T2: & \quad \text{BEGIN} \quad A = 1.06 \times A, \quad B = 1.06 \times B \quad \text{END}
\end{align*}
\]

1\textsuperscript{st} xact transfers $100 from B’s account to A’s

2\textsuperscript{nd} xact credits both accounts with 6% interest

Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2?

\[
$2000 \times 1.06 = $2120
\]

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. But, the net effect must be equivalent to these two transactions running serially in some order.
Example (Cont.)

Legal outcomes: A=1166, B=954 or A=1160, B=960
Consider a possible interleaved schedule:

| T1:      | A=A+100,       | B=B-100 |
| T2:      | A=1.06*A,      | B=1.06*B |

This is OK (same as T1;T2). But what about:

| T1:      | A=A+100,       | B=B-100  |
| T2:      | A=1.06*A, B=1.06*B |

Result: A=1166, B=960; A+B = 2126, bank loses $6
The DBMS’s view of the second schedule:

| T1:      | R(A), W(A),     | R(B), W(B) |
| T2:      | R(A), W(A), R(B), W(B) |
Anomalies with Interleaved Execution

Reading Uncommitted Data (WR Conflicts, “dirty reads”):

| T1: R(A), W(A), R(B), W(B), Abort |
| T2: R(A), W(A), C |

Unrepeatable Reads (RW Conflicts):

| T1: R(A), R(A), W(A), C |
| T2: R(A), W(A), C |
Anomalies (Continued)

Overwriting Uncommitted Data (WW Conflicts):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td>W(A), W(B), C</td>
<td>W(B), C</td>
</tr>
<tr>
<td>T2:</td>
<td>W(A), W(B), C</td>
<td></td>
</tr>
</tbody>
</table>
Concurrency Control

How to avoid such anomalies? “lock” data

*Strict Two-phase Locking (Strict 2PL) Protocol*

- obtain an *S (shared)* lock on object before reading
- obtain an *X (exclusive)* lock on object before writing

(i) obtain locks automatically
(ii) if a xact holds an X lock on object no other xact can acquire S or X
(iii) if a xact holds an S lock, no other xact can acquire X (but only S)

2 phases: first acquire and then release all at the end
important: no lock is ever acquired after one has been released
Transaction Management

Overview of ACID

Concurrency control

Logging and recovery

Readings: Chapter 16.7
Atomicity of Transactions

Two possible outcomes of executing a transaction:

– Transaction might *commit* after completing all its actions
– or it could *abort* (or be aborted by the DBMS) after executing some actions

DBMS guarantees that transactions are *atomic*.

– From user’s point of view: transaction always either executes all its actions, or executes no actions at all
Mechanisms for Ensuring Atomicity

One approach: LOGGING
- DBMS logs all actions so that it can undo the actions of aborted transactions

Another approach: SHADOW PAGES
- (ask me after class if you’re curious)

Logging used by modern systems, because of the need for audit trail and for efficiency
Aborting a Transaction (i.e., Rollback)

If a xact \( T_i \) is aborted, all its actions must be undone.

If \( T_j \) reads object last written by \( T_i \), \( T_j \) must be aborted!

- Most systems avoid such *cascading aborts* by releasing locks only at end of the transaction (i.e., strict locking).
- If \( T_i \) writes an object, \( T_j \) can read it only after \( T_i \) finishes.

To *undo* actions of an aborted transaction, DBMS maintains *log* which records every write.

Log is also used to recover from system crashes:
- All active Xacts at time of crash are aborted when system comes back up.
The Log

Log consists of “records” that are written sequentially
  – Typically chained together by transaction id
  – Log is often archived on stable storage

Need for UNDO and/or REDO depend on Buffer Manager
  – **UNDO** required if: uncommitted data can overwrite stable version of committed data (STEAL buffer management)
  – **REDO** required if: transaction can commit before all its updates are on disk (NO FORCE buffer management)
The Log (cont.)

The following actions are recorded in the log:

– *if $T_i$ writes an object*, write a log record with:
  • If UNDO required need “before image"
  • IF REDO required need “after image”

– *$T_i$ commits/aborts*: a log record indicating this action
Logging (cont.)

Write-Ahead Logging protocol

- Log record must go to disk *before* the changed page!
- All log records for a transaction (including its commit record) must be written to disk before the transaction is considered “Committed”

All logging and CC-related activities are handled transparently by the DBMS
(Review) Goal: The **ACID** properties

**Atomicity:** All actions in the transaction happen, or none happen

**Consistency:** If each transaction is consistent, and the DB starts consistent, it ends up consistent

**Isolation:** Execution of one transaction is isolated from that of other transactions

**Durability:** If a transaction commits, its effects persist

What happens if system **crashes** between **commit** and **flushing modified data to disk**?
Durability - Recovering From a Crash

Three phases:

- **Analysis**: Scan the log (forward from the most recent *checkpoint*) to identify all transactions that were active at the time of the crash.
- **Redo**: Redo updates as needed to ensure that all logged updates are in fact carried out and written to disk.
- **Undo**: Undo writes of all transactions that were active at the crash, working backwards in the log.

At the end – all committed updates and only those updates are reflected in the database.

Some care must be taken to handle the case of a crash occurring during the recovery process!
Summary

Concurrency control and recovery are among the most important functions provided by a DBMS

Concurrency control is automatic

- System automatically inserts lock/unlock requests and schedules actions of different Xacts
- Property ensured: resulting execution is equivalent to executing the Xacts one after the other in some order

Write-ahead logging (WAL) and the recovery protocol are used to:

1. undo the actions of aborted transactions, and
2. restore the system to a consistent state after a crash