CS460: Intro to Database Systems

Class 20: Transactional Management Overview

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS460/
Administrativia – what lies ahead

WA6 – on normalization (deadline 11/24)
    uploaded a few days ago

PA2 – Row-store vs Column-store & Query Opt. (deadline 11/28)
    *uploaded a week ago*

WA7 (last WA) – on transaction management (deadline 12/6)
    coming on 11/24

PA3 (last PA) – on Key-Value Stores (deadline 12/8)
    coming on 11/26

**Final:** last week of semester, on Friday 12/11
Transaction Management

Overview of ACID

Readings: Chapter 16.1

Concurrency control

Logging and recovery
Components of a DBMS

- Query Compiler
- Transaction Manager
- Schema Manager
- Execution Engine
- Logging/Recovery
- Concurrency Control
- Buffer Manager
- Storage Manager
- LOCK TABLE
- BUFFERS
- BUFFER POOL

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:

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 \((read(A), write(B), \ldots)\)
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 \[\xrightarrow{\text{transaction } T}\] 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 >= 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:

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

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:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=A+100,</td>
<td>A=1.06*A,</td>
</tr>
<tr>
<td>B=B-100</td>
<td>B=1.06*B</td>
</tr>
</tbody>
</table>

Result: A=1166, B=960; A+B = 2126, bank loses $6

The DBMS’s view of the second schedule:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A), W(A),</td>
<td>R(A), W(A),</td>
</tr>
<tr>
<td>R(B), W(B)</td>
<td>R(B), W(B)</td>
</tr>
</tbody>
</table>

Remember: correct outcome: A+B=$2120
I Anomalies with Interleaved Execution

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

<table>
<thead>
<tr>
<th>T1: R(A), W(A),</th>
<th>R(B), W(B), Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), C</td>
<td></td>
</tr>
</tbody>
</table>

Unrepeatable Reads (RW Conflicts):

<table>
<thead>
<tr>
<th>T1: R(A),</th>
<th>R(A), W(A), C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T2: R(A), W(A), C</td>
<td></td>
</tr>
</tbody>
</table>
Anomalies (Continued)

Overwriting Uncommitted Data (WW Conflicts):

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

A gets its value from T2
B gets its values from T1

A correct execution would take both values from T2 or both from T1
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
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

why? to ensure atomicity!
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 depends on Buffer Manager
– **UNDO required if:** uncommitted data can overwrite 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 \textit{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

**A** atomicity: All actions in the transaction happen, or none happen

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

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

**D** 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!
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

next: concurrency control in detail!