Transaction Processing Overview (continued)

Transaction processing studied in depth in
CS W4112-Database System Implementation
Aborting a Transaction

Consider a transaction T1:

\[ T1: R1(A) W1(A) W1(B) R1(C) \text{Abort1} \]

What should we do to remove all effects of T1 from the database as we terminate the transaction?

\[ T1: R1(A) W1(A) W1(B) R1(C) W1(B) W1(A) \text{Abort1} \]

- As part of the abort operation, we write back/restore the old values of B and A that T1 received, which T1 can do because it is still holding exclusive locks on B and A (strict 2PL…)
- The old values are in the database log, in which the old and new values of every write to the database are recorded
- Finally T1 releases all locks that T1 acquired

Cascading Aborts

- If a transaction Ti is aborted, all its actions have to be undone
- Furthermore, if Tj read an object last written by Ti, Tj must be aborted as well
- Strict 2PL avoids such cascading aborts because transactions only release their locks at the very end (i.e., at commit or abort time)
  So if Ti writes an object, it holds on to its exclusive lock on the object until the end; Tj cannot read the object until after Ti commits or aborts
Unrecoverable Schedules

Consider this schedule for transactions T1 and T2, with the “reading uncommitted data” problem we saw:

T1: R1(A) \textbf{W1(A)} \textbf{Abort2}
T2: \textbf{R2(A)} R2(B) W2(B) Commit2

Is the schedule serializable? Yes

Is there a problem? Yes! T2 reads a value of A written by T1, T2 commits (so because of durability, T2 persists forever), and then T1 aborts (so because of atomicity, all of T1’s actions should be erased)

---

Recoverable Schedules

A schedule is recoverable if its transactions commit only after—and if—all transactions whose changes they read have committed

Strict 2PL does not allow unrecoverable schedules, so the schedule in the previous slide is not possible under Strict 2PL

Only serializable, recoverable schedules that avoid cascading aborts are possible under Strict 2PL
The Phantom Problem

Consider the following schedule for T1 and T2, following Strict 2PL:

**T1**: locks all pages with sailors with
rating=1 and finds oldest such sailor is **71 years old**

**T2**: inserts a new sailor with rating=1 and age=96

and deletes oldest sailor with rating=2, who happens to be **80 years old**

**Problem**: Yes! Schedule is not equivalent to serial schedule T1; T2 (the oldest sailors for rating=1 and rating=2 that T1 observes would have been 71 and 80 years old, respectively) or to serial schedule T2; T1 (the oldest sailors for rating=1 and rating=2 that T1 observes would have been 96 and 63 years old, respectively)

Dynamic Databases and Phantoms

- This schedule is possible under Strict 2PL yet it’s not serializable!
- Problem: implicit assumption that transaction T1 has locked all sailors, current or future, with rating=1, not just existing sailors
- One way to handle such dynamic databases with Strict 2PL to guarantee true serializability is to use **predicate locking**: lock all current or future tuples satisfying a predicate (e.g., rating=1 OR rating=2 for T1)
### Transaction Isolation Levels in SQL

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>

- Transactions have an **access mode**: READ WRITE vs. READ ONLY
- “Read Uncommitted” transactions must be READ ONLY and don’t use locks (if concurrency control is based on Strict 2PL)
- For all other isolation levels, exclusive locks are held until the end of the transactions
- For “Read Committed” transactions, shared locks are released immediately after the corresponding reads
- “Repeatable Reads” and “Serializable” transactions follow Strict 2PL
- “Serializable” transactions use predicate locking

See [https://www.postgresql.org/docs/9.3/static/transaction-iso.html](https://www.postgresql.org/docs/9.3/static/transaction-iso.html) for PostgreSQL support

### ACID Properties of Transactions

- Atomicity
- Consistency
- Isolation
- Durability

- Atomicity and durability are handled by the recovery manager
- We have already discussed how to terminate a transaction during normal database operation
- Now: what should we do when the system crashes, to guarantee atomicity and durability?
Recovery After a Crash

Assumptions:
• Disk is safe, RAM is not (and goes away when system crashes)
• DBMS uses Strict 2 Phase locking

The Database Log

• The log is a single, sequential, append-only file that keeps information on actions by all transactions
• Log record for a write operation:

<table>
<thead>
<tr>
<th>TID</th>
<th>pageID</th>
<th>offset</th>
<th>length</th>
<th>old value</th>
<th>new value</th>
</tr>
</thead>
</table>

TID=transaction ID; pageID, offset, and length refer to the block on disk, the position within the block, and the length that is written
• When a transaction writes an object, the log record includes both the old and the new values for the object
• When a transaction commits or aborts, a log record documents this action
• The most recent records of the log are in main memory, while the rest of the log is stored stably on disk (and often replicated as well)
Write-Ahead Logging, or WAL

For efficiency, we allow writes of data and log to happen in memory without an immediate disk-write counterpart: dangerous if not done properly!

Write-Ahead Logging rules:
• We must force the log record for an update to disk before the corresponding data page for the update makes it to disk
• We must write all log records for a transaction—including the commit record—to disk before declaring the transaction committed

Note: The log is a sequential file, so if we push a log record to disk, all earlier log records are also pushed to disk

Recovering from a Crash

During recovery:
• **Redo updates by committed transactions** T1, T2, and T3 (their updates might not have made it to disk by the time of the crash); use log for this
• **Undo updates by in-progress transactions** T4 and T5 (their updates might have made it to disk); use log for this; these transactions get terminated and restarted from scratch
### ARIES Recovery Algorithm: 3 Phases

- **Analysis**: Scan the log to identify all transactions that had committed or aborted, or were in progress at crash time.
- **Redo**: Redo all updates—by using the new values in the corresponding log records—to ensure that all logged updates are indeed carried out, in chronological order.
- **Undo**: Undo all updates—by using the old values in the corresponding log records—of in-progress transactions, in reverse chronological order.

### Summary of Transaction Processing

- ACID properties of transactions are critically important for sensitive applications.
- Transaction processing—with the enforcement of the ACID properties—is done largely transparently by DBMS.
- This is a big advantage of using DBMSs over ad-hoc software.
- CS W4112-Database System Implementation covers transaction processing in depth.
COMS E6111
Advanced Database Systems

Offered again in 2019-2020 academic year

Prerequisites:
COMS W4111 (not W4112); fluency in Java or Python

Sample of Topics Covered

• Information Retrieval
• Web Search
• Data Mining
• Data Warehousing, OLAP, Decision Support
• Spatial Data Management
• Time Series Analysis
• Information Extraction
• …
General Structure of Course (subject to change)

- Regular lectures, but with more discussion
- No homework
- Many readings through research papers
- 3 projects (in Java or Python)
- Midterm and final
- Undergraduate students can register