Transaction Management

Yanlei Diao
UMass Amherst

Slides Courtesy of R. Ramakrishnan and J. Gehrke

DBMS Architecture

Motivating Examples

- Banking
- Ticketing
  - Have you tried to get popular concert tickets online?
- E-commerce
  - Have you experienced online shopping during sales?
- Voting
- Telecommunications
- Online gaming...

Concurrent User Queries

- Concurrent execution of queries for improved performance.
  - E.g., when Query 1 is doing I/O, Query 2 can utilize CPU.
  - Improve average response time (average delay that a user query experiences).
  - Improve system throughput (number of user queries processed in each time unit).

Transactions

- User programs (e.g., your php code) include queries but may do many things on data retrieved.
  - E.g., add a user rating, update avg rating.
  - E.g., search for a ticket, think about it..., and buy it.
  - E.g., operations on Bob’s bank account.
- DBMS is only concerned about what data is read from/ written to the database backend.
- A transaction is DBMS’ abstract view of a user program: a sequence of reads and writes of database objects
  - They are translated from a series of queries.
  - We use A, B, C... to denote objects, e.g. tables, tuples

Concurrency

- Many users submit xacts, but each user thinks of his as executing by itself.
  - DBMS interleaves reads and writes of xacts for concurrency.
- Consistency: each xact starts and ends with a consistent state (i.e., satisfying all integrity constraints).
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.
- Isolation: execution of one xact appears isolated from others.
  - Nobody else can see the data in its intermediate state, e.g., account A being debited but B not being credited.
Recovery

- A transaction might commit after completing all its actions, or it could be aborted after executing some actions.
- Atomicity: either all actions of a xact are performed or none of them is (all-or-none).
  - DBMS logs all actions so that it can undo the actions of aborted xacts.
- Durability: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS logs all actions (a form of redundant storage) so that it can redo the actions of committed xacts.

James Gray & Turing Award

- Jim Gray won Turing Award in 1998 for
  "for seminal contributions to database and transaction processing research and technical leadership in system implementation"

Outline

- Transaction management overview
  - Serializability & recoverability
- Lock-based concurrency control
- Recovery

Example

- Consider two transactions:
  T1: BEGIN A=A+100, B=B-100 END
  T2: BEGIN A=1.06*A, B=1.06*B END
  - No guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
  - However, the net effect must be equivalent to these two transactions running serially in some order!

Example (Contd.)

- Consider a possible interleaving schedule:
  T1: A=A+100, B=B-100
  T2: A=1.06*A, B=1.06*B
  - But what about:
    T1: A=A+100, B=B-100
    T2: A=1.06*A, B=1.06*B
    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)

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions.
- Equivalent schedules: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- Serializable schedule: A schedule that is equivalent to some serial execution of the transactions.
  - If each transaction preserves consistency, every serializable schedule preserves consistency.
Serializability

- Serializability theory concerns the schedules of transactions that are not (explicitly) aborted.
- Given a set of such xacts submitted on the fly, ideally want to allow any serializable schedule.
  - Recognizing any serializable schedule is highly complex, if possible.
  - Instead, allow only a subset of serializable schedules that are easy to detect.

Conflict Serializability

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.

Dependency Graph

- Precedence graph:
  - One node per Xact;
  - Edge from Xact Ti to Xact Tj if an action of Ti precedes and conflicts with one of Tj’s actions (RW, WR, WW operations on the same object).
- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic.

Example

- Schedule is not conflict serializable:
  - The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
- Solution: abort one xact to break the cycle, and restart it!

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

Recoverability

- Recoverability theory concerns schedules that involve aborted transactions.

T1: R(A), W(A) Abort
T2: R(A), W(A) Commit

Unrecoverable!
- A schedule S is recoverable if each xact commits only after all xacts from which it read have committed.

Recoverability (Contd.)

- S avoids cascading rollback if each xact may read only those values written by committed xacts.

T1: R(A), W(A) Abort
T2: R(A), W(A) Abort

Recoverable, but with cascading aborts.
Recoverability (Contd.)

- S is strict if each xact may read and write only objects previously written by committed xacts.
  - No cascading aborts.
  - Actions of aborted xacts can be simply undone by restoring the original values of modified objects.

An Exercise:

- Regarding only committed xacts:
  - Serializable?
  - Conflict serializable?

- Regarding both committed & aborted xacts:
  - Recoverable?
  - Avoids cascading aborts?
  - Strict?

Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency controls
- Recovery

(1) Locking Protocol: Strict 2PL

- Strict Two-Phase Locking (Strict 2PL) Protocol:
  1. Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
  2. If an Xact holds an X lock on an object, other Xacts cannot get a lock (S or X) on that object and become blocked.
  3. All locks held by a xact are released when it completes. Other blocked xacts can resume now.

Strict 2PL (contd.)

- Theorem: Strict 2PL allows only schedules whose precedence graph is acyclic.
  - Strict 2PL only allows conflict serializable schedules!

- Strict 2PL is strict with respect to recoverability.
  - A schedule is strict if each xact may read and write only objects previously written by committed xacts.
  - Strict 2PL is recoverable without anomalies related to aborted transactions.
  - Hence, it simplifies transaction aborts.
Nonstrict Two-Phase Locking Protocol

1. Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
2. If an Xact holds an X lock on an object, other Xacts cannot get a lock (S or X) on that object and become blocked.
3. A xact cannot request additional locks once it releases any locks.
   - It releases locks earlier, so blocked xacts can resume earlier.

Nonstrict 2PL (contd.)

- **Thm:** Nonstrict 2PL ensures acyclicity of precedence graph.
  - Nonstrict 2PL allows only conflict serializable schedules.
  - An equivalent serial schedule is given by the order of xacts entering their shrinking phase.
- Nonstrict 2PL is recoverable but **not strict**!
  - A schedule S is recoverable if each xact commits only after all xacts from which it read have committed.
  - Nonstrict 2PL involves complex abort processing, but allows xacts to go through more quickly.

Deadlocks

- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.

- **Two ways of dealing with deadlocks:**
  - Deadlock detection
  - Deadlock prevention

Deadlock Detection

- **Create a waits-for graph**:
  - Nodes are Xacts.
  - There is an edge from Xact Ti to Xact Tj if Ti is waiting for Tj to release a lock.
  - Cycles indicate deadlocks!

- Periodically check for cycles in the waits-for graph.
  - Resolve a deadlock by aborting a transaction on the cycle and releasing all its locks.

Deadlock Detection (Contd.)

| T1:  | S(A), R(A), S(B) |
| T2:  | X(B), W(B)      |
| T3:  | S(C), R(C)      |
| T4:  | X(B)            |

Now T1, T2, T3 are all blocked, waiting infinitely... Abort one!

Waits-for vs. Precedence Graphs

- **Precedence graph** is a theoretical tool to detect conflict serializable schedules.

- **Real implementation:** (strict) 2PL + deadlock handling
  - 2PL prevents conflicting actions from being materialized in the DB by blocking xacts.
  - Waits-for graph fixes infinite blocking when xact’s are waiting for each other to release locks.
**Deadlock Prevention**

- Assign priorities based on timestamps.
  - The older the timestamp, the higher the xact’s priority.
- **Wait-Die**: Ti wants a lock that Tj holds. If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts.
  - Lower priority xacts can never wait.
- **Wound-wait**: Ti wants a lock that Tj holds. If Ti has higher priority, Tj aborts; otherwise Ti waits.
  - Higher priority xacts never wait.
- If a transaction re-starts, make sure it has its original timestamp so its priority increases.

**Outline**

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Recovery

**Recovery in DBMS: Motivation**

- **Atomicity: all-or-none**
  - Transactions may abort (“rollback”).
- **Durability:**
  - Effects of committed xacts should survive crashes.
- Desired behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted (effects not seen).

**Assumptions**

- Concurrency control is in effect.
  - **Strict 2PL**, in particular.
- Updates are happening “in place”.
  - i.e. updates of an object are written from memory to the only copy of it on disk.
- A simple scheme to guarantee atomicity & durability?
Handling the Buffer Pool

- Force every write to disk at commit time?
  - Provides durability.
  - Poor response time. Why?
    - No force, how can we ensure durability?
- Steal buffer-pool frames from uncommitted Xacts?
  - If not, poor throughput. Why?
  - If steal, how can we ensure atomicity?

Logging (a form of redundant storage)

- Log: A history of actions executed by DBMS
  - Records for REDO/UNDO information for every update
    - <XID, pageID, offset, length, old data, new data>
  - All commits and aborts
  - And additional control info (which we’ll see soon).

- Writing log records to disk is more efficient than data pages
  - Sequential writes to log (put it on a separate disk).
  - Minimal info written to log, often smaller than a data record; multiple updates fit in a single log page.

Protocol: Write-Ahead Logging

- Write-Ahead Logging (WAL) Protocol:
  1. Must force the log record for an update before the corresponding data page gets to disk (when steal).
     - Guarantees atomicity
  2. Must write all log records for a Xact before commit.
     - Guarantees durability

- Exactly how is logging (and recovery) done?
  - We’ll study the ARIES algorithm.

ARIES

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN written to disk so far.

ARIES

- Each log record has a unique Log Sequence Number (LSN).
  - LSNs always increasing.
- Each data page contains a pageLSN.
  - The LSN of the most recent log record for an update to that page.
- System keeps track of flushedLSN.
  - The max LSN written to disk so far.

Big Picture: What’s Stored Where

- Data pages each w. a pageLSN
- LogRecords LSN, prevLSN XID type page length offset before-image after-image
- Xact Table (TT) lastLSN
- Dirty Page Table (DPT)
  - flushedLSN
  - flushed
  - In RAM
1. Transaction Commit

- When an Xact is running,
  - Generate a log record with LSN for each update operation.
  - Update Xact table and Dirty Page table in memory.
- Upon commit,
  - Write commit record to log.
  - WAL (2): All log records up to Xact’s lastLSN are flushed.
    - Guarantees that flushedLSN ≥ lastLSN.
    - Log writes are sequential; many log records per log page.
  - When commit returns, write end record to log.

2. Simple Transaction Abort

- For now, consider an explicit abort of a Xact.
  - No crash involved.
- “Play back” the log in reverse order, UNDOing updates.
  - Get lastLSN of Xact from Xact table.
  - Follow chain of log records backward via prevLSN field.
    - To perform UNDO, also need to have a lock on data!
  - Logging continues in UNDOs!

1. Before starting UNDO, write an Abort log record.
  - For recovering from crash during UNDO!

Abort (Contd.)

2. Before restoring old value of a page, write a Compensation Log Record (CLR):
  - CLR has one extra field, undonextLSN, pointing to the next LSN to undo (i.e. the prevLSN of the record we’re undoing now).
3. At end of UNDO, write an end log record.

Example of Rollback

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>update: T1 writes P2</td>
</tr>
<tr>
<td>40</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>45</td>
<td>CLR: Undo T1 LSN 30</td>
</tr>
<tr>
<td>50</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>60</td>
<td>T1 ends</td>
</tr>
</tbody>
</table>

3. Crash Recovery: Big Picture

- Analysis figures which Xacts had committed or failed at crash time.
  - Checkpoint: a snapshot of xact table and dirty page table.
  - Start from most recent checkpoint (via master record).
- REDO all actions; repeat history
  - Start from smallest recLSN in Dirty Page Table.
- UNDO effects of failed Xacts.
  - Back to oldest LSN of a running xact at crash.

(1) Recovery: The Analysis Phase

- First, reconstruct state at checkpoint.
  - Get begin_checkpoint record via the master record.
  - Find its end_checkpoint record, read the xact table and dirty page table (DPT).
- Reconstruct Xact Table and Dirty Page Table at crash; scan log forward (from last checkpoint).
  - Xact Table (TT):
    - End record: Remove xact from Xact Table.
    - Other records: Add xact to Xact Table (if not there), set its lastLSN to this LSN, change xact status upon commit/abort.
  - Dirty Page Table (DPT):
    - Update record: If page P not in Dirty Page Table, add P to D.P.T., set its recLSN (earliest update) to this LSN.
(2) Recovery: The REDO Phase

- Repeat history to reconstruct DB state at crash:
  - Reapply all updates, even those of aborted Xacts and redo CLRs.

  a. Scan forward from the log record containing smallest recLSN in D.P.T.
  b. For each update or CLR log record, REDO the action.

- Optimizations are in textbook, but not required in this class.
- No additional logging!

(3) Recovery: The UNDO Phase

- Take a set of loser xacts, undo all in reverse order of LSN!

ToUndo = \{ l | l: lastLSN of a "loser" xact\}

Repeat:

a. Choose largest LSN among ToUndo.

b. If this LSN is a CLR and undonextLSN == NULL
   - Write an End record for this Xact.

b. If this LSN is a CLR, and undonextLSN != NULL
   - Add undonextLSN to ToUndo

- Else this LSN is an update. Undo the update, write a CLR, add prevLSN to ToUndo.

Until ToUndo is empty.

Example of Recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>45</td>
<td>T1 ends</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
</tr>
</tbody>
</table>

RAM

Xact Table (TT) lastLSN, status
Dirty Page Table (DPT) recLSN (earliest)
flushedLSN

prevLSNs

Example of Recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>40</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>45</td>
<td>T1 ends</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
</tr>
</tbody>
</table>

RAM

Xact Table (TT) lastLSN, status
Dirty Page Table (DPT) recLSN (earliest)
flushedLSN

prevLSNs

Crash During Restart!

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00,05</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T3 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>40,45</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
</tr>
</tbody>
</table>

RAM

Xact Table (TT) lastLSN, status
Dirty Page Table (DPT) recLSN (earliest)
flushedLSN

undonextLSN

Example of Recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>40,45</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
</tr>
</tbody>
</table>

RAM

Xact Table (TT) lastLSN, status
Dirty Page Table (DPT) recLSN (earliest)
flushedLSN

undonextLSN

Example of Recovery

<table>
<thead>
<tr>
<th>LSN</th>
<th>LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>begin_checkpoint</td>
</tr>
<tr>
<td>05</td>
<td>end_checkpoint</td>
</tr>
<tr>
<td>10</td>
<td>update: T1 writes P5</td>
</tr>
<tr>
<td>20</td>
<td>update: T2 writes P3</td>
</tr>
<tr>
<td>30</td>
<td>T1 aborts</td>
</tr>
<tr>
<td>40,45</td>
<td>CLR: Undo T1 LSN 10</td>
</tr>
<tr>
<td>50</td>
<td>update: T3 writes P1</td>
</tr>
<tr>
<td>60</td>
<td>update: T2 writes P5</td>
</tr>
<tr>
<td>CRASH, RESTART</td>
<td></td>
</tr>
</tbody>
</table>

RAM

Xact Table (TT) lastLSN, status
Dirty Page Table (DPT) recLSN (earliest)
flushedLSN

undonextLSN
Main Principles behind ARIES

- **Write-ahead logging (WAL)**
  - Any change to an object is recorded in the log. The log is written to disk before the change to the object is written, or upon commit.

- **Repeating history during REDO**
  - On restart after a crash, repeat all actions before the crash, brings the system back to the exact state that it was in before the crash.
  - Then undo the xacts still active at crash time.

- **Logging changes during UNDO**
  - Changes made to DB while undoing are logged to ensure such an action isn’t repeated in the event of repeated restarts.

- **Fuzzy checkpointing to expedite recovery**
  - An efficient way to create a snapshot of Xact Table & D.P.T.