Transaction Management: Concurrency Control

Yanlei Diao
UMass Amherst
April 24 and 29, 2008
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Deadlock management
- Efficient B+tree locking
DBMS Architecture

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor
- Lock Manager
- Access Methods
- Buffer Manager
- Log Manager
- Disk Space Manager

Concurrency Control

Recovery
Motivating Example

- Your movie database is a great success with 1 million users!
  - Everyday 7pm-8pm is the prime time of database access.
  - 20% users rate movies just seen.
  - 80% users search for top-rated movies in their favorite genres (of their favorite directors, etc.) to rent.

- As the DBA, you face many questions:
  - What is the workload of the database backend?
  - Execute queries in serial or in parallel?
  - Any anomalies may occur?
Concurrent User Programs

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses (I/O) are frequent and relatively slow, so when task 1 is doing I/O, we want to run task 2 to utilize the CPU and do useful work.
  - Good for *average response time* (average delay that a user task experiences)
  - Good for *system throughput* (#. of user tasks processed in each time unit)
Transactions

- A user’s program may do many things on the data retrieved.
  - E.g., retrieving synopsis of movies; writing reviews
  - E.g., operations on Bob’s bank account.
- But the DBMS is only concerned about what data is read from/written to the database.
- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.
  - E.g. transfer of money from account A to account B, including debiting A and crediting B.
ACID Properties of Transactions

- **Concurrency**: many users submit transactions, but each user thinks of his as executing by itself.
  - DMBS achieves concurrency by *interleaving* reads and writes of various transactions.

- **Consistency**: each transaction starts and ends with a consistent state; i.e. it can’t break ICs.
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.

- **Isolation**: execution of one transaction 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.
ACID Properties (Contd.)

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.

- **Atomicity**: either all actions of a transaction are performed or none of them is (all-or-none).
  - DBMS logs all actions so that it can undo the actions of aborted transactions.

- **Durability**: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS logs all actions so that it can redo the actions of committed transactions.
Example

- Consider two transactions \((Xacts)\):

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

  - The 1st xact is transferring $100 from B’s account to A’s.
  - The 2nd xact is crediting both accounts with a 6% interest payment.
  - There is 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 \textit{serially} in some order!
Example (Contd.)

- Consider a possible interleaving schedule:

  - T1: \( A = A + 100, \) \( B = B - 100 \)
  - T2: \( A = 1.06^*A, \) \( B = 1.06^*B \)

- This is OK. 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 (on the set of objects in the DB) 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.
Anomalies with Interleaved Execution

- **Reading Uncommitted Data** ("dirty reads"): 

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

- **Unrepeatable Reads**: 

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

- Overwriting Uncommitted Data:

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

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Deadlock management
- Efficient B+tree locking
Serializability

- Serializability theory concerns the schedules of transactions that are not (explicitly) aborted.
- Given a set of such xacts, 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.

- Given a set of xacts, conflict serializable schedules are a *subset* of serializable schedules.
  - There are serializable schedules that can’t be detected using conflict serializability.


**Dependency Graph**

- **Precedence graph:**
  - One node per Xact;
  - Edge from Xact $T_i$ to Xact $T_j$ if an action of $T_i$ precedes and conflicts with one of $T_j$'s actions (RW, WR, WW operations on the same object).

- **Theorem:** Schedule is conflict serializable if and only if its precedence graph is acyclic.
The schedule is not conflict serializable:

- The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
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.)

\[
\begin{array}{ll}
\text{T1: } R(A), W(A) & \text{Abort} \\
\text{T2: } & R(A), W(A) \text{ Abort}
\end{array}
\]

Recoverable, but with cascading aborts.

- S \textit{avoids cascading rollback} if each xact may \textit{read} only those values written by committed xacts.
Recoverability (Contd.)

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

Recoverable, no cascading aborts, but update of A by T2 will be lost!

- 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.
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Deadlock management
- Efficient B+tree locking
Strict 2PL

- **Strict Two-Phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a S *(shared)* lock on object before reading, an X *(exclusive)* lock on object before writing.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  - All locks held by a transaction are released when the transaction completes.
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.
- Strict 2PL is recoverable without anomalies related to aborted transactions. Hence, it simplifies transaction aborts.
Nonstrict 2PL

- Two-Phase Locking Protocol
  - Each Xact must obtain a S (shared) lock on object before reading, an X (exclusive) lock on object before writing.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  - A transaction cannot request additional locks once it releases any locks.
Nonstrict 2PL (contd.)

- **Theorem**: Nonstrict 2PL ensures acyclicity of precedence graph.
  - Nonstrict 2PL only allows *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*!
  - Involves complex abort processing.
  - But allows xacts to go through more quickly.
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Deadlock management
- Efficient B+tree locking
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.
  - Note the difference from the precedence graph for conflict serializability.

- Periodically check for cycles, indicating deadlocks, 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), X(C)
T3: S(C), R(C)
T4: X(A), X(B)

Diagram:

T1 → T2
T4 → T3
T1 → T3
T1 → T4
T2 → T3

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
- Deadlock management
- Efficient B+tree locking
Locking in B+ Trees

- How can we efficiently lock B+tree nodes (pages)?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.
Two Useful Observations

1. Higher levels of the tree only *direct searches* for leaf pages (mostly read-only).

2. For inserts, a node on a path from root to leaf must be locked in X mode, only if a *split can propagate up to it from the modified leaf*.
   - Similar point holds w.r.t. deletes.

- Exploit these observations to design efficient locking protocols that guarantee serializability even though they violate 2PL.
A Simple Tree Locking Algorithm

- **Search:** Start at root and go down; repeatedly, lock child then unlock parent.
  - Searches never go back up. "Crabbing", i.e. holding at most two locks on the parent and current nodes, is enough.

- **Insert/Delete:** Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is **safe**:
  - If child is safe, release all locks on ancestors.
  - Otherwise, hold X locks up to the closed safe ancestor or the root.

- **Safe node:** Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.
Example

Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*
A Better Tree Locking Algorithm (See Bayer-Schkolnick paper)

- **Search**: As before.
- **Insert/Delete**:
  - Set locks as if for **Search**, get to leaf, and set X lock on leaf.
  - If leaf is not **safe**, release all locks, and restart Xact using previous Insert/Delete protocol.
- Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful. In practice, better than previous alg.
Example

Do:
1) Delete 38*
2) Insert 25*
3) Insert 45*, then 46*

20 35 44