Transaction Management: Concurrency Control

Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke
DBMS Architecture

- Query Parser
- Query Rewriter
- Query Optimizer
- Query Executor
- Disk Space Manager
- Access Methods
- Buffer Manager
- Lock Manager
- Log Manager
- Concurrency Control
- Recovery
- DB
Online Transaction Processing (OLTP)

- **Banking:** Multiple programs can touch the same data
- **Ticketing:** Have you tried to get popular concert tickets online?
- **E-commerce:** Have you experienced online shopping during sales?
- **Voting**
- **Telecommunications**
- **Online gaming**
- **A lot of (small) reads and writes on the same data**
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Efficient B+tree locking
- Optimistic concurrency control
Concurrent User Programs

- **Concurrent execution of user programs**: good for performance.
  - When task 1 is doing I/O, run task 2 to utilize the CPU.
  - Improve *average response time* (average delay that a user task experiences)
  - Improve *system throughput* (number of user tasks processed in each time unit)
Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob’s bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it..., and buy it.

- But the DBMS is only concerned about what data is read from/written to the database.

- A transaction is DBMS’s abstract view of a user program, simply, a sequence of reads and writes of DB objects.
ACID Properties for Concurrency

- Many users submit xacts, but each user thinks of his as executing by itself.
  - DMBS *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.
**ACID Properties for 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 so that it can *redo* the actions of committed xacts.
Outline

- Transaction management overview

- Serializability & recoverability

- Lock-based concurrency control

- Efficient B+tree locking

- Other locking issues

- Optimistic concurrency control
Example

- Consider two transactions:

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

- 1st xact transfers $100 from B’s account to A’s.
- 2nd xact credits both accounts with a 6% interest payment.
- 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:

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

  What about:

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

  The DBMS’s view of the second schedule:

  \[
  \begin{align*}
  \text{T1:} & \quad R(A), W(A), \quad R(B), W(B) \\
  \text{T2:} & \quad R(A), W(A), R(B), W(B)
  \end{align*}
  \]
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, ideally want to allow any serializable schedule.
  - Recognizing any serializable schedule as Xact’s are submitted online 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 potentially *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 potentially 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.
Example

\begin{tabular}{|l|l|}
\hline
T1: & R(A), W(A), R(B), W(B) \\
T2: & R(A), W(A), R(B), W(B) \\
\hline
\end{tabular}

- The schedule is not conflict serializable:
  - The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
Weaker Condition on Serializability

- Conflict serializability is *sufficient but not necessary* for serializability.

**S1: interleaved schedule**

- T1: R(A), W(A)
- T2: W(A)
- T3: W(A)

**S2: serial schedule**

- T1: R(A), W(A)
- T2: W(A)
- T3: W(A)
Recoverability

- **Recoverability theory** concerns schedules that involve **aborted** transactions.

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

Unrecoverable!

- A schedule S is **recoverable** if each xact *commits* only after all xacts from which it read have committed.
Recoverability (Contd.)

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

Recoverable, but with cascading aborts.

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

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

Recoverable, no cascading aborts, but update by T2 may be overwritten when aborting T1. So need to consider other xacts during abort!

- **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.
Venn Diagram for Schedules

All schedules

- Recoverable
- Avoid cascading aborts
- Strict

Serializable

- Conflict serializable
- Serial

Committed Xacts

Also Aborted Xacts
Outline

- Transaction management overview
- Serializability & recoverability
  - Lock-based concurrency control
- Efficient B+tree locking
- Other locking issues
- Optimistic concurrency control
(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, no other Xact can get a lock (S or X) on that object.
  3. All locks held by a transaction are released when the transaction completes.

<table>
<thead>
<tr>
<th>Compatibility</th>
<th>Shared</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Exclusive</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

# of locks

\[
\begin{align*}
\text{C} & \quad \text{time} \\
\end{align*}
\]
**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.
  - Strict 2PL is recoverable without anomalies related to aborted transactions.
  - Hence, it simplifies transaction aborts.
Nonstrict 2PL

- **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, no other Xact can get a lock (S or X) on that object.
  3. Duration of locks: 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.
(2) 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), X(C)
T3: S(C), R(C)
T4: X(A), 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.
- **Waits-for graph** is a mechanism to detect deadlocks.
- Real implementation: *(strict) 2PL + deadlock handling*
  - 2PL prevents conflicting actions from being materialized in the DB by blocking xact’s.
  - 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.
(3) The Phantom Problem

- If we consider *insertion and update* of records in a DB, even *Strict 2PL* won’t assure serializability:
  - T1 locks *all* pages containing sailor records with *rating* = 1, and finds oldest sailor, say, *age* = 71.
  - T2 next *inserts* a new sailor: *rating* = 1, *age* = 96.
  - T1 now reads the oldest sailor again; now, *age* = 96 appears as a *phantom*!

- What is broken in our xact model/protocol?
- How can we fix it?
Illustrating the Problem

BEGIN TRANSACTION

   SELECT max(age) FROM Sailors WHERE rating = 1;

   ...

   SELECT max(age) FROM Sailors WHERE rating = 1;

END TRANSACTION

BEGIN TRANSACTION

   INSERT INTO Sailors(sid, sname, rating, age) VALUES (123, 'Fred', 1, 96);

END TRANSACTION

BEGIN TRANSACTION

   UPDATE Sailors S SET rating = S.rating - 1 WHERE NOT EXISTS
   (SELECT * FROM Reserves R WHERE R.date >= '01/01/2016' and R.sid = S.sid);

END TRANSACTION
A Fundamental Reason

- **Problem**: T1 assumes that it has locked all sailor records with \textit{rating} = 1.
  - This is only true if nobody adds/updates records while T1 is running!
  - Phantoms can occur with both inserts and updates.
  - E.g., update a sailor’s \textit{rating} from 3 to 1.

- Need to enforce the assumption that T1 indeed holds locks on \textit{all} records satisfying a condition, e.g., \textit{rating}=1.
Fundamental Reason (cont’d)

- Fundamental reason for the phantom problem:
  - The old transaction model consists of reads and writes to *individual* data items.
  - In practice, transactions include queries that *dynamically* define sets of items based on *predicates*.
  - When the query is executing, all the records satisfying this predicate at a particular time must be locked.
  - Locking individual items, however, cannot prevent later addition of a record satisfying this predicate.

- Solution: extend lockable objects to *index pages* and further to *arbitrary predicates*!
  - Index locking: used in practice
  - Predicate locking: not very practical
Index Locking

- If there is an index on the `rating` field, T1 should lock the *index page* containing data entries with `rating = 1`.
  - If there are no data entries with `rating = 1`, T1 must lock the index page where such a data entry *would* be, if it existed!

- If there is no suitable index, T1 must lock the entire file/table to prevent new pages from being added.
  - so that no new records with `rating = 1` can be added.
Predicate Locking

- Grant locks on some logical predicates
  - e.g. $age > 2 \times sal$, or $(age > 50 \text{ or } age < 30) \text{ and } sal > 10K$

- Index locking is a special case of predicate locking
  - An existing index matches the predicate.
  - It supports efficient implementation of the predicate lock.

- In general, predicate locking has a lot of overhead.
  - For each record, we need to check if it satisfies a complex predicate.
  - Therefore, it is not commonly used.
Outline

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