Transactions

- Many enterprises use databases to store information about their state
  - E.g., balances of all depositors
- When an event occurs in the real world that changes the state of the enterprise, a program is executed to change the database state in a corresponding way
  - E.g., balance must be updated when you deposit
- Such a program is called a transaction

Transactions as part of a larger program

- A user’s 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
- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.
  - Any one execution of a user program
  - Different from the notion of execution of program outside the DBMS
Why the Transaction Concept?

• Consider two programs against the DBMS:
  – P1: BEGIN   A:=A+100, B:=B-100 END
  – P2: BEGIN   A:=A-500, B:=B+500 END
  – E.g., the programs might be moving money from one account to the other
• In the DBMS view each program has two statements, each statement has two actions, one R and one W
  – P1: R(A), W(A), R(B), W(B)
  – P1: R(A), W(A), R(B), W(B)
• In SQL read → SELECT
• In SQL write → UPDATE (or INSERT or DELETE)

Why the Transaction Concept? (cont’d)

• In an ordinary program, no guarantee that actions of the programs will not interleave
  P1: R(A), A:=A+100 W(A)
  P2: R(A), A:=A-500 W(A)
  Time →
• What if system crashes in the middle of the execution of one of these programs?
  – P1: R(A), W(A), and then network connection is lost before R(B), W(B) can be executed

Topics

• The Transaction Concept
  – Properties of Transactions
• Isolation
• Implementing Concurrency Control
• Transaction Support in SQL
• Oracle & Isolation Levels

Special Properties of Transactions

• Transaction execution must maintain the correctness of the database
• Therefore additional requirements are placed on the execution of transactions beyond those placed on ordinary programs
  – Atomicity
  – Consistency
  – Isolation
  – Durability
  (explained next)

ACID properties
Transactions: What & How

• What do transactions provide:
  – Transactions must ensure 4 ACID properties

• How are the properties supported:
  – Concurrency Control subsystem of DBMS
    • Mechanism by which DBMS handles concurrent executions
  – Recovery subsystem of DBMS
    • Mechanism by which DBMS handles partial transactions
      – e.g., txns interrupted by system crash
  – SQL commands for Use by Program Implementer

ACID Transactions – Atomicity

• What:
  – A real-world event either happens or does not happen
  – A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions
  – Either all actions in a transaction are carried out or none

• How:
  – DBMS logs all actions so that it can undo the actions of aborted transactions

• Who is responsible:
  – DBMS Recovery Manager

ACID Transactions - Durability

• What:
  – If the User is informed that a txn completed, then its effect must persist, even if system crashes

• How
  – DBMS logs all actions so that it can redo the actions of txns

• Who is responsible
  – DBMS Recovery Manager

ACID Transactions - Consistency

• What:
  – Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins
    • DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements
    • Beyond this, the DBMS does not really understand the semantics of the data. e.g., it does not understand money transfer between bank accounts.

• How:
  – Proper coding of programs

• Who is responsible:
  – Database Programmer
Ensuring Consistency

• Rules of the enterprise generally limit the occurrence of certain real-world events
  – Student cannot register for a course if current number of registrants = maximum allowed
• Correspondingly, allowable database states are restricted.
  – \( \text{cur}_{\text{reg}} \leq \text{max}_{\text{reg}} \)
• We have already studied how Integrity Constraints can express many of these limitations

Ensuring Consistency (cont’d)

• Transaction designer must ensure that
  IF the database is in a state that satisfies all integrity constraints when execution of a transaction is started
  THEN when the transaction completes:
  • All integrity constraints are once again satisfied (constraints can be violated in intermediate states)
  • New database state satisfies specifications of transaction

Transactions & Ensuring Consistency

• Example 1:
  – In the bank example, to ensure consistency the operations:
  • Deduction from Checking and
  • Deposit into Saving
  – Must both happen or neither should occur
  – The programmer must include both these operations in one transaction
• Example 2:
  – One way to make sure that \( \text{cur}_{\text{reg}} \leq \text{max}_{\text{reg}} \) is to use one transaction to:
  • insert a row in the Transcript table to register a student
  • issue a query to get \( \text{cur}_{\text{reg}}, \text{check against max}_{\text{reg}} \)
  • abort txn if the condition is false, otherwise commit the txn

ACID Transactions - Isolation

• What:
  – The User should be able to understand each txn without considering effect of any concurrently executing txn
  – Txns are isolated from effects of concurrently executing txns
• How:
  – Concurrency Control
• Who is responsible:
  – DBMS Concurrency Control mechanism
ACID Properties

- The DBMS is responsible for ensuring atomicity, durability, and (the requested level of) isolation
  - Hence it provides the abstraction of failure-free, non-concurrent environment, greatly simplifying the task of the transaction designer
- The transaction designer is responsible for ensuring the consistency of each transaction, but doesn’t need to worry about concurrency and system failures

Database Consistency

- What:
  - Every transaction sees a consistent database instance
- How:
  - Follows from
    - Consistency
    - Isolation
    - Atomicity

Topics

- The Transaction Concept
- Isolation
  - Schedules and Serialization
  - Anomalies with Interleaved Execution
- Implementing Concurrency Control
- Transaction Support in SQL
- Oracle & Isolation Levels

Isolation & Concurrency

- Isolation:
  - Each transaction can be considered as executing by itself
  - DB user need only understand his/her transaction
- One way to achieve isolation:
  - Only one transaction at a time!
- But DBMS needs to provide concurrent txns
  - Better response time
  - Better throughput
How is Concurrency Achieved

- DBMS considers a txn as made up of a list of actions
- **Action:**
  - read or write of DB object
  - abort, commit
- Concurrency achieved by interleaving actions of various transactions
- **Schedule:**
  - List of (interleaved) actions

Interleaved Execution

Example

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

- Consider a possible 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

- **Complete schedule:**
  - Schedule that contains either a commit or an abort for each transaction whose actions are listed in the schedule
- **Serial schedule:**
  - Schedule that does not interleave the actions of different transactions
Example: Serial Schedule

Before transactions, let \( A = 1000 \), \( B = 2000 \)

Two transactions:

T1: \( A := A + 100 \), \( B := B - 100 \)

T2: \( A := 1.06 \times A \), \( B := 1.06 \times B \)

For these 2 txns, there are 2 serial schedules:

- T1 executes and then T2 executes
  - Result: \( A = 1166 \), \( B = 2014 \)
- T2 executes and then T1 executes
  - Result: \( A = 1160 \), \( B = 2020 \)

Scheduling Transactions (cont’d)

- Equivalent schedules:
  - For any database state, the effect (on the set of objects in the database) 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
  - Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Example S: Serializable Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A (in DB)</th>
<th>B (in DB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A) 1000</td>
<td>1000</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A) 1100</td>
<td>1100</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A) 1100</td>
<td>1100</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A) 1166</td>
<td>1166</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B) 2000</td>
<td>1166</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B) 1900</td>
<td>1166</td>
<td>1900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B) 1900</td>
<td>1166</td>
<td>1900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B) 2014</td>
<td>1166</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example NS: Non-Serializable Schedule

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A (in DB)</th>
<th>B (in DB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R(A) 1000</td>
<td>1000</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(A) 1000</td>
<td>1000</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(A) 1060</td>
<td>1060</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B) 2000</td>
<td>1100</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>R(B) 2000</td>
<td>1100</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B) 1900</td>
<td>1100</td>
<td>1900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W(B) 2014</td>
<td>1100</td>
<td>1900</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Serializability & Commutativity

• The schedule in Example S should be allowed while the schedule in Example NS shouldn’t be allowed
• Two actions commute if, when executed in either order:
  – The values returned by both are the same and
  – The database is left in the same final state
• In general actions from different transactions commute if either of the following holds:
  – They refer to different data items
  – They are both read requests

Serializability & Commutativity (cont’d)

• Two schedules are equivalent if
  – one can be derived from the other by a series of simple interchanges of commutative operations
• A schedule is serializable if it is equivalent to a serial schedule

Concurrency Control

• Performance requirements might not be achievable if schedules are serializable
• In addition to serializable, DBMSs implement less stringent isolation levels
  – Serializable schedules correct for all applications
  – Less stringent levels do not guarantee correctness for all applications, but are correct for some
• The concurrency control of a DBMS is responsible for implementing isolation levels
• Application programmer is responsible for choosing appropriate level

Topics

• The Transaction Concept
• Isolation
  – Schedules and Serialization
  – Anomalies with Interleaved Execution
• Implementing Concurrency Control
• Transaction Support in SQL
• Oracle & Isolation Levels
Anomalies: Dirty Read

- Occurs in a schedule that allows reading uncommitted data
- Example:
  - Before transactions, let $A = 1000$, $B = 2000$
  - Two transaction:
    - T1: $A := A + 100$, $B := B - 100$
    - T2: $A := 1.06^*A$, $B := 1.06^*B$
  - For these 2 txns, there are 2 serial schedules:
    - T1 executes and then T2 executes
    - Result: $A = 1166$, $B = 2014$
    - T2 executes and then T1 execute
    - Result: $A = 1160$, $B = 2020$

Example: Schedule w. Dirty Read

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>$A := A + 100$</td>
<td>$B := B - 100$</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>$W(A) 1100$</td>
<td>$R(A) 1100$</td>
<td>1100</td>
<td>2000</td>
</tr>
<tr>
<td>T2</td>
<td>$A := 1.06^*A$</td>
<td>$B := 1.06^*B$</td>
<td>1166</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>$W(A) 1166$</td>
<td>$R(B) 2000$</td>
<td>1166</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>$R(B) 2120$</td>
<td>$W(B) 2020$</td>
<td>2120</td>
<td>2020</td>
</tr>
</tbody>
</table>

- Dirty read of $R(A)$ in T2
  - Database is not in a consistent state at end of txns

Anomalies: Unrepeatable Reads

- Example:
  - Before transactions, let $A = 1000$, $B = 2000$
  - Two transaction:
    - T1: $A := A + 100$, $B := B - 100$
    - T2: $C := A + B$
  - For these 2 txns, there are 2 serial schedules:
    - T1 executes and then T2 executes
    - Result: $A = 1100$, $B = 1900$, $C = 3000$
    - T2 executes and then T1 execute
    - Result: $A = 1100$, $B = 1900$, $C = 3000$

Exm: Schedule w. Unrepeatable Reads

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>$A := A + 100$</td>
<td>$B := B - 100$</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>$W(A) 1100$</td>
<td>$R(A) 1000$</td>
<td>1100</td>
<td>2000</td>
</tr>
<tr>
<td>T2</td>
<td>$C := A + B$</td>
<td>$W(B) 1900$</td>
<td>1100</td>
<td>1900</td>
</tr>
<tr>
<td></td>
<td>$W(B) 1900$</td>
<td>$R(B) 1900$</td>
<td>1100</td>
<td>1900</td>
</tr>
<tr>
<td></td>
<td>$W(C) 2900$</td>
<td>Commit</td>
<td>1100</td>
<td>1900</td>
</tr>
</tbody>
</table>

- Unrepeatable read: in T2 T(R(A)) would be different from the 1st R(A) if op was repeated after R(B)
  - Database is not in a consistent state at end of txns
Anomalies: Lost Update

- **Cause:** Overwriting Uncommitted Data
- **Example:**
  - Before transactions, A = 1000, B = 2000
  - Two transaction:
    - T1: A := A + 100, B := B - 100
    - T2: A := 1.06*A, B := 1.06*B
  - For these 2 txns, there are 2 serial schedules:
    - T1 executes and then T2 executes
      - Result: A = 1166, B = 2014
    - T2 executes and then T1 executes
      - Result: A = 1160, B = 2020

---

**Exm: Schedule w. Lost Update**

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
<th>A (in DB)</th>
<th>B (in DB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>1000</td>
<td></td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>W(A)</td>
<td>1060</td>
<td></td>
<td>1060</td>
<td>2000</td>
</tr>
<tr>
<td>W(B)</td>
<td>1100</td>
<td></td>
<td>1100</td>
<td>2000</td>
</tr>
<tr>
<td>R(B)</td>
<td>1900</td>
<td></td>
<td>1900</td>
<td></td>
</tr>
<tr>
<td>W(B)</td>
<td>2014</td>
<td></td>
<td>1100</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Topics**

- The Transaction Concept
- Isolation
- Implementing Concurrency Control
  - Lock-Based Concurrency Control
  - Issues in Lock-based CC
- Transaction Support in SQL
- Oracle & Isolation Levels

---

**Lock-Based Concurrency Control**

- DBMS must implement isolation levels, including the strictest level of allowing only serializable schedules
  - Application developer can choose the appropriate level
- One mechanism: use locks
- Lock:
  - a small bookkeeping object associated with a database object
- Locking protocol:
  - Set of rules to be followed by each transaction
Two Phase Locking Protocols

• **Strict Two-phase Locking (Strict 2PL) Protocol**:
  - Each txn must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - If an txn holds an X lock on an object, no other txn can get a lock (S or X) on that object.

• Strict 2PL allows only serializable schedules

Strict Two-Phase Locking

• Intuition about correctness of the protocol:
  - Locks that are required for non-commutative operations are in conflict.
  - Therefore, active transactions cannot execute operations that do not commute.

• Does strict 2PL allow all serializable schedules?
  - No. Strict 2PL allows only serializable schedules, but not all serializable schedules are allowed by strict 2PL.
  - E.g., look at Example S (Slide 27)

Nonstrict Two-Phase Locking Protocol

• Relaxes the second rule:
  - Each txn must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes.
  - A transaction cannot request additional locks once it release any lock.
  - If an txn holds an X lock on an object, no other txn can get a lock (S or X) on that object.

• Allows only serializable schedules?

Example: Non-Strict 2 Phase CC

<table>
<thead>
<tr>
<th>T1:</th>
<th>T2:</th>
<th>A (in DB)</th>
<th>B (in DB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:=A+100</td>
<td>W(A) 1000</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>B:=B-100</td>
<td>W(A) 1100</td>
<td>1100</td>
<td>2000</td>
</tr>
<tr>
<td>T2:</td>
<td>R(B) 2000</td>
<td>1100</td>
<td>2000</td>
</tr>
<tr>
<td>A:=1.06*A</td>
<td>W(B) 1900</td>
<td>1100</td>
<td>1900</td>
</tr>
<tr>
<td>B:=1.06*B</td>
<td>R(A) 1100</td>
<td>1100</td>
<td>1900</td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>W(A) 1166</td>
<td></td>
<td>1166</td>
<td>2000</td>
</tr>
<tr>
<td>R(B) 2000</td>
<td></td>
<td>1166</td>
<td>2000</td>
</tr>
<tr>
<td>W(B) 2120</td>
<td></td>
<td>1166</td>
<td>2120</td>
</tr>
<tr>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Non-Strict Concurrency Controls

- Non-strict controls: locks can be released before completion
- Problem:
  - Transaction 2 has read a value that gets rolled back when Transaction 1 is aborted
  - Transaction 1 has an effect even though it is aborted
  - Hence, atomicity is violated
- Non-strict 2 phase CC does not guarantee serializability

Lock Release

Two-Phase locking: All locks are acquired before any lock is released

Strict: Transaction holds all locks until completion

Locking Protocols and Anomalies in Non-Serializable Schedules

- In terms of lock release what causes the anomalies?
- Dirty read
  - Write lock released early
- Non-Repeating Read
  - Read lock released early
- Lost Update
  - Read lock released early
  - In the example on Slide 38, write lock is also given up early. But examples can be constructed with only early release of read lock causing the anomaly

Topics

- The Transaction Concept
- Isolation
- Implementing Concurrency Control
  - Lock-Based Concurrency Control
  - Issues in Lock-based CC
- Transaction Support in SQL
- Oracle & Isolation Levels
Blocking Locks

- With lock-based concurrency control protocols an application can be blocked waiting for locks
  - Txn 1 needs a lock on a data object
  - The lock on the data object is held by Txn 2
  - Txn 1 will have to wait until the lock is released by Txn 2
- Example in Oracle
  - Get exclusive lock in SQL Plus session 1
    - SQL> lock table foo in exclusive mode
  - Try getting exclusive lock in SQL Plus session 2
    - SQL> lock table foo in exclusive mode
    - This session will be blocked

Deadlocks

- When a transaction can hold locks and request another lock (e.g., in two-phase locking), a cycle of
  waiting transactions can result:
  - Suppose two transactions are both trying to update the value of \( x \) (for example to deposit in the same bank
    account)
    \[ R_1(x) R_2(x) \text{ Request}_W_1(x) \text{ Request}_W_2(x) \]
- Deadlock: Cycle of transactions waiting for locks to be released by each other.
  - Txn 1 waiting for a lock another Txn 2 is holding
  - Now Txn 2 requests a lock Txn 1 is holding
- Example in Oracle
  - Get exclusive lock on A in one SQL Plus session 1
  - Get exclusive lock on B in another SQL Plus session 2
  - Try getting exclusive lock on B in SQL Plus session 1
  - Try getting exclusive lock on A in SQL Plus session 2
  - A transaction in the cycle must be aborted by DBMS
    - Otherwise the transactions involved will wait forever
  - DBMS uses deadlock detection algorithms or timeout to deal with this
    - E.g., terminate a requests to break the cycle

Deadlocks (cont’d)

- Example in Oracle
  - Get exclusive lock on A in one SQL Plus session 1
  - Get exclusive lock on B in another SQL Plus session 2
  - Try getting exclusive lock on B in SQL Plus session 1
  - Try getting exclusive lock on A in SQL Plus session 2
  - A transaction in the cycle must be aborted by DBMS
    - Otherwise the transactions involved will wait forever
  - DBMS uses deadlock detection algorithms or timeout to deal with this
    - E.g., terminate a requests to break the the cycle

Lock Granularity

- Cost of implementing locks:
  - Space: data structure in DBMS for each lock
  - Time: handling of lock request and release
- Locks can be associated with different size items:
  - Row: fine granularity
  - Page
  - Table: coarse granularity
- Tradeoff:
  - Coarse granularity locks generally have lower cost
  - But fine granularity locks allow greater concurrency
Phantoms & Strict 2-Phase CC with Row Level Locking

SELECT *
FROM Transcript T
WHERE T.CrsCode = 'CS305' AND T.Semester = 'F2004'

- Locking entire table restricts concurrency
- But locking only rows returned yields new **anomaly**
  - T1: execute SELECT
  - T2: insert a new row satisfying WHERE clause
  - T1: execute SELECT again
- Inserted row is called a **phantom**
- Strict two-phase **row locking**
  - Does not prevent phantoms
  - Does not guarantee serializable schedules

Topics

- The Transaction Concept
- Isolation
- Implementing Concurrency Control
  - Transaction Support in SQL
  - Oracle & Isolation Levels

Transaction Support in SQL-92

- Transaction start
  - Automatically started when certain statements execute
    - E.g., UPDATE, INSERT
- Transaction end
  - Explicit end:
    - COMMIT: commits the transaction
    - ROLLBACK: aborts the transaction
  - Implicit commits
    - E.g., DDL in Oracle commits txn; logging out of a SQL Plus Session commits

Savepoints

- Can selectively rollback to a savepoint instead of aborting the whole transaction
  - SAVEPOINT foo
  - ...
  - ROLLBACK TO SAVEPOINT foo
ANSI Standard Isolation Levels

- Each transaction has an isolation level
- Isolation level determines extent to which a given transaction is exposed to the actions of others
  - SET TRANSACTION ISOLATION LEVEL X
- The isolation levels are defined in terms of anomalies

Isolation Levels & Anomalies

- Four isolation levels
- Also termed as ‘degrees’

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Lost Update</th>
<th>Phantom Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>No</td>
<td>Possible</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Possible</td>
<td>Possible</td>
<td>Yes</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Locking Protocols & Isolation Levels

- Early (non-strict) lock release is used to implement less strict levels
  - Short-term locks - held for duration of single statement
  - Long-term locks - held until transaction completes (strict)
- At all levels, any write locks are obtained for long-term

Locking Implementation of Isolation Levels

- READ UNCOMMITTED
  - No read locks
  - Dirty reads possible since transaction can read a write-locked item
  - No modifications to the database allowed with this level
  - Can’t execute INSERT, UPDATE, DELETE
  - Can’t acquire write lock
- READ COMMITTED
  - Short-term read locks on rows
  - Non-repeatable reads possible since transaction releases read lock after reading
Locking Implementation of
Isolation Levels (cont’d)

• REPEATABLE READ
  – Long-term read locks on rows
  – Doesn’t lock objects that txn wants to have unchanged
    • Can run into Phantom problem

• SERIALIZABLE
  – Lock objects that txn wants to have unchanged
  – Combination of table, row, and index locks
    • Prevents against Phantom problem

Topics

• The Transaction Concept
• Isolation
• Implementing Concurrency Control
• Transaction Support in SQL
• Oracle & Isolation Levels

Isolation Levels Supported in Oracle

• Of the 4 ANSI standard isolation levels
  Oracle DBMS supports two:
  – READ COMMITTED
  – SERIALIZABLE
• The default isolation level is:
  – READ COMMITTED
• Internally, Oracle implements concurrency control using a mechanism different than just using locks

Multi-version Concurrency Control

• We won’t get into the details, but here are some points relevant to an application developer
• By default read locks are never acquired by DBMS
  – Readers don’t block writers
  – Writers don’t block readers
• The old value of an item is not overwritten when it is updated. Instead, a new version is created
  – Hence the name multi-version concurrency control
• Statement level read consistency
  – Query result is consistent with respect to the time query started
  – SQL statement is isolated from any changes made since the time statement execution started
Multi-version Concurrency Control: Example

- T1 issues SQL stmt to read all rows in table Accounts
- Before T1 gets done, T2 updates Balance in one row of Accounts
  - The DBMS maintains the previous version of Balance in a special area
- When T1 gets to reading the updated row, it does not block
  - Recall read lock not acquired
- T1 is given the value of Balance that existed at the time the SQL stmt started
  - Value returned by DBMS by reading the previous version that DBMS had stored in the special area

Lock Requests by Application?

- Generally all the locking/unlocking is done by the DBMS
- If serializability is important, then choose SERIALIZABILITY isolation level
  - Application should never have to explicitly request locks
- Typically though the (default level) READ COMMITTED is used
  - For some cases application may explicitly request locks

Dealing with Anomalies in READ COMMITTED

- Recall in READ COMMITTED transactions 3 anomalies are possible
  - Non-repeatable reads, lost updates, phantom problem
- For many transactions non-repeatable reads or phantoms aren’t big problems
  - E.g., Consider booking a flight online where for a flight a table lists all the seats and whether a seat is available
  - Flight(Flight_num, Seat_num, Customer_id)
  - If a seat is available CustomerId is NULL
- Are phantoms a problem? Would there be a need to have repeatable reads for multiple read stmts in the transaction?

Lost Update: Example

- Lost updates though can be a problem
- Consider two users A and B concurrently trying to reserve a seat
- On behalf of User A, T1 reads the table and displays available seat to User A
- On behalf of User B, T2 reads the table and displays available seat to User to User B

SELECT Seat_num
FROM Flight
WHERE Flight_num = 'CO718' AND Customer_id IS NULL

- No read locks obtained by T1 or T2
Lost Update: Example

- User A decides to reserve seat 12D
  - T1 updates row, marks it as taken by User A and commits
  - Write lock obtained and given up on commit

```
UPDATE Flight
SET Customer_id = 'User_A'
WHERE Seat_num = '12D'
AND Flight_num = 'CO718'
```

- User B decides to reserve the seat 12D
  - T2 updates row, marks it as taken by User B and commits
  - Write lock obtained and given up on commit

Possible Solution: Request Lock Explicitly

- One solution to the lost update problem is to lock rows before showing to a User

```
SELECT Seat_num
FROM Flight
WHERE Flight_num = 'CO718' AND Customer_id IS NULL
FOR UPDATE NOWAIT
```

- T1 will lock the rows for available seats
- When T2 executes this statement, the execution fails with error message that the “resource” is being used
  - If NOWAIT is not specified, T2 will block until T1 commits

Possible Solution: Request Lock Explicitly (cont’d)

- So what are the pros and cons of the specific solution shown in the previous slide?
- Pros:
  - Lost update problem prevented
  - If a User is shown a seat, s/he will be able to reserve it
    - This is an example of a pessimistic approach: avoid failure at an early stage and reduce concurrency to achieve this
- Cons:
  - All the available seats are locked while the first user makes a choice. Concurrency reduced substantially!

Alternative Solutions

1. Use such a pessimistic approach only when a few row need to be locked
   - E.g., Don’t lock rows when displaying all the available seats. Lock a row when the User has chosen a specific seat and is now going through confirmation screens
2. For the specific example, the UPDATE statement can be modified to prevent lost update

```
UPDATE Flight
SET Customer_id = 'User_A'
WHERE Seat_num = '12D'
AND Flight_num = 'CO718'
AND Customer_id IS NULL
```

- This is an example of an optimistic approach: expect the best case and don’t reduce concurrency. If failure later, inform the User
Topics

• The Transaction Concept
• Isolation
• Implementing Concurrency Control
• Transaction Support in SQL
• Oracle & Snapshot Isolation

Summary

• Transactions are one of the most important features supported by DBMS
• We discussed the four important ACID properties of transactions and who is responsible for each of these properties
  – Atomicity
  – Consistency
  – Isolation
  – Durability

Summary (cont’d)

• There is a tension between providing isolation and providing concurrency
  – More isolation can mean lowered concurrency
• Lock based protocols are a widely used mechanism for implementing concurrency control
• DBMS support multiple levels of isolation
  – Less strict levels of isolation increase concurrency but a transaction may not be isolated from other concurrent transactions
  – Different anomalies possible with different isolation levels

Summary (cont’d)

• An application programmer can decide the level of isolation needed by a transaction coded by them
  – Set the isolation level before executing the transaction
  – Code the transaction with that isolation level in mind
    • For less strict isolation level will have to deal with the potential anomalies of the level
• Concurrency control is implemented differently by different DBMS vendors
  – Behavior of a transactional application can be different on different DBMSs
  – If an application needs to run on different DBMSs, make sure to understand the default behavior of transactions on each of these