CS143: Transactions

Professor Junghoo “John” Cho
Motivation (1)

• Crash recovery
  • Example: Transfer $1M from Susan to Jane

  S1: UPDATE Account SET balance = balance - 1000000 WHERE owner = `Susan`
  S2: UPDATE Account SET balance = balance + 1000000 WHERE owner = `Jane`

  System crashes after S1 but before S2. What now?
**Motivation (2)**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = balance</td>
<td>B = balance</td>
<td>100</td>
</tr>
<tr>
<td>A = A - 10</td>
<td>B = B - 20</td>
<td>100</td>
</tr>
<tr>
<td>Give out $10</td>
<td>Give out $20</td>
<td>90</td>
</tr>
<tr>
<td>balance = A</td>
<td>balance = B</td>
<td>90</td>
</tr>
</tbody>
</table>

- **Q:** How can DBMS guarantee that these “bad” scenarios will never happen?
Transaction

• A sequence of SQL statements that are executed as “one unit”
• Two key commands related to transaction
  • After a sequence of SQL commands, user can issue either COMMIT or ROLLBACK
    • COMMIT
      • “I am done. Commit everything that I have done!”
      • All changes made by the transaction must be stored permanently
    • ROLLBACK
      • “I changed mind. Ignore what I just did!”
      • Undo all changes made by the transaction
Creating a Transaction

• All SQL commands until COMMIT/ROLLBACK become one transaction.
ACID Property of Transaction

• DBMS guarantees **ACID** property on all transactions
  • Atomicity: “all or nothing”
    • Either ALL OR NONE of the operations in a transaction is executed
    • If system crashes in the middle of a transaction, all changes are “undone”
  • Consistency
    • If the database was in a “consistent” state before transaction, it is still in a consistent state after the transaction
  • Isolation
    • Even if multiple transactions run concurrently, the final result is the same as each transaction runs in isolation in a sequential order
  • Durability
    • All changes made by “committed” transaction will remain even after system crash
Autocommit Mode

• Sometimes, it is too inconvenient to declare transactions explicitly
• Autocommit mode
  • When ON: Every SQL statement automatically becomes one transaction
  • When OFF: As usual
    • All SQL commands through COMMIT/ROLLBACK become one transaction
Setting Autocommit Mode

• Oracle: SET AUTOCOMMIT ON/OFF (default is off)
• MySQL: SET AUTOCOMMIT = {0|1} (default is on. InnoDB only)
• MS SQL Server: SET IMPLICIT_TRANSACTIONS OFF/ON (default is off)
  • IMPLICIT_TRANSACTION ON means AUTOCOMMIT OFF
• DB2: UPDATE COMMAND OPTIONS USING c ON/OFF (default is on)
• In JDBC: connection.setAutoCommit(true/false) (default is on)
• In Oracle, MySQL, and MS SQL Server, “BEGIN TRANSACTION” command temporarily disables autocommit mode until COMMIT or ROLLBACK
SQL Isolation Levels

• By default, RDBMS guarantees ACID for transactions
• Some applications may not need ACID and may want to allow minor “bad scenarios” to gain more “concurrency”
• By specifying “SQL Isolation Level,” app developer can specify what type of “bad scenarios” can be allowed for their apps
  • Dirty read, non-repeatable read, and phantom
Dirty Read

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>1000</td>
</tr>
<tr>
<td>Eddie</td>
<td>1000</td>
</tr>
<tr>
<td>Esther</td>
<td>1000</td>
</tr>
<tr>
<td>John</td>
<td>1000</td>
</tr>
<tr>
<td>Melanie</td>
<td>1000</td>
</tr>
</tbody>
</table>

• T1: UPDATE Employee SET salary = salary + 100;
   T2: SELECT salary FROM Employee WHERE name = ‘Amy’;

• Q: Under ACID, once T1 update Amy’s salary, can T2 read Amy’s salary?

• Some applications may be OK with *dirty read*
  • Among 4 SQL isolation levels, READ UNCOMMITTED allows dirty read
## SQL Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Dirty read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Y</td>
</tr>
<tr>
<td>Read committed</td>
<td>N</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>N</td>
</tr>
<tr>
<td>Serializable</td>
<td>N</td>
</tr>
</tbody>
</table>
Non-repeatable Read

• T1: UPDATE Employee SET salary = salary + 100 WHERE name = ‘John’;

T2: (S1) SELECT salary FROM Employee WHERE name = ‘John’;
...
  (S2) SELECT salary FROM Employee WHERE name = ‘John’;

• Q: Under ACID, can T2 get different values for S1 and S2?

• **Non-repeatable read**: When Ti reads the same tuple multiple times, Ti may get different value

• SQL isolation levels, READ UNCOMMITTED and READ COMMITTED, allow non-repeatable read
## SQL Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Read committed</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Serializable</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Phantom

• T1: INSERT INTO Employee VALUES (Beverly, 1000), (Zack, 1000);
  T2: SELECT SUM(salary) FROM Employee;

<table>
<thead>
<tr>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>1000</td>
</tr>
<tr>
<td>Eddie</td>
<td>1000</td>
</tr>
<tr>
<td>Esther</td>
<td>1000</td>
</tr>
<tr>
<td>John</td>
<td>1000</td>
</tr>
<tr>
<td>Melanie</td>
<td>1000</td>
</tr>
</tbody>
</table>

• Q: Under ACID, what may T2 return?
Phantom

- **Phantom**: When new tuples are inserted, statements may or may not see (part of) them
  - Preventing phantom can be very costly
  - Exclusive lock on the entire table or a range of tuples
- Except the isolation level SERIALIZABLE, phantoms are allowed
## SQL Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Read committed</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Serializable</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
Access Mode

• A transaction can be declared to be *read only*, when it has SELECT statements only (no INSERT, DELETE, UPDATE)

• DBMS may use this information to optimize for more concurrency
Declaring SQL Isolation Level

• SET TRANSACTION [READ ONLY] ISOLATION LEVEL <level>
  • e.g., SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
• More precisely “SET TRANSACTION [access mode,] ISOLATION LEVEL <level>”
  • access mode: READ ONLY/READ WRITE (default: READ WRITE)
  • level:
    • READ UNCOMMITTED
    • READ COMMITTED (default in Oracle, MS SQL Server)
    • REPEATABLE READ (default in MySQL, IBM DB2)
    • SERIALIZABLE
  • READ UNCOMMITTED is allowed only for READ ONLY access mode
• Isolation level needs to be set before every transaction
Mixing Isolation Levels

• John’ initial salary = 1000
  T1: UPDATE Employee SET salary = salary + 100; ROLLBACK;
  T2: SELECT salary FROM Employee WHERE name = ‘John’;

• Q: T1: SERIALIZABLE and T2: SERIALIZABLE. What may T2 return?

• Q: T1: SERIALIZABLE and T2: READ UNCOMMITTED. What may T2 return?

• Isolation level is in the eye of the beholding operation
  • Global ACID is guaranteed only when all transactions are SERIALIZABLE
Guaranteeing ACID

• T1: UPDATE Student SET GPA = 3.0 WHERE sid = 30;

• DBMS does not immediately writes the updated disk block back to disk for performance reasons
  • Q: What happens if the system crashes before the block is written back?
Rolling Back to Earlier State

- $T$: read(A) write(A) read(B) write(B)

Q: What if we execute up to “read(A) write(A) read(B)” and decide to ROLLBACK? How can we go back to the “old value” of $A$?
Partial Execution

• $T$: read(A) write(A) read(B) write(B)

Q: What if system executes up to “read(A) write(A)”, and system crashes? What should the system do when it reboots? How does the system know whether $T$ did not finish?
Logging: Intuition

• In a separate log file, save the following log records before $T_i$ takes any action:

<table>
<thead>
<tr>
<th>Log record</th>
<th>When</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;T_i, \text{start}&gt;$</td>
<td>Before transaction $T_i$ starts</td>
</tr>
<tr>
<td>$&lt;T_i, \text{commit/abort}&gt;$</td>
<td>Before transaction $T_i$ is committed/aborted</td>
</tr>
<tr>
<td>$&lt;T_i, X, \text{old-value, new-value}&gt;$</td>
<td>Before a statement in $T_i$ changes value of $X$ from “old-value” to “new-value”</td>
</tr>
</tbody>
</table>

• These records are used during ROLLBACK or during crash recovery
Logging Example

T1
1. x = read(A)
2. x = x - 50
3. write(A, x)
4. y = read(B)
5. y = y + 50
6. write(B, y)
7. commit

T2
1. z = read(C)
2. z = z * 2
3. write(C, z)
4. commit

Memory
- A: 100
- B: 100
- C: 100

Log file
1. <T1, start>
2. <T1, A, 100, 50>
3. <T2, start>
4. <T2, C, 100, 200>
5. <T2, commit>
6. <T1, B, 100, 150>
7. <T1, commit>

Disk
- A: 100
- B: 100
- C: 100

Log file
1. <T1, start>
2. <T1, A, 100, 50>
3. <T2, start>
4. <T2, C, 100, 200>
5. <T2, commit>
6. <T1, B, 100, 150>
7. <T1, commit>
Rules for Log-Based Recovery

1. DBMS generates a log record before start and end and modification by $T_i$
2. Before $T_i$ is committed, all log records until $T_i$’s commit must be flushed to disk
3. Before any modified tuple is written back to disk, all log records through the tuple modification must be flushed to disk first
   • Example: the log record $<T_i, A, 5, 10>$ should be written to the disk before the tuple $A$ is updated to 10 in disk
4. During ROLLBACK, DBMS reverts to old values of tuples using log records
5. During crash recovery, DBMS does:
   a) “re-execute” all actions in the log file from the beginning to the end and
   b) “rolls back” all actions from non-committed transactions in the reverse order
Example: Recovery

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = read(A)</td>
<td>z = read(C)</td>
</tr>
<tr>
<td>x = x - 50</td>
<td>z = z * 2</td>
</tr>
<tr>
<td>write(A, x)</td>
<td>write(C, z)</td>
</tr>
<tr>
<td></td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>y = read(B)</td>
<td>1 &lt;T1, start&gt;</td>
</tr>
<tr>
<td>y = y + 50</td>
<td>2 &lt;T1, A, 100, 50&gt;</td>
</tr>
<tr>
<td>write(B, y)</td>
<td>3 &lt;T2, start&gt;</td>
</tr>
<tr>
<td>commit</td>
<td>4 &lt;T2, C, 100, 200&gt;</td>
</tr>
</tbody>
</table>

Disk

| A: 100 | B: 100 | C: 100 |

Log file

1 <T1, start>
2 <T1, A, 100, 50>
3 <T2, start>
4 <T2, C, 100, 200>
Example: Recovery

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = read(A)</td>
<td>z = read(C)</td>
</tr>
<tr>
<td>x = x - 50</td>
<td>z = z * 2</td>
</tr>
<tr>
<td>write(A, x)</td>
<td>write(C, z)</td>
</tr>
<tr>
<td>y = read(B)</td>
<td></td>
</tr>
<tr>
<td>y = y + 50</td>
<td>commit</td>
</tr>
<tr>
<td>write(B, y)</td>
<td>commit</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th></th>
<th>A: 50</th>
<th>B: 100</th>
<th>C: 100</th>
</tr>
</thead>
</table>

Log file

1 <T1, start>
2 <T1, A, 100, 50>
3 <T2, start>
4 <T2, C, 100, 200>
5 <T2, commit>
Example: Recovery

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = read(A)</td>
<td>z = read(C)</td>
</tr>
<tr>
<td>x = x - 50</td>
<td>z = z * 2</td>
</tr>
<tr>
<td>write(A, x)</td>
<td>write(C, z)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>write(C, z)</td>
<td>commit</td>
</tr>
</tbody>
</table>

Log file

1 <T1, start>  
2 <T1, A, 100, 50>  
3 <T2, start>  
4 <T2, C, 100, 200>  
5 <T2, commit>  
6 <T1, B, 100, 150>  
7 <T1, commit>
Summary

• DBMS uses a log file to ensure ACID for transactions
  • Helps rolling back partially executed transactions
  • Helps recovery after crash

• Before modifying any data, DBMS generates a log record
• Before commit, DBMS flushes log records to disk to ensure durability
• During recovery, records in the log file are “replayed” to put the system in the supposed state