Outline

1 Why We Need Transactions
   Concurrency
   Failures

2 Transactions
   Properties
   Formal Definition
   Completion States

3 Transactions in SQL
   Transaction Specification
   Isolation Levels

Notes

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________
Why We Need Transactions

- A database is a shared resource accessed by many users and processes concurrently.
  - Both queries and modifications
- Not managing this concurrent access to a shared resource will cause problems (not unlike in operating systems)
  - Problems due to concurrency
  - Problems due to failures
Problems Caused by Concurrency

Accounts(Anum, CId, BranchId, Balance)

- Application 1: You are depositing money to your bank account.
  
  ```
  update Accounts 
  set Balance = Balance + 100 
  where Anum = 9999
  ```

- Application 2: The branch is calculating the balance of the accounts.
  
  ```
  select Sum(Balance) 
  from Accounts
  ```

**Problem – Inconsistent reads**

If the applications run concurrently, the total balance returned to application 2 may be inaccurate.
Another Concurrency Problem

- Application 1: You are depositing money to your bank account at an ATM.
  
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999

- Application 2: Your partner is withdrawing money from the same account at another ATM.
  
  update Accounts
  set Balance = Balance - 50
  where Anum = 9999

Problem – Lost Updates

If the applications run concurrently, one of the updates may be “lost”, and the database may be inconsistent.
Yet Another Concurrency Problem

- Application 1:
  
  ```sql
  update Employee
  set Salary = Salary + 1000
  where WorkDept = 'D11'
  ```

- Application 2:
  
  ```sql
  select * from Employee
  where WorkDept = 'D11'

  select * from Employee
  where Lastname like 'A%'
  ```

Problem – Non-Repeatable Reads

If there are employees in D11 with surnames that begin with “A”, Application 2’s queries may see them with different salaries.
We need to worry about interaction between two applications when

- one **reads** from the database while the other **writes** to (modifies) the database;
- both **write** to (modify) the database.

We do **not** worry about interaction between two applications when both **only** read from the database.
Problems Caused by Failures

- Update all account balances at a bank branch.

```sql
update Accounts
set Balance = Balance * 1.05
where BranchId = 12345
```

**Problem**
If the system crashes while processing this update, some, but not all, tuples with `BranchId = 12345` (i.e., some account balances) may have been updated.

**Problem**
If the system crashes after this update is processed but before all of the changes are made permanent (updates may be happening in the buffer), the changes may not survive.
Another Failure-Related Problem

- transfer money between accounts:

  ```sql
  update Accounts
  set Balance = Balance - 100
  where Anum = 8888
  
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  ```

**Problem**

If the system fails between these updates, money may be withdrawn but not redeposited.
We need to worry about **partial** results of applications on the database when a crash occurs.

We need to make sure that when applications are completed their changes to the database survive crashes.
**Definition (Transaction)**

An application-specified *atomic* and *durable* unit of work (a *process*).

- Concurrency transparency
- Failure transparency

---

**Notes**

---
Properties of Transactions

Atomic: a transaction occurs entirely, or not at all
Consistency: each transaction preserves the consistency of the database
Isolated: concurrent transactions do not interfere with each other
Durable: once completed, a transaction’s changes are permanent
How do DBMSs Guarantee These

**Isolation:** Concurrency control algorithms and techniques guarantee concurrent transactions do not interfere with each other and don’t see each other’s changes until they complete.

- Some sort of mutual exclusion is typically implemented (i.e., locking) but alternatives exist

**Atomicity & Durability:** Recovery management guarantees that committed transactions are durable (despite failures), and that aborted transactions have no effect on the database.

- DBMS logs every action securely so that it can consult the log later to determine what to do.

**Good news/Bad news...**

We will not study these; they are covered in CS448.
Let
- $o_i(x)$ be some operation of transaction $T$ operating on data item $x$, where $o_i \in \{\text{read, write}\}$ and $o_i$ is atomic;
- $OS \cup o_i$;
- $N \in \{\text{abort, commit}\}$

Transaction $T$ is a partial order $T = \{\Sigma, \prec\}$ where

1. $\Sigma = OS \cup \{N\}$,
2. For any two operations $o_i, o_j \in OS$, if $o_i = r(x)$ and $o_j = w(x)$ for any data item $x$, then either $o_i \prec o_j$ or $o_j \prec o_i$,
3. $\forall o_i \in OS, o_i \prec N$. 
Example

Consider a transaction $T$:

$$T = \{ \text{Read}(x), \text{Read}(y), x \leftarrow x + y, \text{Write}(x), \text{commit} \}$$

Then

$$\Sigma = \{ r[x], r[y], w[x], c \}$$

$$\preceq = \{ (r[x], w[x]), (r[y], w[x]), (w[x], c), (r[x], c), (r[y], c) \}$$

DAG representation

```
  r[x]
     /\    
    /  \   
   /    \  
  r[y]  w[x] --> c
```

Notes

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________
How Do Transactions Help?

- Application 1: You are depositing money to your bank account at an ATM.

  ```sql
  update Accounts
  set Balance = Balance + 100
  where Anum = 9999
  ```

- Application 2: Your partner is withdrawing money from the same account at another ATM.

  ```sql
  update Accounts
  set Balance = Balance - 50
  where Anum = 9999
  ```

**Isolation**

If each of these applications run as a transaction, their effects would be isolated from each other – Application 2 can’t see Application 1’s update until Application 1 completes.
How Do Transactions Help?

- Update all account balances at a bank branch.

```sql
update Accounts
set Balance = Balance * 1.05
where BranchId = 12345
```

**Atomicity**

If the application runs as a transaction, either all the accounts will get updated or none of them will.
**Transaction Completion**

**COMMIT**: Any updates a transaction has made become permanent and visible to other transactions. Before COMMIT, changes are tentative.

- Atomicity: commit is the “all” in “all-or-nothing” execution.
- Durability: updates will survive crashes.

**ABORT**: Any updates a transaction may have made are undone (erased), as if the transaction never ran at all.

- Isolation: abort is the “nothing” in “all-or-nothing” execution.

A transaction that has started but has not yet aborted or committed is said to be *active.*

---

**Notes**
Transactions in SQL

- A new transaction is begun when an application first executes an SQL command.
- Two SQL commands are available to terminate a transaction:
  - `commit`: commits the transaction
  - `rollback`: abort the transaction
- A new transaction begins with the application’s next SQL command after `commit` or `rollback`.
The start of a new SQL expression (SELECT, UPDATE, INSERT, DELETE, CREATE) automatically starts a transaction – no explicit command required, but the termination needs to be specified.

```
SELECT * FROM Employee WHERE WorkDept = 'D11'
COMMIT

UPDATE Employee
SET Salary = Salary + 1000 WHERE WorkDept = 'D11'
COMMIT
```
Example Transaction – Embedded SQL

... main() ...

EXEC SQL WHENEVER SQLERROR GOTO error;
EXEC SQL UPDATE Employee
    SET Salary = Salary + 1000
    WHERE WorkDept = 'D11';
EXEC SQL COMMIT;
return(0);
...
error:
    printf("update failed, sqlcode = %ld\n",SQLCODE);
EXEC SQL ROLLBACK;
return(-1);
...

Notes
Explicitly Aborting Transaction

main() { ... 
  EXEC SQL BEGIN DECLARE SECTION;
    int actno1, actno2;  real amount;
  EXEC SQL END DECLARE SECTION;
  gets(actno1,actno2,amount);
  EXEC SQL UPDATE Accounts
    SET Balance = Balance + :amount WHERE Anum = :actno2;
  SELECT Balance INTO tempbal FROM Accounts
    WHERE Anum = :actno1;
  if (tempbal - :amount)<0 {
    printf("insufficient funds");
    EXEC SQL ROLLBACK;
    return(-1);}
  else {
    EXEC SQL UPDATE Accounts
      SET Balance = Balance + :amount WHERE Anum = :actno1;
    EXEC SQL COMMIT;
    printf("funds transfer completed");
    return(0); } }
Setting Transaction Properties

```
set transaction <transaction mode>
    [, <transaction mode>] [, <transaction mode>]
```

transaction mode ::= <diagnostic size>
                   | <access mode>
                   | <isolation level>

- Diagnostic size determines how many error conditions can be recorded.
- Access mode indicates whether the transaction is READ ONLY or READ WRITE (default).
- Isolation level determines how the interactions of transactions are to be managed (remember the concurrency problems).
SQL Isolation Levels

- Different isolation levels deal with different concurrency problems.
- Four isolation levels are supported, with the highest being serializability:
  
  **Level 0 (Read Uncommitted):** transaction may see uncommitted updates
  
  **Level 1 (Read Committed):** transaction sees only committed changes, but non-repeatable reads are possible
  
  **Level 2 (Repeatable Read):** reads are repeatable, but “phantoms” are possible
  
  **Level 3 (Serializability)**
Level 3 – Serializability

- This is the strongest form of isolation level.
- Concurrent transactions must appear to have been executed sequentially, i.e., one at a time, in some order. If $T_i$ and $T_j$ are concurrent transactions, then either:
  1. $T_i$ will appear to precede $T_j$, meaning that $T_j$ will “see” any updates made by $T_i$, and $T_i$ will not see any updates made by $T_j$, or
  2. $T_i$ will appear to follow $T_j$, meaning that $T_i$ will see $T_j$’s updates and $T_j$ will not see $T_i$’s.
Serializability: An Example

- An interleaved execution of two transactions, $T_1$ and $T_2$:
  \[ H_a = w_1[x] \text{ } r_2[x] \text{ } w_1[y] \text{ } r_2[y] \]
- An equivalent serial execution of $T_1$ and $T_2$:
  \[ H_b = \begin{array}{c}
  w_1[x] \text{ } w_1[y] \\
  \text{ } T_1 \\
  r_2[x] \text{ } r_2[y] \\
  \text{ } T_2
  \end{array} \]
- An interleaved execution of $T_1$ and $T_2$ with no equivalent serial execution:
  \[ H_c = w_1[x] \text{ } r_2[x] \text{ } r_2[y] \text{ } w_1[y] \]

$H_a$ is serializable because it is equivalent to $H_b$, a serial schedule. $H_c$ is not serializable.
Transactions and Histories

- Two operations conflict if:
  1. they belong to different transactions,
  2. they operate on the same object, and
  3. at least one of the operations is a write

- Two types of conflicts:
  1. Read-Write
  2. Write-Write

- An execution history over a set of transactions $T_1 \ldots T_n$ is an interleaving of the operations of $T_1 \ldots T_n$ in which the operation ordering imposed by each transaction is preserved.

- Two important assumptions:
  1. Transactions interact with each other only via reads and writes of objects
  2. A database is a fixed set of independent objects
Serializability

**Definition ((Conflict) Equivalence)**

Two histories are *(conflict) equivalent* if

- they are over the same set of transactions, and
- the ordering of each pair of conflicting operations is the same in each history

**Definition ((Conflict) Serializability)**

A history $H$ is said to be *(conflict) serializable* if there exists some *serial* history $H'$ that is *(conflict) equivalent* to $H$
Testing for Serializability

$r_1[x] \ r_3[x] \ w_4[y] \ r_2[u] \ w_4[z] \ r_1[y] \ r_3[u] \ r_2[z] \ w_2[z] \ r_3[z] \ r_1[z] \ w_3[y]$

Is this history serializable?

**Theorem**

A history is serializable iff its serialization graph is acyclic.
Serialization Graphs

Serialization graph $SG_H = (V, E)$ for schedule $H$ is defined as:

1. $V = \{ T \mid T$ is a committed transaction in $H \}$
2. $E = \{ T_i \rightarrow T_j \text{ if } o_{ij} \in T_i \text{ and } o_{kl} \in T_k \text{ conflict and } o_{ij} <_H o_{kl} \}$

$r_1[x] r_3[x] w_4[y] r_2[u] w_4[z] r_1[y] r_3[u] r_2[z] w_2[z] r_3[z] r_1[z] w_3[y]$
The history above is equivalent to

\[ w_4[y] \ w_4[z] \ r_2[u] \ r_2[z] \ w_2[z] \ r_1[x] \ r_1[y] \ r_1[z] \ r_3[x] \ r_3[z] \ w_3[y] \]

That is, it is equivalent to executing \( T_4 \) followed by \( T_2 \) followed by \( T_1 \) followed by \( T_3 \).
Level 0 – Read Uncommitted

Transaction at this level may see uncommitted updates of other transactions.

- **Dirty read**: Transaction $T_i$ may read the update of uncommitted transaction $T_j$.
- If $T_j$ later aborts, the value that $T_i$ read is incorrect.
- Database may be corrupted as well.
main() { ... 
  EXEC SQL BEGIN DECLARE SECTION;
    int actno1, actno2; real amount;
  EXEC SQL END DECLARE SECTION;
  gets(actno1,actno2,amount);
  EXEC SQL UPDATE Accounts
    SET Balance = Balance + :amount WHERE Anum = :actno2;
  SELECT Balance INTO tempbal FROM Accounts
    WHERE Anum = :actno1;
  if (tempbal - :amount)<0 {
    printf("insufficient funds");
    EXEC SQL ROLLBACK;
    return(-1);}
  else {
    EXEC SQL UPDATE Accounts
    SET Balance = Balance + :amount WHERE Anum = :actno1;
    EXEC SQL COMMIT;
    printf("funds transfer completed");
    return(0); }  }  

Notes
Read Uncommitted Example

Start Balance(777) = $300, Balance(888) = $100, Balance(999) = $200

$T_1(888, 999, $150)$  $T_2(999, 777, $250)$

Add $150 to 999 ($350)

Test Balance of 888 ($100)

Rollback: Deduct $150 from 999 ($-50)

Add $250 to 777 ($550)

Test Balance of 999 ($350)

Deduct $250 from 999 and Commit ($100)
Level 1 – Read Committed

Transaction at this level will not see uncommitted updates of other transactions, but non-repeatable reads are possible.

- **Non-repeatable read**: Transaction $T_i$ reads a value from the database. Transaction $T_j$ updates that value. When $T_i$ reads the value again, it will see different value.
- $T_i$ is reading $T_j$’s value after $T_j$ commits (so no dirty reads).
- However, $T_j$’s update is in between two reads by $T_i$.

We have seen an example early on.
Level 2 – Repeatable Reads

Transaction at this level will not have repeatable reads problem (i.e., multiple reads will return the same value), but phantoms are possible.

- Transaction $T_i$ reads a row from a table (perhaps based on a predicate in WHERE clause).
- Transaction $T_j$ inserts some tuples into the table.
- $T_i$ issues the same read again and reads the original row and a number of new rows that it did not see the first time (these are the phantom tuples).
Repeatable Reads Example

Application 1:
\[
\text{select } * \\
\text{from Employee} \\
\text{where WorkDept = 'D11'}
\]

\[
\text{select } * \\
\text{from Employee} \\
\text{where Salary > 50000}
\]

Application 2:
\[
\text{insert into Employee} \\
\quad \text{values}('000123','Sheldon','Q','Jetstream','D11','05/01/00',52000.00)
\]

Problem
Application 1’s second query may see Sheldon Jetstream, even though its first query does not.
### Interaction Between Transactions at Different Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Read Committed</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Serializable</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Snapshot Isolation

A transaction will see a consistent snapshot of the database when it started executing.

- A transaction reads the committed values from the database when it starts.
- If it does not make any updates, no problem.
- If it makes updates that do not conflict by any updates made by any other transaction, it can commit.
- If it makes updates that do conflict by an update made by another transaction, it has to rollback.

Read-Write conflicts are avoided; only Write-Write conflicts are managed.