The Transaction Concept

- Multiple online users:
  - Gives rise to the concurrency problem.
- Component unreliability:
  - Gives rise to the failure problem.
- Database designers confronted these problems in the context of managing persistent data:
  - Online transaction processing system
  - Design, implementation, and operation of large application systems with hundreds of terminals, tens of computers, providing service with no downtime, guaranteeing application correctness and data consistency.

The Transaction Concept

- Transactions provide an integrative framework in the presence of many "moving parts".
- Distributed transaction-oriented systems are the enabling technology:
  - Distributed and Networked applications
  - E-commerce and Workflow systems
  - Large-scale Information Infrastructures

- Without transactions, distributed systems/networked applications cannot be made to work.
The Transaction Concept

Transactions were originally developed in the context of DBMS as a paradigm to deal with:

- Concurrent access to shared data
- Failures of different kinds/types.

Typical and canonical application scenarios in the context of banking application: Debit/Credit operations, and fund Transfers.

The key problem solved in an elegant manner:

- Subtle and difficult issue of keeping data consistent in the presence of concurrency and failures while ensuring performance, reliability, and availability.

OLTP Example: Debit/Credit

```c
void main () {
  EXEC SQL BEGIN DECLARE SECTION
  int BAL, AID, amount;
  EXEC SQL END DECLARE SECTION;
  scanf("%d %d", &AID, &amount);  // USER INPUT
  EXEC SQL Select Balance into :BAL From Account Where Account_Id = :AID;  // READ FROM DB
  BAL = BAL + amount; // update BALANCE
  EXEC SQL Update Account
    Set Balance = :b
    Where Account_Id = :AID;  // WRITE TO DB
  EXEC SQL Commit Work;
}
```

Concurrent Executions: Lost Update Anomaly

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select Balance</td>
<td>1</td>
</tr>
<tr>
<td>From Account</td>
<td></td>
</tr>
<tr>
<td>Where Account_Id = :a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Select Balance</td>
<td>3</td>
</tr>
<tr>
<td>From Account</td>
<td></td>
</tr>
<tr>
<td>Where Account_Id = :a</td>
<td></td>
</tr>
<tr>
<td>Update Account</td>
<td>4</td>
</tr>
<tr>
<td>Set Balance = :b</td>
<td></td>
</tr>
<tr>
<td>Where Account_Id = :a</td>
<td></td>
</tr>
<tr>
<td>Update Account</td>
<td>5</td>
</tr>
<tr>
<td>Set Balance = :b</td>
<td></td>
</tr>
<tr>
<td>Where Account_Id = :a</td>
<td></td>
</tr>
<tr>
<td>Update Account</td>
<td>6</td>
</tr>
<tr>
<td>Set Balance = :b</td>
<td></td>
</tr>
<tr>
<td>Where Account_Id = :a</td>
<td></td>
</tr>
</tbody>
</table>

Observation: concurrency or parallelism may cause inconsistencies, requires concurrency control for "isolation"
Funds Transfer: Inconsistent DATA

```c
void main () {
    /* read user input */
    scanf("%d %d %d", &srcid, &tgtid, &amount);
    EXEC SQL Update Account
        Set Balance = Balance - :amount Where AccId = :srcid;
    EXEC SQL Update Account
        Set Balance = Balance + :amount Where AccId = :tgtid;
    EXEC SQL Commit Work;
}
```

Observation: failures may cause inconsistencies, require recovery for "atomicity" and "durability."

Database Correctness

Database Fundamentals

Dirty Agrawal
Department of Computer Science
UC Santa Barbara

Reminder: Database System Layers
+ Basic Ingredients
- Elementary Operations (read and write)
- Transactions (i.e., transaction program executions)
- Execution histories
- Characterization of correct executions
- Protocols (i.e., online algorithms to ensure correctness)

+ Transaction Page Model: Syntax

Page Model of Transaction:
A transaction T is a partial order of steps (actions) of the form r[x] or w[x], where x ∈ D and reads and writes as well as multiple writes applied to the same object are ordered.
We write T = (op, <) for transaction T with step set op and partial order <.

Example: r[x] w[x] r[y] w[y]

+ Transaction Page Model: Semantics

Interpretation of jth step, pj, of T:
If pj = r[x], then interpretation is assignment vj := x to local variable vj

If pj = w[x] then interpretation is assignment x := f(vj1, ..., vjk) with unknown function f, and j1, ..., jk denoting T's prior read steps.
+ Lost Update Problem

<table>
<thead>
<tr>
<th>P1 Time</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r (x)</td>
<td>1/5</td>
</tr>
<tr>
<td>x := x + 100</td>
<td>1</td>
</tr>
<tr>
<td>w (x)</td>
<td>5/4</td>
</tr>
<tr>
<td>r (x)</td>
<td>9/5</td>
</tr>
<tr>
<td>x := x + 200</td>
<td>5</td>
</tr>
<tr>
<td>x := x + 100</td>
<td>5</td>
</tr>
</tbody>
</table>

update “lost”

Observation: problem is the interleaving r(x) r(x) w(x) w(x)

+ Dirty Read Problem

<table>
<thead>
<tr>
<th>P1 Time</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r (x)</td>
<td>1/5</td>
</tr>
<tr>
<td>x := x + 100</td>
<td>1</td>
</tr>
<tr>
<td>w (x)</td>
<td>5/4</td>
</tr>
<tr>
<td>x := x - 100</td>
<td>9/5</td>
</tr>
<tr>
<td>failure &amp; rollback</td>
<td>5</td>
</tr>
</tbody>
</table>

cannot rely on validity of previously read data

Observation: transaction rollbacks could affect concurrent transactions

+ Correctness Requirements: ACID

- ATOMICITY:
  - All-or-none property of user programs

- CONSISTENCY
  - User program is a consistent unit of execution

- ISOLATION
  - User programs are isolated with the side-effects of other user programs

- DURABILITY:
  - Effects of user programs are persistent forever
Transactions Executions: History

- History:
  - Contains all operations from all transactions
  - Distinct termination for every transaction
  - Preserves the order of operations of all transactions
  - Termination is the final step
  - Conflicting operations are ordered

Notion of Transaction Histories

- Goal:
  - A technique/algorithm/scheduler that prevents incorrect or bad execution.
  - Develop the notion of correctness – or characterize what does correct execution means.
  - This characterization will be based on the histories of transaction execution:

  ![Good vs Bad Histories](image)

Transaction Executions: Histories

Let $T = \{T_1, \ldots, T_n\}$ be a set of transactions, where each $T_i \in T$ has the form $T_i = \langle \text{op}_i, <_i \rangle$.

A history for $T$ is $H = (\text{op}(H), \leq_h)$ such that:
1. $\text{op}(s) \subseteq \bigcup_{i=1}^{n} \text{op}_i \cup \bigcup_{i=1}^{n} \{a, c\}$
2. for all $i, 1 \leq i \leq n$:
   - $\text{op}(s) \subseteq \{a, c\}$
   - $\bigcup_{i=1}^{n} <_i \subseteq \leq_h$
3. for all $i, 1 \leq i \leq n$:
   - $\bigcup_{i=1}^{n} <_i \subseteq \leq_h$
4. for all $i, 1 \leq i \leq n$:
   - $\bigcup_{i=1}^{n} <_i \subseteq \leq_h$
5. for all $p, q \in \text{op}(s)$ s.t. at least one of them is a write and both access the same data item: $p \leq_h \langle q, a \rangle$ or $p \leq_h \langle a, q \rangle$.
**History Example**

1. \( R1[x] \) writes \( w1[x] \) to \( c1 \).
2. \( R1[y] \) writes \( w2[y] \) to \( c2 \).
3. \( R2[y] \) writes \( w3[y] \) to \( c3 \).
4. \( R3[x] \) writes \( w3[x] \) to \( c3 \).

**Correctness**

- Syntactical semantics for schedules based on an intuitive notion:
  - Each transaction is a correct mapping, i.e.,

    - DB consistent
    - Transaction \( T \) consistent

  Hence, serial execution of transactions will be correct.

**Serial History**

- A history \( H \) is serial if for any two transactions \( T_i \) and \( T_j \) in \( H \), all operations of \( T_i \) are ordered in \( H \) before all operations of \( T_j \) or vice-versa.
General Idea

- Notion of equivalence of two histories $H_1$ and $H_2$.

- Use this notion of equivalence to accept all histories which are “equivalent” to some serial history as being correct.

- How to establish this equivalence notion?

Semantics

- Equivalence via a notion of semantics:
  - We do not know the semantics of transaction programs

  - We need a general notion that can capture all potential transaction semantics

  - Need a general enough and powerful notion that can capture all possible semantics of transactions.