Concurrency and Recovery

- DBMS should enable multiple clients to access the database concurrently
  - This can lead to problems with correctness of data because of interleaving of operations from different clients
  - System should ensure correctness (concurrency control)

Integrity or correctness of data

- Would like data to be “accurate” or “correct” at all times

<table>
<thead>
<tr>
<th>EMP</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
<td></td>
</tr>
<tr>
<td>Gray</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - x $\rightarrow$ y holds in R
  - Domain(x) = {Red, Blue, Green}
  - $\alpha$ is valid index for attribute x of R
  - no employee should make more than twice the average salary

Definition:

- Consistent state: satisfies all constraints
- Consistent DB: DB in consistent state
Constraints (as we use here) may not capture “full correctness”

Example 1  Transaction constraints
• When salary is updated, new salary > old salary
• When account record is deleted, balance = 0

Note: could be “emulated” by simple constraints, e.g.,
account  

Constraints (as we use here) may not capture “full correctness”

Example 2  Database should reflect real world

Observation: DB cannot be consistent always!
Example: \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)
Deposit $100 in \( a_2 \): \( a_2 \leftarrow a_2 + 100 \)
\( TOT \leftarrow TOT + 100 \)

Transactions
• Transaction: Sequence of operations executed by one concurrent client that preserve consistency
Transaction: collection of actions that preserve consistency

Big assumption:
If T starts with consistent state + T executes in isolation ⇒ T leaves consistent state

Correctness (informally)
- If we stop running transactions, DB left consistent
- Each transaction sees a consistent DB

Transactions - ACID
- Atomicity
  - Either all or no commands of transaction are executed (their changes are persisted in the DB)
- Consistency
  - After transaction DB is consistent (if before consistent)
- Isolation
  - Transactions are running isolated from each other
- Durability
  - Modifications of transactions are never lost

How can constraints be violated?
- Transaction bug
- DBMS bug
- Hardware failure
e.g., disk crash alters balance of account
- Data sharing
e.g.: T1: give 10% raise to programmers
  T2: change programmers ⇒ systems analysts

How can we prevent/fix violations?
- Part 13 (Recovery):
  - due to failures
- Part 14 (Concurrency Control):
  - due to data sharing
Will not consider:
- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair
  That is, solutions studied here do not need to know constraints

Data Items:
- **Data Item / Database Object / ...**
- Abstraction that will come in handy when talking about concurrency control and recovery
- Data Item could be
  - Table, Row, Page, Attribute value

Operations:
- Input (x): block containing x → memory
- Output (x): block containing x → disk
- Read (x,t): do input(x) if necessary
  \[ t \leftarrow \text{value of } x \text{ in block} \]
- Write (x,t): do input(x) if necessary
  \[ \text{value of } x \text{ in block} \leftarrow t \]

Key problem: Unfinished transaction (Atomicity)
Example: Constraint: A=B
\[ T_1: \ A \leftarrow A \times 2 \]
\[ B \leftarrow B \times 2 \]

\[ \begin{array}{ll}
  \text{A: 8} & \text{B: 8} \\
  \end{array} \]

memory disk
Transactions in SQL

- **BEGIN WORK**
  - Start new transaction
  - Often implicit

- **COMMIT**
  - Finish and make all modifications of transactions persistent

- **ABORT/Rollback**
  - Finish and undo all changes of transaction

**Example**

```sql
BEGIN WORK;
UPDATE accounts
SET bal = bal + 40
WHERE acc = 10;
COMMIT;

UPDATE accounts
SET bal = bal - 40
WHERE acc = 9;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal * 1.05;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal * 1.05
WHERE acc = 10;
COMMIT;
```

**Example**

Bank customer transfers money from account 9 to account 10

Bank adds interest to all accounts
BEGIN WORK;
UPDATE accounts
SET bal = bal + 40
WHERE acc = 10;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal - 40
WHERE acc = 9;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal * 1.05;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal + 40
WHERE acc = 10;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal - 40
WHERE acc = 9;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal + 40
WHERE acc = 10;
COMMIT;

BEGIN WORK;
UPDATE accounts
SET bal = bal * 1.05;
COMMIT;

Modeling Transactions and their Interleaving

• Transaction is sequence of operations
  - read: \( r_i(x) = \) transaction \( i \) read item \( x \)
  - write: \( w_i(x) = \) transaction \( i \) wrote item \( x \)
  - commit: \( c_i = \) transaction \( i \) committed
  - abort: \( a_i = \) transaction \( i \) aborted

Schedules

• A schedule \( S \) for a set of transactions \( T = \{T_1, \ldots, T_n\} \) is an partial order over operations of \( T \) so that
  - \( S \) contains a prefix of the operations of each \( T_i \)
  - Operations of \( T_i \) appear in the same order in \( S \) as in \( T_i \)
  - For any two conflicting operations they are ordered

Note

• For simplicity: We often assume that the schedule is a total order
How to model execution order?

- Schedules model the order of the execution for operations of a set of transactions

Conflicting Operations

- Two operations are conflicting if
  - At least one of them is a write
  - Both are accessing the same data item

Intuition

- The order of execution for conflicting operations can influence result!

Conflicting Operations

- Examples
  - $w_1(X), r_2(X)$ are conflicting
  - $w_1(X), w_2(Y)$ are not conflicting
  - $r_1(X), r_2(X)$ are not conflicting
  - $w_1(X), w_1(X)$ are not conflicting

Complete Schedules = History

- A schedule $S$ for $T$ is complete if it contains all operations from each transaction in $T$

- We will call complete schedules histories

Complete Schedule

\[
T_1 = r_1(a_1), w_1(a_2), r_1(a_3), w_1(a_4), c_1
\]

\[
T_2 = r_1(a_2), w_1(a_3), r_1(a_4), w_1(a_1), c_1
\]

Incomplete Schedule

\[
S = r_1(a_3), r_1(a_5), w_1(a_6)
\]

Not a Schedule

\[
S = r_1(a_1), r_1(a_3), c_1
\]

Conflicting operations

- Conflicting operations $w_1(a_3)$ and $w_2(a_3)$
- Order of these operations determines value of $a_3$
- $S_1$ and $S_2$ do not generate the same result

\[
S_1 = w_1(a_3) \rightarrow w_2(a_3)
\]

\[
S_2 = w_2(a_3) \rightarrow w_1(a_3)
\]
Why Schedules?

- Study properties of different execution orders
  - Easy/Possible to recover after failure
  - Isolation
  - -> preserve ACID properties
- Classes of schedules and protocols to guarantee that only “good” schedules are produced