CS 525: Advanced Database Organization

12: Transaction Management

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab
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)
Concurrency and Recovery

- DBMS should enable reestablish correctness of data in the presence of failures
  - System should restore a correct state after failure (recovery)
Integrity or correctness of data

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>52</td>
</tr>
<tr>
<td>Green</td>
<td>3421</td>
</tr>
<tr>
<td>Gray</td>
<td>1</td>
</tr>
</tbody>
</table>
Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - x → y holds in R
  - Domain(x) = {Red, Blue, Green}
  - α 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

| Acct # | .... | balance | deleted? |
Constraints (as we use here) may not capture “full correctness”

Example 2 Database should reflect real world
in any case, continue with constraints...

Observation: DB cannot be consistent always!

Example: \( a_1 + a_2 + \ldots + a_n = TOT \) (constraint)

Deposit $100 in \( a_2 \):

\[
\begin{align*}
    a_2 & \leftarrow a_2 + 100 \\
    TOT & \leftarrow TOT + 100
\end{align*}
\]
**Example:** \( a_1 + a_2 + \ldots + a_n = \text{TOT (constraint)} \)

Deposit $100 in \( a_2 \):

\[
\begin{align*}
\text{a}_2 & \quad \leftarrow \quad a_2 + 100 \\
\text{TOT} & \quad \leftarrow \quad \text{TOT} + 100
\end{align*}
\]

\[
\begin{array}{c|c|c}
\text{a}_2 & \text{TOT} & \text{TOT} + 100 \\
\hline
50 & 1000 & 1100
\end{array}
\]
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
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: $A=B$

\[
T_1: A \leftarrow A \times 2 \\
B \leftarrow B \times 2
\]
\[ T_1: \begin{align*}
&\text{Read } (A,t); \quad t \leftarrow t \times 2 \\
&\text{Write } (A,t); \\
&\text{Read } (B,t); \quad t \leftarrow t \times 2 \\
&\text{Write } (B,t); \\
&\text{Output } (A); \\
&\text{Output } (B); 
\end{align*} \]
T₁: Read (A,t); \( t \leftarrow t \times 2 \)
Write (A,t);
Read (B,t); \( t \leftarrow t \times 2 \)
Write (B,t);
Output (A);
Output (B);

A: 8  16
B: 8  16

memory

disk
\[ T_1: \] Read \((A,t)\); \( t \leftarrow t \times 2 \)
Write \((A,t)\);
Read \((B,t)\); \( t \leftarrow t \times 2 \)
Write \((B,t)\);
Output \((A)\);
Output \((B)\);

failure!

\begin{align*}
\text{memory} & \quad \text{disk} \\
A: & 8 \quad 16 \\
B: & 8 \quad 16 \\
\end{align*}
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
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
  WHERE acc = 10;

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

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

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

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

    COMMIT;

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

    COMMIT;

Bank customer transfers money from account 9 to account 10.
BEGIN WORK;
  UPDATE accounts
  SET bal = bal + 40
  WHERE acc = 10;

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

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

Bank adds interest to all accounts

Example

time

BEGIN WORK;

CS 525
COMPUTER SCIENCE

Notes 12 - Transaction Management
BEGIN WORK;
    UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;

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

    COMMIT;

Potential Problems:
1. Transactions are interrupted
   • No reduction in bal of acc 9
   • Only some accounts got interest
2. Interleaving of Transaction
   • Acc 9 too much interest (before 40 has been deducted)

SET bal = bal * 1.05;
COMMIT;
Modeling Transactions and their Interleaving

• Transaction is sequence of operations
  – **read**: $r_i(x) = \text{transaction } i \text{ read item } x$
  – **write**: $w_i(x) = \text{transaction } i \text{ wrote item } x$
  – **commit**: $c_i = \text{transaction } i \text{ committed}$
  – **abort**: $a_i = \text{transaction } i \text{ aborted}$
BEGIN WORK;
    UPDATE accounts
        SET bal = bal + 40
    WHERE acc = 10;

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

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

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

Assume we have accounts: a₁, a₂, a₉, a₁₀

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

T₁=r₁(a₁₀), w₁(a₁₀), r₁(a₉), w₁(a₉), c₁

T₂=r₂(a₁), w₂(a₁), r₂(a₂), w₂(a₂), r₂(a₉), w₂(a₉), r₂(a₁₀), w₂(a₁₀), c₁
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**
$T_1 = r_1(a_{10}), w_1(a_{10}), r_1(a_9), w_1(a_9), c_1$

$T_2 = r_2(a_1), w_2(a_1), r_2(a_2), w_2(a_2), r_2(a_9), w_2(a_9), r_2(a_{10}), w_2(a_{10}), c_1$

**Complete Schedule**

$S = r_2(a_1), r_1(a_{10}), w_2(a_1), r_2(a_2), w_1(a_{10}), w_2(a_2), r_2(a_9), w_2(a_9), r_1(a_9), w_1(a_9), c_1, r_2(a_{10}), w_2(a_{10}), c_1$

**Incomplete Schedule**

$S = r_2(a_1), r_1(a_{10}), w_2(a_1), w_1(a_{10})$

**Not a Schedule**

$S = r_2(a_1), r_1(a_{10}), c_1$
Conflicting operations

- Conflicting operations \( w_1(a_{10}) \) and \( w_2(a_{10}) \)
- Order of these operations determines value of \( a_{10} \)
- \( S_1 \) and \( S_2 \) do not generate the same result

\[ T_1 = r_1(a_{10}), w_1(a_{10}), r_1(a_9), w_1(a_9), c_1 \]

\[ T_2 = r_2(a_1), w_2(a_1), r_2(a_2), w_2(a_2), r_2(a_9), w_2(a_9), r_2(a_{10}), w_2(a_{10}), c_1 \]

\[ S_1 = \ldots w_2(a_1) \ldots w_1(a_{10}) \]

\[ S_2 = \ldots w_1(a_1) \ldots w_2(a_{10}) \]
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