Overview

- Transactions
- Concurrency Control
- Locking
- Transactions in SQL
Problems

- System crashes during transaction
  - database remains in inconsistent (intermediate) state
  - solution: recovery (next lecture)
- Multiple transactions executed at same time
  - other applications have access to inconsistent (intermediate) state
  - solution: concurrency control (this lecture)
  - Example: 10 parallel clients use the same server

A Sample Transaction

1: Begin_Transaction
2: get (K1, K2, CHF) from terminal
3: Select BALANCE Into S1 From ACCOUNT Where ACCOUNTNR = K1;
4: S1 := S1 - CHF;
5: Update ACCOUNT Set BALANCE = S1 Where ACCOUNTNR = K1;
6: Select BALANCE Into S2 From ACCOUNT Where ACCOUNTNR = K2;
7: S2 := S2 + CHF;
8: Update ACCOUNT Set BALANCE = S2 Where ACCOUNTNR = K2;
9: Insert Into BOOKING(ACCOUNTNR, DATE, AMOUNT, TEXT)
   Values (K1, today, -CHF, 'Transfer');
10: Insert Into BOOKING(ACCOUNTNR, DATE, AMOUNT, TEXT)
    Values (K2, today, CHF, 'Transfer');
12: If S1<0 Then Abort_Transaction
11: End_Transaction

Transaction = Program that takes database from one consistent state to another consistent state
Transactions

- A *transaction* = sequence of statements (operation) that either all succeed, or all fail (read, write operations)
- Transactions have the ACID properties:
  - A = atomicity
  - C = consistency
  - I = isolation
  - D = durability
ACID

- **Atomicity**: the operation sequence is either executed completely or not at all
- **Consistency**: the operation sequence takes the database from any consistent state to another consistent state (with respect to integrity constraints)
- **Isolation**: intermediate states of transactions are not visible to other transactions (equivalence to single user mode)
- **Durability**: effects of completed transactions are not lost due to hardware or software failures

Transaction Management

- Isolation (+ Consistency) => **Concurrency Control**
  - Concurrent transaction should appear as if they were executed serially (i.e. in sequence)
  - Performance problems?
- Atomicity + Durability => Recovery
Model for Transactions

• Assumption: the database is composed of *elements*
  – Usually 1 element = 1 block
  – Can be smaller (=1 record) or larger (=1 relation)

• Assumption: each transaction reads/writes some elements

Concurrency Control

• Interleaving the operations of different transactions can lead to anomalies

• Canonical problems
  – Lost Update
  – Dirty Read
  – Unrepeatable Read
# Lost Update

T1, T2: deposit on account ‘acc1’

<table>
<thead>
<tr>
<th>Transactions</th>
<th>T2:</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(acc1)</td>
<td></td>
<td>acc1</td>
</tr>
<tr>
<td>acc1 := acc1 + 10</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>write(acc1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Changes of T2 are lost
  "Schedule"

R₁(acc₁) R₂(acc₁) W₂(acc₁) W₁(acc₁)  11

# Dirty Read

T1: two deposits on account ‘acc1’, T2: sum of all accounts

<table>
<thead>
<tr>
<th>Transactions</th>
<th>T2:</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(acc1)</td>
<td></td>
<td>acc1</td>
</tr>
<tr>
<td>acc1 := acc1 + 10</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>write(acc1)</td>
<td></td>
<td>sum</td>
</tr>
<tr>
<td>write(sum)</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>commit</td>
<td></td>
<td>40</td>
</tr>
</tbody>
</table>

- T2 sees dirty data of T1
  "Schedule"

R₁(acc₁) W₁(acc₁) R₂(acc₁) W₂(sum) W₁(acc₁)  12
Unrepeatable Read

T1: multiple read from account ‘acc1’, T2: deposit account ‘acc1’

<table>
<thead>
<tr>
<th>Transactions</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1:</td>
<td></td>
</tr>
<tr>
<td>read(acc1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>T2:</td>
<td></td>
</tr>
<tr>
<td>read(acc1)</td>
<td>20</td>
</tr>
<tr>
<td>acc1 := acc1 + 20</td>
<td></td>
</tr>
<tr>
<td>write(acc1)</td>
<td>40</td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>read(acc1)</td>
<td></td>
</tr>
<tr>
<td>sum := sum + acc1</td>
<td></td>
</tr>
<tr>
<td>write(sum)</td>
<td>40</td>
</tr>
<tr>
<td>commit</td>
<td></td>
</tr>
</tbody>
</table>

• T1 reads different values for acc1

"Schedule"

Schedules

• Schedule = an interleaving of actions (read/write) from a set of transactions, where the actions of any single transaction are in the original order

• Complete Schedule = add commit or abort at end
Complete Schedule

Transactions
T1: T2:
read(acc1)
   read(acc1)
   acc1 := acc1 + 20
   write(acc1)
   commit
read(acc1)
sum := sum + acc1
write(sum)
commit

Schedule
read1(acc1)
read2(acc1)
write2(acc1)
commit2
read1(acc1)
write1(sum)
commit1

Initial State of DB + Schedule → Final State of DB

Serial Schedule

• One transaction at a time, no interleaving

T1: T2:
read(acc1)
   read(acc1)
   acc1 := acc1 + 20
   write(acc1)
   commit
read(acc1)
read(acc1)
sum := sum + acc1
write(sum)
commit

• Final state is consistent (if transactions are, too)
• Different serial schedules give different final states
Serializable Schedule

- Schedule with interleaved transactions that produces the same result as some serial schedule
- "Good" schedules
- Canonical problems before were non-serializable schedules

Checking Serializability

- Idea: which actions can be swapped in a schedule?
- The following cannot be swapped without changing the result (conflict)
  - Actions within the same transaction
  - Actions in different transactions on the same object if at least one action is a write operation
- Try to transform into serial schedule by swapping: then serializable
Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r_1(a)</td>
<td>r_2(a)</td>
</tr>
<tr>
<td>w_1(a)</td>
<td>r_2(a)</td>
</tr>
<tr>
<td>r_1(b)</td>
<td>r_2(b)</td>
</tr>
</tbody>
</table>

Can we find a serial schedule?

More definitions

- A schedule of a set of transactions is **serializable** if it is equivalent to a **serial schedule**
- Transactions in a serial schedule are **isolated**
Performance difference

• Serial schedule

• Serializable schedule

  – What is the main difference?
  – Why does a DBMS aim for serializability?

Conflicts

• Conflicting actions: pairs of actions on same object from different transactions where at least one is write

• Two schedules are conflict-equivalent if they have the same conflicts

• A schedule is conflict-serializable if it is conflict-equivalent to a serial schedule
Example

same conflicts, thus conflict-equivalent

serial schedule

Serializability Graph

- **Node** for each transaction $T_i$
- **Edge** from $T_i$ to $T_j$ if there is an action of $T_i$ that precedes and “conflicts” with an action of $T_j$
- Theorem: A schedule is conflict serializable iff its Serializability Graph is acyclic.
Example

Checking Serializability

- **optimistic**: validate serializability after transaction is executed using the serializability graph, otherwise abort transactions
  - possibly many aborts
- **pessimistic**: make sure that never a non-serializable schedule occurs while transaction is executed
  - locking
Locking

- Transactions obtain locks on objects x
  - S-locks (shared) for read: slock(x)
  - X-locks (exclusive) for write: xlock(x)

<table>
<thead>
<tr>
<th>lock requested</th>
<th>-</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock held</td>
<td></td>
<td>Ok</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>Ok</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>Ok</td>
<td></td>
</tr>
</tbody>
</table>

compatibility of locks

2Phase Locking (2PL)

- Before accessing an object, a lock is acquired
- Locks of concurrent transactions must be compatible
- A transaction can acquire only one lock per object
- At end of transaction all locks have to be released
- Locks can be released only if no further locks are required
2PL

- Phase 1: get lock
- Phase 2: release lock

2PL

- Theorem: 2PL ensures that the serializability graph of the schedule is acyclic
  - Guarantees conflict serializability
Strict 2PL

- Hold all locks until end of transaction
  - avoids "domino effect": T1 releases locks, T2 reads released objects, T1 aborts
  - when are no more locks required?
  - required for recovery (see next week)

Dining Philosophers Problem
Deadlocks

- 2PL can lead to deadlocks
  - Different transactions wait for each other to release locks
- Represent the waiting relationship as waiting graph
  - Directed edge from $T_i$ to $T_j$ if $T_i$ waits for $T_j$

Resolving Deadlocks

- 2PL cannot avoid deadlocks
- If the waiting graph contains cycles
  - abort one of the transactions (e.g. younger one)
The Phantom Problem

• T1 locks all pages containing professor records in faculty I&C, and finds oldest (say, age=59).
• T2 inserts a new professor; faculty I&C, age=65.
• T2 deletes oldest professor in faculty STI (say, age=73), and commits.
• T1 now locks all pages containing professors in faculty STI, and finds oldest (say, age=61)

Analysis of Phantom Problem

• Schedule is not serial!
• Problem: T1 assumes it has locked ALL professors in faculty I&C
  – only true if no new ones are inserted
  – 2PL applied to data objects does not work
• Solution
  – choose the right locks (e.g. use a predicate)
  – not a problem with 2PL per se
4 Isolation Levels

• **Read Uncommitted**
  – Can see uncommitted changes of other transactions
  – Dirty Read, Unrepeatable Read
  – Recommended only for statistical functions

• **Read Committed**
  – Can see committed changes of other transactions
  – No Dirty read, but unrepeatable read possible
  – Acceptable for query/decision-support

• **Repeatable Read**
  – No dirty or unrepeatable read
  – May exhibit phantom phenomenon

• **Serializable**

---

4 Different Isolation Levels

<table>
<thead>
<tr>
<th></th>
<th>Dirty Read</th>
<th>Nonrepeatable Read</th>
<th>Phantom Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Uncommitted</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Not possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Not possible</td>
<td>Not possible</td>
<td>-Possible (but unlikely)</td>
</tr>
<tr>
<td>Serializable</td>
<td>Not possible</td>
<td>Not possible</td>
<td>Not possible</td>
</tr>
</tbody>
</table>
Implementation of Isolation Levels

- Read Uncommitted
  - no S-locks
- Read Committed
  - S-locks can be released anytime
  - X-locks strict 2PL
- Repeatable Read
  - strict 2PL on all data
- Serializable
  - strict 2PL on all data and indices

Transactions in SQL 92

- Start Transaction
  - No explicit statement (though START TRANSACTION can be used)
  - Implicitly started by a SQL statement
- End Transaction
  - By COMMIT or ROLLBACK
  - Automatically with AUTOCOMMIT when SQL statement completed
### MySQL Examples

```sql
start transaction;
update account set balance=balance-1000 where number=2;
update account set balance=balance+1000 where number=1;
commit;

lock tables account write;
select balance from account where number = 2;
update account set balance = 1500 where number = 2;
unlock tables;
```

### Setting Properties of Transactions

```
SET TRANSACTION
  [READ ONLY | READ WRITE]
ISOLATION LEVEL
  [READ UNCOMMITTED | Serializable | REPEATABLE READ | READ COMMITTED]
```
Summary

• Isolation is required to achieve concurrency control
• Locks are often used to avoid problems