Lecture 4: Transactions

Wednesday, October 20, 2010

Homework 3

The key concepts here:

- Connect to db and call SQL from java
- Dependent joins
- Integrate two databases
- Transactions

Amount of work:

- 20 SQL queries+180 lines Java ≈ 12 hours (?)

Review Questions

Query Answering Using Views, by Halevy

- Q1: define the problem
- Q2: how is this used for physical data independence ?
- Q3: what is *data integration* and what is its connection to query answering using views ?

Outline

- Transaction basics
- Recovery
- Concurrency control

Reading Material for Lectures 4 & 5

From the main textbook (Ramakrishnan and Gehrke):

- Chapters 16, 17, 18

From the second textbook (Garcia-Molina, Ullman, Widom):

- Chapters 17.2, 17.3, 17.4
- Chapters 18.1, 18.2, 18.3, 18.8, 18.9

Transactions

- The problem: An application must perform *several* writes and reads to the database, as a unity
- Solution: multiple actions of the application are bundled into one unit called *Transaction*

Turing Awards to Database Researchers

- Charles Bachman 1973 for CODASYL
- Edgar Codd 1981 for relational databases
- Jim Gray 1998 for transactions

The World Without Transactions

- Write to files to ensure durability
- Rely on operating systems for scheduling, and for concurrency control

- What can go wrong ?
  - System crashes
  - Anomalies (three are famous)

Crashes

Client 1:

```
UPDATE Accounts
SET balance= balance - 500
WHERE name= 'Fred'

UPDATE Accounts
SET balance = balance + 500
WHERE name= 'Joe'
```

Crash !

What's wrong ?

1st Famous Anomaly: Lost Updates

Client 1:	Client 2:
UPDATE Customer SET rentals= rentals + 1 WHERE cname= 'Fred'	UPDATE Customer SET rentals= rentals + 1 WHERE cname= 'Fred'

Two people attempt to rent two movies for Fred, from two different terminals. What happens ?

2^{nd} Famous Anomaly: Inconsistent Read

Client 1: move from gizmo→gadget

```
UPDATE Products
SET quantity = quantity + 5
WHERE product = 'gizmo'


UPDATE Products
SET quantity = quantity - 5
WHERE product = 'gadget'
```

Client 2: inventory….

```
SELECT sum(quantity)
FROM Product
```

3rd Famous Anomaly: Dirty Reads

Client 1: transfer $100 acc1→ acc2

```
X = Account1.balance
Account2.balance += 100

If (X>=100) Account1.balance −=100
else { /* rollback ! */
        account2.balance −= 100
        println("Denied !")
```

Client 2: transfer $100 acc2 → acc3

```
Y = Account2.balance
Account3.balance += 100

If (Y>=100) Account2.balance −=100
else { /* rollback ! */
        account3.balance −= 100
        println("Denied !")
```

What's wrong ?

The Three Famous anomalies

- Lost update
  - Two tasks T and T' both modify the same data
  - T and T' both commit
  - Final state shows effects of only T, but not of T'
- Dirty read
  - T reads data written by T' while T' has not committed
  - What can go wrong: T' write more data (which T has already read), or T' aborts
- Inconsistent read
  - One task T sees some but not all changes made by T'

Transactions: Definition

- **A transaction** = one or more operations, which reflects a single real-world transition
  – Happens completely or not at all; all-or-nothing
- Examples
  – Transfer money between accounts
  – Rent a movie; return a rented movie
  – Purchase a group of products
  – Register for a class (either waitlisted or allocated)
- By using transactions, all previous problems disappear

Transactions in Applications

START TRANSACTION

[SQL statements]

COMMIT or ROLLBACK (=ABORT)

May be omitted: first SQL query starts txn

In ad-hoc SQL: each statement = one transaction

Revised Code

```
Client 1: transfer $100  acc1→ acc2
START TRANSACTION
X = Account1.balance;    Account2.balance += 100

If (X>=100) { Account1.balance -=100;  COMMIT }
else {println("Denied !"; ROLLBACK)
```

```
Client 1: transfer $100  acc2→ acc3
START TRANSACTION
X = Account2.balance;    Account3.balance += 100

If (X>=100) { Account2.balance -=100;  COMMIT }
else {println("Denied !"; ROLLBACK)
```

ACID Properties

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID: Atomicity

- Two possible outcomes for a transaction
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

- That is, transaction’s activities are all or nothing

ACID: Isolation

- A transaction executes concurrently with other transaction
- Isolation: the effect is as if each transaction executes in isolation of the others

ACID: Consistency

- The database satisfies integrity constraints
  - Account numbers are unique
  - Stock amount can't be negative
  - Sum of *debits* and of *credits* is 0
- Consistency = if the database satisfied the constraints at the beginning of the transaction, and if the application is written correctly, then the constraints must hold at the end of the transactions
- Introduced as a requirement in the 70s, but today we understand it is a consequence of atomicity and isolation

ACID: Durability

- The effect of a transaction must continue to exists after the transaction, or the whole program has terminated
- Means: write data to disk
- Sometimes also means recovery

Reasons for Rollback

- Explicit in the application
  - E.g. use it freely in HW 3
- System-initiated abort
  - System crash
  - Housekeeping, e.g. due to timeouts

Simple Log-based Recovery

These simple recovery algorithms are based on *Garcia-Molina, Ullman, Widom*

- Undo logging 17.2
- Redo logging 17.3
- Redo/undo 17.4

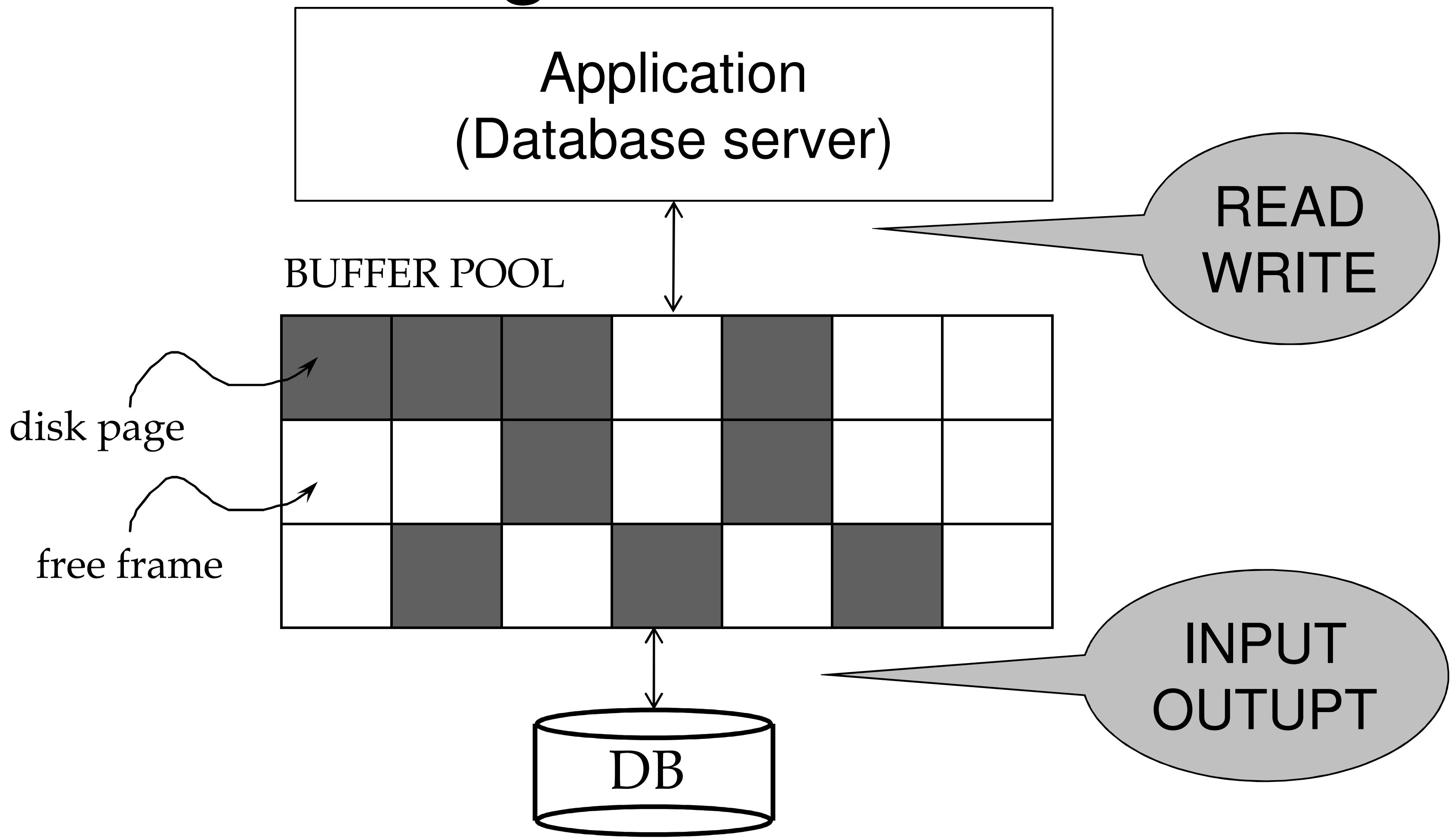
Disk Access Characteristics

- Disk latency = time between when command is issued and when data is in memory

- Disk latency = seek time + rotational latency
  - Seek time = time for the head to reach cylinder
    - 10ms – 40ms
  - Rotational latency = time for the sector to rotate
    - Rotation time = 10ms
    - Average latency = 10ms/2
- Transfer time = typically 40MB/s
- Disks read/write one block at a time

Large gap between disk I/O and memory ➔ Buffer pool

Buffer Management in a DBMS



Buffer manager maintains a table of <pageid, frame#> pairs

Page Replacement Policies

- LRU = expensive
  - Next slide
- Clock algorithm = cheaper alternative
  - Read in the book

Both work well in OS, but not always in DB

Least Recently Used (LRU)

P5, P2, P8, P4, P1, P9, P6, P3, P7

Read(P6)

P6, P5, P2, P8, P4, P1, P9, P3, P7

Read(P10)

Input(P10)

P10, P6, P5, P2, P8, P4, P1, P9, P3

Buffer Manager

DBMS build their own buffer manager and don’t rely on the OS

- Better control for transactions
  - Force pages to disk
  - Pin pages in the buffer
- Tweaks to LRU/clock algorithms for specialized accesses, s.a. sequential scan

Recovery

Type of Crash	Prevention
Wrong data entry	Constraints and Data cleaning
Disk crashes	Redundancy: e.g. RAID, archive
Fire, theft, bankruptcy…	Remote backups
System failures: e.g. power	DATABASE RECOVERY

Key Principle in Recovery

- Write-ahead log =
  - A file that records every single action of all running transactions
  - *Force* log entry to disk
  - After a crash, transaction manager reads the log and finds out exactly what the transactions did or did not

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

Primitive Operations of Transactions

- READ(X,t)
  - copy element X to transaction local variable t
- WRITE(X,t)
  - copy transaction local variable t to element X

- INPUT(X)
  - read element X to memory buffer
- OUTPUT(X)
  - write element X to disk

Example

```
START TRANSACTION
READ(A,t);
t := t*2;
WRITE(A,t);
READ(B,t);
t := t*2;
WRITE(B,t)
COMMIT;
```

Atomicity:
BOTH A and B
are multiplied by 2

READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t)

	Transaction	Buffer pool		Disk	
Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash occurs after OUTPUT(A), before OUTPUT(B)
We lose atomicity

Action	t	Mem A	Mem B	Disk A	Disk B
INPUT(A)		8		8	8
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		8	8
INPUT(B)	16	16	8	8	8
READ(B,t)	8	16	8	8	8
t:=t*2	16	16	8	8	8
WRITE(B,t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	[illegible]
OUTPUT(B)	16	16	16	16	[illegible]

Crash !

The Log

- An append-only file containing log records
- Multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that didn't commit
  - Undo other transactions that didn't commit
- Three kinds of logs: undo, redo, undo/redo

Undo Logging

Log records

- <START T>
  - transaction T has begun
- <COMMIT T>
  - T has committed
- <ABORT T>
  - T has aborted
- <T,X,v>
  - T has updated element X, and its *old* value was v

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	[illegible]	
COMMIT						<COMMIT T>

Crash !

WHAT DO WE DO ?

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						T>

Crash !

WHAT DO WE DO ?

After Crash

- In the first example:
  - We UNDO both changes: A=8, B=8
  - The transaction is atomic, since none of its actions has been executed

- In the second example
  - We don't undo anything
  - The transaction is atomic, since both it's actions have been executed

Undo-Logging Rules

U1: If T modifies X, then <T,X,v> must be written to disk before OUTPUT(X)

U2: If T commits, then OUTPUT(X) must be written to disk before <COMMIT T>

- Hence: OUTPUTs are done *early*, before the transaction commits

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
INPUT(A)		8		8	8	
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
INPUT(B)	16	16	8	8	8	
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
COMMIT						<COMMIT T>

Recovery with Undo Log

After system's crash, run recovery manager

- Idea 1. Decide for each transaction T whether it is completed or not
  - <START T>….<COMMIT T>…. = yes
  - <START T>….<ABORT T>……. = yes
  - <START T>……………………… = no
- Idea 2. Undo all modifications by incomplete transactions

Recovery with Undo Log

Recovery manager:

- Read log <u>from the end</u>; cases:

  <COMMIT T>: mark T as completed

  <ABORT T>: mark T as completed

  <T,X,v>: if T is not completed
  then write X=v to disk
  else ignore

  <START T>: ignore

Recovery with Undo Log

…
…
<T6,X6,v6>
…
…
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

Crash

Question1: Which updates are undone ?

Question 2:
What happens if there is a second crash, during recovery ?

Question 3:
How far back do we need to read in the log ?

Recovery with Undo Log

- Note: all undo commands are *idempotent*
  - If we perform them a second time, no harm is done
  - E.g. if there is a system crash during recovery, simply restart recovery from scratch

Recovery with Undo Log

When do we stop reading the log ?

- We cannot stop until we reach the beginning of the log file
- This is impractical

Instead: use checkpointing

Checkpointing

Checkpoint the database periodically

- Stop accepting new transactions
- Wait until all current transactions complete
- Flush log to disk
- Write a <CKPT> log record, flush
- Resume transactions

Undo Recovery with Checkpointing

During recovery,
Can stop at first
<CKPT>

…
…
<T9,X9,v9>
…
…
(all completed)
<CKPT>
<START T2>
<START T3
<START T5>
<START T4>
<T1,X1,v1>
<T5,X5,v5>
<T4,X4,v4>
<COMMIT T5>
<T3,X3,v3>
<T2,X2,v2>

other transactions

transactions T2,T3,T4,T5

Nonquiescent Checkpointing

- Problem with checkpointing: database freezes during checkpoint
- Would like to checkpoint while database is operational
- Idea: nonquiescent checkpointing

Quiescent = being quiet, still, or at rest; inactive
Non-quiescent = allowing transactions to be active

Nonquiescent Checkpointing

- Write a <START CKPT(T1,…,Tk)> where T1,…,Tk are all active transactions
- Continue normal operation
- When all of T1,…,Tk have completed, write <END CKPT>

Undo Recovery with Nonquiescent Checkpointing

During recovery, Can stop at first <CKPT>

…
…
…
…
…
…
<START CKPT T4, T5, T6>
…
…
…
…
<END CKPT>
…
…
…

earlier transactions plus T4, T5, T6

T4, T5, T6, plus later transactions

later transactions

Q: do we need <END CKPT> ?

Implementing ROLLBACK

- A transaction ends in COMMIT or ROLLBACK
- Use the undo-log to implement ROLLBCACK

- LSN = Log Seqence Number
- Log entries for the same transaction are linked, using the LSN's
- Read log in reverse, using LSN pointers

Redo Logging

Log records

- <START T> = transaction T has begun
- <COMMIT T> = T has committed
- <ABORT T>= T has aborted
- <T,X,v>= T has updated element X, and its *new* value is v

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Redo-Logging Rules

R1: If T modifies X, then both <T,X,v> and <COMMIT T> must be written to disk before OUTPUT(X)

- Hence: OUTPUTs are done *late*

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Recovery with Redo Log

After system's crash, run recovery manager

- Step 1. Decide for each transaction T whether we need to redo or not
  - <START T>….<COMMIT T>…. = yes
  - <START T>….<ABORT T>……. = no
  - <START T>……………………… = no
- Step 2. Read log from the beginning, redo all updates of *committed* transactions

Recovery with Redo Log

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
…
…

Nonquiescent Checkpointing

- Write a <START CKPT(T1,…,Tk)> where T1,…,Tk are all active transactions
- Flush to disk all blocks of committed transactions (*dirty blocks*), while continuing normal operation
- When all blocks have been flushed, write <END CKPT>

Note: this differs significantly from ARIES (next lecture)

Redo Recovery with Nonquiescent Checkpointing

Step 1: look for
The last
<END CKPT>

All OUTPUTs
of T1 are guaranteed
to be on disk

Cannot
use

…
<START T1>
…
<COMMIT T1>
…
<START T4>
…
<START CKPT T4, T5, T6>
…
…
…
…
<END CKPT>
…
…
…
<START CKPT T9, T10>
…

Step 2: redo
from the
earliest
start of
T4, T5, T6
ignoring
transactions
committed
earlier

Comparison Undo/Redo

- Undo logging:
  - OUTPUT must be done early
  - If <COMMIT T> is seen, T definitely has written all its data to disk (hence, don't need to redo) – inefficient
- Redo logging
  - OUTPUT must be done late
  - If <COMMIT T> is not seen, T definitely has not written any of its data to disk (hence there is not dirty data on disk, no need to undo) – inflexible
- Would like more flexibility on when to OUTPUT: undo/redo logging (next)

Undo/Redo Logging

Log records, only one change

- <T,X,u,v>= T has updated element X, its *old* value was u, and its *new* value is v

Undo/Redo-Logging Rule

UR1: If T modifies X, then <T,X,u,v> must be written to disk before OUTPUT(X)

Note: we are free to OUTPUT early or late relative to <COMMIT T>

Action	T	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
REAT(A,t)	8	8		8	8	
t:=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8,16>
READ(B,t)	8	16	8	8	8	
t:=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Can OUTPUT whenever we want: before/after COMMIT

Recovery with Undo/Redo Log

After system's crash, run recovery manager

- Redo all committed transaction, top-down
- Undo all uncommitted transactions, bottom-up

Recovery with Undo/Redo Log

<START T1>
<T1,X1,v1>
<START T2>
<T2, X2, v2>
<START T3>
<T1,X3,v3>
<COMMIT T2>
<T3,X4,v4>
<T1,X5,v5>
…
…

Concurrency Control

Problem:

- Many transactions execute concurrently
- Their updates to the database may interfere

Scheduler = needs to schedule transactions

Concurrency Control

Basic definitions

- Schedules: serializable and variations

Next lecture:

- Locks
- Concurrency control by timestamps 18.8
- Concurrency control by validation 18.9

The Problem

- Multiple concurrent transactions T_1, T_2, …
- They read/write common elements A_1, A_2, …
- How can we prevent unwanted interference ?

The SCHEDULER is responsible for that

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Lost Update

T_1: READ(A)

T_1: A := A+5

T_1: WRITE(A)

T_2: READ(A);

T_2: A := A*2

T_2: WRITE(A);

RW conflict and WW conflict

Inconsistent Reads

T_1: A := 20; B := 20;
T_1: WRITE(A)

T_2: READ(A);
T_2: READ(B);

T_1: WRITE(B)

WR conflict and RW conflict

Dirty Read

T_1: WRITE(A)

T_1: ABORT

T_2: READ(A)

WR conflict

Unrepeatable Read

T_1: WRITE(A)

T_2: READ(A);

T_2: READ(A);

RW conflict and WR conflict

Schedules

A *schedule* is a sequence of interleaved actions from all transactions

Example

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Serializable Schedule

A schedule is *serializable* if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is NOT a serial schedule, but is ***serializable***

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s + 200
	WRITE(A,s)
	READ(B,s)
	s := s + 200
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

Schedule is serializable because t=t+100 and s=s+200 commute

We don't expect the scheduler to schedule this

Ignoring Details

- Assume worst case updates:
  - We never commute actions done by transactions
- As a consequence, we only care about reads and writes
  - Tran [illegible] W(A)'s

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Conflicts

Two actions by same transaction T_i: $\boxed{r_i(X); w_i(Y)}$

Two writes by T_i, T_j to same element $\boxed{w_i(X); w_j(X)}$

Read/write by T_i, T_j to same element $\boxed{w_i(X); r_j(X)}$ $\boxed{r_i(X); w_j(X)}$

A "conflict" means: you can't swap the two operations

Conflict Serializability

- A schedule is *conflict serializable* if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$\Downarrow$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

The Precedence Graph Test

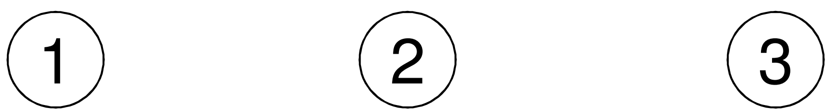
Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions T_i
- Edge from T_i to T_j if T_i makes an action that conflicts with one of T_j and comes first
- The test: if the graph has no cycles, then it is conflict serializable !

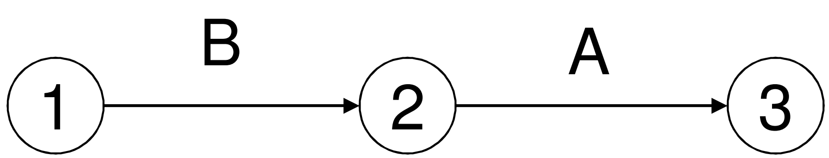
Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

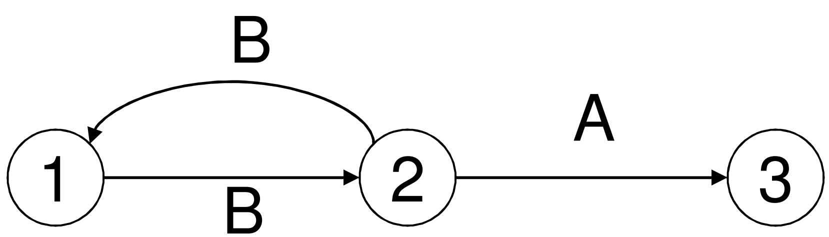
Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

1 2 3

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

View Equivalence

- A serializable schedule need not be conflict serializable, even under the “worst case update” assumption

$w_1(X); w_2(X); w_2(Y); w_1(Y); w_3(Y);$

Lost write

$w_1(X); w_1(Y); w_2(X); w_2(Y); w_3(Y);$

Equivalent, but can’t swap

View Equivalent

T1	T2	T3
W1(X)		
	W2(X)	
	W2(Y)	
	CO2	
W1(Y)		
CO1		
	Lost	W3(Y)
		CO3

T1	T2	T3
W1(X)		
W1(Y)		
CO1		
	W2(X)	
	W2(Y)	
	CO2	
		W3(Y)
		CO3

Serializable, but not conflict serializable

View Equivalence

Two schedules S, S' are *view equivalent* if:

- If T reads an initial value of A in S, then T also reads the initial value of A in S'
- If T reads a value of A written by T' in S, then T also reads a value of A written by T' in S'
- If T writes the final value of A in S, then it writes the final value of A in S'

View-Serializability

A schedule is *view serializable* if it is view equivalent to a serial schedule

Remark:

- If a schedule is *conflict serializable*, then it is also *view serializable*
- But not vice versa

Schedules with Aborted Transactions

- When a transaction aborts, the recovery manager undoes its updates
- But some of its updates may have affected other transactions !

Schedules with Aborted Transactions

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Cannot abort T1 because cannot undo T2

Recoverable Schedules

A schedule is *recoverable* if:

- It is conflict-serializable, and
- Whenever a transaction T commits, all transactions who have written elements read by T have already committed

Recoverable Schedules

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Nonrecoverable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
Abort	
	Commit

Recoverable

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T

- A schedule is said to *avoid cascading aborts* if whenever a transaction read an element, the transaction that has last written it has already committed.

Avoiding Cascading Aborts

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
. . .	
	. . .

With cascading aborts

T1	T2
R(A)	
W(A)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	. . .

Without cascading aborts

Review of Schedules

Serializability

- Serial
- Serializable
- Conflict serializable
- View serializable

Recoverability

- Recoverable
- Avoiding cascading deletes