CSEP544 Lecture 4: Transactions

Tuesday, April 21, 2009
HW 3

• Database 1 = IMDB on SQL Server

• Database 2 = you create a CUSTOMER db on postgres
  – Customers
  – Rentals
  – Plans
Overview

Today:
• Overview of transactions (R&G Chapter 16)
• Recovery from crashes (Ullman’s book, then R&G Chapter 18)

Next week
• Concurrency control
Transactions

• **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
Inconsistent Read

/* Client 1: move 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
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 ?
Example: Lost Update

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

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

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

• Dirty read
  – T reads data written by T’ while T’ has not committed
  – What can go wrong: T’ writes more or aborts
  – Inconsistent read: T sees only some changes by T’

• 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’

• Many other anomalies exists, with or without name
  – E.g. write skew
Protection against 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 ?
Definition of Transactions

• A transaction = one or more operations, which reflects a single real-world transition
  – Happens completely or not at all

• 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)

Default: each statement = one transaction

May be omitted: first SQL query starts txn
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)}
Using Transactions

Very easy to use:

• START TRANSACTION
• COMMIT
• ROLLBACK

What they mean:

• Popular culture: ACID
• Theory: serializability (next lecture)
ACID Properties

- **Atomicity**: Either all changes performed by transaction occur or none occurs.
- **Consistency**: A transaction as a whole does not violate integrity constraints.
- **Isolation**: Transactions appear to execute one after the other in sequence.
- **Durability**: If a transaction commits, its changes will survive failures.
What Could Go Wrong?

• **Concurrent** operations
  – Will discuss next time

• **Failures** can occur at any time
  – Will discuss today

• **Transactions** are intimately connected to the *buffer manager* (will discuss next)
The Mechanics of Disk

Mechanical characteristics:
- Rotation speed (5400 RPM)
- Number of platters (1-30)
- Number of tracks (≤10000)
- Number of bytes/track (10^5)

Unit of read or write: disk block
Once in memory: page
Typically: 4k or 8k or 16k
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
RAID

Several disks that work in parallel
• Redundancy: use parity to recover from disk failure
• Speed: read from several disks at once

Various configurations (called levels):
• RAID 1 = mirror
• RAID 4 = n disks + 1 parity disk
• RAID 5 = n+1 disks, assign parity blocks round robin
• RAID 6 = “Hamming codes”
Design Question

• Consider the following query:

```
SELECT S1.temp, S2.pressure
FROM TempSensor S1, PressureSensor S2
WHERE S1.location = S2.location
AND S1.time = S2.time
```

• How can the DBMS execute this query given
  – 1 GB of memory
  – 100 GB TempSensor and 10 GB PressureSensor
Buffer Manager

Files and access methods
Buffer pool manager

Main memory

Disk space manager

1 page corresponds
to 1 disk block

READ
WRITE

INPUT
OUTPUT

Page requests from higher-level code

Buffer pool

Disk page

Free frame

Disk = collection of blocks

choice of frame dictated by replacement policy

• Data must be in RAM for DBMS to operate on it!
• Buffer pool = table of <frame#, pageid> pairs
Buffer Manager

• Enables higher layers of the DBMS to assume that needed data is in main memory

• Needs to decide on page replacement policy
  – LRU, clock algorithm, or other

• Both work well in OS, but not always in DB
Least Recently Used (LRU)

- Order pages by the time of last accessed
- Always replace the least recently accessed

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

Access P6

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

LRU is expensive (why ?); the clock algorithm is good approx
Buffer Manager

• Why not use the OS for the task??
• Reason 1: Correctness
  – DBMS needs fine grained control for transactions
  – Needs to force pages to disk for recovery purposes
• Reason 2: Performance
  – DBMS may be able to anticipate access patterns
  – Hence, may also be able to perform prefetching
  – May select better page replacement policy
Transaction Management and the Buffer Manager

Transaction manager operates on buffer pool

- **Recovery**: ‘log-file write-ahead’, then careful policy about which pages to force to disk

- **Concurrency control**: locks at the page level, multiversion concurrency control
Connection to ACID

• Recovery from crashes: ACID
  – Will discuss today

• Concurrency control: ACID
  – Will discuss next week
Recovery

From which events below can DBMS recover?

• Wrong data entry
• Disk failure
• Fire / earthquake / bankruptcy / … .
• System failure, transaction failure:
  – Power failure
  – Rollback
## Recovery

<table>
<thead>
<tr>
<th>Type of Crash</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong data entry</td>
<td>Constraints and Data cleaning</td>
</tr>
<tr>
<td>Disk crashes</td>
<td>Redundancy: e.g. RAID, archive</td>
</tr>
<tr>
<td>Fire, theft, bankruptcy…</td>
<td>Buy insurance, Change jobs…</td>
</tr>
<tr>
<td>System/transaction failures</td>
<td>DATABASE RECOVERY</td>
</tr>
</tbody>
</table>

Most frequent
System Failures

• Each transaction has *internal state*

• When system crashes, internal state is lost
  – Don’t know which parts executed and which didn’t
  – Need ability to *undo* and *redo*

• Remedy: use a **log**
  – File that records every single action of each transaction
Problem Illustration

Client 1:

START TRANSACTION
INSERT INTO SmallProduct(name, price)
    SELECT pname, price
    FROM Product
    WHERE price <= 0.99

DELETE Product
    WHERE price <= 0.99

COMMIT

What do we do now?

Crash!
Transactions

• Assumption: db 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
<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Transaction 1

- **Action**
  - `INPUT(A)`
  - `READ(A,t)`
  - `t:=t*2`
  - `WRITE(A,t)`
  - `READ(B,t)`
  - `t:=t*2`
  - `WRITE(B,t)`
  - `OUTPUT(A)`
  - `OUTPUT(B)`

### Table

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(A,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>t</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
</tr>
<tr>
<td>-------------</td>
<td>----</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>t</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
</tr>
<tr>
<td>-----------------</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table:

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Code:

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

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>t</td>
<td>Mem A</td>
<td>Mem B</td>
<td>Disk A</td>
<td>Disk B</td>
</tr>
<tr>
<td>---------------</td>
<td>----</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Transaction

<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>

### Buffer pool

### Disk

---

**Action:** 
- `READ(A,t); t := t*2; WRITE(A,t);`
- `READ(B,t); t := t*2; WRITE(B,t);`
```plaintext
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
```
<table>
<thead>
<tr>
<th>Action</th>
<th>t</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

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

- **STEAL or NO-STEAL**
  - Can an update made by an uncommitted transaction overwrite the most recent committed value of a data item on disk?

- **FORCE or NO-FORCE**
  - Should all updates of a transaction be forced to disk before the transaction commits?

- Easiest for recovery: NO-STEAL/FORCE

- Highest performance: STEAL/NO-FORCE
Solution: Use a Log

- **Log**: append-only file containing log records
- Enables the use of STEAL and NO-FORCE
- For every update, commit, or abort operation
  - Write physical, logical, or physiological log record
  - Note: multiple transactions run concurrently, log records are interleaved
- After a system crash, use log to:
  - Redo some transaction that did commit
  - Undo other transactions that didn’t commit
Write-Ahead Log

• Rule 1: (WAL Rule) All log records pertaining to a page are written to disk before the page is overwritten on disk

• Rule 2: All log records for transaction are written to disk before the transaction is considered committed
  – Why is this faster than FORCE policy?

• Committed transaction: transactions whose commit log record has been written to disk
Undo Logging

Log records

• `<START T>`
  – Transaction T has begun

• `<COMMIT T>`
  – T has committed

• `<ABORT T>`
  – T has aborted

• `<T,X,v>` -- Update record
  – T has updated element X, and its *old* value was v
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START T&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT(A)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

4/21/2009 csep 544 48
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td></td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

WHAT DO WE DO?
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
</tbody>
</table>

4/21/2009

WHAT DO WE DO?

Crash!
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

Undo-logging Rule: If T commits, then OUTPUT(X) must be written to disk before <COMMIT T>

• Hence: OUTPUTs are done *early*, before the transaction commits
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT(A)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8&gt;</td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>INPUT(B)</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>COMMIT</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

4/21/2009

CSEP 544
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 from the end; 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

Question 1 in class:
Which updates are undone?

Question 2 in class:
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 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>

other transactions

transactions T2, T3, T4, T5

4/21/2009
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. Flush log to disk

- Continue normal operation

- When all of T1,..,Tk have completed, write `<END CKPT>`. Flush log to disk
Undo Recovery with Nonquiescent Checkpointing

During recovery, Can stop at first <CKPT>

Q: do we need <END CKPT> ?

csep 544
Implementing ROLLBACK

• Recall: a transaction can end in COMMIT or ROLLBACK
• Idea: use the undo-log to implement ROLLBACK
• How?
• LSN = Log Sequence Number
• Log entries for the same transaction are linked, using the LSN’s
Redo Logging

Log records

• \(<\text{START T}\> = \text{transaction T has begun}\)
• \(<\text{COMMIT T}\> = \text{T has committed}\)
• \(<\text{ABORT T}\>= \text{T has aborted}\)
• \(<\text{T,X,v}\>= \text{T has updated element X, and its new value is v}\)
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,16&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Redo-Logging Rules

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

• Hence: OUTPUTs are done \textit{late}
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;START T&gt;</td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,16&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;COMMIT T&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
Recovery with Redo Log

After system’s crash, run recovery manager

• Step 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

• 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>
...
...

4/21/2009
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 written, write <END CKPT>
Redo Recovery with Nonquiescent Checkpointing

Step 1: look for
The last
(END CKPT>

All OUTPUTs of T1 are known to be on disk

Cannot use

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

All OUTPUTs of T1 are known to be on disk

<START T1>

<COMMIT T1>

<START T4>

<START CKPT T4, T5, T6>

<END CKPT>

<START CKPT T9, T10>

...
Nonquiescent Checkpointing

• This checkpointing methods is only for the simple redo-log
• We will discuss later the checkpointing method for ARIES, which differs significantly
• The book describes ARIES only
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 \textit{old} value was \(u\), and its \textit{new} value is \(v\)
<table>
<thead>
<tr>
<th>Action</th>
<th>T</th>
<th>Mem A</th>
<th>Mem B</th>
<th>Disk A</th>
<th>Disk B</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;START T&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAT(A,t)</td>
<td>8</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(A,t)</td>
<td>16</td>
<td>16</td>
<td></td>
<td>8</td>
<td>8</td>
<td>&lt;T,A,8,16&gt;</td>
</tr>
<tr>
<td>READ(B,t)</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>t:=t*2</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>WRITE(B,t)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>&lt;T,B,8,16&gt;</td>
</tr>
<tr>
<td>OUTPUT(A)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;COMMIT T&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT(B)</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

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>
...
...

4/21/2009
csep 544
ARIES Method

• Read R&K Chapter 18

• Three pass algorithm
  – **Analysis pass**
    • Figure out what was going on at time of crash
    • List of dirty pages and active transactions
  – **Redo pass (repeating history principle)**
    • Redo all operations, even for transactions that will not commit
    • Get back to state at the moment of the crash
  – **Undo pass**
    • Remove effects of all uncommitted transactions
    • Log changes during undo in case of another crash during undo
ARIES Method Illustration

Figure 3: The Three Passes of ARIES Restart

[Figure 3 from Franklin97]
ARIES Method Elements

- Each page contains a pageLSN
  - Log Sequence Number of log record for latest update to that page
  - Will serve to determine if an update needs to be redone

- Physiological logging
  - page-oriented REDO
    - Possible because will always redo all operations in order
  - logical UNDO
    - Needed because will only undo some operations
ARIES Data Structures

• **Transaction table**
  – Lists all running transactions (active transactions)
  – For each txn: lastLSN = most recent update by transaction

• **Dirty page table**
  – Lists all dirty pages
  – For each dirty page: recoveryLSN = first LSN that caused page to become dirty

• **Write ahead log contains log records**
  – LSN, prevLSN = previous LSN for same transaction
  – other attributes
# ARIES Data Structures

## Dirty pages

<table>
<thead>
<tr>
<th>pageID</th>
<th>recLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>P5</td>
<td>2</td>
</tr>
<tr>
<td>P6</td>
<td>3</td>
</tr>
<tr>
<td>P7</td>
<td>1</td>
</tr>
</tbody>
</table>

## Log

<table>
<thead>
<tr>
<th>LSN</th>
<th>prevLSN</th>
<th>transID</th>
<th>pageID</th>
<th>Log entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>T100</td>
<td>P7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>T200</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T200</td>
<td>P6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>T100</td>
<td>P5</td>
<td></td>
</tr>
</tbody>
</table>

## Active transactions

<table>
<thead>
<tr>
<th>transID</th>
<th>lastLSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>T100</td>
<td>4</td>
</tr>
<tr>
<td>T200</td>
<td>3</td>
</tr>
</tbody>
</table>
ARIES Method Details

• Steps under normal operations
  – Add log record
  – Update transactions table
  – Update dirty page table
  – Update pageLSN
Checkpoints

• Write into the log
  – Contents of transactions table
  – Contents of dirty page table

• Enables REDO phase to restart from earliest recoveryLSN in dirty page table
  – Shortens REDO phase
Analysis Phase

• Goal
  – Determine point in log where to start REDO
  – Determine set of dirty pages when crashed
    • Conservative estimate of dirty pages
    – Identify active transactions when crashed

• Approach
  – Rebuild transactions table and dirty pages table
  – Reprocess the log from the beginning (or checkpoint)
    • Only update the two data structures
  – Find oldest recoveryLSN (firstLSN) in dirty pages tables
Redo Phase

• Goal: redo all updates since firstLSN

• For each log record
  – If affected page is not in Dirty Page Table then do not update
  – If affected page is in Dirty Page Table but recoveryLSN > LSN of record, then no update
  – Else if pageLSN > LSN, then no update
  • Note: only condition that requires reading page from disk
  – Otherwise perform update
Undo Phase

- **Goal**: undo effects of aborted transactions
- Identifies all loser transactions in trans. table
- **Scan log backwards**
  - Undo all operations of loser transactions
  - Undo each operation unconditionally
  - All ops. logged with compensation log records (CLR)
  - Never undo a CLR
    - Look-up the UndoNextLSN and continue from there
Handling Crashes during Undo

Figure 4: The Use of CLRs for UNDO

[Figure 4 from Franklin97]
Summary

• Transactions are a useful abstraction

• They simplify application development

• DBMS must maintain ACID properties in face of
  – Concurrency
  – Failures