

Lecture 16: Transactions (Recovery)

Wednesday, May 16, 2012

Announcements

- **Makeup lectures:**
 - Friday, May 18, 10:30-11:50, CSE 405
 - Friday, May 25, 10:30-11:50, CSE 405
- **No lectures:**
 - Monday and Wednesday (May 21 and 23)
- **Updated time:**
 - Wednesday, May 30, 9-10:30, CSE 405
- **Paper reviews**
 - May 25: provenance
 - May 30: privacy
- **Project presentations:**
 - Monday, May 28, 1:30-4:30 and
 - Tuesday, May 29, 8:30-12
- **Homework 3:**
 - **POSTED!**
 - Due Sunday, June 3 at midnight

Outline

- Transaction basics
- Recovery
- Concurrency control (next lecture)

Textbook (Ramakrishnan): Ch. 16, 17, 18

Read BOTH the slides and the book!

Transaction

Definition: a transaction is a sequence of updates to the database with the property that either all complete, or none completes (all-or-nothing).

BEGIN TRANSACTION

[SQL statements]

COMMIT or **ROLLBACK (=ABORT)**

May be omitted:
first SQL query
starts txn

In ad-hoc SQL: each statement = one transaction

Implementing Transactions

What can go wrong ?

- System crash
 - Division by 0, power failure
- Interferences with other users
 - “Anomalies” – 3 have famous names

System Crash

Client 1:
BEGIN TRANSACTION
UPDATE Account1
SET balance= balance – 500



Crash !

UPDATE Account2
SET balance = balance + 500
COMMIT

1st Famous Anomaly: Lost Update

Client 1:

BEGIN TRANSACTION

UPDATE Account1

SET balance= balance+\$100

COMMIT

Client 2:

BEGIN TRANSACTION

UPDATE Account1

SET balance=balance-\$100

COMMIT

Lost update: two TXN's update the same element, but only one succeeds.

2nd Famous Anomaly: Inconsistent Read

```
Client 1: transfer $100  
BEGIN TRANSACTION  
  UPDATE Account1  
  SET balance= balance+$100  
  
  UPDATE Account2  
  SET balance= balance+$100  
COMMIT
```

```
Client 2: check total balance  
BEGIN TRANSACTION  
  SELECT sum(balance)  
  FROM All_Accounts  
COMMIT
```

Inconsistent read: TXN sees some updates by another TXN, but not all updates.

3rd Famous Anomaly: Dirty Reads

```
-- Client 1:  
BEGIN TRANSACTION  
UPDATE Account1  
SET balance= balance+$100  
  
...  
ROLLBACK
```

```
-- Client 2: get cash $100  
BEGIN TRANSACTION  
X = Account1.balance  
If (X>=100)  
    { ...dispense money...  
      COMMIT }  
...
```

Dirty read: TXN reads a value written by another transaction that later aborts.

ACID Properties

- **A**tomic
 - State shows either all the effects of txn, or none of them
- **C**onsistent
 - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
 - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
 - Once a txn has committed, its effects remain in the database

Outline

- Recovery from failures (the A in ACID)
 - Today
- Concurrency Control (the C in ACID)
 - Next lecture

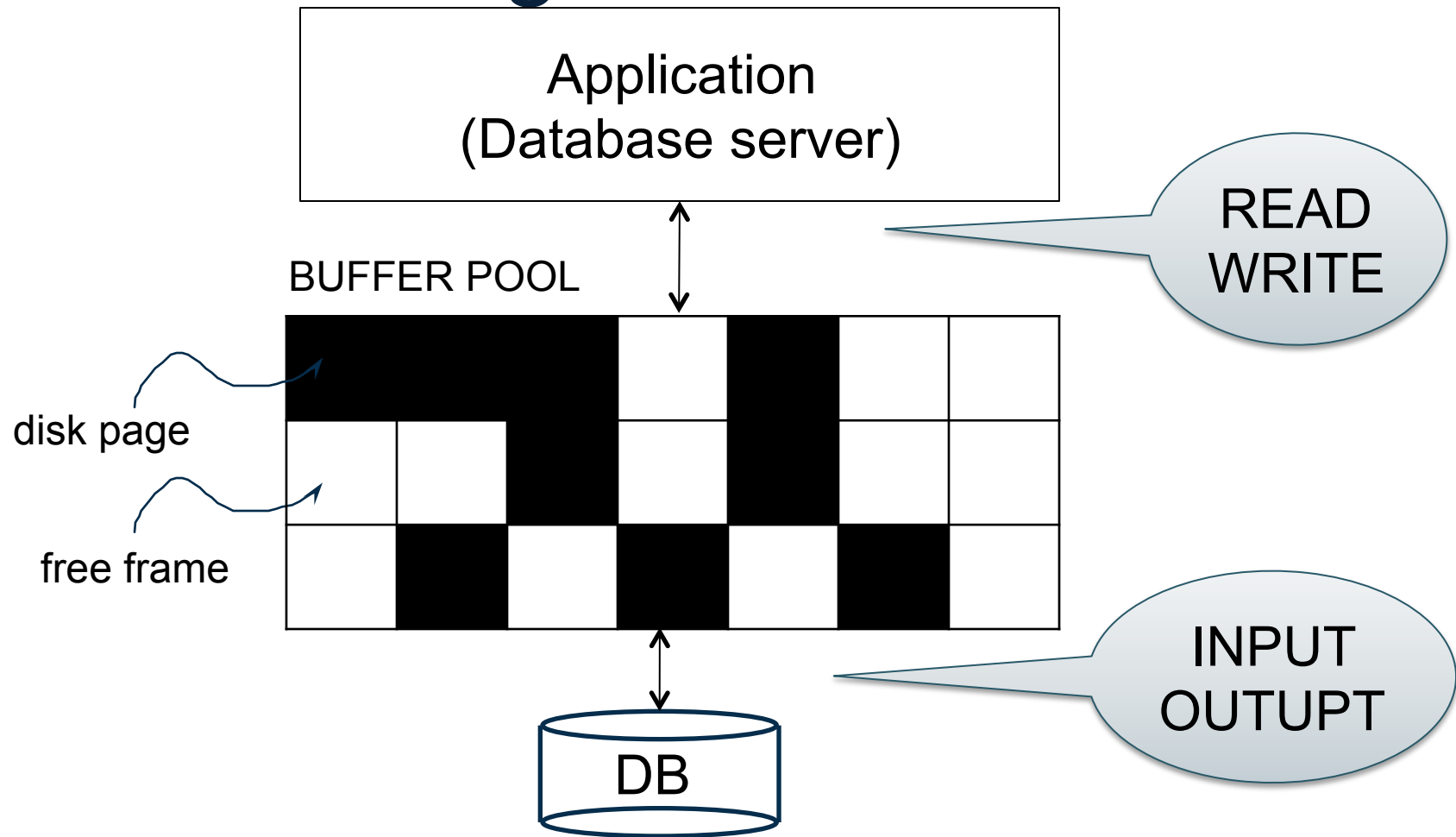
Log-based Recovery

Basics (based on Garcia-Molina Ch. 17.2, 17.3, 17.4)

- **Undo** logging
- **Redo** logging

Aries: (Ramakrishnan Ch. 18)

Buffer Management in a DBMS



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

Transactions

- Assumption: the database is composed of *elements*.
- 1 element can be either:
 - 1 page = physical logging
 - 1 record = logical logging
- Aries uses both (will discuss later)

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

Running Example

```
BEGIN TRANSACTION
```

```
READ(A,t);
```

```
t := t*2;
```

```
WRITE(A,t);
```

```
READ(B,t);
```

```
t := t*2;
```

```
WRITE(B,t)
```

```
COMMIT;
```

Initially, $A=B=8$.

Atomicity requires that either
(1) T commits and $A=B=16$, or
(2) T does not commit and $A=B=8$.

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
COMMIT					

Is this bad ?

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
COMMIT					

Crash !

Is this bad ?

Yes it's bad: A=16, B=8....

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
COMMIT					

Crash !

Is this bad ?

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
COMMIT					

Crash !

Is this bad ?

Yes it's bad: $A=B=16$, but not committed

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
COMMIT					

Crash !

Is this bad ?

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
COMMIT					

Crash !

Is this bad ?

No: that's OK

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
COMMIT					



Crash !

Atomic Transactions

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

- **FORCE**: Pages of committed transactions must be forced to disk before commit
- **NO-STEAL**: Pages of uncommitted transactions cannot be written to disk

Easy to implement (how?) and ensures atomicity

No-Force/Steal

- **NO-FORCE**: Pages of committed transactions need not be written to disk
- **STEAL**: Pages of uncommitted transactions may be written to disk

In either case, Atomicity is violated; need WAL

Write-Ahead Log

The Log: append-only file containing log records

- Records every single action of every TXN
- Force log entry to disk
- After a system crash, use log to recover

Three types: UNDO, REDO, UNDO-REDO

UNDO Log

FORCE and STEAL

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	UNDO 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	UNDO 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>



WHAT DO WE DO ?

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO 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>



WHAT DO WE DO ?

We **UNDO** by setting B=8 and A=8

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO 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>

What do we do now ?

Crash !

Action	t	Mem A	Mem B	Disk A	Disk B	UNDO 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>

What do we do now ?

Nothing: log contains COMMIT

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>



Question 1: Which updates are undone ?

Question 2:
How far back do we need to read in the log ?

Question 3:
What happens if there is a second crash, during recovery ?

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

When must we force pages to disk ?



Action	t	Mem A	Mem B	Disk A	Disk B	UNDO 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>

RULES: log entry before OUTPUT before COMMIT

Undo-Logging Rules

U1: If T modifies X, then $\langle T, X, v \rangle$ must be written to disk before $\text{OUTPUT}(X)$

U2: If T commits, then $\text{OUTPUT}(X)$ must be written to disk before $\langle \text{COMMIT } T \rangle$



FORCE

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

REDO Log

NO-FORCE and NO-STEAL

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Yes, it's bad: A=16, B=8

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



Crash !

Is this bad ?

Yes, it's bad: lost update

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



Crash !

Is this bad ?

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



Crash !

Is this bad ?

No: that's OK.

Action	t	Mem A	Mem B	Disk A	Disk B
READ(A,t)	8	8		8	8
t:=t*2	16	8		8	8
WRITE(A,t)	16	16		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
COMMIT					
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16



Crash !

Redo Logging

One minor change to the undo log:

- $\langle T, X, v \rangle =$ T has updated element X, and its new value is v

Action	t	Mem A	Mem B	Disk A	Disk B	REDO 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						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Action	t	Mem A	Mem B	Disk A	Disk B	REDO 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						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	



How do we recover ?

Action	t	Mem A	Mem B	Disk A	Disk B	REDO 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						<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	



How do we recover ?

We **REDO** by setting A=16 and B=16

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>
```

Crash !

Show actions during recovery

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

When must we force pages to disk ?



Action	t	Mem A	Mem B	Disk A	Disk B	REDO 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		NO-STEAL				<COMMIT T>
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

RULE: OUTPUT after COMMIT

Redo-Logging Rules

R1: If T modifies X, then both $\langle T, X, v \rangle$ and $\langle \text{COMMIT } T \rangle$ must be written to disk before $\text{OUTPUT}(X)$

NO-STEAL

- Hence: OUTPUTs are done late

Comparison Undo/Redo

- Undo logging: OUTPUT must be done early:
 - Inefficient
- Redo logging: OUTPUT must be done late:
 - Inflexible

ARIES

Aries

- ARIES pieces together several techniques into a comprehensive algorithm
- Developed at IBM Almaden, by Mohan
- IBM botched the patent, so everyone uses it now
- Several variations, e.g. for distributed transactions

ARIES Recovery Manager

- A redo/undo log
- **Physiological logging**
 - Physical logging for REDO
 - Logical logging for UNDO
- Efficient checkpointing



Why ?

ARIES Recovery Manager

Log entries:

- $\langle \text{START } T \rangle$ -- when T begins
- Update: $\langle T, X, u, v \rangle$
 - T updates X, old value=u, new value=v
 - In practice: undo only and redo only entries
- $\langle \text{COMMIT } T \rangle$ or $\langle \text{ABORT } T \rangle$
- CLR's – we'll talk about them later.

ARIES Recovery Manager

Rule:

- If T modifies X, then $\langle T, X, u, v \rangle$ must be written to disk before OUTPUT(X)

We are free to OUTPUT early or late

LSN = Log Sequence Number

- **LSN** = identifier of a log entry
 - Log entries belonging to the same TXN are linked
- Each page contains a **pageLSN**:
 - LSN of log record for latest update to that page

ARIES Data Structures

- **Active Transactions Table**

- Lists all active TXN's
- For each TXN: **lastLSN** = its most recent update LSN

- **Dirty Page Table**

- Lists all dirty pages
- For each dirty page: **recoveryLSN** (**recLSN**) = first LSN that caused page to become dirty

- **Write Ahead Log**

- LSN, **prevLSN** = previous LSN for same txn

$W_{T100}(P7)$
 $W_{T200}(P5)$
 $W_{T200}(P6)$
 $W_{T100}(P5)$

ARIES Data Structures

Dirty pages

pageID	recLSN
P5	102
P6	103
P7	101

Log (WAL)

LSN	prevLSN	transID	pageID	Log entry
101	-	T100	P7	
102	-	T200	P5	
103	102	T200	P6	
104	101	T100	P5	

Active transactions

transID	lastLSN
T100	104
T200	103

Buffer Pool

P8	P2	...
	...	
P5 PageLSN=104	P6 PageLSN=103	P7 PageLSN=101

ARIES Normal Operation

T writes page P

- What do we do ?

ARIES Normal Operation

T writes page P

- What do we do ?
- Write $\langle T, P, u, v \rangle$ in the **Log**
- $\text{pageLSN} = \text{LSN}$
- $\text{prevLSN} = \text{lastLSN}$
- $\text{lastLSN} = \text{LSN}$
- $\text{recLSN} = \text{if isNull then LSN}$

ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- What do we do ?

Buffer manager wants INPUT(P)

- What do we do ?

ARIES Normal Operation

Buffer manager wants to OUTPUT(P)

- Flush log up to **pageLSN**
- Remove P from **Dirty Pages** table

Buffer manager wants INPUT(P)

- Create entry in **Dirty Pages** table
recLSN = NULL

ARIES Normal Operation

Transaction T starts

- What do we do ?

Transaction T commits/aborts

- What do we do ?

ARIES Normal Operation

Transaction T starts

- Write **<START T>** in the **log**
- New entry T in **Active TXN**;
lastLSN = null

Transaction T commits/aborts

- Write **<COMMIT T>** in the **log**
- Flush **log** up to this entry

Checkpoints

Write into the log

- Entire **active transactions table**
- Entire **dirty pages table**

Recovery always starts by analyzing latest checkpoint

Background process periodically flushes dirty pages to disk

ARIES Recovery

1. Analysis pass

- Figure out what was going on at time of crash
- List of dirty pages and active transactions

2. 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

3. Undo pass

- Remove effects of all uncommitted transactions
- Log changes during undo in case of another crash during undo

ARIES Method Illustration

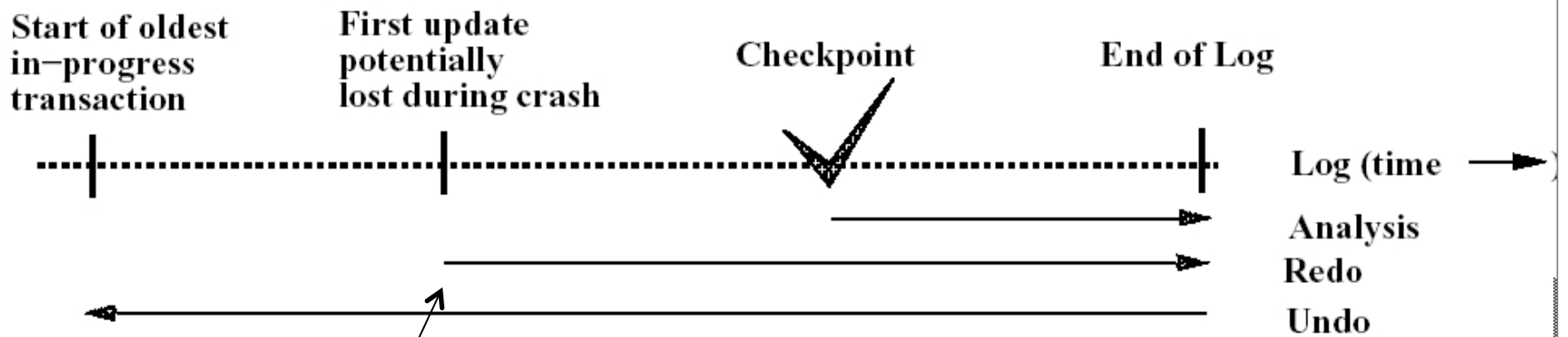


Figure 3: The Three Passes of ARIES Restart

First undo and first redo log entry might be in reverse order

[Figure 3 from Franklin97]

1. 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 **active transactions table** and **dirty pages table**
 - Reprocess the log from the checkpoint
 - Only update the two data structures
 - Compute: **firstLSN** = smallest of all **recoveryLSN**

1. Analysis Phase

Log

Checkpoint

(crash)



firstLSN= ????

Where do we start the REDO phase ?

Dirty pages

pageID	recLSN	pageID

Active txn

transID	lastLSN	transID

1. Analysis Phase

Log

Checkpoint

(crash)



$\text{firstLSN} = \min(\text{recLSN})$

**Dirty
pages**

pageID	recLSN	pageID

**Active
txn**

transID	lastLSN	transID

1. Analysis Phase

Log

Checkpoint

(crash)



firstLSN

**Dirty
pages**

pageID	recLSN	pageID

Replay
history

pageID	recLSN	pageID

**Active
txn**

transID	lastLSN	transID

transID	lastLSN	transID

2. Redo Phase

Main principle: replay history

- Process Log forward, starting from **firstLSN**
- Read every log record, sequentially
- Redo actions are not recorded in the log
- Needs the **Dirty Page Table**

2. Redo Phase: Details

For each **Log** entry record **LSN: $\langle T, P, u, v \rangle$**

- Re-do the action $P=u$ and $WRITE(P)$
- But which actions can we skip, for efficiency ?

2. Redo Phase: Details

For each **Log** entry record **LSN**: $\langle T, P, u, v \rangle$

- If **P** is not in **Dirty Page** then **no update**
- If **recLSN** $>$ **LSN**, then **no update**
- Read page from disk:
If **pageLSN** $>$ **LSN**, then **no update**
- Otherwise perform update

2. Redo Phase: Details

What happens if system crashes during REDO ?

2. Redo Phase: Details

What happens if system crashes during REDO ?

We REDO again ! Each REDO operation is *idempotent*: doing it twice is the as as doing it once.

3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?

3. Undo Phase

- Cannot “unplay” history, in the same way as we “replay” history
- WHY NOT ?
- Need to support ROLLBACK: selective undo, for one transaction
- Hence, *logical* undo v.s. *physical* redo

3. Undo Phase

Main principle: “logical” undo

- Start from end of **Log**, move backwards
- Read only affected log entries
- Undo actions *are* written in the Log as special entries: **CLR** (Compensating Log Records)
- **CLRs** are redone, but never undone

3. Undo Phase: Details

- “Loser transactions” = uncommitted transactions in **Active Transactions Table**
- **ToUndo** = set of **lastLSN** of loser transactions

3. Undo Phase: Details

While **ToUndo** not empty:

- Choose most recent (largest) **LSN** in **ToUndo**
- If **LSN** = regular record **<T,P,u,v>**:
 - Undo v
 - Write a **CLR** where **CLR.undoNextLSN** = **LSN.prevLSN**
- If **LSN** = **CLR record**:
 - Don't undo !
- if **CLR.undoNextLSN** not null, insert in **ToUndo** otherwise, write **<END TRANSACTION>** in log

3. Undo Phase: Details

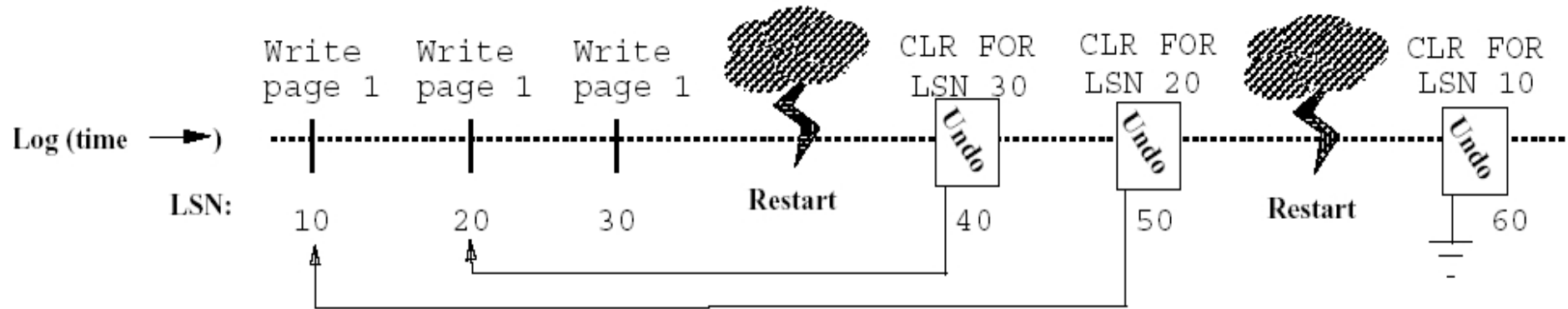


Figure 4: The Use of CLR for UNDO

[Figure 4 from Franklin97]

3. Undo Phase: Details

What happens if system crashes during UNDO ?

3. Undo Phase: Details

What happens if system crashes during UNDO ?

We do not UNDO again ! Instead, each CLR is a REDO record: we simply redo the undo

Physical v.s. Logical Logging

Why are redo records physical ?

Why are undo records logical ?

Physical v.s. Logical Logging

Why are redo records physical ?

- Simplicity: replaying history is easy, and idempotent

Why are undo records logical ?

- Required for transaction rollback: this not “undoing history”, but selective undo