Recovery

Databases, Aarhus University

Ira Assent

Intended learning outcomes

- Be able to
  - Describe characteristics of different recovery strategies
  - Identify the use of recovery techniques in the ARIES system

ira@cs.au.dk

Recap: More CC

- Multiversion timestamping
  - Always work with most recent version that is in the past of the transaction
    - highest write_TS(X_i) of all versions of X less than or equal to TS(T)
    - Reads always possible; if own timestamp larger than current, update
  - Abort if version attempted to write already read by future transaction, i.e., read _TS(X_i) > TS(T); otherwise create new version
- Validation (optimistic) concurrency control
  - Read phase with read and write on local copies; before commit, validation phase: if issue, abort; otherwise, write phase applies changes to database
  - Validate T_i, for each T_j committed or in validation

    (1) T_i completes all three phases before T_j begins

    (2) T_j completes its write phase before T_i starts its write phase, and T_j does not write to any item read by T_i : write_set(T_j) ∩ read_set(T_i) = ∅

    (3) T_j completes its read phase before T_i completes its read phase and T_j does not write to any item that is either read or written by T_i :

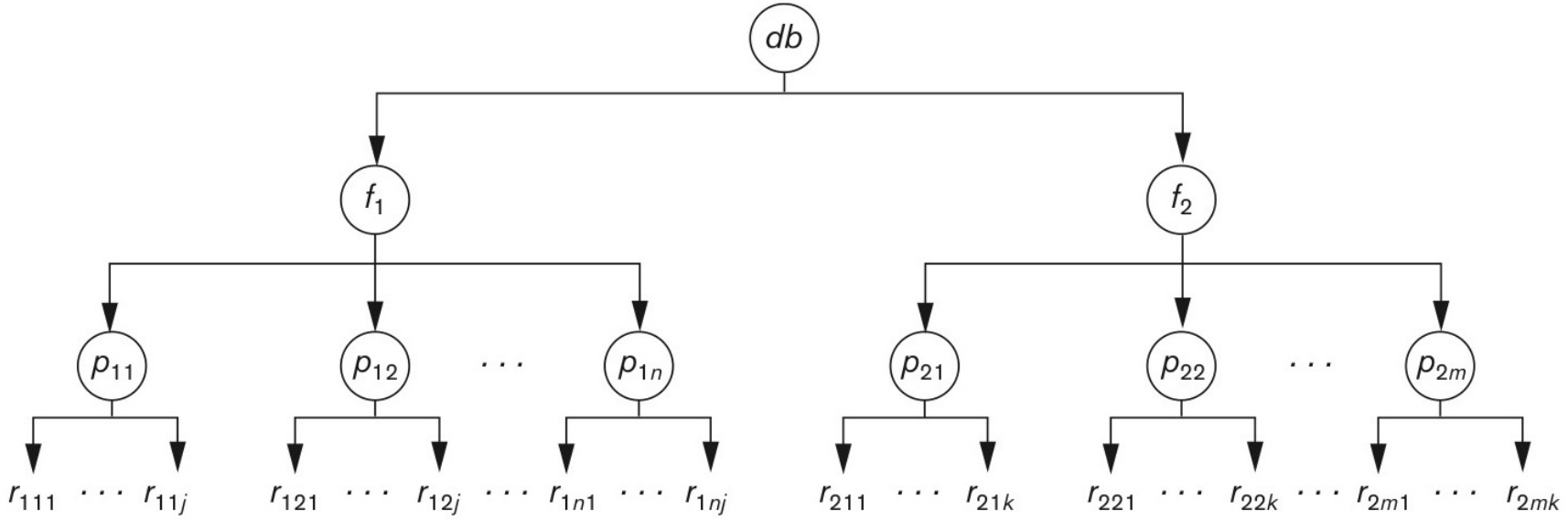
    write_set(T_j) ∩ read_set(T_i) = ∅

    write_set(T_j) ∩ write_set(T_i) = ∅
  - as soon as one condition holds, validates; if all fail, abort T_i

Recap: Multiple Granularity Locking

Granularity of data items: lockable unit of data

coarse (entire database), fine (a tuple or an attribute of a relation)

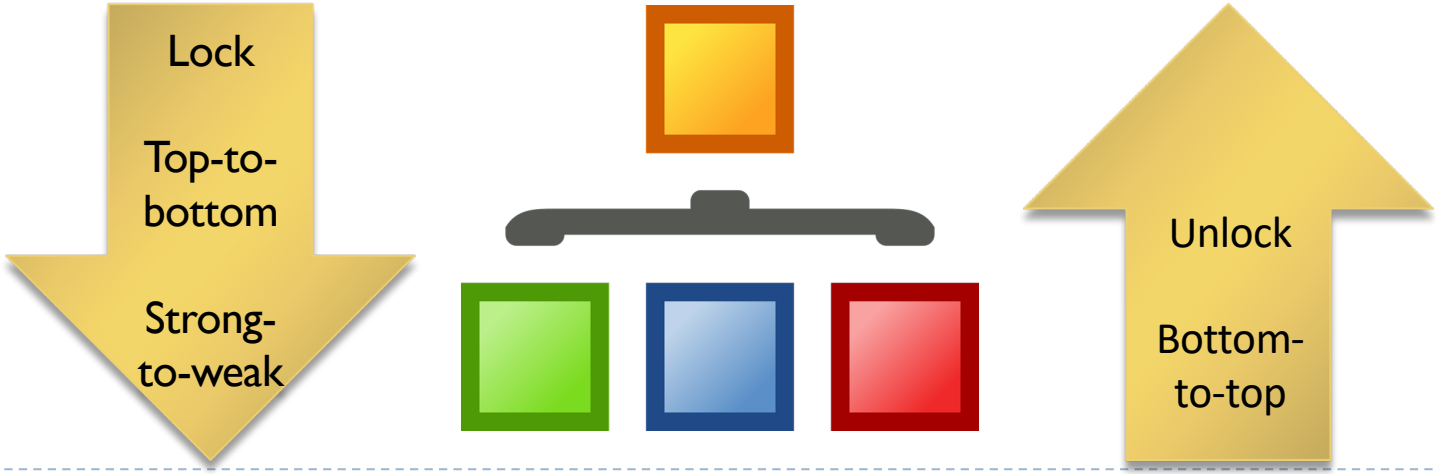


- **Intention-shared (IS)**: shared lock(s) will be requested on some descendent nodes(s)
- **Intention-exclusive (IX)**: exclusive lock(s) will be requested on some descendent node(s)
- **Shared-intention-exclusive (SIX)**: current node is shared locked and exclusive lock(s) will be requested on some descendent nodes(s)

ira@cs.au.dk

Producing multigranularity schedules

- Rules for producing serializable schedules using multiple granularity locking
  1. The lock compatibility must be adhered to
  2. The root of the tree must be locked first
  3. A node can be locked by a transaction T in S or IS mode only if its parent node is already locked by T in IS or IX mode
  4. A node can be locked by T in X, IX, or SIX mode only if its parent node is already locked by T in IX or SIX mode
  5. T can lock a node only if it has not unlocked any node (2PL policy)
  6. T can unlock a node only if none of its children are currently locked by T



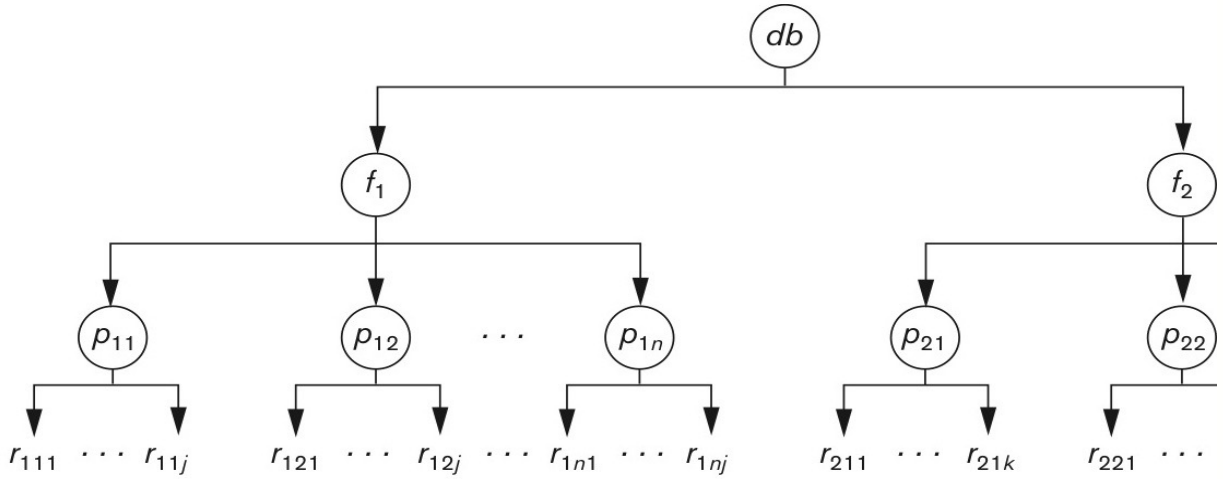
ira@cs.au.dk

Example

T_1 wants to update record r_{111} and r_{211}

T_2 wants to update all records on page p_{12}

T_3 wants to read record r_{11j} and the entire f_2 file



T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)

Certify Locks Concept

- Multiversion Two-Phase Locking Using Certify Locks
  - Allow a transaction T' to read a data item X while it is write-locked by a conflicting transaction T
  - This is accomplished by maintaining two versions of each data item X where one version must have been written by some committed transaction
    - The second "local version" created when a transaction acquires a write lock
    - This means a write operation always creates a new version of X
  - So, three *modes* of locks: read, write, certify
    - Write lock is no longer fully exclusive, reads possible
    - Certify lock new: fully exclusive

ira@cs.au.dk

The steps in Certify Locks Concurrency Control

1. X is committed version of a data item
2. When T wishes to write X, T creates a second version X' after obtaining write lock on X
3. Other transactions continue to read X
4. When T is ready to commit, it obtains certify lock on X'
5. The committed version X becomes X'
6. T releases its certify lock on X', which is new committed version of X

- In standard locking, compatibility:
  - if read locked, other read locks possible
- In certify locking,
  - If read or write locked, more read locks possible
  - Certify locks exclusive
    - form of lock upgrade from write lock

	Read	Write
Read	Yes	No
Write	No	No

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

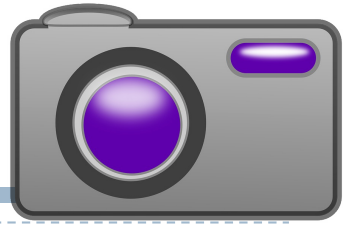
ira@cs.au.dk

Certify locks: pros and cons

- In multiversion 2PL with certify locks
  - read and write operations from conflicting transactions can be processed concurrently
    - Improves concurrency
  - But may delay transaction when it is ready to commit because of obtaining certify locks on all its writes
  - Avoids cascading abort but like strict two-phase locking scheme conflicting transactions may get deadlocked

ira@cs.au.dk

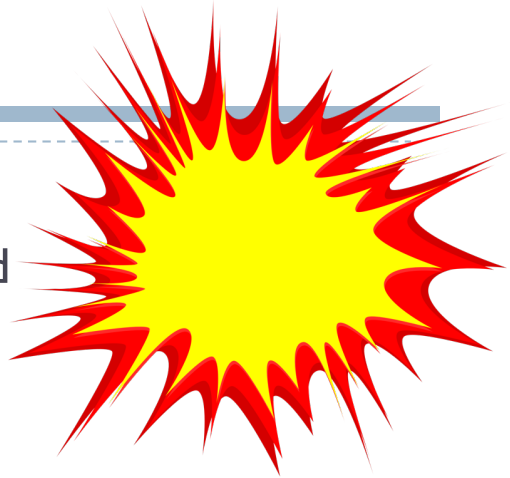
Snapshot isolation



- Transactions see data items based on committed values of the items in the database snapshot (or database state) when transaction starts
- Any changes after transaction start not seen
- No read locks, only write locks
- Writes create older versions in temporary version store (tempstore) along with creation timestamp
  - So that other transactions that started before can read the respective older version
  - Usually implemented using pointers from item to list of recent versions
- No phantom reads, dirty read, or nonrepeatable read as only committed values are seen
  - Rare anomalies can occur (not covered here, complex to detect)
    - Either ignore these issues or
    - Resolve in program (cumbersome) or
    - Use variant: serializable snapshot isolation (SSI) e.g. in PostgreSQL
  - Tradoff between runtime performance and accepting rare anomalies

ira@cs.au.dk

Recovery



- Handling failures
  - DBMS is responsible for handling of any transaction submitted
- Types of failures
  - System crash
    - Hardware (e.g. memory), software, network error…
  - Transaction or system error
    - Division by zero, integer overflow, erroneous parameter values, logical programming error
  - Local errors, non-handled exception condition
    - Data not found, insufficient account balance for withdrawal….
  - Concurrency control enforcement
    - Aborted transactions (e.g. deadlock handlling, serializability violations,…)
  - Disk failure
    - Issue with read or write during transaction operation
  - Physical problems / catastrophes
    - Fire, theft, sabotage, incorrect mounting of tape, air-condition failure,…
- Maintain ACID properties when things go wrong

ira@cs.au.dk

Recovery: what to in case of crash?

Example: transfer $50 from account A to account B

```
1 Transaction starts
2 Read A
3 A ← A - 50
4 Write A
5 Read B
6 B ← B + 50
7 Write B
8 Transaction ends
```

A. Re-execute the entire transaction
B. Do not re-execute the entire transaction
C. We cannot execute the remaining statements
D. Execute the remaining statements

ira@cs.au.dk

Recovery

- Initial (incorrect) procedures
  - Re-execute transaction
    - If crashed after instruction 4 (or later), incorrect values (A = 130,...)
  - Do not re-execute the transaction
    - If crashed before instruction 7, incorrect values (..., B = 100)
- Problems
  - Do not know which instruction executed last
  - Multiple, concurrent users
  - Buffer management

Example: transfer $50 from account A to account B

```
1 Transaction starts
2 Read A
3 A ← A - 50
4 Write A
5 Read B
6 B ← B + 50
7 Write B
8 Transaction ends
```

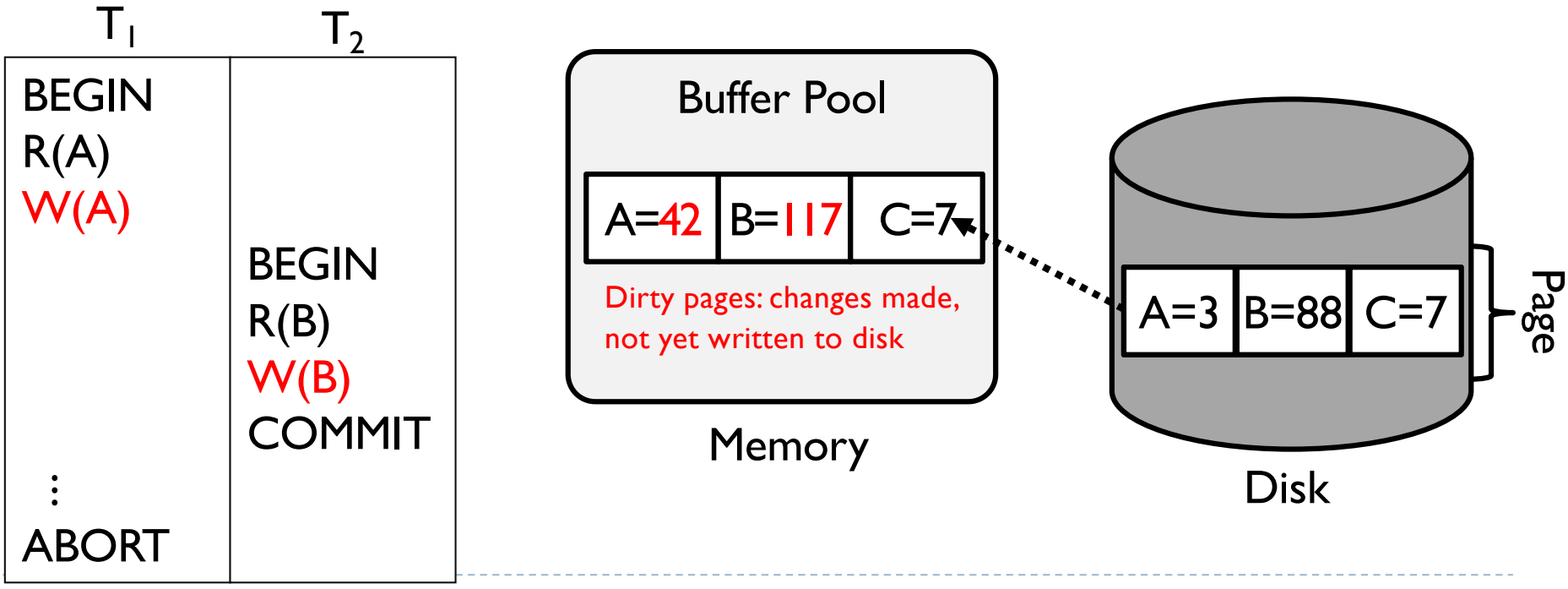
Sample values of database variables at various points in execution:

Last Instruction	A	B
2	180	100
5	130	100
7	130	150

ira@cs.au.dk

Buffer Pool Management

- Database operates
  - Load data from disk (I/O) to do work in main memory
    - Creates "dirty pages" that contain changes not yet written to disk
  - Operating System (OS) will write to disk at some point in time
    - DBMS may need to force some writes to ensure durability



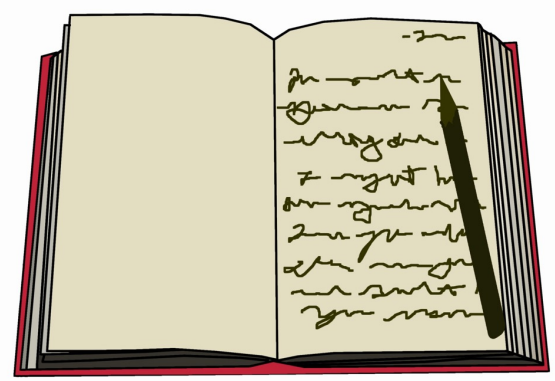
ira@cs.au.dk

Log-Based Recovery

- General idea of logs: keep track of modifications of the data
- Log manager to record important events in log:
  - When transaction *T* starts: <*T* starts> written to log
  - When *T* modifies data by Write(*X*):

    <*T*, *X*, old: *V*, new: *W*> written to log

    where *T* transaction ID, *X* data item, *V* old value, *W* new value
    - Important: log stored in stable storage
- When *T* reaches last statement, <*T* commits> written to log, and *T commits*
  - Commits precisely when commit entry (after all previous entries for T) output to log



ira@cs.au.dk

Example

Initial values: A=100, B=300, C=5, D=60, E=80

T_1 is executed followed by T2

T_1	T_2
$R_1(A)$	$R_2(A)$
$A \leftarrow A+50$	$A \leftarrow A+10$
$R_1(B)$	$W_2(A)$
$B \leftarrow B+100$	$R_2(D)$
$W_1(B)$	$D \leftarrow D-10$
$R_1(C)$	$R_2(E)$
$C \leftarrow 2*C$	$R_2(B)$
$W_1(C)$	$E \leftarrow E+B$
$A \leftarrow A+B+C$	$W_2(E)$
$W_1(A)$	$D \leftarrow D+E$
	$W_2(D)$

The Log

```
1. <T1 starts>
2. <T1, B, old:  300, new: 400>
3. <T1, C, old:  5, new: 10>
4. <T1, A, old:  100, new:  560>
5.  <T1 commits>
6.  <T2 starts>
7.  <T2,  A, old: 560, new: 570>
8.  <T2,  E, old: 80, new: 480>
9.  <T2,  D, old: 60, new: 530>
10. <T2 commits>
```

Output of B can occur any time after entry 2 output to log, etc.

ira@cs.au.dk

Buffering of Log Entries and Data Items

- We have so far assumed that log entries are put on stable storage when they are created
- Can we lift this restriction and only place log records in the cache? This would yield better performance!
  - Yes!

- We must impose the following restrictions
  - Transaction *T* cannot commit before <*T* commits> has been output to stable storage
  - <*T* commits> cannot be placed on stable storage before all other entries pertaining to *T* are on stable storage
  - A block of data items cannot be output to stable storage until all log entries pertaining to the data items are output

ira@cs.au.dk

In case of failure

1. Redo all transactions for which log has either "start" or "commit" entries
2. Redo all transactions for which log has both "start" and "commit" entries, undo all transactions for which log has "start" entry but no "commit" entry
3. Undo all transactions for which log has either "start" or "commit" entries
4. Undo all transactions for which log has both "start" and "commit" entries, redo all transactions for which log has "start" entry but no "commit" entry

ira@cs.au.dk

The Undo/Redo Algorithm

- Following a failure, the following is done
  - Redo all transactions for which log has both "start" and "commit" entries
  - Undo all transactions for which log has "start" entry but no "commit" entry
- Remarks
  - In a multitasking system, more than one transaction may need to be undone
  - If a system crashes during the recovery stage, the new recovery must still give correct results (*idempotent*)
  - In this algorithm, a large number of transactions need to be redone, since we do not know what data items are on disk

ira@cs.au.dk

Example

- If system crashes, recovery action depends on the last instruction (actually) written on log

Last Instruction	Action	Consequence
l = 0	Nothing	Neither T1 nor T2 has run
1 ≤ l ≤ 4	*Undo T1*: Restore the values of variables listed in 1-l old values	T1 has not run
5 ≤ l ≤ 9	*Redo T1*: Set the values of the variables listed in l-4 to values created by T1 *Undo T2*: Restore the values of variables listed in l-9 to those before T2 started execution	T1 ran T2 has not run
l=10	*Redo T1* *Redo T2*	T1 and T2 both ran

1. <T1 starts>
2. <T1, B, old: 300, new: 400>
3. <T1, C, old: 5, new: 10>
4. <T1, A, old: 100, new: 560>
5. <T1 commits>
6. <T2 starts>
7. <T2, A, old: 560, new: 570>
8. <T2, E, old: 80, new: 480>
9. <T2, D, old: 60, new: 530>
10. <T2 commits>

ira@cs.au.dk

Undo/Redo (immediate database modification)

- Goal: Maximize efficiency during normal operation
  - Some extra work required during recovery
- Allows maximum flexibility of buffer manager
  - Database outputs entirely asynchronous, other than having to happen after the corresponding log entry outputs
- Most complex at recovery time
  - Must implement both undo and redo
- Note: requires before and after images in log, or only after images if an initial before image is written to the log before first write

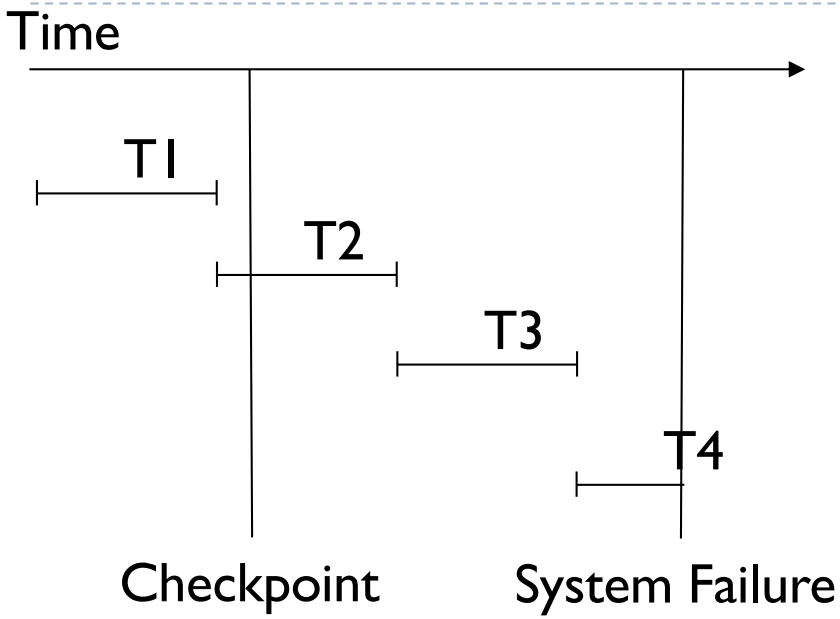
ira@cs.au.dk

Checkpointing

- Checkpointing speeds up recovery by flushing dirty pages to disk
- Periodically
  1. Output the log buffers to the log
  2. Force database buffers to the disk
  3. Output an entry <checkpoint> on the log
- During recovery
  - Undo all transactions that have not committed
  - Redo all transactions that have committed after checkpoint

ira@cs.au.dk

Recovery with checkpoints



1. T1 and T3 ok, T2 redone, T4 undone
2. T1 and T3 ok, T2 and T4 undone
3. T1 ok, T2 and T3 redone, T4 undone
4. T1 ok, T3 redone, T2 and T4 undone

ira@cs.au.dk

No-Undo/Redo (deferred database modification)

- Algorithm
  - Do not output values to disk until commit log entry on stable storage
  - All writes go to log and to database cache
  - Sometime after commit, cached values are output to disk
- Advantages
  - Faster during recovery: no undo
  - No before images needed in log
- Disadvantages
  - Database outputs must wait
  - Lots of extra work at commit time

ira@cs.au.dk

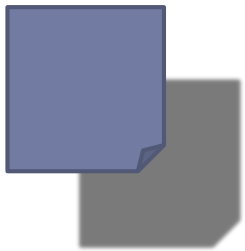
Undo/No-Redo

- Algorithm
  - All changed data items output to disk before commit
    - Requires that write entry first be output to (stable) log
  - At commit:
    - Output (*flush*) all changed data items in cache
    - Add commit entry to log
- Advantages
  - No after images are needed in log
  - No transactions need to be redone
- Disadvantages
  - Hot spot data requires a flush for each committed write
    - Implies lots of I/O traffic

ira@cs.au.dk

No-Undo/No-Redo

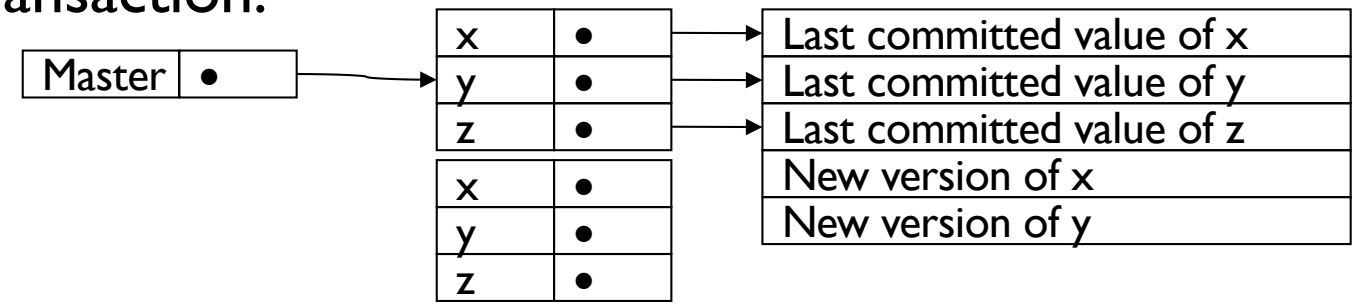
- Algorithm
  - No-undo → do not change the database during a transaction
  - No-redo → on commit, write changes to the database in a single atomic action
- Advantages
  - Recovery is instantaneous
  - No recovery code need be written
- Atomic database writes of many pages is accomplished using *shadow paging* technique



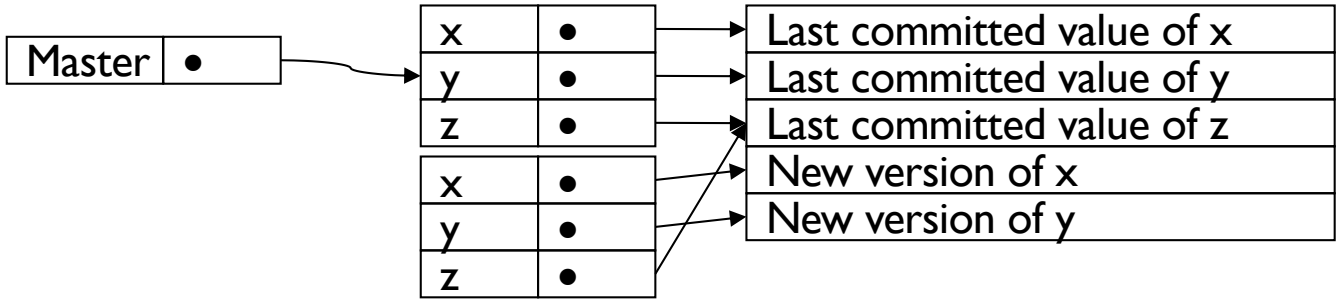
ira@cs.au.dk

No-Undo/No-Redo example

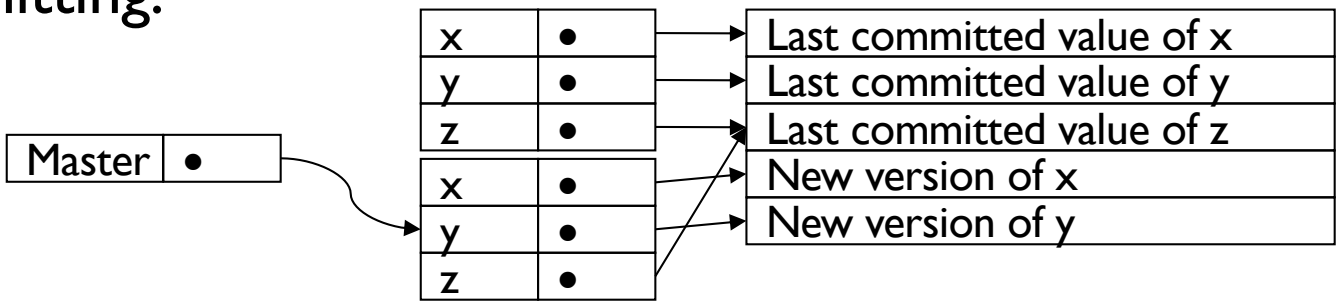
- During a transaction:



- After preparing new directory for commit:



- After committing:



ira@cs.au.dk

No-Undo/No-Redo, cont.

- Commit requires writing on disk of one bit
- Recovery is very fast: one disk read
- Problems:
  - Access to stable storage is indirect (though directories may be possibly stored in main memory)
  - Garbage collection of stable storage is required
  - Original layout of data is destroyed
  - Concurrent transactions are difficult to support

ira@cs.au.dk

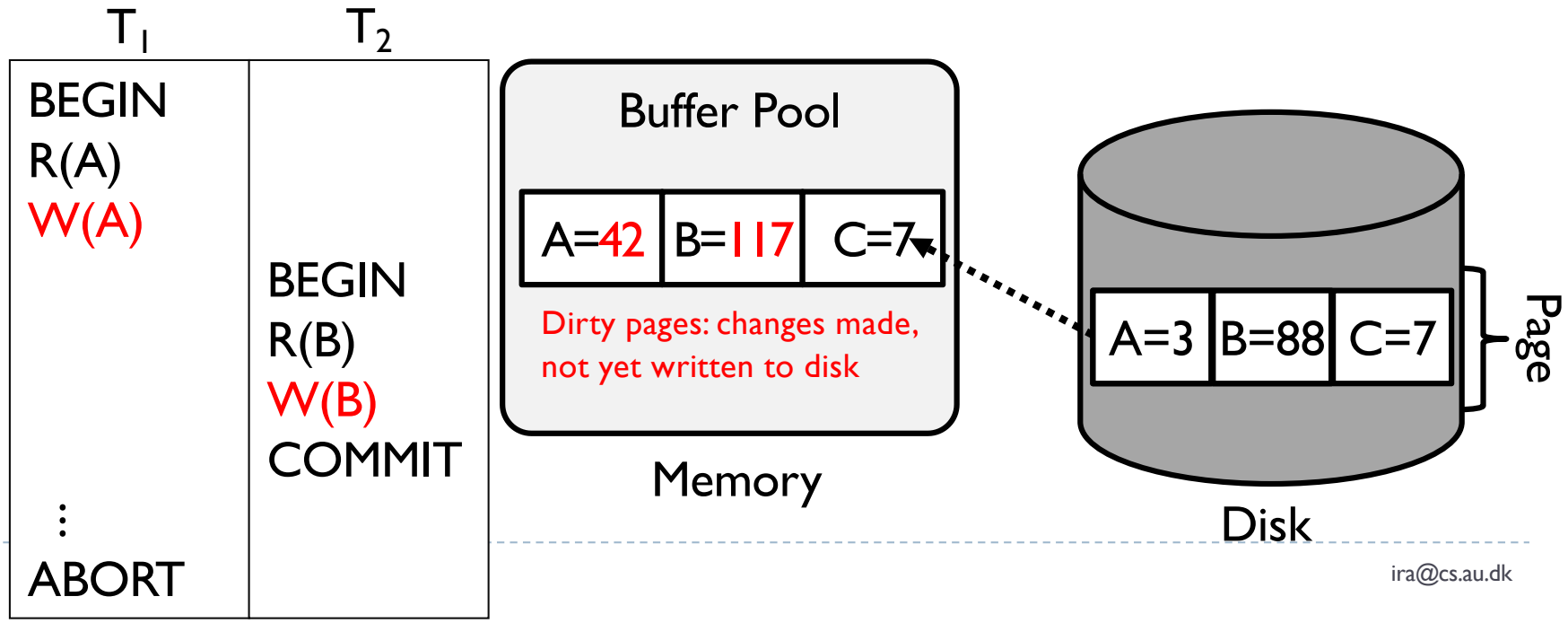
Pros/cons of strategies

- Undo/Redo: redo transactions where log has "start" and "commit", Undo transactions where log has "start" but no "commit"
  - Maximize efficiency during normal operation, some extra work required during recovery, requires before/after information in log
- No-Undo/Redo: deferrred database modification, i.e., no output on disk until commit log on stable storage
  - Faster during recovery: no undo, no before information in log, but database outputs must wait, more work at commit time
- Undo/No-Redo: changes to disk before commit, requires that write entry first be output to (stable) log
  - No after images are needed in log, no redo, but hot spot data requires a flush for each committed write (lots of I/O traffic)
- No-Undo/No-Redo: changes only on shadow pages (copies), on commit changes written to database in a single atomic action
  - Recovery instantaneous, nothing to be done, but access to stable storage is indirect, original layout of data destroyed, garbage collection required, and concurrent transactions difficult to support

ira@cs.au.dk

Strategies for Buffer Pool Management

- When are pages written to disk?
  - T1 writes A, T3 wants to load page D
    - Can we make room for D by written A back to disk?
      - Steal strategy allows this
  - T2 is committed
    - Should we ensure that page B is written back to disk before T2 leaves the system?
      - Force strategy ensures this



ira@cs.au.dk

What does undo/redo correspond to?

A. Steal/Force
B. No-Steal/No-Force
C. Steal/No-Force
D. No-Steal/Force

ira@cs.au.dk

Policies for writing pages to disk

Steal/No-Steal and Force/No-Force

- Possible ways for flushing database cache to database disk:
  1. Steal: Cache can be flushed before transaction commits
  2. No-Steal: Cache cannot be flushed before transaction commit
  3. Force: Cache is flushed (forced) to disk before commit
  4. No-Force: Cache may be flushed after transaction commits
- Correspond to different ways for handling recovery:
  - Steal/No-Force (Undo/Redo)
  - Steal/Force (Undo/No-redo)
  - No-Steal/No-Force (Redo/No-undo)
  - No-Steal/Force (No-undo/No-redo)

ira@cs.au.dk

The ARIES Recovery Algorithm

- Concrete example for the use of recovery techniques in actual database systems, provides more detailed view of the individual steps
- The ARIES system is used in many relational database related IBM products
- Based on 3 concepts:
  - WAL (Write Ahead Logging)
    - In-place updating writes to the same original location on disk
      - Most commonly used strategy (avoids shadowing)
    - Maintain "before image" in log, flush to disk before overwritten with "after image" on disk
    - Steal/no-force policy (undo/redo)
  - Repeating history during redo:
    - Retrace all actions prior to crash to reconstruct the database state up to this point
  - Logging changes during undo:
    - Avoids repeating completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

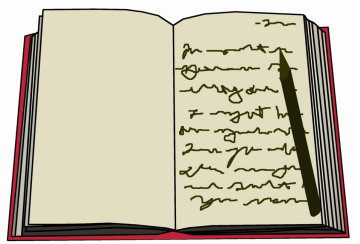
ira@cs.au.dk

ARIES log records

- Log consists of records identified by unique Log Sequence Number (LSN)
  - LSN increases monotonically and indicates disk address of the log record it is associated with
    - In addition, each data page stores LSN of the latest log record corresponding to a change for that page

Lsn	Last_lsn	Tran_id	Type	Page_id
1	0	T_1	update	*C*
2	0	T_2	update	*B*
3	1	T_1	commit	
4	begin checkpoint			
5	end checkpoint			
6	0	T_3	update	*A*
7	2	T_2	update	*C*
8	7	T_2	commit	

- A log record is written for:
  - (a) data update
  - (b) transaction commit
  - (c) transaction abort
  - (d) undo
  - (e) transaction end
  - The event is recorded as the "type" in the log record
- In case of undo a compensating log record is written



ira@cs.au.dk

ARIES log sequence numbers

- A log record stores
  - (a) the previous LSN of that transaction (Last_lsn)
    - Links the log record of each transaction (like a back pointer to previous record of same transaction); allows tracing back individual transactions among all entries
  - (b) the transaction ID (Tran_id)
  - (c) the type of log record

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	*C*	···
2	0	T_2	update	*B*	···
3	1	T_1	commit		···
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	*A*	···
7	2	T_2	update	*C*	···
8	7	T_2	commit		···

ira@cs.au.dk

ARIES log for writes

- In addition to
  1. (a) Previous LSN of transaction
  2. (b) Transaction ID
  3. (c) Type of log record
- For a write operation, also log:
  4. Page ID for the page that includes the item
  5. Length of the updated item
  6. Its offset from the beginning of the page
  7. Before image (BFIM) of the item, i.e., the value before the modification
  8. After image (AFIM) of the item, i.e., the value after the modification

ira@cs.au.dk

ARIES tables

- Transaction table
  - Contains an entry for each active transaction
  - Information such as transaction ID, transaction status and the LSN of the most recent log record for the transaction
- Dirty Page table:
  - Contains an entry for each dirty page in the buffer
  - includes page ID and LSN corresponding to the earliest update to that page

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

ira@cs.au.dk

ARIES checkpointing

- Checkpointing
  - Write begin_checkpoint record in the log
  - Write end_checkpoint record in the log
    - With this record the contents of transaction table and dirty page table are appended to the end of the log
  - Writes the LSN of the begin_checkpoint record to a special file
    - This special file is accessed during recovery to locate the last checkpoint information
- To reduce the cost of checkpointing and allow the system to continue to execute transactions, use "fuzzy checkpointing"
  - DBMS can continue to execute transactions while checkpointing

Lsn	Last_lsn	Tran_id
1	0	T_1
2	0	T_2
3	1	T_1
4	begin checkpoint	
5	end checkpoint	
6	0	T_3
7	2	T_2
8	7	T_2

ira@cs.au.dk

ARIES recovery

- The following phases are part of recovery
  - **Analysis phase**
    - Start at begin_checkpoint record and proceed to end of log
      - Access transaction table and dirty page table when encountering end_checkpoint record
        - Note that during this phase some other log records may be written to the log and transaction table may be modified
    - Analysis phase compiles the set of redo and undo to be performed
      - identifies dirty pages in buffer, and set of transactions active at time of crash; determines appropriate point in log where redo is to start
  - **Redo phase**
    - Start from point in the log up to where all dirty pages have been flushed
    - Move forward to the end of the log
      - Any change that appears in the dirty page table is redone
  - **Undo phase**
    - Start from end of log and proceed backward while performing appropriate undo
      - For each undo write a compensating record in the log
- The recovery completes at the end of undo phase

ira@cs.au.dk

ARIES example

a) Log at point of crash

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

b) Transaction and dirty page tables at time of checkpoint

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

c) Transaction and dirty page tables after analysis phase

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	7
B	2
A	6

ira@cs.au.dk

ARIES: what should happen in case of abort?

1. Write „abort" log record, undo updates in reverse order, write "end" log record
2. Write „abort" log record, undo updates in same order, write "end" log record
3. Write „abort" log record, undo updates in reverse order, adding corresponding log records, write "end" log record
4. Write „abort" log record, undo updates in the same order, adding corresponding log records, write "end" log record

Summary

- Intended learning outcomes
  - Be able to
    - Describe characteristics of different recovery strategies
    - Identify the use of recovery techniques in the ARIES system

ira@cs.au.dk

What was this all about?

Guidelines for your own review of today's session

- Recovery has to handle…
- The Undo/Redo algorithm consists of…
  - Its advantages/disadvantages are…
- Buffer pool management is concerned with…
- Policies for writing to disk are…
  - They correspond to the following redo/undo cases…
- The ARIES algorithm aims to…
  - It uses the following three concepts…
  - It maintains the following information while the transactions operate… in the following tables…
  - It works in the following steps:…

ira@cs.au.dk