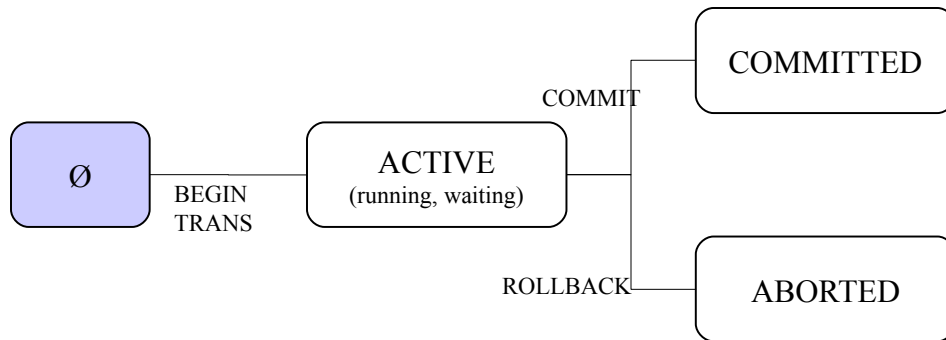


LOG TUNING

Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

Atomicity and Durability



- Every transaction either commits or aborts. It cannot change its mind
- Even in the face of failures:
 - Effects of committed transactions should be permanent;
 - Effects of aborted transactions should leave no trace.

UNSTABLE STORAGE

DATABASE BUFFER				
	P _i		P _j	



STABLE STORAGE

Failures: Crash

- Processor failure, software bug
 - Program behaves unpredictably, destroying contents of main (*volatile*) memory
 - Contents of mass store (*non-volatile memory*) generally unaffected
 - Active transactions interrupted, database left in inconsistent state
- Server supports atomicity by providing a *recovery procedure* to restore database to consistent state
 - Since rollforward is generally not feasible, recovery rolls active transactions back

Failures: Abort

- Causes:
 - User (*e.g.*, cancel button)
 - Transaction (*e.g.*, deferred constraint check)
 - System (*e.g.*, deadlock, lack of resources)
- The technique used by the recovery procedure supports atomicity
 - Roll transaction back

Failures: Media

- Durability requires that database state produced by committed transactions be preserved
- Possibility of failure of mass store implies that database state must be stored redundantly (in some form) on independent non-volatile devices

Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

Buffer Management

- Modified pages are called dirty pages
- Steal vs. No Steal
 - Steal: Dirty pages modified by non-committed transactions might be written to disk
 - No Steal: Dirty pages modified by non-committed transactions might NOT be written to disk
- Force vs. No Force
 - Force: Dirty pages modified by transaction T are forced to disk when T commits
 - No Force: Dirty pages modified by transaction T are NOT forced to disk when T commits.

Handling the Buffer Pool

	No Steal	Steal
Force	Trivial	
No Force		Desired

Handling the Buffer Pool

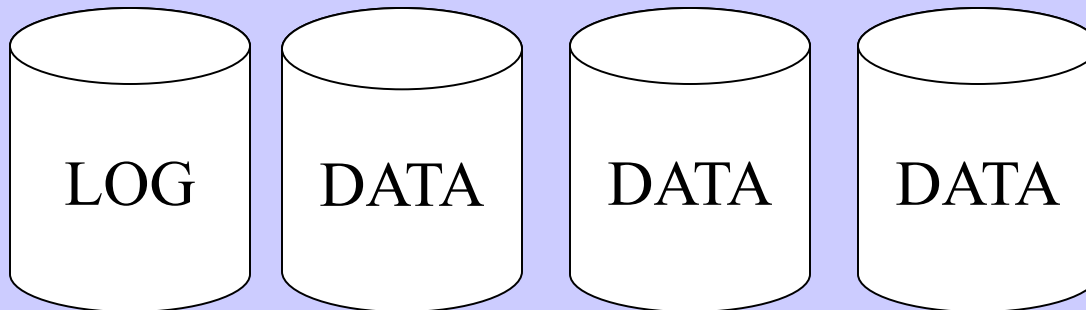
	No Steal	Steal
Force		Undo
No Force	Redo	Undo Redo

Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

UNSTABLE STORAGE

DATABASE BUFFER				
	P _i		P _j	



RECOVERY

STABLE STORAGE

Logging

Database State =

Current state of data disks + log

- How does a log look like?
- How is data written to disk?
 - Key aspect of DBMS performance
- How is the log used to guarantee atomicity and durability?
 - Recovery procedure

Log

- Sequence of records (sequential file)
 - Modified by appending (no updating)
- Contains information from which database can be reconstructed
 - Read by routines that handle abort and crash recovery

Log

- Each modification of the database causes an *update record* to be appended to log
- Update record contains:
 - Identity of data item modified
 - Identity of transaction (tid) that did the modification
 - *Before image* (undo record) – copy of data item before update occurred
 - *After image* (redo record) – copy of data item after update occurred
 - Referred to as *physical logging*

Log

	x	y	z	u	y	w	z
	T ₁	T ₁	T ₂	T ₃	T ₁	T ₄	T ₂
	17	A	2.4	18	ab	3	4.5

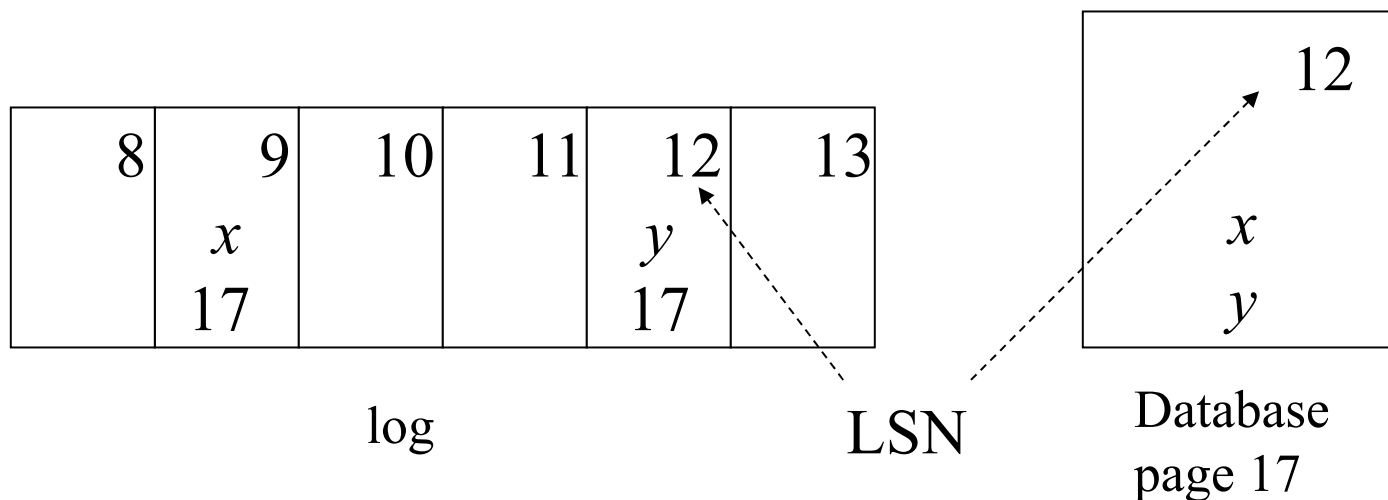
*most recent
database update*

Update records in a log

<XID, pageID, offset, length, old data, new data>

Log Sequence Number (LSN)

- Log records are numbered sequentially
 - Concurrency control is in effect
 - LSN corresponds to schedule time stamp
- Each database page contains the LSN of the update record describing the most recent update of any item in the page



Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

1. Must force the log record for an update *before* the corresponding data page gets to disk.

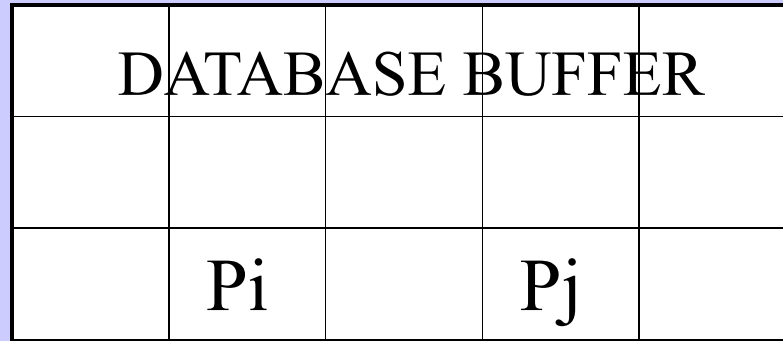
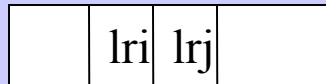
(STEAL)

2. Must write all log records for a Transaction *before commit*.

(NO FORCE)

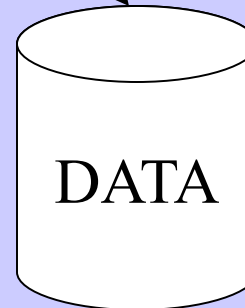
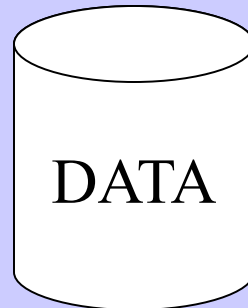
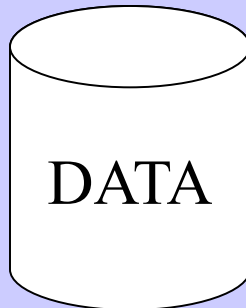
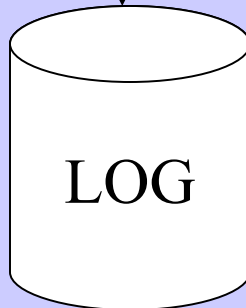
UNSTABLE STORAGE

LOG BUFFER



WRITE
log records before commit

WRITE
modified pages after commit



RECOVERY

STABLE STORAGE

Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

Transaction Abort Using Log

- Scan log backwards using tid to identify transaction's update records
 - Reverse each update using before image
 - Reversal done in last-in-first-out order
- In a strict system modified (new) values unavailable to concurrent transactions (as a result of long term exclusive write locks); hence rollback makes transaction atomic
- **Problem:** terminating scan (log can be long)
- **Solution:** append a *begin record* for each transaction, containing tid, prior to its first update record

Transaction Abort Using Log

	B	U	U	U	U	U	U	U
		<i>x</i>	<i>y</i>	<i>z</i>	<i>u</i>	<i>y</i>	<i>w</i>	<i>z</i>
	T ₁	T ₁	T ₁	T ₂	T ₃	T ₁	T ₄	T ₂
		17	A	2.4	18	ab	3	4.5

↑
abort T₁

Key:

B – begin record

U – update record

- **Abort Procedure:** Scan back to begin record using update records to reverse changes

Crash Recovery Using Log

- Abort all transactions active at time of crash
(STEAL/FORCE)
- **Problem:** How do you identify them?
- **Solution:** *abort record* or *commit record* appended to log when transaction terminates
- **Recovery Procedure:**
 - Scan log backwards - if T's first record is an update record, T was active at time of crash. Roll it back
 - *A transaction is not committed until its commit record is in the log*

Crash Recovery Using Log

	B	U	U	U	U	U	C	U	A	U
		<i>x</i>	<i>y</i>	<i>z</i>	<i>u</i>	<i>y</i>		<i>w</i>		<i>z</i>
	T ₁	T ₁	T ₁	T ₂	T ₃	T ₁	T ₃	T ₄	T ₁	T ₂
		17	A	2.4	18	ab		3		4.5

↑
crash

Key:

B – begin record

U – update record

C – commit record

A – abort record

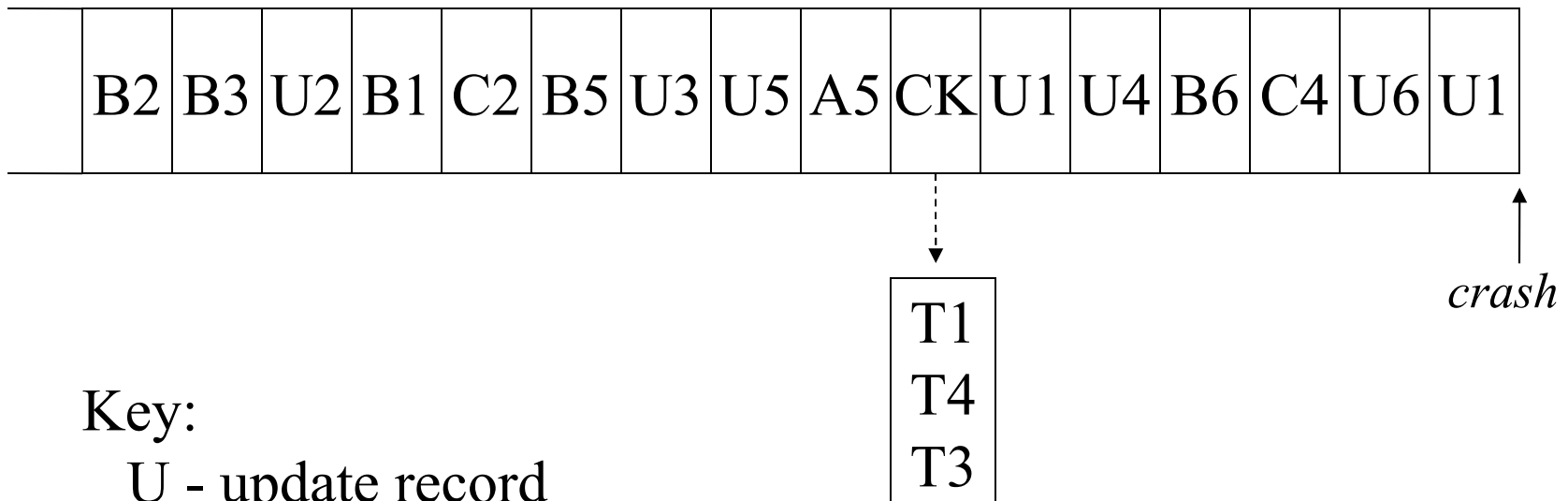
- T₁ and T₃ were not active at time of crash

Crash Recovery Using Log

- **Problem:** Scan must retrace entire log
- **Solution:** Periodically append *checkpoint record* to log. Contains tid's of all active transactions at time of append
 - Backward scan goes at least as far as last checkpoint record appended
 - Transactions active at time of crash determined from log suffix that includes last checkpoint record
 - Scan continues until those transactions have been rolled back

Example

← Backward scan



Key:

U - update record

B - begin record

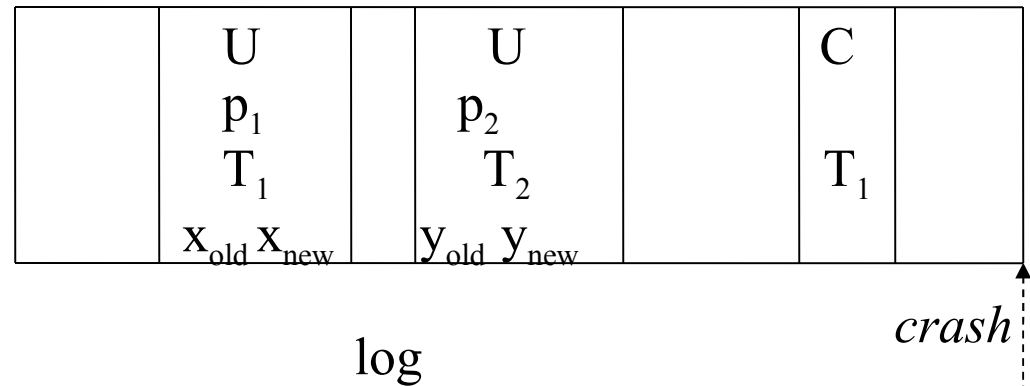
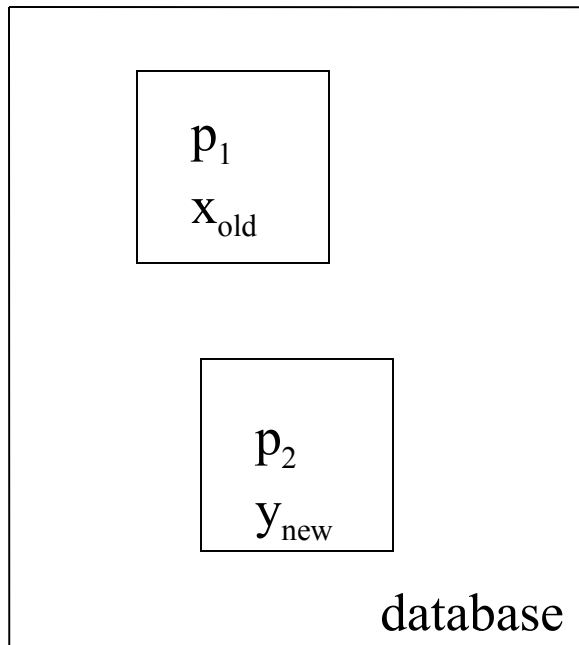
C - commit record

A - abort record

CK - checkpoint record

T_1 , T_3 and T_6 active
at time of crash

Recovery With Steal/No-Force



crash

T₁ committed
 T₂ active
 p₂ flushed
 p₁ not flushed

p₁ must be rolled forward using x_{new}

p₂ must be rolled back using y_{old}

Sharp Checkpoint

- **Problem:** How far back must log be scanned in order to find update records of committed transactions that must be rolled forward?
- **Solution:** Before appending a checkpoint record, CK, to log buffer, halt processing and force all dirty pages from cache
 - Recovery process can assume that all updates in records prior to CK were written to database (only updates in records after CK *might* not be in database)

Recovery with Sharp Checkpoint

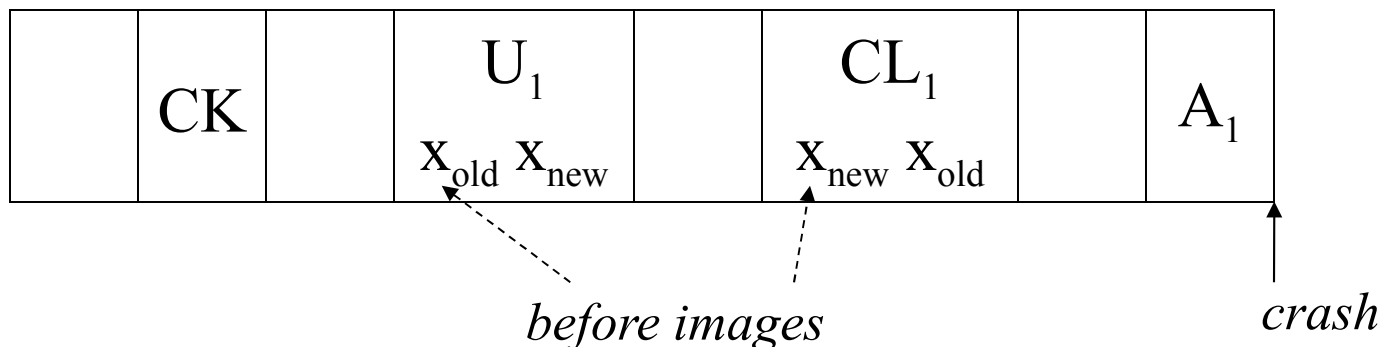
- **Pass 1:** Log is scanned backward to most recent checkpoint record, CK, *to identify transactions active at time of crash.*
- **Pass 2:** Log is scanned forward from CK to most recent record. The *after images* in *all* update records are used *to roll the database forward.*
- **Pass 3:** Log is scanned backwards to begin record of oldest transaction active at time of crash. The *before images* in the update records of these transactions are used *to roll these transactions back.*

Recovery with Sharp Checkpoint

- **Issue 1:** Database pages containing items updated after CK was appended to log *might* have been flushed before crash
 - No problem – with *physical* logging, roll forward using after images in pass 2 is *idempotent*.
 - Rollforward in this case is unnecessary, but not harmful

Recovery with Sharp Checkpoint

- **Issue 2:** Some update records after CK might belong to an aborted transaction, T_1 . These updates will not be rolled back in pass 3 since T_1 was not active at time of crash
 - Treat rollback operations for aborting T_1 as ordinary updates and append *compensating log records* to log



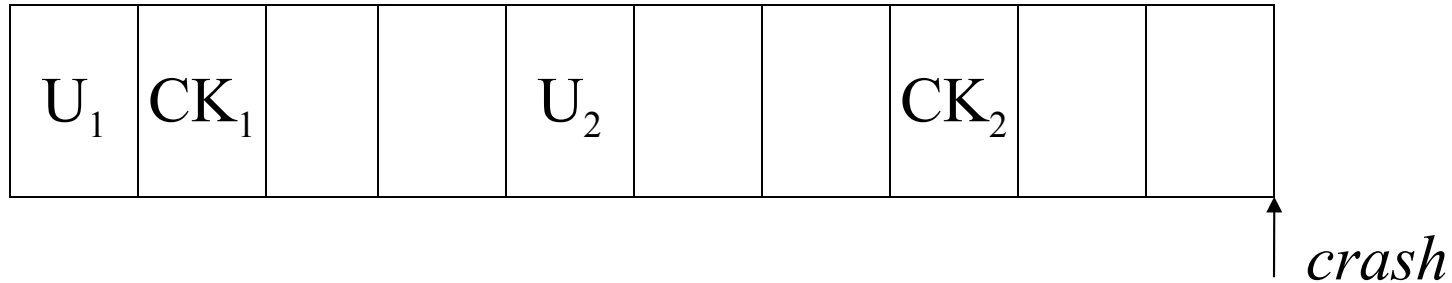
Recovery with Sharp Checkpoint

- **Issue 3:** What if system crashes during recovery?
 - Recovery is restarted
 - If physical logging is used, pass 2 and pass 3 operations are idempotent and hence can be redone

Fuzzy Checkpoints

- **Problem:** Cannot stop the system to take sharp checkpoint (write dirty pages).
 - Use *fuzzy checkpoint*: Before writing CK, record the identity of all dirty pages (do not flush them) in volatile memory
 - All recorded pages must be flushed before next checkpoint record is appended to log buffer

Fuzzy Checkpoints



- Page corresponding to U_1 is recorded at CK_1 and will have been flushed by CK_2
- Page corresponding to U_2 is recorded at CK_2 , but *might* not have been flushed at time of crash
 - Pass 2 must start at CK_1

Logical Logging

- **Problem:** With physical logging, simple database updates can result in multiple update records with large before and after images
 - **Example** – “insert t in T ” might cause reorganization of a data page and an index page for each index. Before and after images might be entire pages
- **Solution:** Log the operation and its inverse instead of before and after images
 - **Example** - store “insert t in T ”, “delete t from T ” in update record

Logical Logging

- **Problem 1:** Logical operations might not be idempotent (e.g., “UPDATE T SET x = x+5”)
 - Pass 2 roll forward does not work (it makes a difference whether the page on mass store was updated before the crash or after the crash)
- **Solution:** Do not apply operation in update record i to database item in page P during pass 2 if $P.LSN > i$

Logical Logging

- **Problem 2: Operations are not atomic**
 - A crash during the execution of a non-atomic operation can leave the database in a *physically* inconsistent state
 - **Example** - “insert t in T ” requires an update to both a data and an index page. A crash might occur after t has been inserted in T but before the index has been updated
 - Applying a logical redo operation in pass 2 to a physically inconsistent state is not likely to work
 - **Example** - There might be two copies of t in T after pass 2

Physiological Logging

- **Solution:** Use *physical-to-a-page, logical-within-a-page logging* (physiological logging)
 - A logical operation involving multiple pages is broken into multiple logical mini-operations
 - Each mini-operation is confined to a single page and hence is atomic
 - **Example** - “insert t in T ” becomes “insert t in a page of T ” and “insert pointer to t in a page of index”
 - Each mini-operation gets a separate log record
 - Since mini-operations are not idempotent, use LSN check before applying operation in pass 2

Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

ARIES

- Steal NoForce
- 3 Phases crash recovery
- Fuzzy checkpoints
- Physiological logging

ARIES algorithms, developed by C.Mohan at IBM Almaden in the early 90's

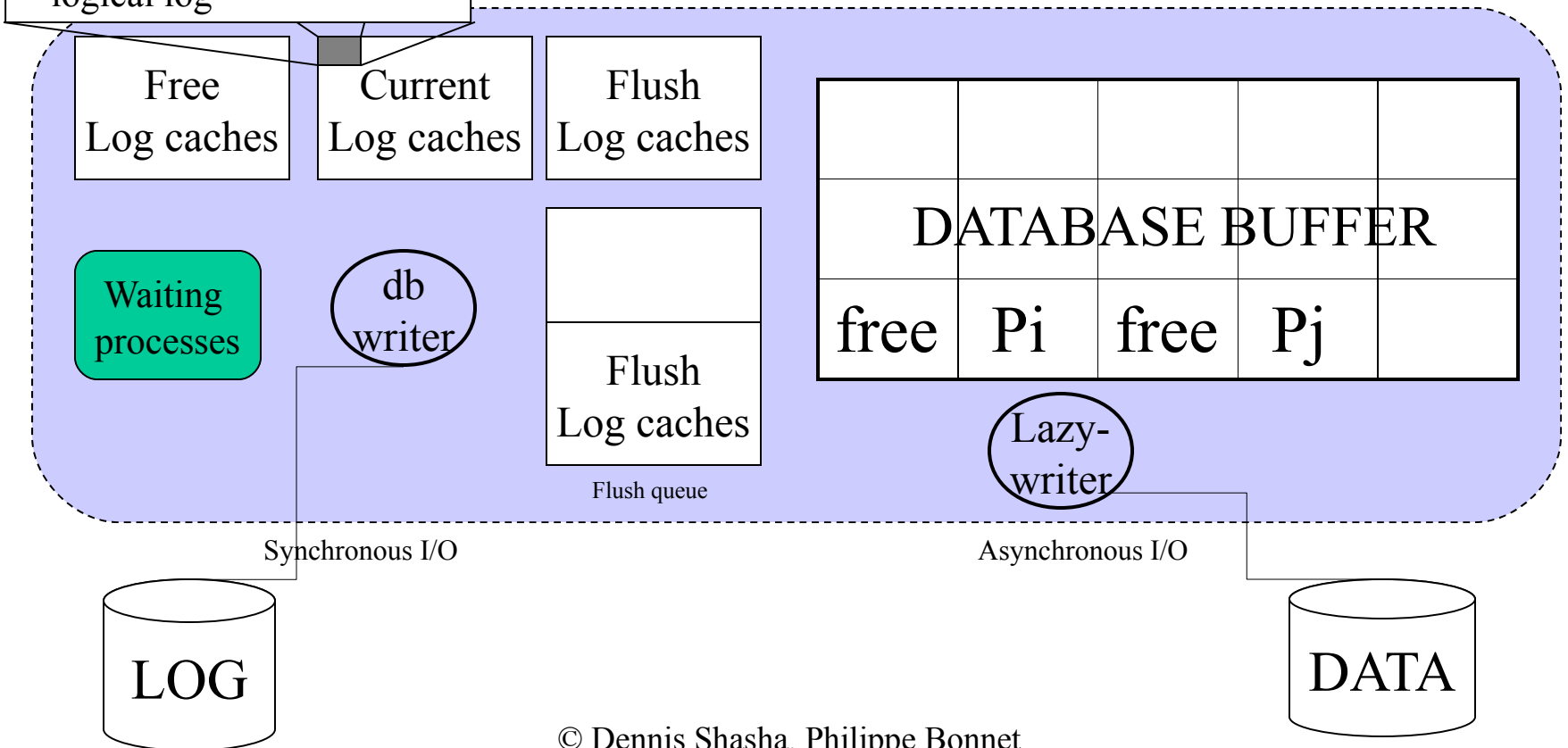
http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

Logging in SQL Server

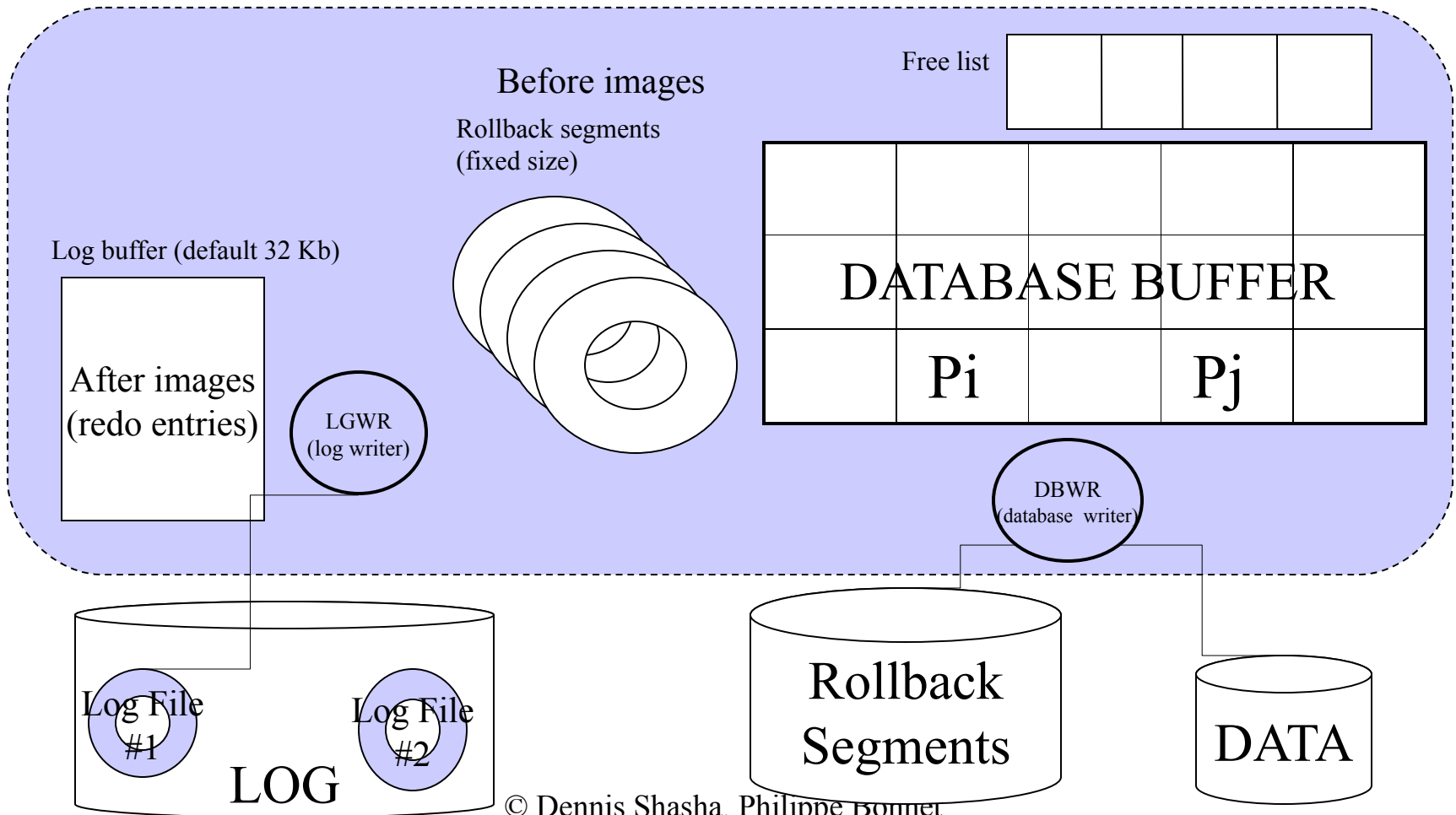
Log entries:

- LSN
- before and after images or logical log

DB2 UDB uses a similar scheme



Logging in Oracle



Agenda

- Atomicity and Durability
- Handling the Buffer Pool
- Logging
 - Log records
 - Writing to the log
 - Recovery
 - ARIES
- Tuning the writes

Put the Log on a Separate Disk

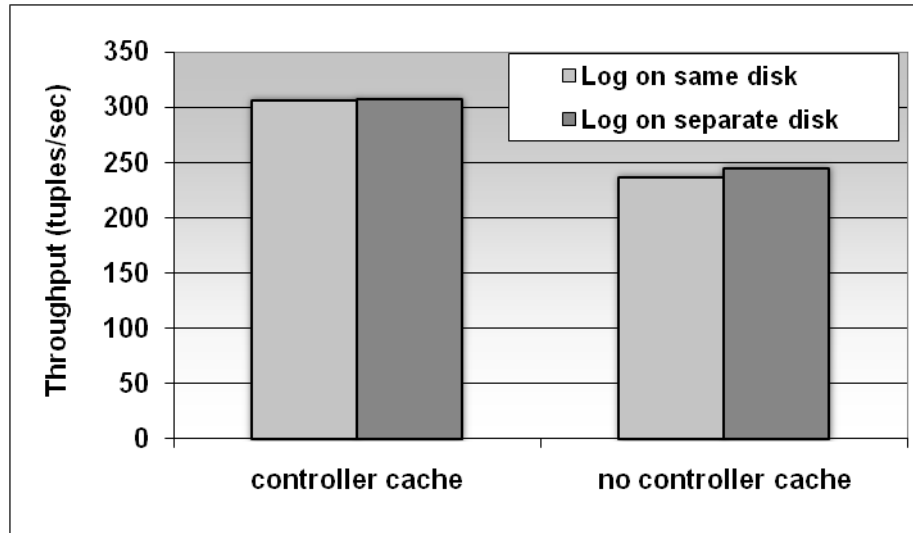
- Writes to log occur sequentially
- Writes to disk occur (at least) 100 times faster when they occur sequentially than when they occur randomly

A disk that has the log should have no other data

+ sequential I/O

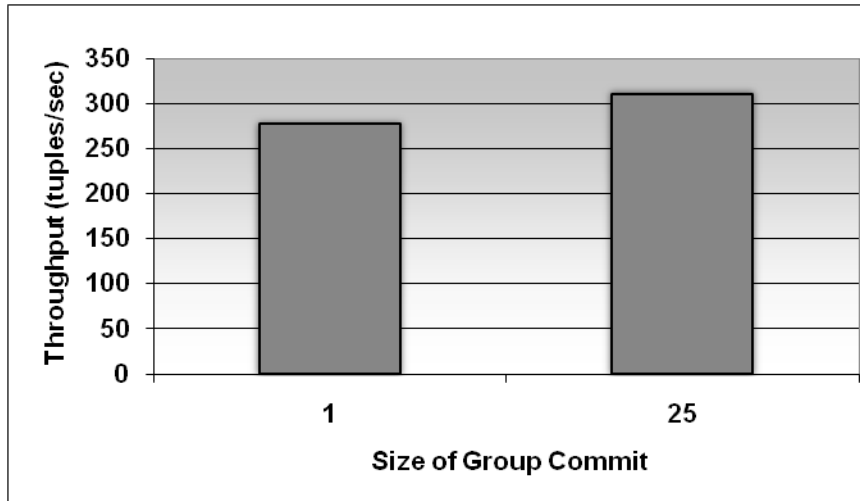
+ log failure independent of database failure

Put the Log on a Separate Disk



- 300 000 transactions. Each contains an insert statement.
 - DB2 UDB v7.1
- 5 % performance improvement if log is located on a different disk
- Controller cache hides negative impact
 - mid-range server, with Adaptec RAID controller (80Mb RAM) and 2x18Gb disk drives.

Group Commits

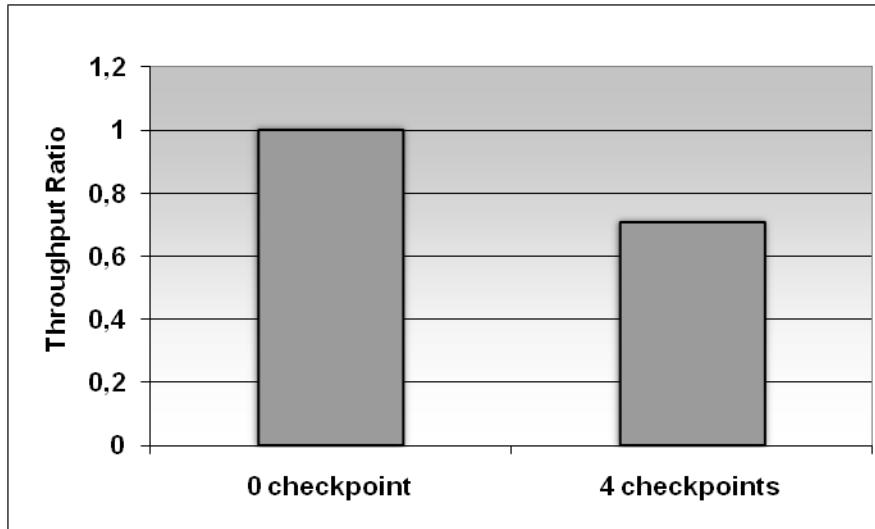


- 300 000 transactions. Each contains an insert statement.
 - DB2 UDB v7.1
- Log records of many transactions are written together
 - Increases throughput by reducing the number of writes
 - at the cost of increased mean response time.

Tuning Database Writes

- Dirty data is written to disk
 - When the number of dirty pages is greater than a given parameter (Oracle 8)
 - When the number of dirty pages crosses a given threshold (less than 3% of free pages in the database buffer for SQL Server 7)
 - When a checkpoint is performed
 - At regular intervals
 - When the log is full (Oracle 8).

Tune Checkpoint Intervals



- A checkpoint (partial flush of dirty pages to disk) occurs at regular intervals or when the log is full:
 - Impacts the performance of on-line processing
 - + Reduces the size of log
 - + Reduces time to recover from a crash
- 300 000 transactions. Each contains an insert statement.
 - Oracle 8i for Windows 2000

Reduce the Size of Large Update Transactions

- Consider an update-intensive batch transaction (concurrent access is not an issue):

It can be broken up in short transactions (mini-batch):

- + Easy to recover
- + Does not overflow the log buffers

Example: Transaction that updates, in sorted order, all accounts that had activity on them, in a given day.

Break-up to mini-batches each of which access 10,000 accounts and then updates a global counter.