COURSE 3

Database Recovery

Isolation Levels in SQL Server

- SQL syntax:

```
SET TRANSACTION ISOLATION LEVEL …
```

Isolation Levels in SQL Server

- **`READ UNCOMMITTED`**: no lock when reading;
- **`READ COMMITTED`**: holds locks during statement execution (default) (*Dirty reads*);
- **`REPEATABLE READ`**: holds locks for the duration of transaction (*Unrepeatable reads*);
- **`SERIALIZABLE`**: holds locks and key range locks during entire transaction (*Phantom reads*);
- **`SNAPSHOT`**: work on data snapshot

Degrees of isolation

Degree	0	1	2	2.999	3
Common Name	Chaos	Read Uncommitted	Read Committed	Repeatable Read	Serializable
AKA		Browse	Cursor Stability		Isolated
Lost Updates?	Yes	No	No	No	No
Dirty Reads?	Yes	Yes	No	No	No
Unrepeatable Reads?	Yes	Yes	Yes	No	No
Phantoms?	Yes	Yes	Yes	Yes	No

Recovery and the ACID Properties

The **Recovery Manager** is responsible for :

Atomicity and Durability.

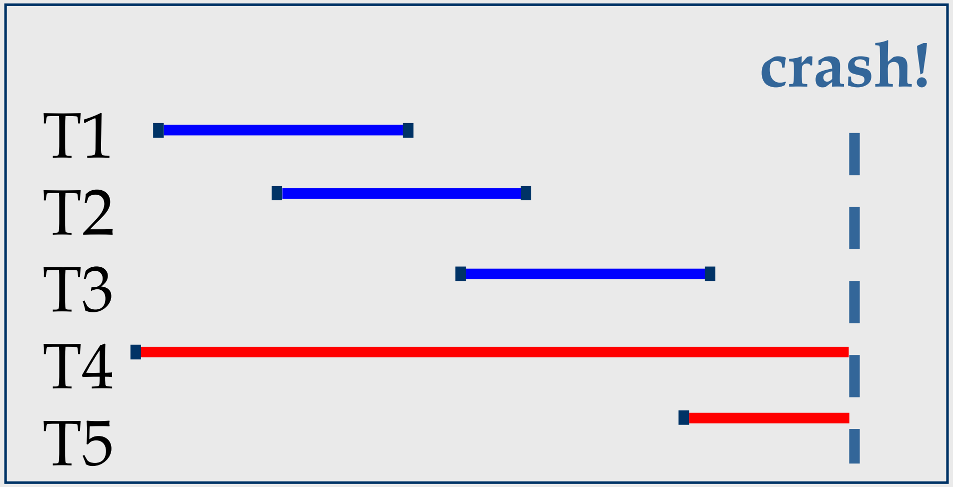
- Atomicity is guaranteed by undoing the actions of the transactions that did not commit.
- Durability is guaranteed by making sure that all actions of committed transactions survive crashes and failures.

Motivation

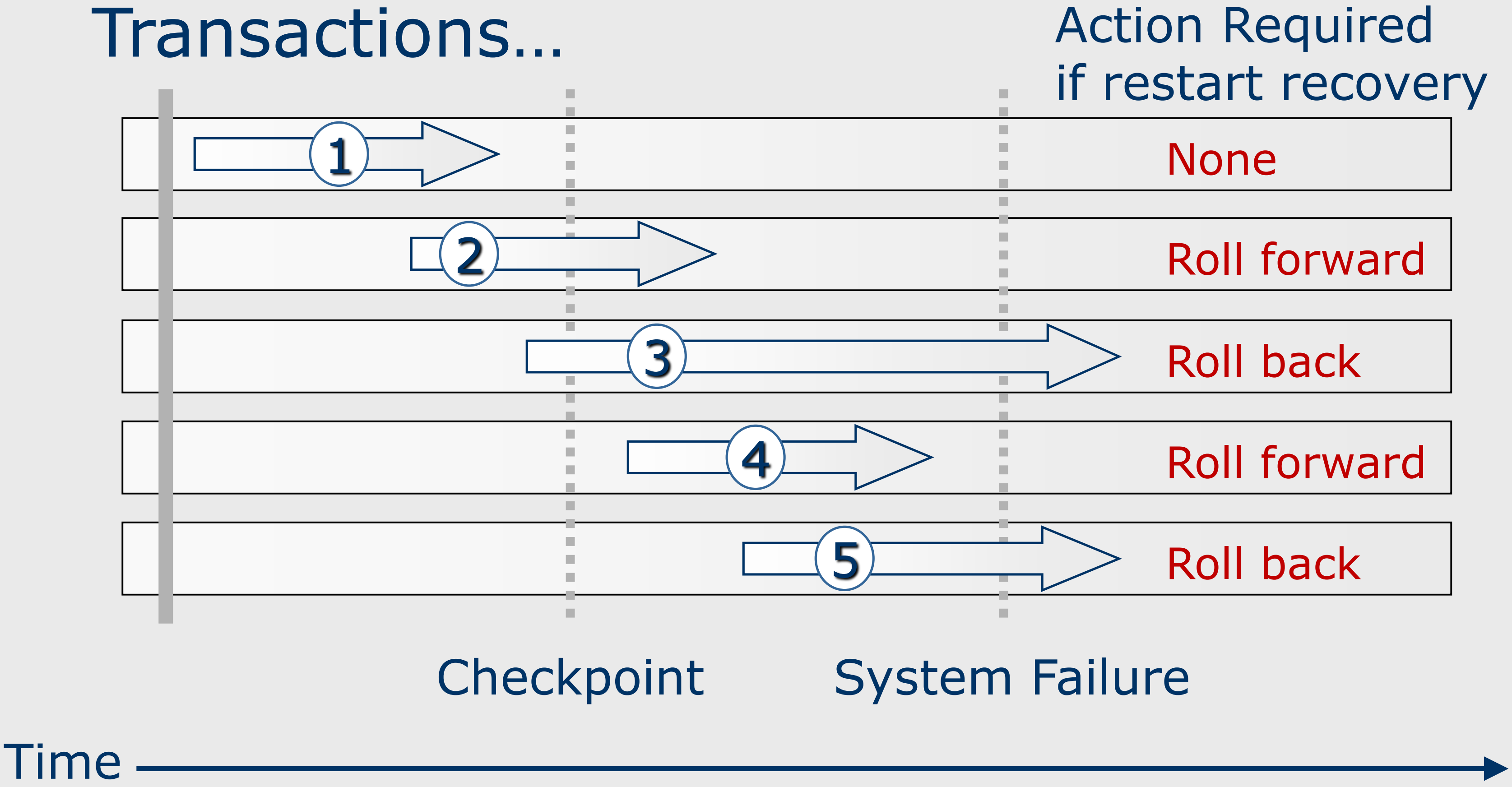
- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

Desired Behavior after system restarts:

- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



Another Motivating Example



Transactions…

Action Required if restart recovery

None

Roll forward

Roll back

Roll forward

Roll back

Checkpoint

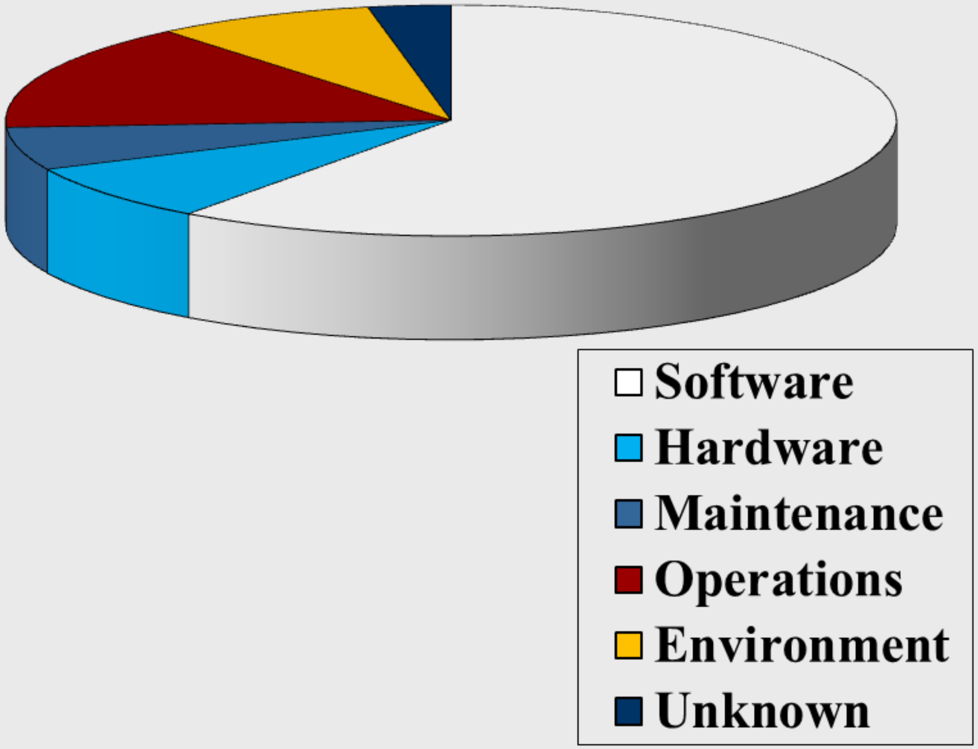
System Failure

Time

Types of Failures

- **System crashes**: due to hardware or software errors, resulting in loss of main memory
- **Media failures**: due to problems with disk head or unreadable media, resulting in loss of parts of secondary storage.
- **Application errors**: due to logical errors in the program which may cause transactions to fail.
- **Natural disasters**: physical loss of media (fire, flood, earthquakes, terrorism, etc.)
- **Sabotage**: intentional corruption or destruction of data
- **Carelessness**: unintentional destruction of data by user or operator.

Failures Impact



General Failure Categories

(1) Transaction failures

- Transaction aborts (unilaterally or due to deadlock)
- 3% of transaction abort abnormally on average (bad input, data not found, overflow, res. limit exceeded)

(2) System failures

- Failure of processor, main memory, power supply…
- Main memory contents are lost but not secondary storage

(3) Media failures

- Failure of secondary storage
- Head crash or controller failure

Recovery

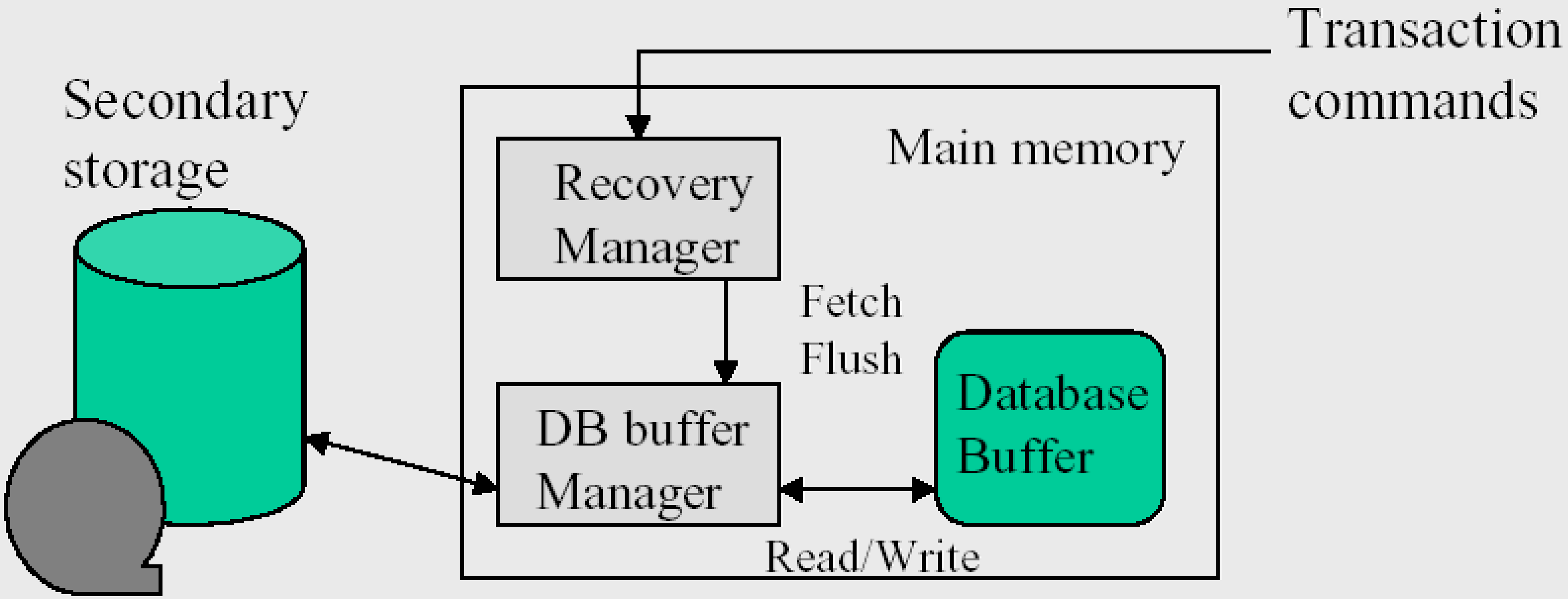
- For non-catastrophic failures (category 1 or 2):
  - Use transaction log to bring database into a consistent state (the way it was before failure)
  - Reverse any changes by **undoing** some operations (e.g., transaction did not commit yet)
  - **Redo** some operations to restore data (e.g., transaction committed but not all of the data was written out to disk yet)
- Catastrophic failure (category 3):
  - Use archival backup to **restore** past copy of database
  - Reconstruct most recent consistent state by **redoing** operations of committed operations from the log (backed up)

Data Recovery for local DB Assumptions:

- Concurrency control is in effect.
  - Strict 2PL, in particular.
- Updates are happening “in place”.
  - i.e. data is overwritten on (deleted from) the disk.

- Is there a simple scheme to guarantee Atomicity & Durability?

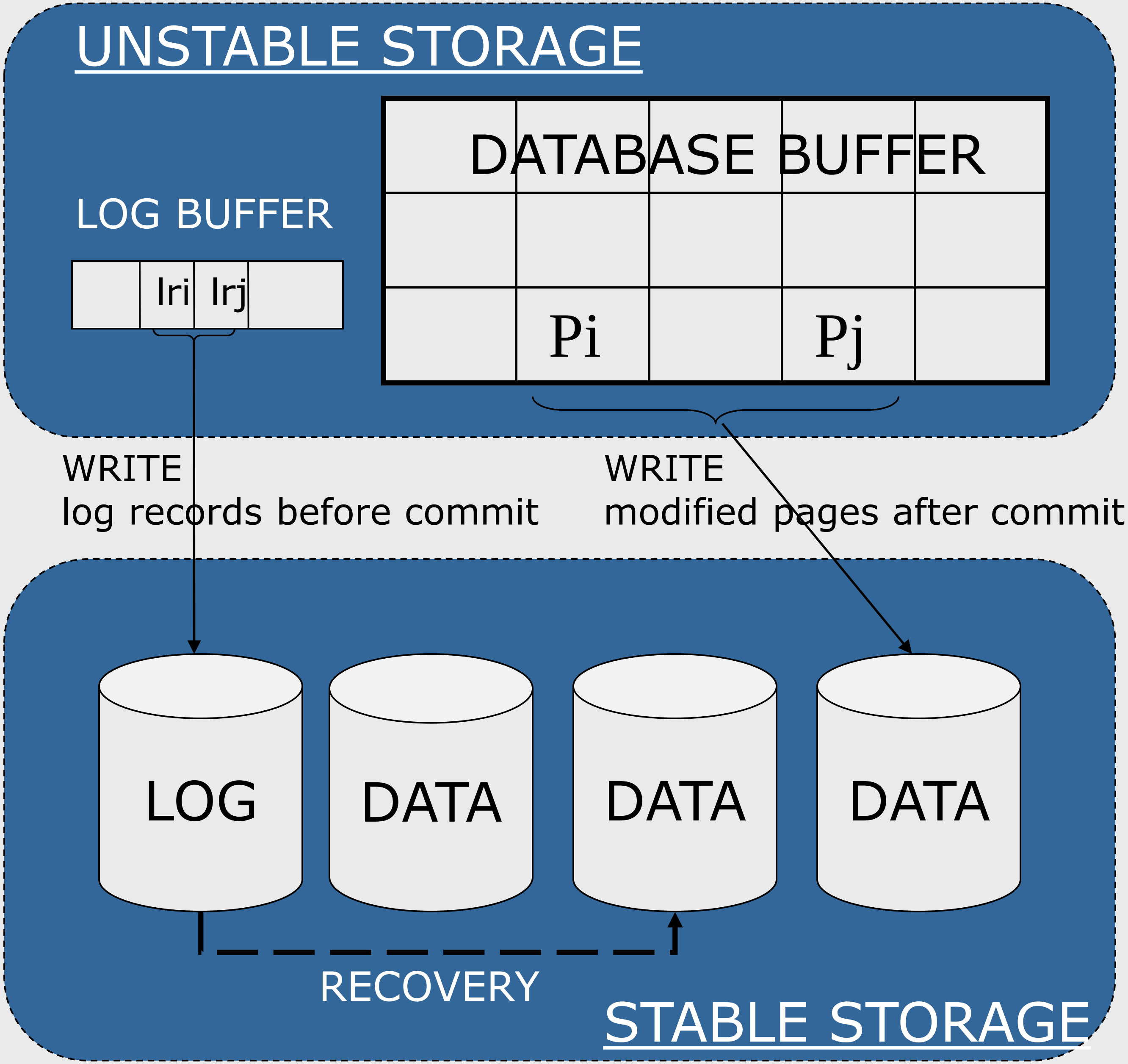
Recovery Manager



- Volatile storage: main memory (DB buffer)
- Stable storage: disks and other media. Resilient to failure and loses its content only in the presence of media failure or intentional attacks (Database and logs)

Action Logging

- Every update action of a transaction must also write a log entry to an append-only file.
- No record needs to be appended in the log if the action merely reads the database
- Why do we need a log? The log is consulted by the system to achieve both atomicity and durability.
- The log is generally stored on a different mass storage device than the database.



UNSTABLE STORAGE

LOG BUFFER

lri lrj

DATABASE BUFFER

Pi

Pj

WRITE
log records before commit

WRITE
modified pages after commit

LOG

DATA

DATA

DATA

RECOVERY

STABLE STORAGE

Database Log

- The log contains records (or entries) that are appended.
- For recovery, the log is read backwards
- An entry in the log contains:
  - A transaction identifier
  - Type of operation
  - Objects accessed to perform action
  - Old value of object (before image)
  - New value of object (after image)
  - …
- Log entries (also called update records) are used to undo changes in case of aborts: using transaction ID replace object value in database with before image

Other Entries in Log

- Log may also contain action such as *begin-transaction, commit-transaction, abort-transaction*.
- If a transaction T is aborted, then rollback → scan log backwards and when T's update records are encountered, the before image is written to DB undoing the change.
  - However, how long do we have to scan backwards looking for T's entries in the log? → we need the ***begin-transaction*** entry indicating where to stop scanning for a transaction actions.
- In a rollback due to a crash, the system needs to distinguish between the transactions that completed and the still active transactions → **commit** and **abort** entries.

Problems With Log

- Does the commit request guarantee durability?
- If a crash occurs after the transaction has made changes and requested the commit, but before the commit record is written in log, the transaction will be aborted by the recovery procedure and the changes are not durable.

- What are the different scenarios?

Making Changes in Database

- Transaction T made changes to x. There is use of DB buffer. There is a crash while operations are performed.
- **Scenario1**: Neither operations made it to secondary storage → No problem. T is aborted.
- **Scenario2**: x is updated on disk but crash occurred before log is updated → No way to rollback since we don't have before image → inconsistent state.
- **Scenario3**: The update made it to the log but the changes to x on disk didn't make it → T aborted and before image is used to overwrite old value → no problem.
- What do we learn from these scenarios?

Write-Ahead Logging (WAL)

- The update record must always be appended to the log before the database is updated. The log is referred to as a ***write-ahead log***.
- The **Write-Ahead Logging** Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to disk.
  2. Must write all log records for a transaction *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.
- Exactly how is logging (and recovery!) done? We'll study the ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) algorithms.

Checkpointing

- In case of a crash, the recovery procedure needs to identify transaction that are still active in order to abort them. This is done by scanning the log backwards.
- How far do we have to scan the log before stopping and guaranteeing that there is not transaction still active?
- To avoid a complete backward scan of the log during recovery, we must include a mechanism to specify where to stop.
- New entry in the log called ***Checkpoint.***

Checkpoints

- The system periodically appends a checkpoint record to the log that lists the current active transactions.
- The recovery process must scan backward at least as far as the most recent checkpoint.
- If T is named in the checkpoint, then T was still active during crash → continue scan backward until *begin-transaction* T.

Checkpoints (cont.)

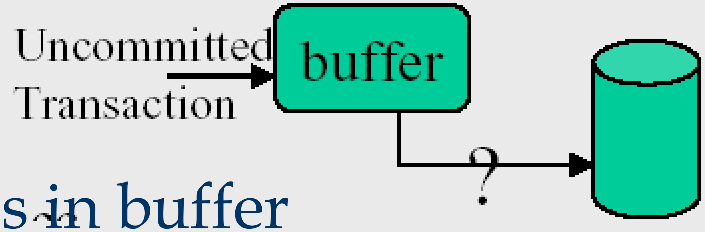
- Actions
  - Suspend execution of all transactions
  - Force-write all main memory buffers that have been modified to disk
  - Write checkpoint record to log, force-write log to disk
  - Resume execution of transactions
- Consequences?
  - If transaction has a commit before checkpoint, no need to redo write operations
- How often to do a checkpoint?
  - Every *m* minutes or *t* transactions

A Few Words on Buffer Management

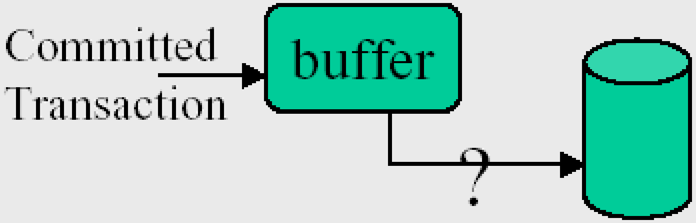
- Data access in terms of disk blocks
  - Each block must be brought into buffer pool if not already there
- May require replacing existing pages if pool is full
  - Remember replacement strategies?
- Use *dirty bit* to determine if page contents have been modified
  - Write back if dirty bit is set, o.w. discard
- Use *pin/unpin bit* to determine if page is candidate for replacement
- Two main strategies for flushing modified buffers back to disk
  - *In-place updating:* data is written back to same original disk location
  - *Shadowing:* before and after image of data on disk (BFIM/AFIM)

Recovery Manager ⇔ Buffer Manager

- Can a Buffer Manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for Recovery Manager to instruct it?
  - **Steal / no-steal** decision
  - No-steal means RM pins pages in buffer



- Does the Recovery Manager force the Buffer Manager to write certain buffer pages in stable database at the end of a transaction's execution?
  - **Force / no-force** decision



Handling the Buffer Pool

- Steal buffer-pool frames from uncommited transactions?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?
- Force every write to disk?
  - Poor response time.
  - But provides durability.

	No Steal	Steal
Force	Trivial	
No Force		Desired

Possible Execution Strategies

- **Steal / No-force**
    - BM may have written some of the updated pages into disk. RM writes a commit
- Steal / force
    - BM may have written some of the updated pages into disk. RM issues a *flush* and writes a commit
- No-steal / no-force
    - None of the updated pages have been written. RM writes a commit and sends unpin to BM for all pinned pages.
- No-steal / force
    - None of the updated pages have been written. RM issues a *flush* and writes a commit

More on Steal and Force

- **STEAL** (why enforcing *Atomicity* is hard)
  - *To steal frame F:* Current page in F (say P) is written to disk; some transaction holds lock on P.
    - What if the transaction with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P).

- **NO FORCE** (why enforcing *Durability* is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.