Ch 18. Crash Recovery

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

SKKU VLDB Lab.

(http://vldb.skku.ac.kr/)

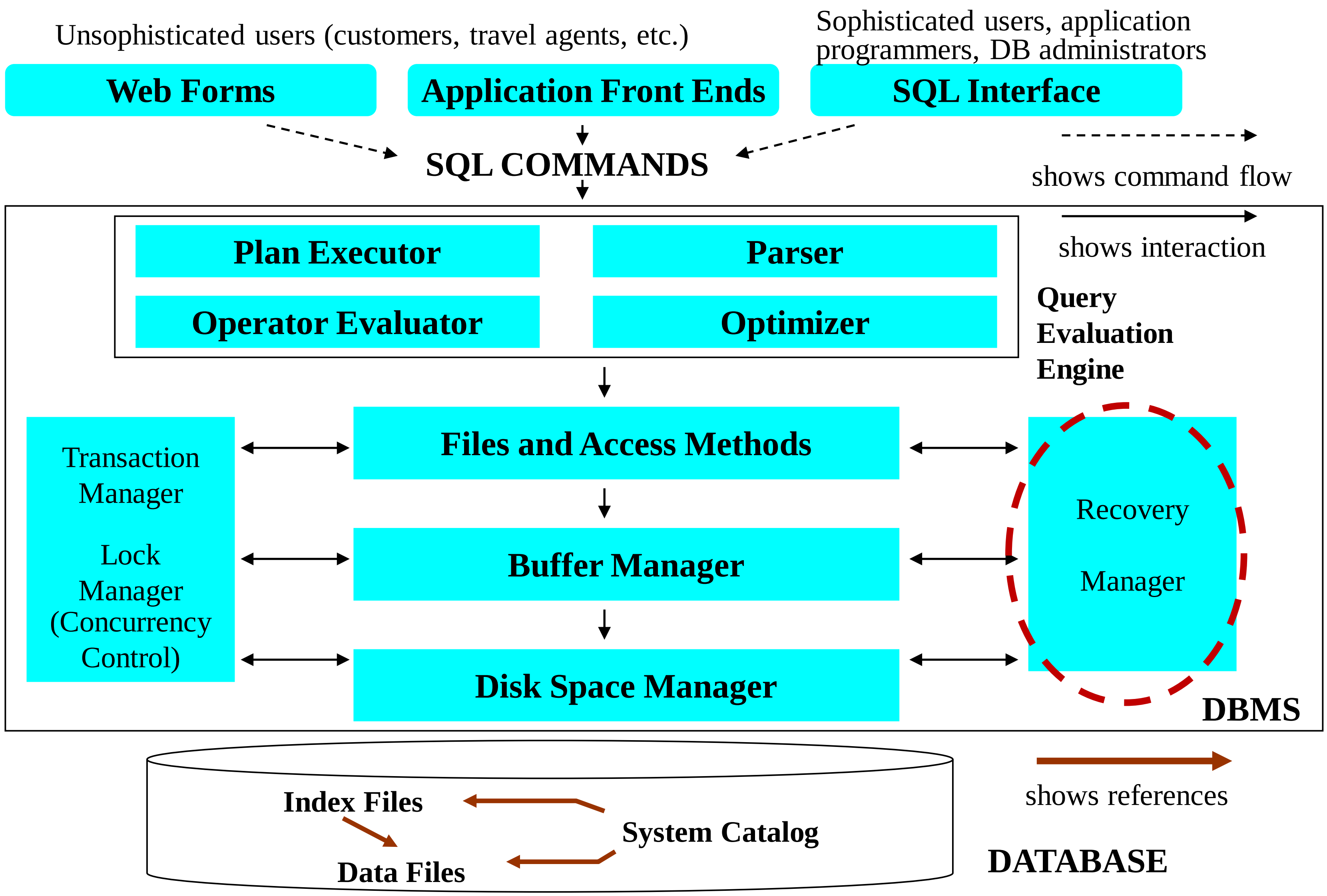


Figure 1.3 Anatomy of an RDBMS

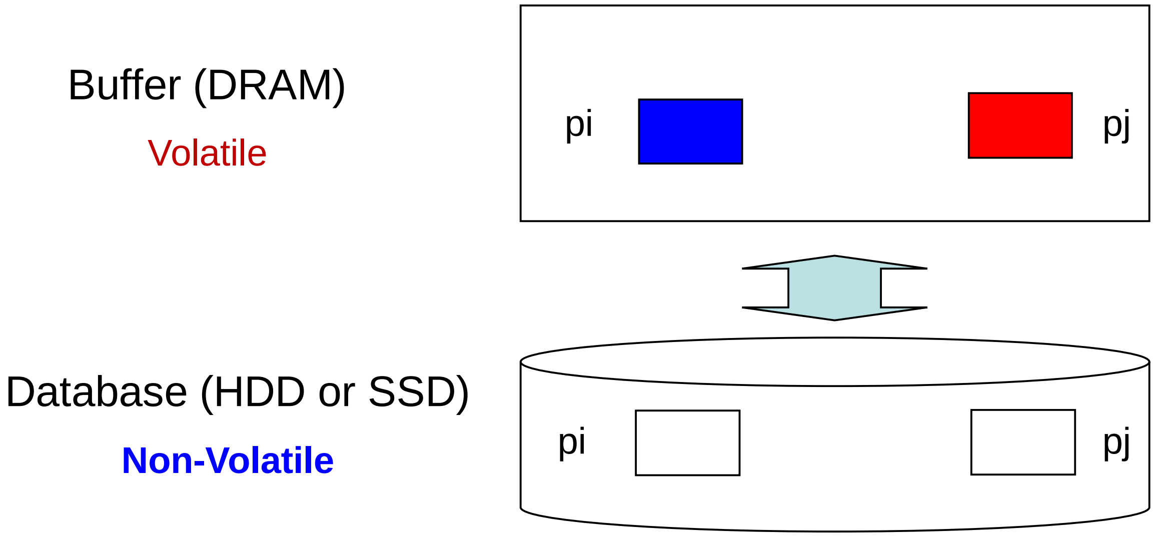
The ACID Properties

- **A**tomicity: All actions in the Xact happen, or none happen.
- **C**onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- **I**solation: Execution of one Xact is isolated from that of other Xacts.
- **D**urability: If a Xact commits, its effects persist.

The **Recovery Manager** guarantees Atomicity & Durability.

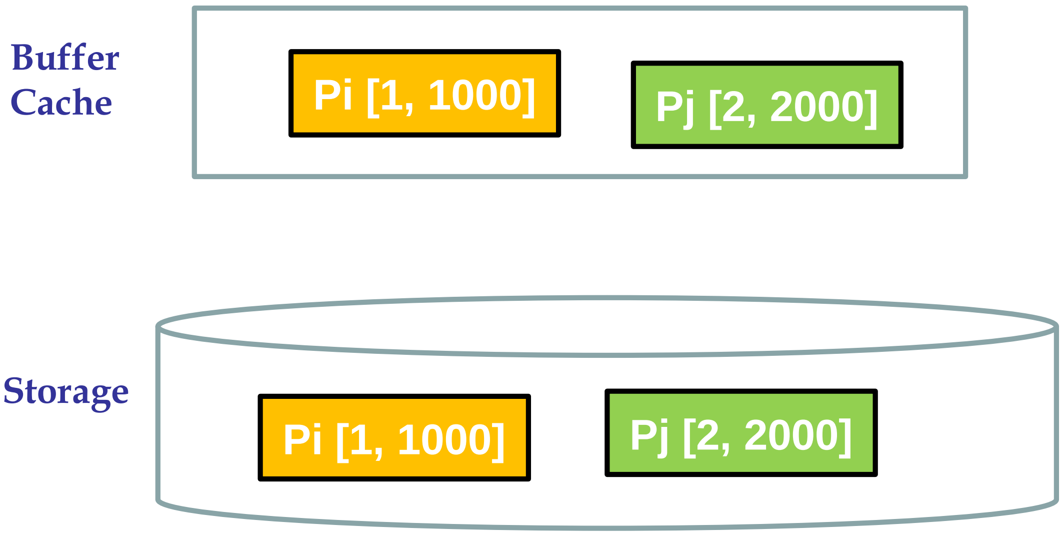
Example: Transactional Atomicity and Durability

- Money transfer
  - BeginTX; A = A – 100; B = B + 100; Commit (or Rollback or Failure)
  - BeginTX; Read A; Update A = A – 100; Read B; Update B = B + 100; ..



Various Failures and Recovery

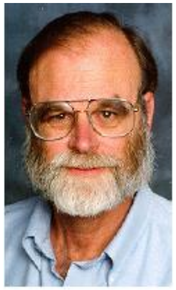
```
create table account (id number, balance number);
insert into account values (1, 1000);
insert into account values (2, 2000);
commit;

Begin_tx;
update account set balance = balance - 100 where id = 1;
update account set balance = balance + 100 where id = 2;
commit; -- or rollback;
```



- NOTE that system can fail at any point of TX execution
- Various DB statuses upon failure time
  - ✓ What if rollback after Pi is updated? -> Undo
  - ✓ Both Pi and Pj are not written back to the storage before system crash → Redo
  - ✓ Updated Pi is already written to the storage before commit and then system crashes; → Undo
  - ✓ Updated pages at DRAM after commit -> Redo
  - ✓ Non-atomic page write
  - ✓ …..
- Database recovery manager has to be able to cope with all the above scenarios.

Failures and Recovery

- Transaction failure
  - Rollback or killed by deadlock → undo recovery
- System failure
  - OS, DBMS bug, Power shortage → redo/undo recovery
  - Atomic write of page(s) is not guaranteed
- Media failure – not the main topic in this slide
  - Disk head crash → archive recovery
- "A very reliable (and available) system with unreliable component?"
  - [John Von Neumann: stored-program, CPU + RAM + Disk]
  - THE GIANT in database field: Jim Gray

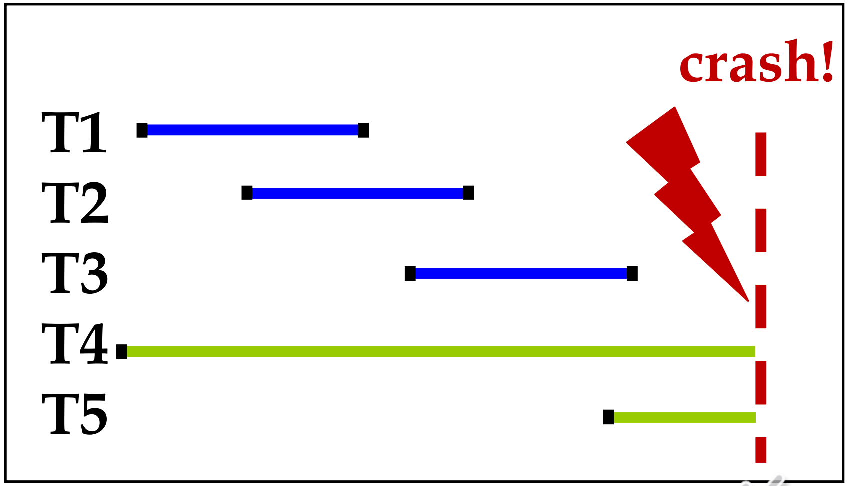


- Replication (though we do not cover) - https://www.khan.co.kr/culture/culture-general/article/202108090600001

Motivation

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

- Desired behavior after system restarts:
  - T1, T2 & T3 should be durable. - **Durability**
  - T4 & T5 should be aborted (effects not seen). - **Atomicity**

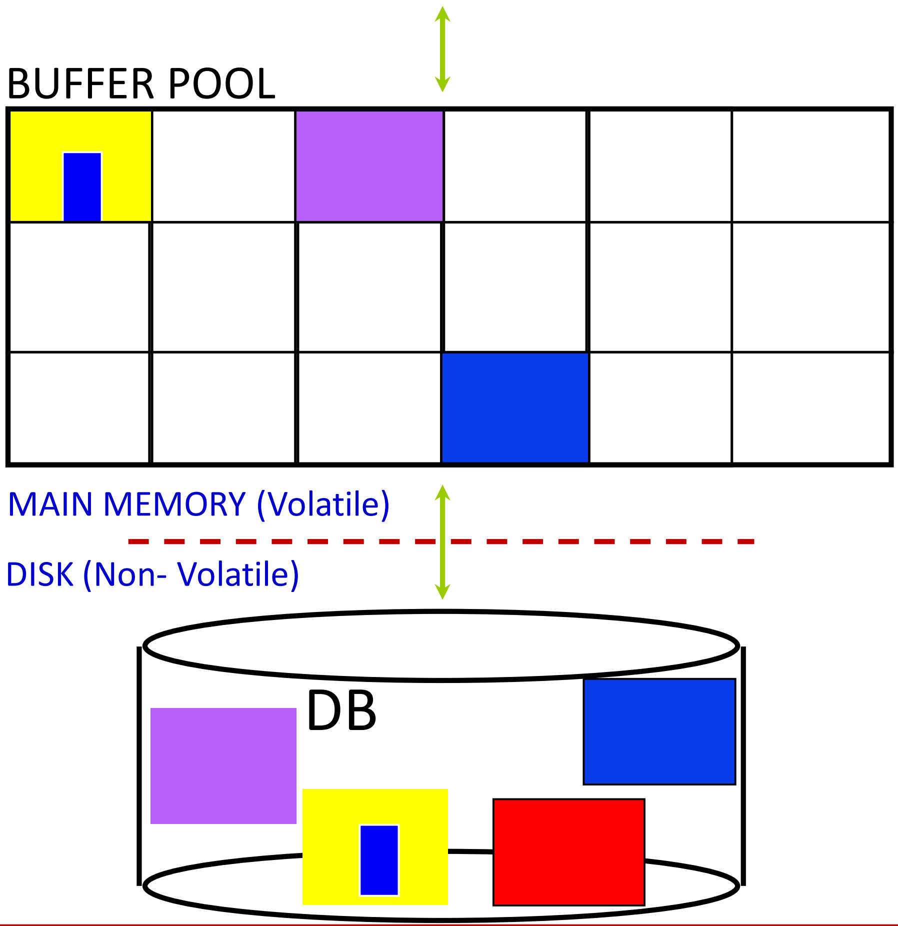


Very Large Data Bases

18.0 Basics

Disk Writes

Updates@Transactions
: SQL Insert/Delete/Update

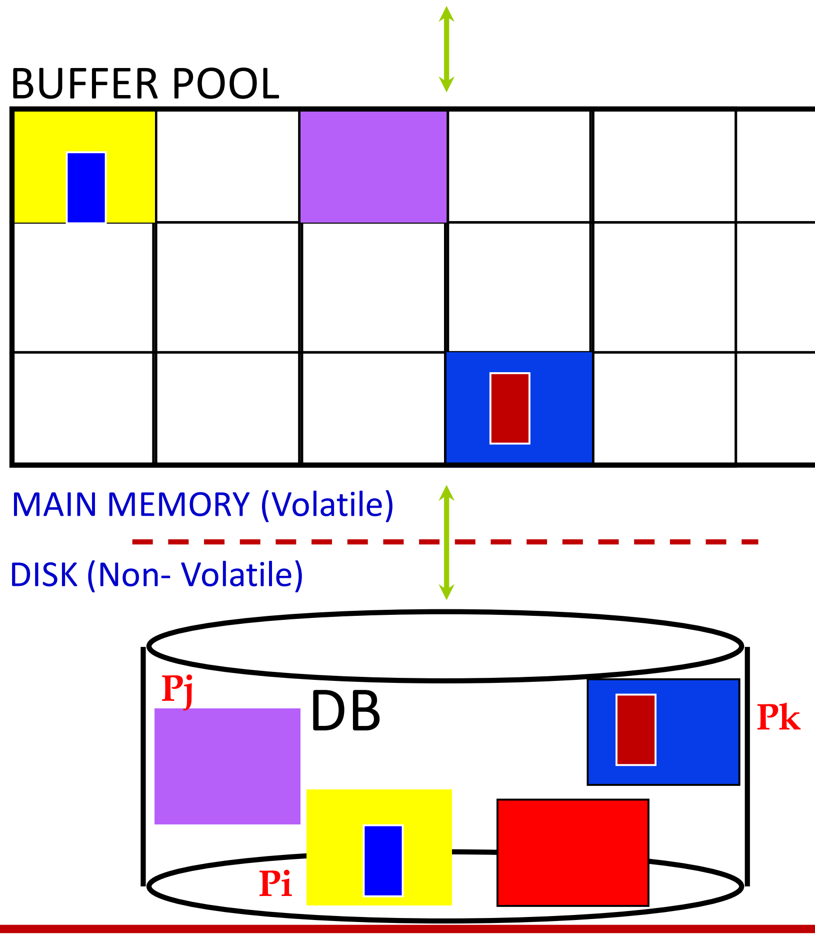


- <u>Three assumptions:</u>
  1. Page: unit of update propagation from host DRAM to storage
  2. <u>In-place update</u> vs. out-of-place update (or Copy-on-write: CoW)
  3. Atomic propagation: single page, multiple pages
     - ✓ How? e.g. shadow page

- <u>Technical issues:</u>
  - Buffer replacement
  - Commit, abort, crash, recovery, redo/undo

Transaction – Commit and Force

Updates@Transactions
: SQL Insert/Delete/Update



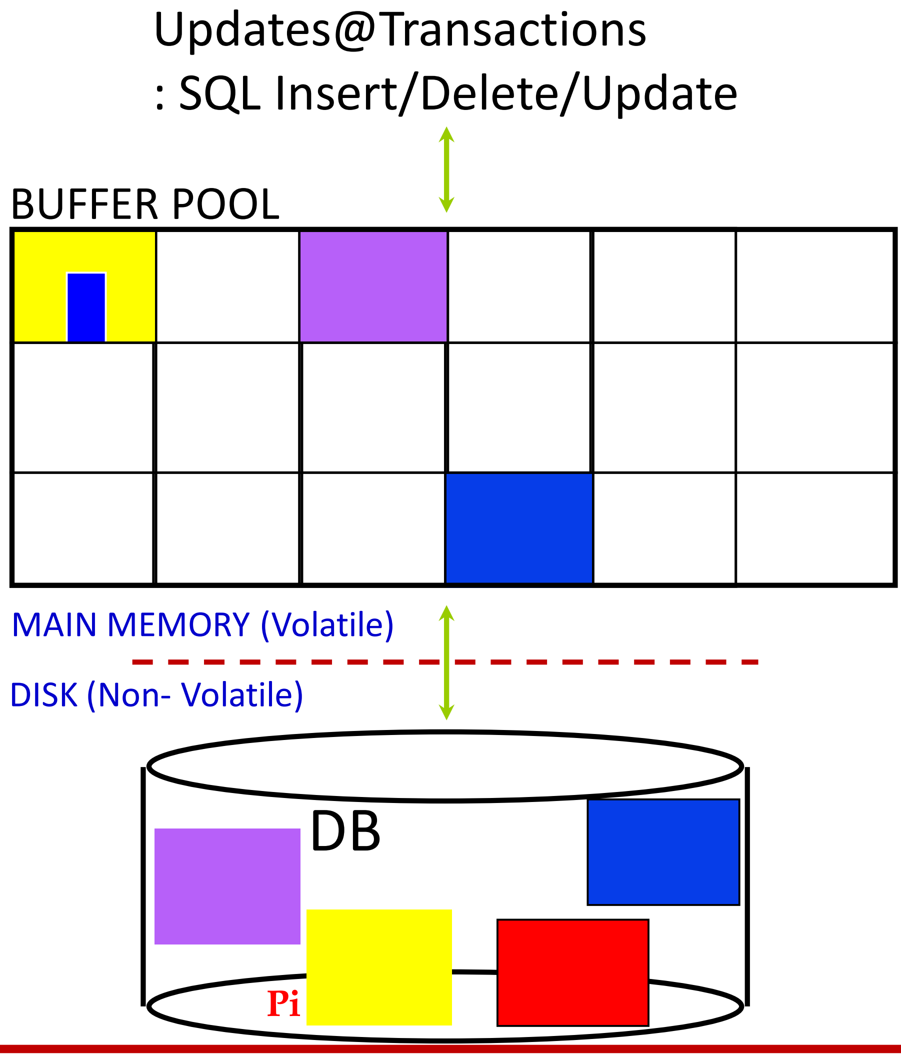
Transaction T

- Begin
- Read page Pi, Pj, Pk (from disk)
- Update Pi, Pk
- Commit (Write Pi, Pk to disk)

➔Achieve "All" and Durability

➔Commit policy: "Force at Commit"

Transaction – Rollbacks and No-Steal



Transaction T

- Begin
- Read page Pi (from disk)
- Update Pi
- *[Never write Pi to disk before commit]*
- Rollback

➔Achieve "Nothing"

➔Buffer management: "No Steal"

Recovery Modes vs. Buffer Policies

- Force every write to disk?
  - Poor response time.
  - But, provides durability.

- Steal buffer frames from uncommited Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

No-Steal and Force – Economical Solution?

- The economics of "volatile RAM and non-volatile disk"
  - No-steal policy is impossible because of limited RAM size
  - Force policy is inefficient since many random writes to disk reduce the throughput

- Therefore, all major DBMSs choose "steal + no-force" policy
  - Steal policy as buffer management
  - No-force policy for commit

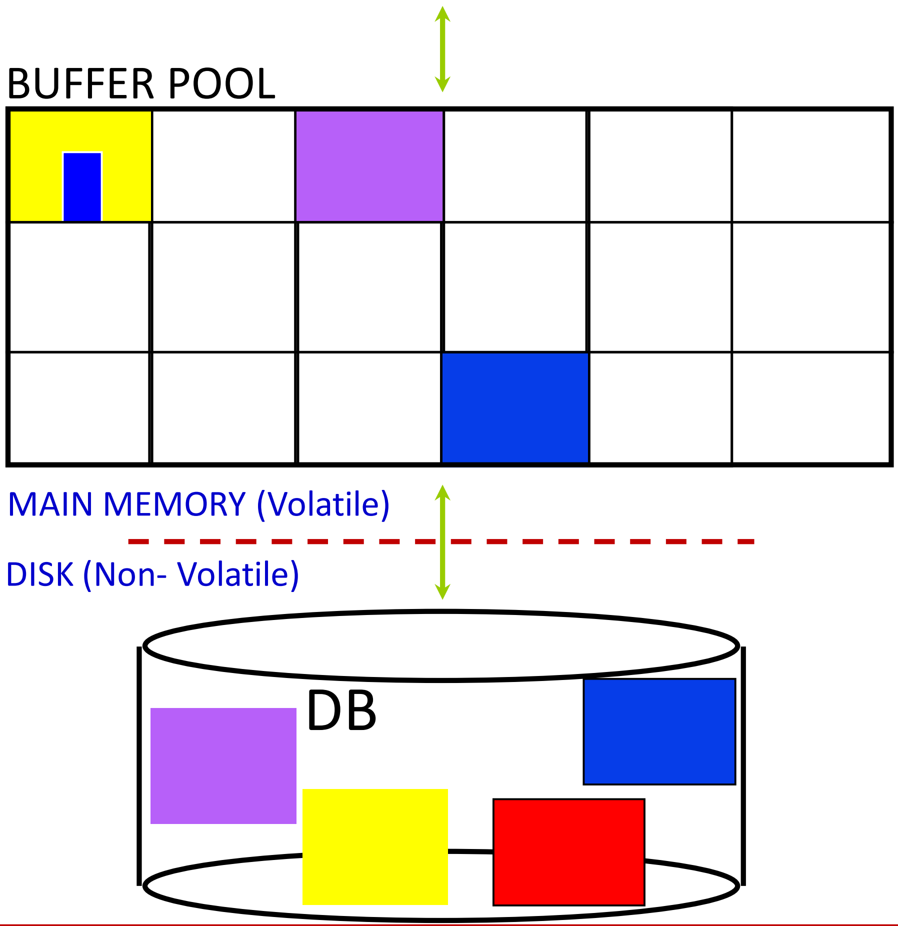
More on Steal and Force

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

- **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.

Transaction – Commit and No-Force

Updates@Transactions
: SQL Insert/Delete/Update



Transaction T

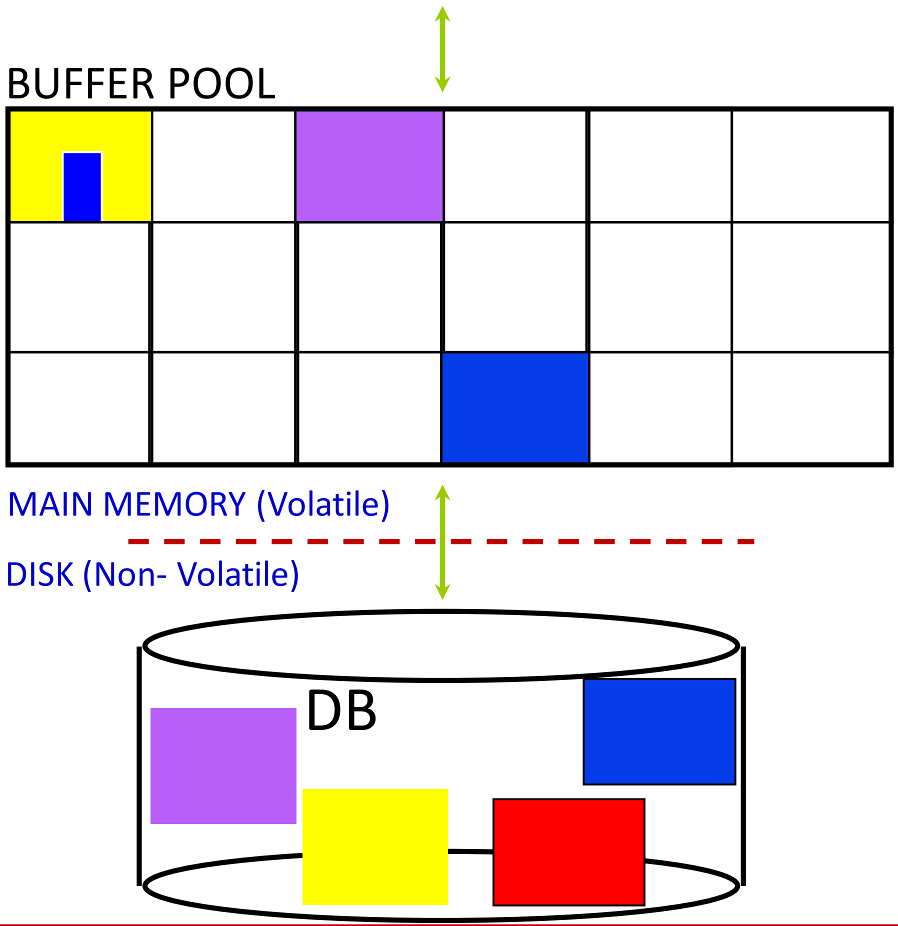
- Begin
- Read page Pi (from disk)
- Update Pi
- Commit
  - *[Do not force Pi to disk]*

➔ In case of system failure, how to achieve "all" and durability?

➔ "Redo log"

Transaction – Rollbacks and Steal

Updates@Transactions
: SQL Insert/Delete/Update



Transaction T

- Begin
- Read page Pi (from disk)
- Update Pi
  - *[prior to commit, Pi can reach disk due to buffer replacement]*
- ……
- *Rollback* (or system failure)

➔ In case of system failure, how to achieve "nothing"?

➔ "Undo log"

Recovery Modes

- Force every write to disk?
  - Poor response time.
  - But provides durability.

	No Steal	Steal
Force	Trivial	
No Force		Desired

- Steal buffer-pool frames from uncommited Xacts?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?

SQLite: Case Study

- Single user, mobile device: very **simple** approach for recovery
- SQLite demo

- Two journal modes for recovery
  - Rollback (RBJ) and WAL journal: a kind of **side file**
    - ✓ Simple and seemingly inefficient: redundant write or journaling
  - But, enable 1) atomic page write, 2) atomic propagation of N pages, and 3) simple UNDO;
  - And, in case of WAL, transaction-consistent checkpoint is supported

- Commit policy: **FORCE**, thus NO REDO
- Buffer policy : **STEAL**, but simple UNDO by RBJ and WAL

SQLite Case Study: Concurrency Control

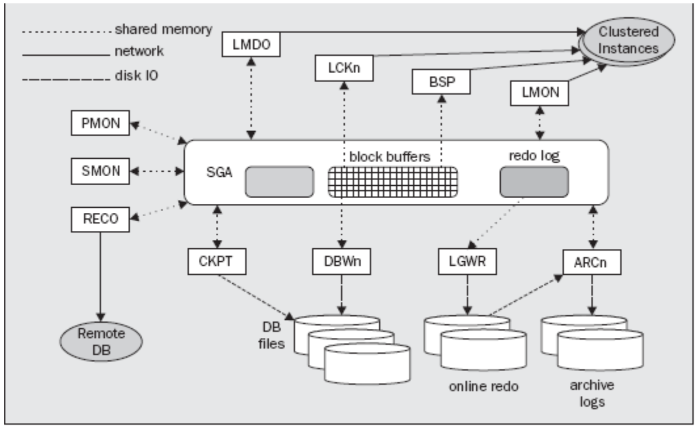
- SQLite takes coarse-grained concurrency control
  - **File-level** vs. Oracle's tuple-level fine-grained CC

- Coarse-Grained CC allows to take the force commit
  - In case of tuple-level CC like Oracle, one page can be be updated concurrently by T1 (committing) and T2(in-progress)
  - THEN, even with RBJ / WAL, there is no way to undo T2's update when system crash is encountered before T2 commits.
  - WHY??

18.2 Log

What You Should Know

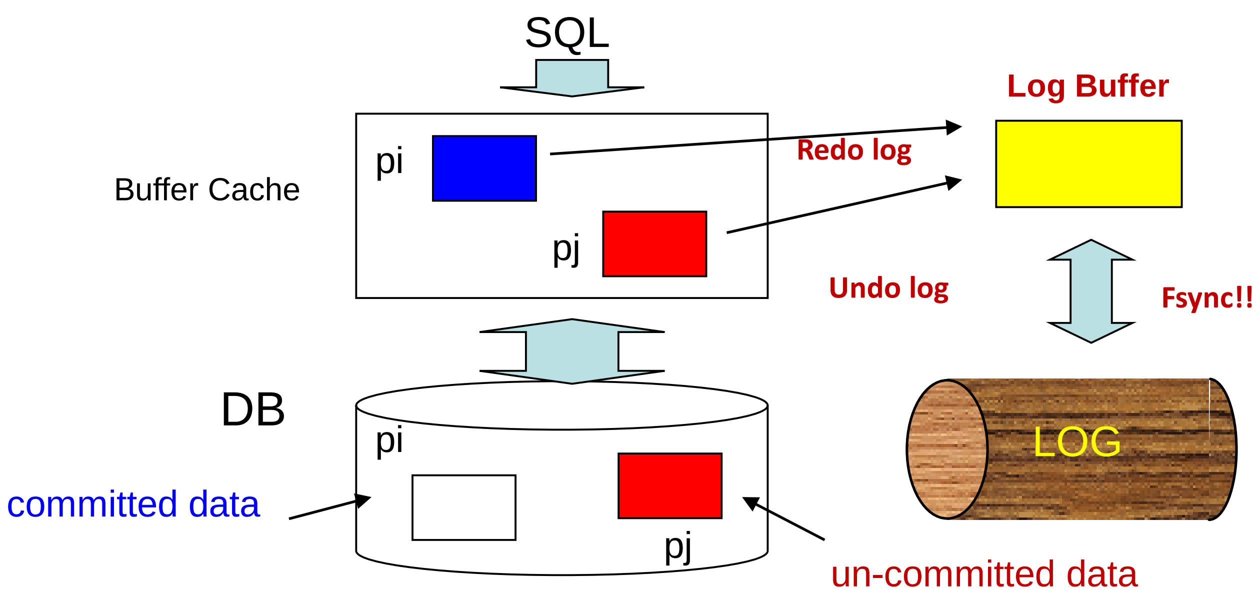
- Log and LSN (Log Sequence Number)
  - Every update per page Pi will generate its corresponding LOG at DRAM log buffer and the LOG's LSN be stored as new **pageLSN** of Pi
- Interplay among <u>buffer manager</u>, <u>database writer</u>, <u>log manager</u>, and <u>recovery manager</u>



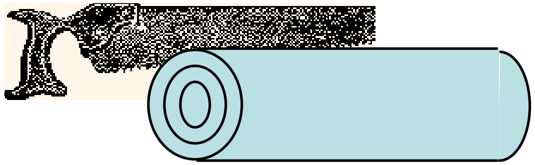
- Log-based UNDO/REDO recovery
  - <u>**Force-log-at-commit**</u> for redo recovery
  - <u>**Write-Ahead Log (WAL)**</u> protocol for undo recovery

Basic Idea: Logging

- Update-in-place + Steal + No-force ➔ {redo + undo} log
  - "Page as unit of update propagation" makes recovery harder
- Log and LSN (Log Sequence Number)
  - Every update per page Pi will generate its corresponding LOG at DRAM log buffer and the LOG's LSN be stored as new **pageLSN** of Pi



Basic Idea: Logging (cont.)

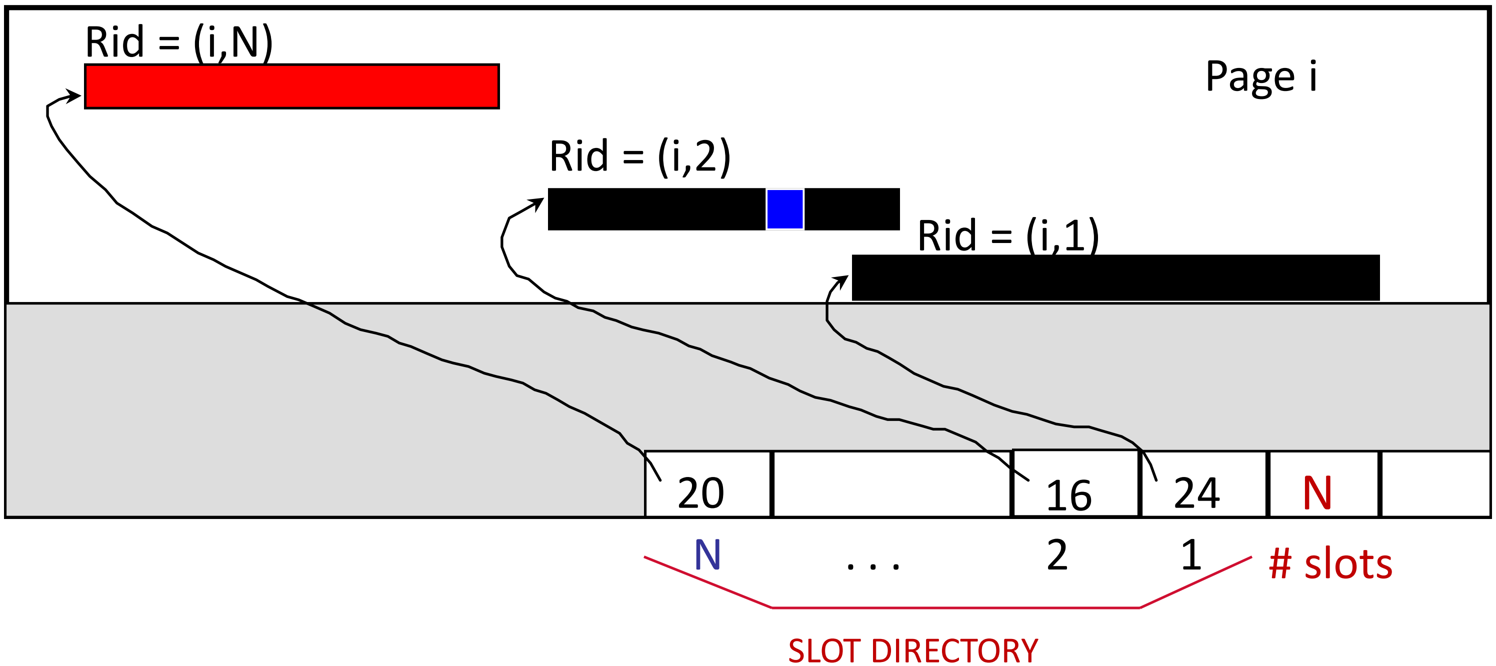


- For every update, store REDO and UNDO info. in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.

- Log: An "ordered" list of REDO/UNDO actions
  - Log record contains:

    <XID, pageID, offset, length, old data, new data>
  - and additional control info (which we'll see soon).

Update and Log

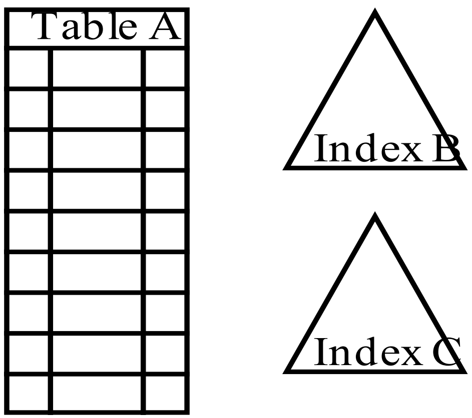


- Page changes - Insert, Update, Delete
- Update log
  - E.g. (LSN,Tid, Pi,Offset,Length, Before-value, After-value)
    - ✓ **LSN** = Log Sequence Number

Three Types of Logs

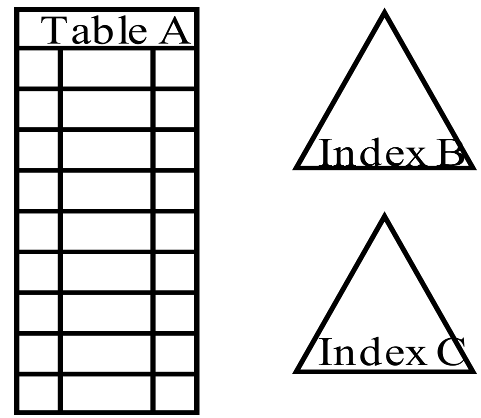
- Physical(value) vs. Logical vs. Physio-logical log

Insert record r into table A



Logical log record

insert, A , r



Physiological log records

insert, A , page 508, r

insert, B, page 72, s

insert, C, page 94, t

Log

- **Database** = Pages in disk + Log
  - Pages in DRAM: unreliable information
  - Log: a redundant information in reliable disk

- In spite of system failure,
  - Redo log guarantees durability for committed transactions, even though their updated pages are not forced out to disk
  - Undo log guarantee atomicity (i.e. “nothing”) for uncommitted or aborting transactions

DO-UNDO-REDO Model

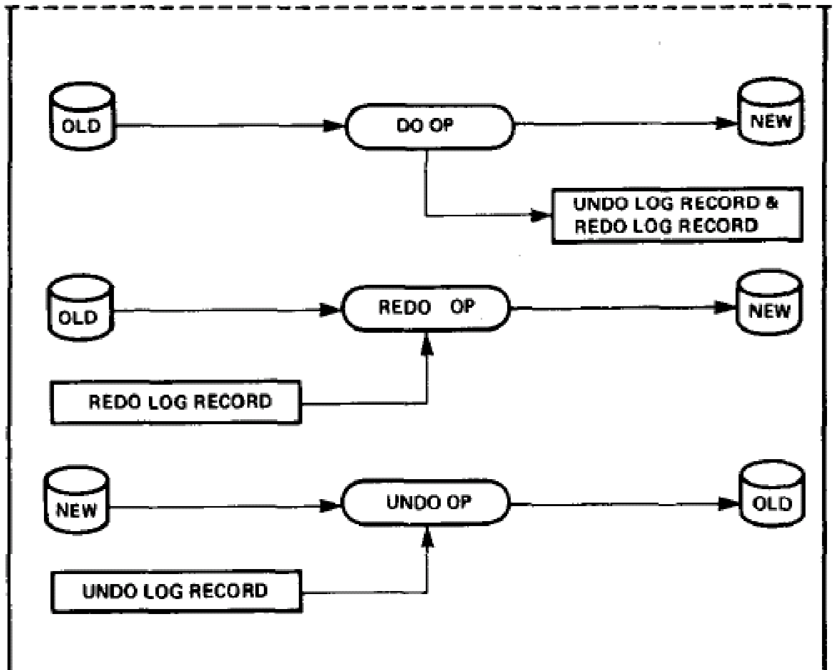


Figure 3. The DO-UNDO-REDO protocol. The execution of each protected action generates a log record which allows the action to be undone or redone. Unprotected

- C.f.
  - "String" by Ariadne and Theseus (Greeks)
  - "Trail of bread crumbs" by Hensel and Gretel

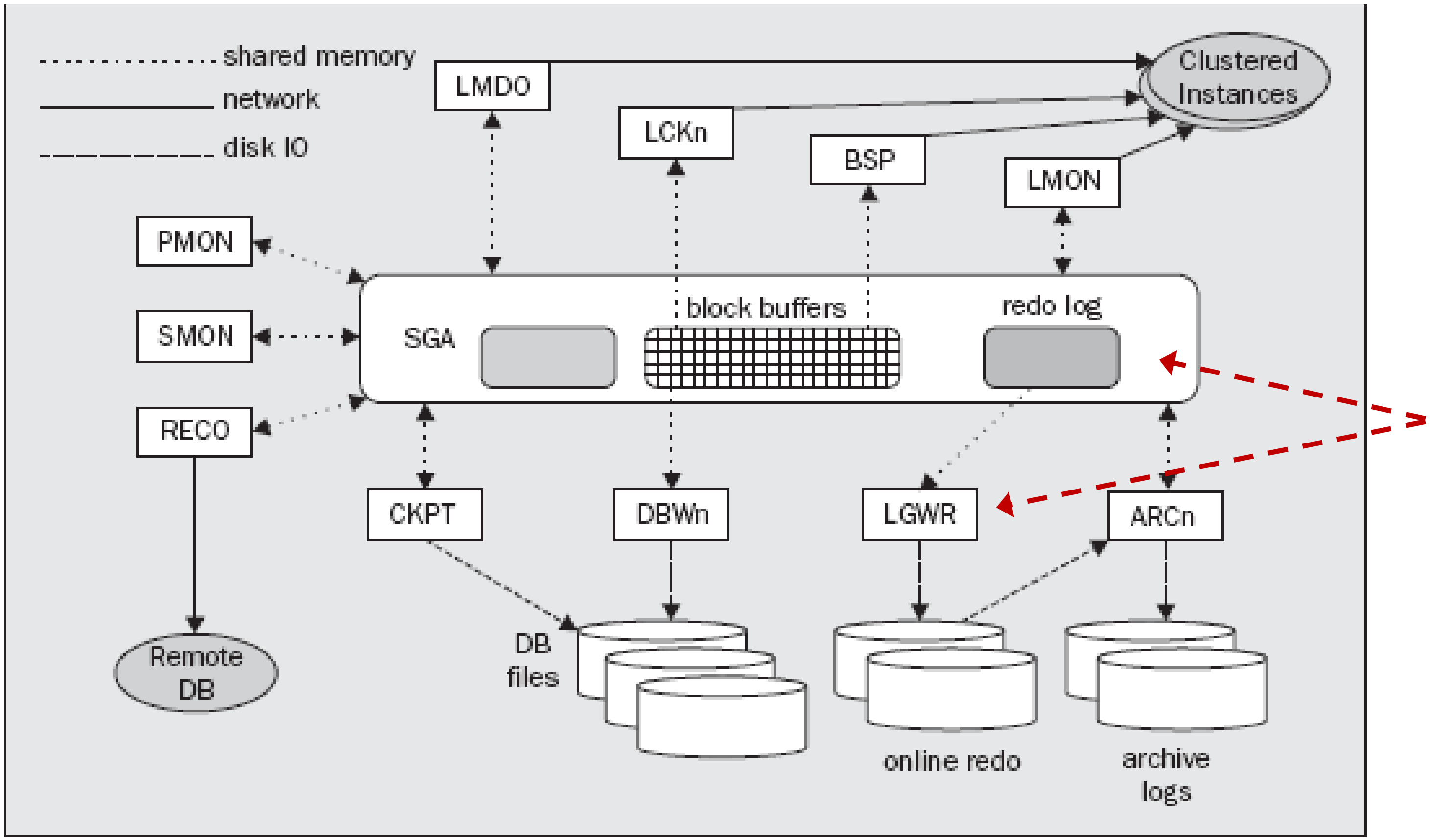
From "The Transaction Concepts: Virtues and Limitations" VLDB 1981

Two Logging Protocols

- WAL (Write-Ahead Log) protocol – for undo
  - Before (over)writing a page to disk, flush undo log in the log buffer
    - ✓ up to the page's pageLSN
  - DBWR calls LGWR to write the undo log (inter-process comm.) – in fact, the overhead (comm. and disk IO) is not big

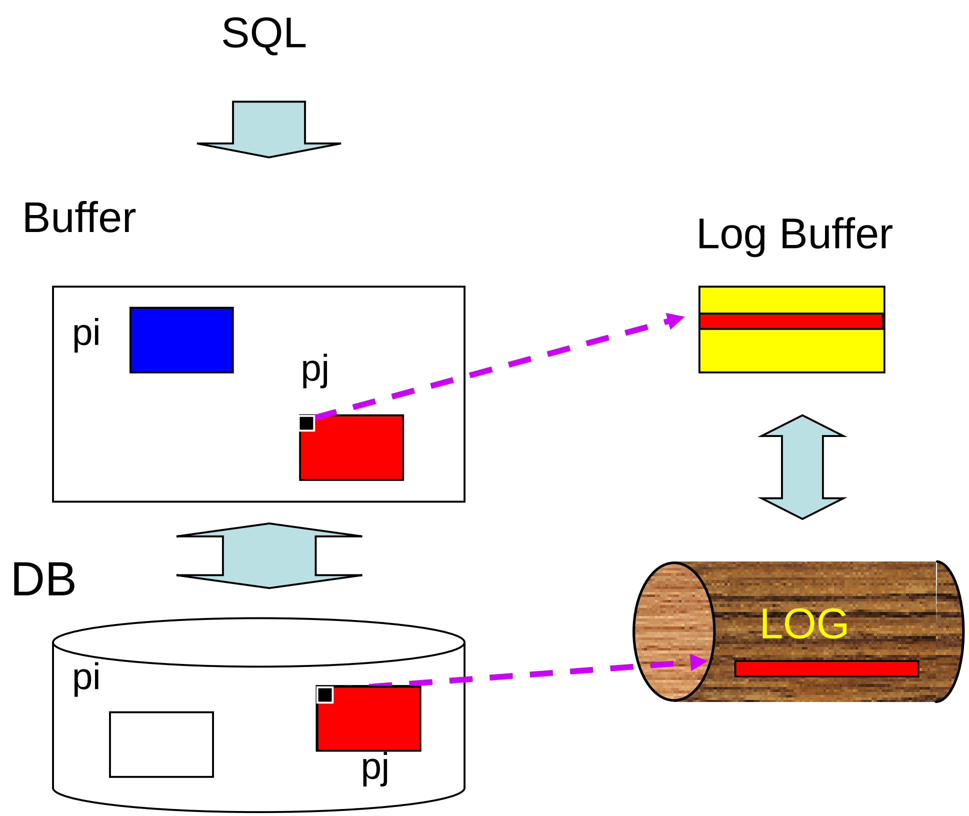
- Log-Force-at-Commit – for redo
  - A transaction is truly committed only after the commit log record is persistently stored in log disk (for this, the cache in log disk is turned off)
  - Performance bottleneck
  - Group commit

Oracle Architecture



WAL Protocol (Chap 18.4)

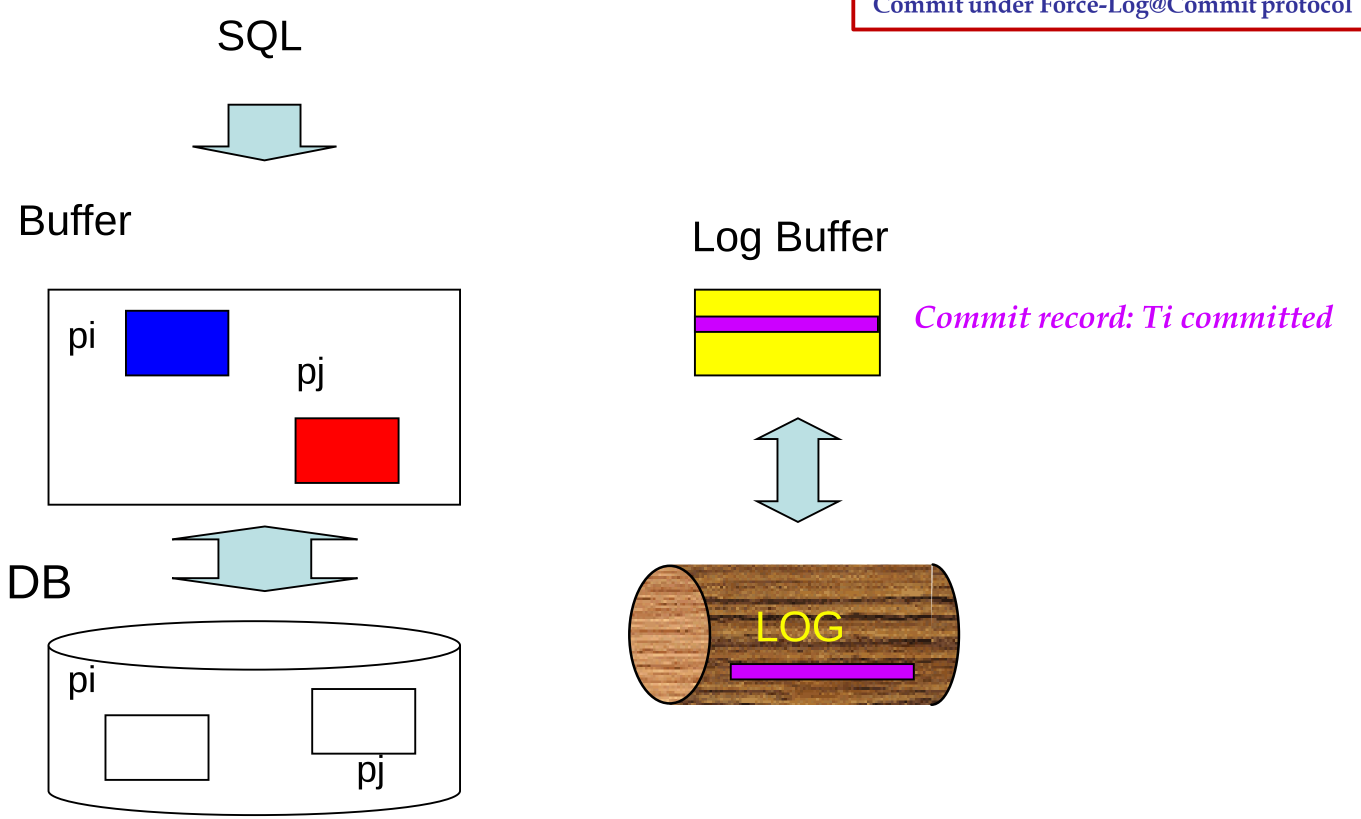
- Before writing Pj to disk



Log Force at Commit

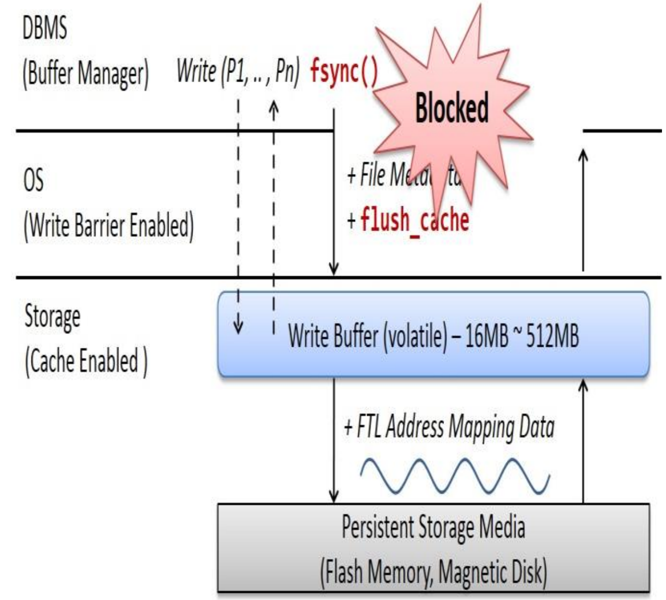
- When a transaction Ti commits

You Should Understand what happens upon normal Update SQL statements and Commit under Force-Log@Commit protocol



Write, Durability, Fsync, Write_barrier and Storage Cache

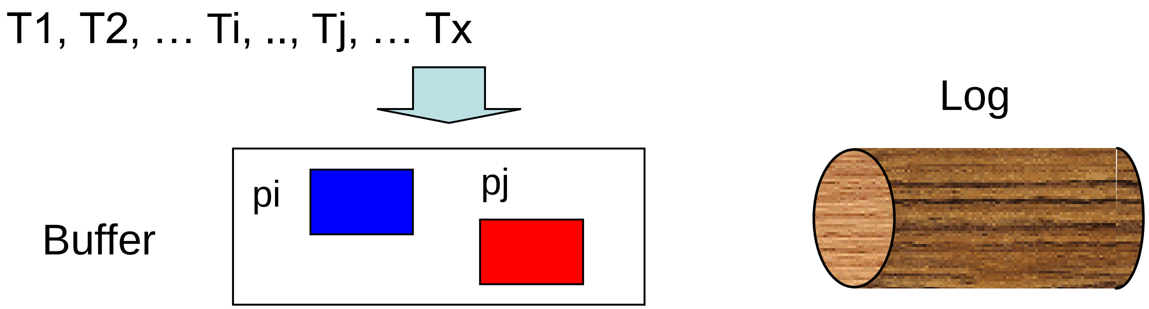
- Durability: changes made in DRAM are persistently stored in non-volatile media
- Common mechanism for D is to issue the f(data)sync() call
  - Flushes dirty pages from OS page cache to storage device
  - If WRITE_BARRIER is enabled, OS sends a FLUSH_CACHE command to storage device and flushes the write cache to persistent media
  - Modern storage device has its own cache



- **0-latency durability**
  - Latency for durability is a challenge for performance
  - HDD → SSD → NVRAM/NVDIMM @ Storage or Host
- What if NVRAM? Do we still need REDO/UNDO recovery? It still looks challenging even with NVRAM!
  - Given NVRAM instead of DRAM, {P1, P2} TX; If we meet crash after updating only P1 in place in NVRAM, can we undo it?

18.5 Checkpointing

- Due to no-force policy, many pages dirtified by already committed transactions remain unflushed in buffer cache.



- Thanks to redo log, those pages are recoverable upon failure
- But, it takes time to recover. And, the time is money!!
- Thus, it is very critical to minimize the recovery time upon failure.
- For this reason, those pages have to be periodically checkpointed.

Checkpointing (2)

- Periodically, the DBMS creates a checkpoint, in order to minimize the time taken to recover in the event of a system crash. Write to log:
  1. begin_checkpoint record: Indicates when chkpt began.
  2. end_checkpoint record: Contains current *Xact table* and *dirty page table*. This is a **`fuzzy checkpoint':**
  - ✓ Other Xacts continue to run; so these tables accurate only as of the time of the begin_checkpoint record.
  - ✓ No attempt to force dirty pages to disk; effectiveness of checkpoint limited by oldest unwritten change to a dirty page. (So it's a good idea to periodically flush dirty pages to disk!)
  3. Store LSN of chkpt record in a safe place (*master* record).

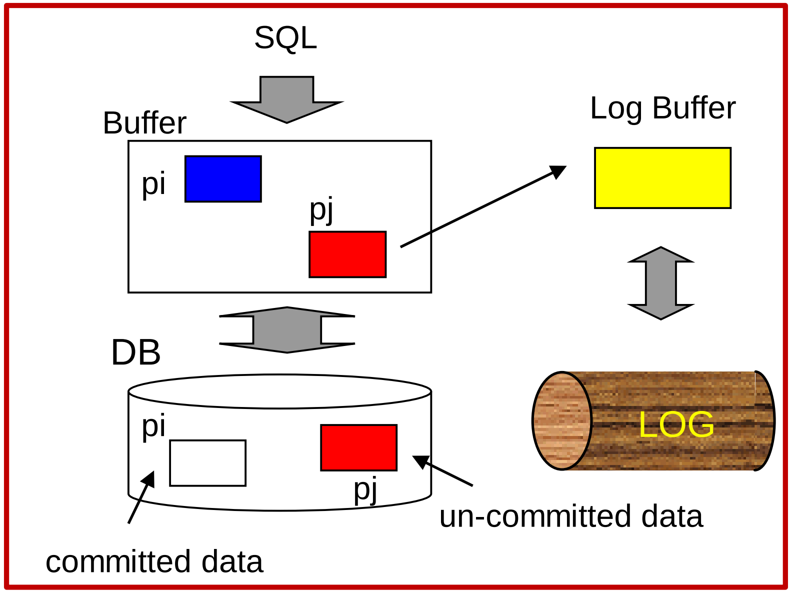
Summary – Normal-time Operations for Recovery

- WAL Protocol for Undo log
- Force-Write Log for Redo log
- Checkpointing

Very Large Data Bases

Recovery Summary – Disk

- Commit policy: Force vs. No Force
- Buffer mgt policy: Steal vs. No Steal
- Disk / RAM economics
  - "WAL, No Force + Steal" in normal exec.
  - "Undo/Redo" in recovery time
- **SQLite vs. Enterprise-Class DBMS**



- TRADE-OFF: **Performance@NORMAL_TIME vs. Recovery@FAILURE**

	No Steal	Steal
No Force		**Fastest**
Force	**Slowest**	

Performance

	No Steal	Steal
No Force	REDO	UNDO REDO
Force	No-UNDO No-REDO	UNDO

- Terribly complex
- Time-consuming

Logging/Recovery

Very Large Data Bases

18. 1 & 18.6 ARIES Algorithm

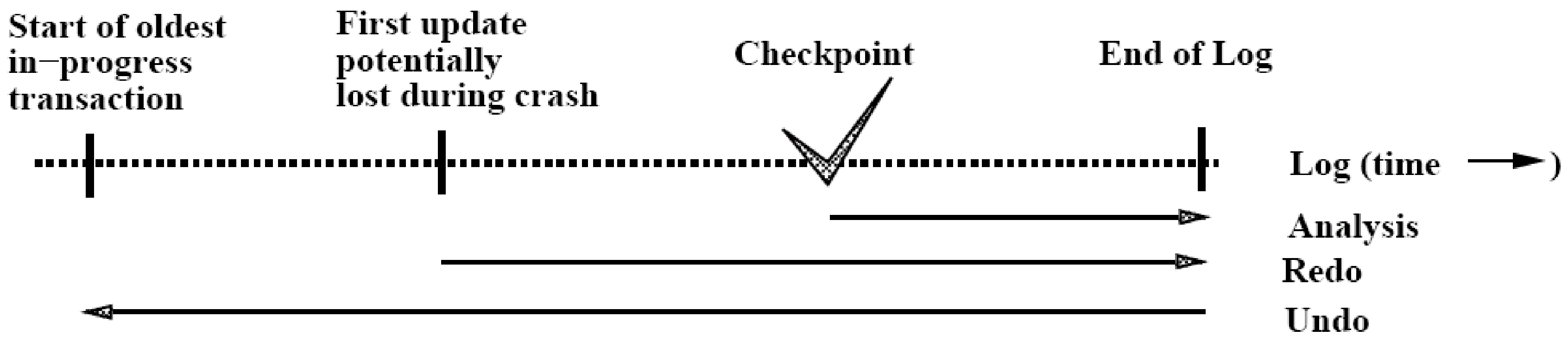
(Algorithms for Recovery and *Isolation* Exploiting Semantics)

Golden Standard for Database Recovery

WAL protocol is known to achieves an appropriate balance
in terms of the trade-off b/w normal-time performance and recovery time
in the environment of DRAM + harddisk.

Research Questions: Does this assumption still hold for NVM / SSD??

Three Phases in ARIES



(Start from the last checkpoint – found via master record)

1. Analysis (forward sequential scan)
2. Redo (forward sequential scan)
   - "Repeat history"
3. Undo (backward sequential scan)

Very Large Data Bases