Database Applications (15-415)

DBMS Internals- Part XIII
Lecture 21, April 14, 2014

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

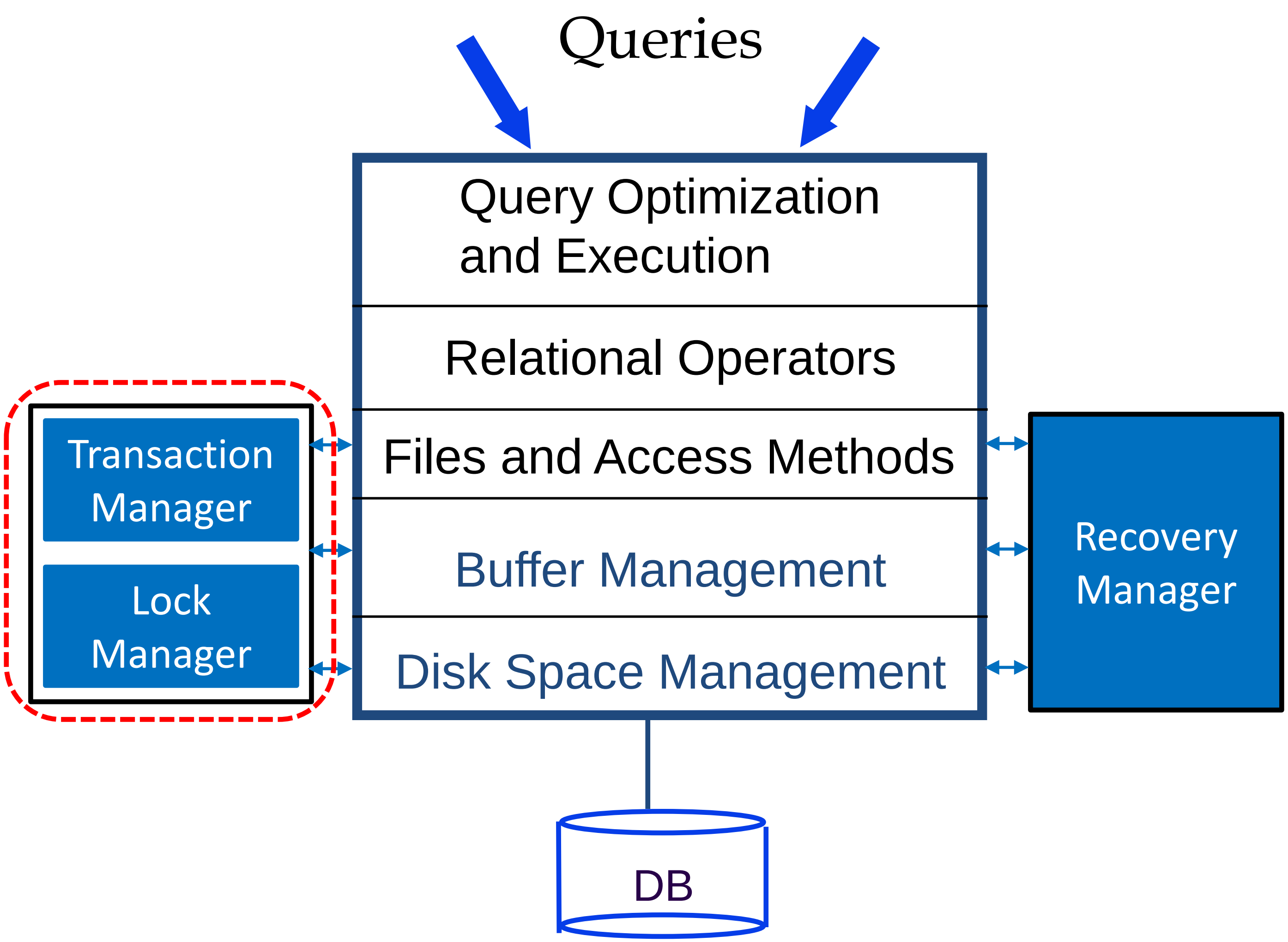
Today…

- Last Session:
  - Transaction Management (*Cont'd*)

- Today's Session:
  - Transaction Management (*finish*)
    - Non-Lock Based Protocols
  - Recovery Management

- Announcements:
  - PS4 is due tomorrow, April 15^{th}, by midnight
  - Please collect your quizzes tomorrow from my office

DBMS Layers



Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Outline

- Concurrency Control without Locking ✓
- The ACID Properties
- The Steal, No-Force Approach
- Logging and the WAL Protocol
- The Log

Locking Protocols on the Scale

- What is the main advantage of locking protocols?
  - They resolve RW, WR and WW conflicts
- What are the main disadvantages of locking protocols?
  - They entail lock management overhead
  - They require deadlock detection and resolution, or prevention mechanisms
  - They induce lock contention for heavily used objects
- If conflicts *are very rare*, the disadvantages of locking protocols might limit performance unnecessarily!

Can we do better?

Optimistic Concurrency Control (Kung & Robinson)

- We can allow all transactions to execute and only check for conflicts before they commit
  - Premise: Most transactions do not conflict with each others

- In particular, transactions can proceed in 3 phases:
  1. Read: read values and write results to private workspaces
  2. Validation: check for conflicts (*abort* in case of conflicts)
  3. Write: make private results public

This is known as "Optimistic" Concurrency Control!

The Validation Phase

- Each transaction ***Ti*** is assigned a numeric ID
  - E.g., A timestamp TS(***Ti***)

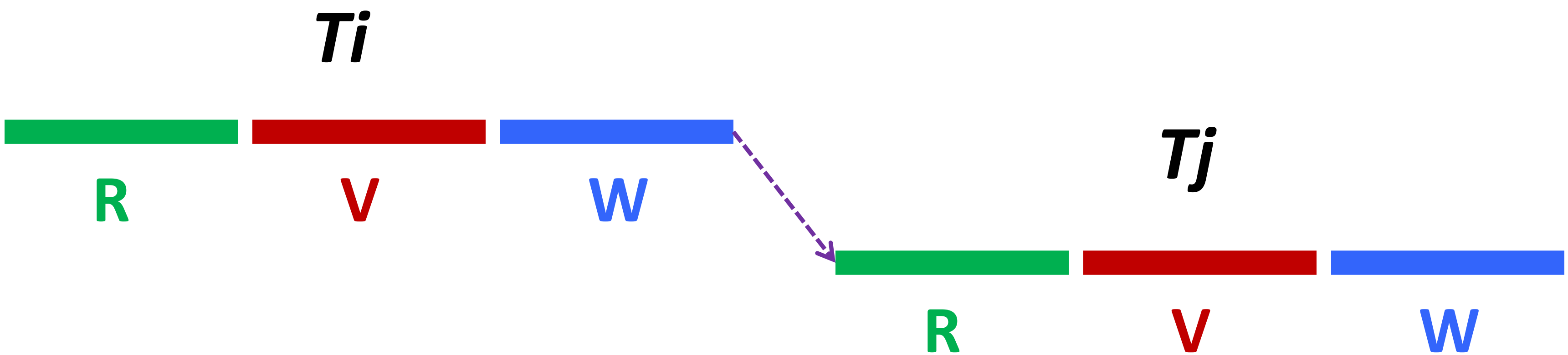
- For each ***Ti***, two sets of objects are maintained:
  - ReadSet(***Ti***): Set of objects read by ***Ti***
  - WriteSet(***Ti***): Set of objects written by ***Ti***

- The validation criterion checks whether the timestamp-ordering of transactions is equivalent to a serial order

- In particular, for every pair of transactions ***Ti*** and ***Tj*** such that TS(***Ti***) < TS(***Tj***), *three* validation conditions must hold (*see next*)

The Validation Phase: *Condition 1*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that ***Ti*** completes before ***Tj*** begins

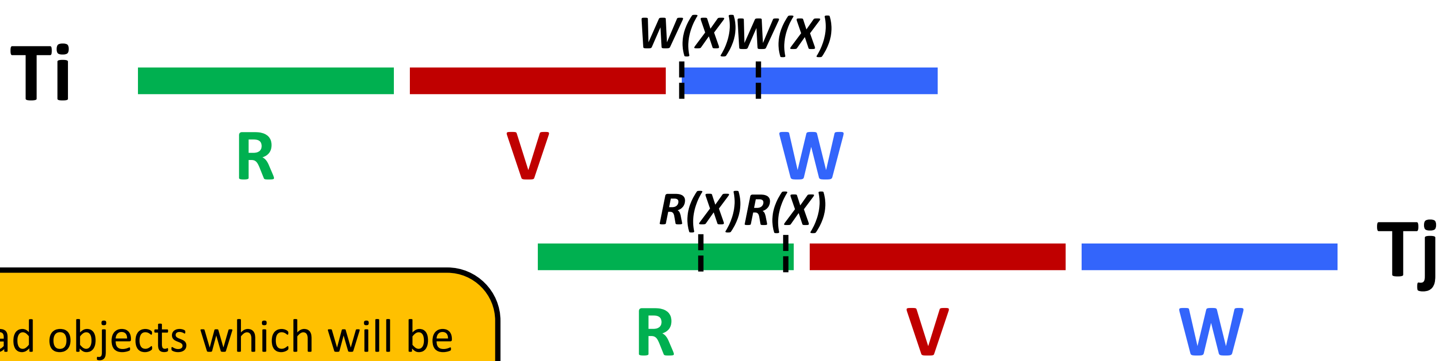


- ***Tj*** can see some of ***Ti***'s changes, but they execute entirely in serial order with respect to each other

- This ensure no RW, WR and WW conflicts!

The Validation Phase: *Condition 2*

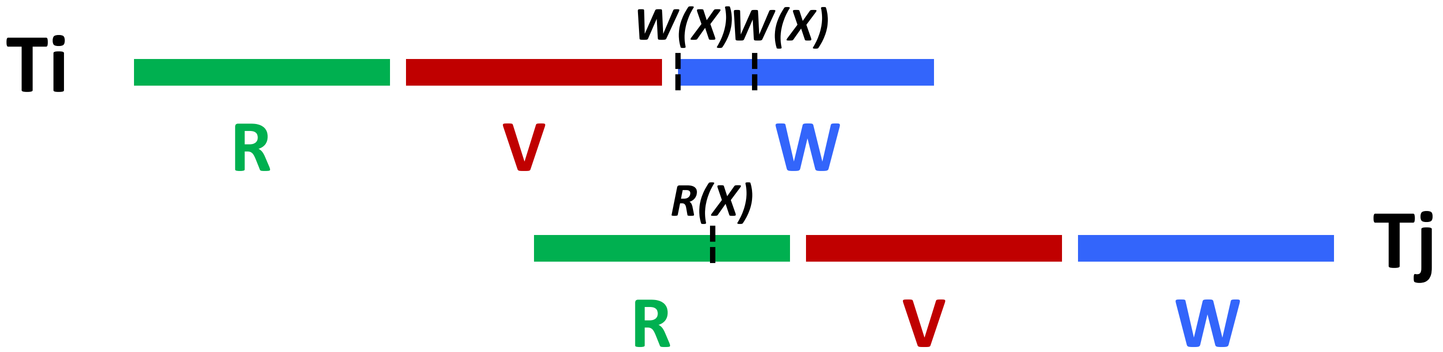
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes before ***Tj*** begins its Write phase
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



Tj can read objects which will be written by ***Ti***; hence, to avoid RW conflicts, WriteSet(***Ti***) ∩ ReadSet(***Tj***) should be empty!

The Validation Phase: *Condition 2*

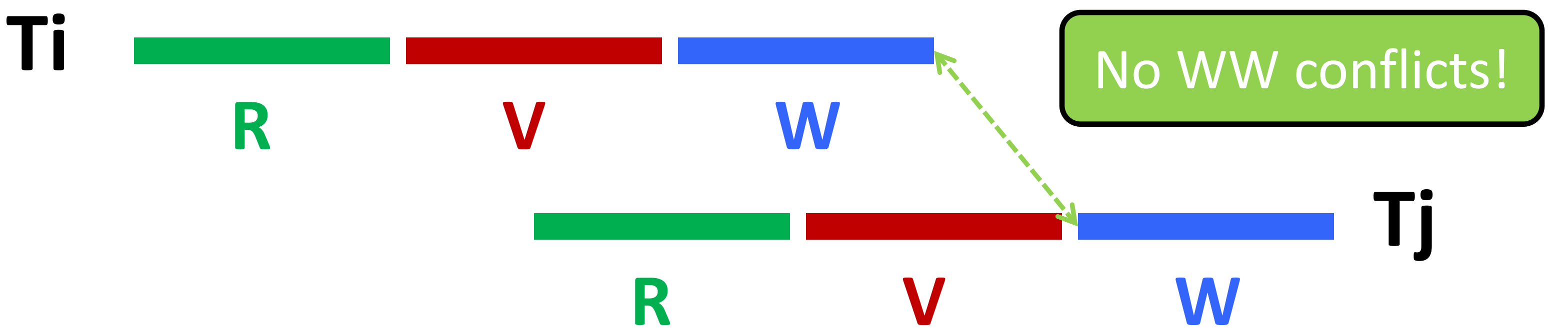
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes before ***Tj*** begins its Write phase
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



Tj can read objects which have been *<u>temporarily</u>* written by ***Ti***; hence, to avoid WR conflicts, WriteSet(***Ti***) ∩ ReadSet(***Tj***) should be empty!

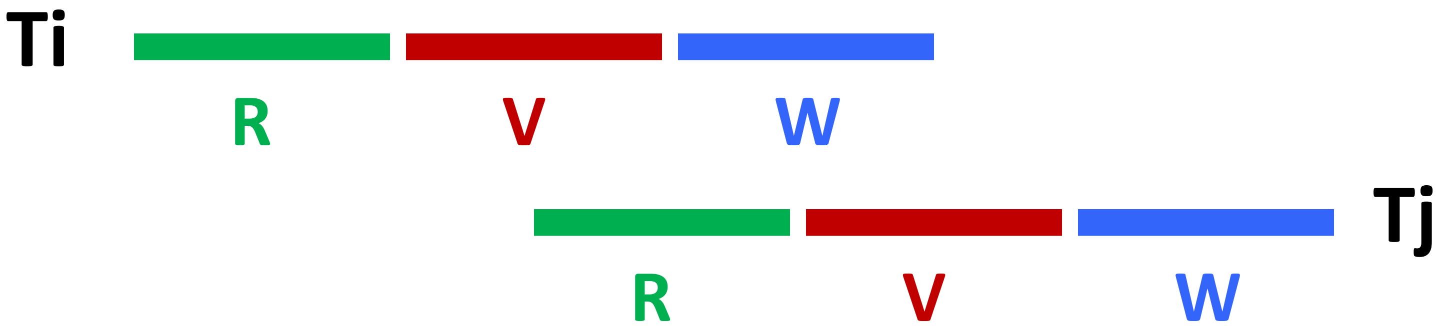
The Validation Phase: *Condition 2*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes before ***Tj*** begins its Write phase
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



The Validation Phase: *Condition 2*

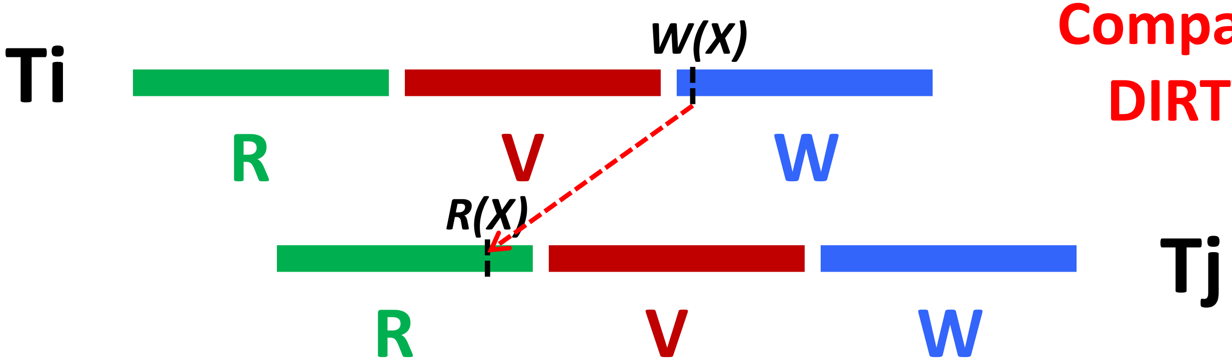
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes before ***Tj*** begins its Write phase
  - And WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



Therefore, ***Condition 2*** ensures that no RW, WR or WW will arise!

The Validation Phase: *Condition 3*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - And WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty

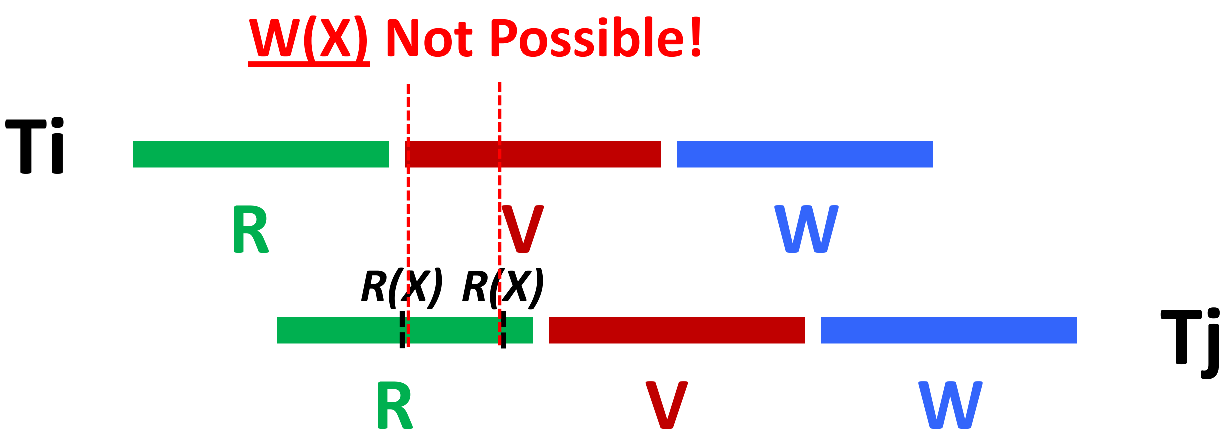


Comparable to a DIRTY READ!

Tj can read objects which will be written by ***Ti***; hence, to avoid WR conflicts, WriteSet(***Ti***) ∩ ReadSet(***Tj***) should be empty!

The Validation Phase: *Condition 3*

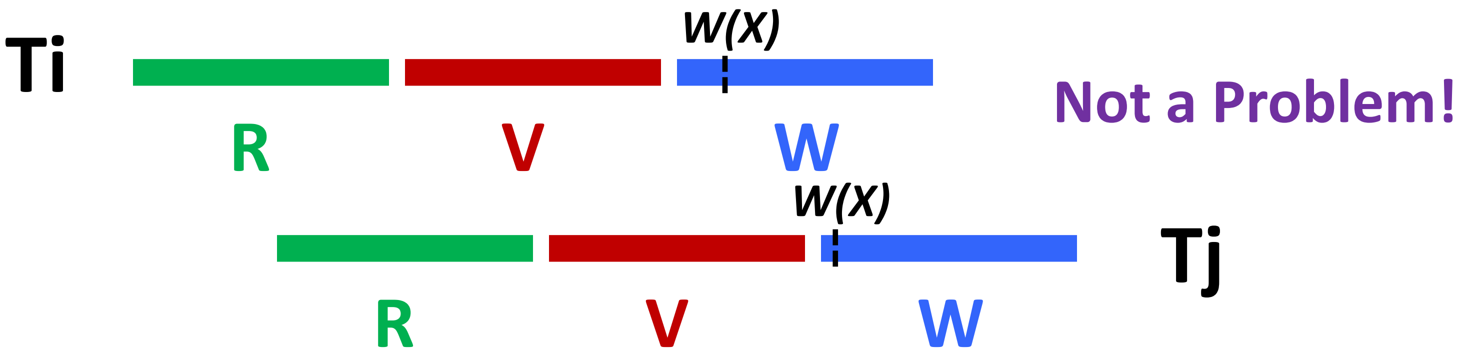
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



An unrepeatable read is not an option; hence, no RW conflicts!

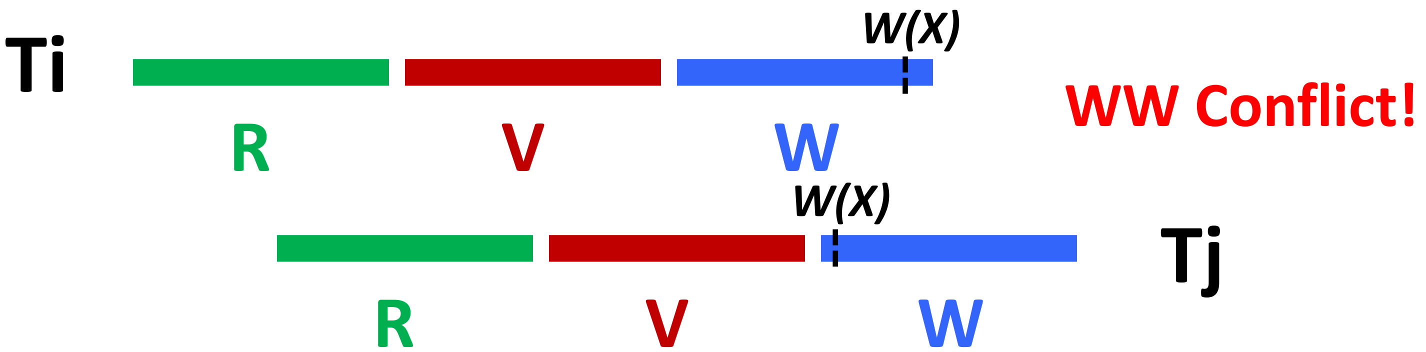
The Validation Phase: *Condition 3*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



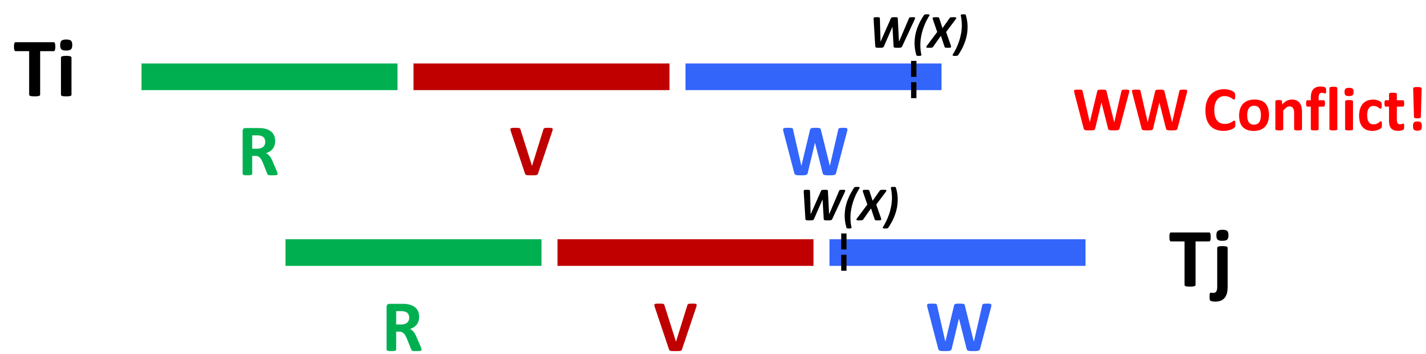
The Validation Phase: *Condition 3*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - And WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



The Validation Phase: *Condition 3*

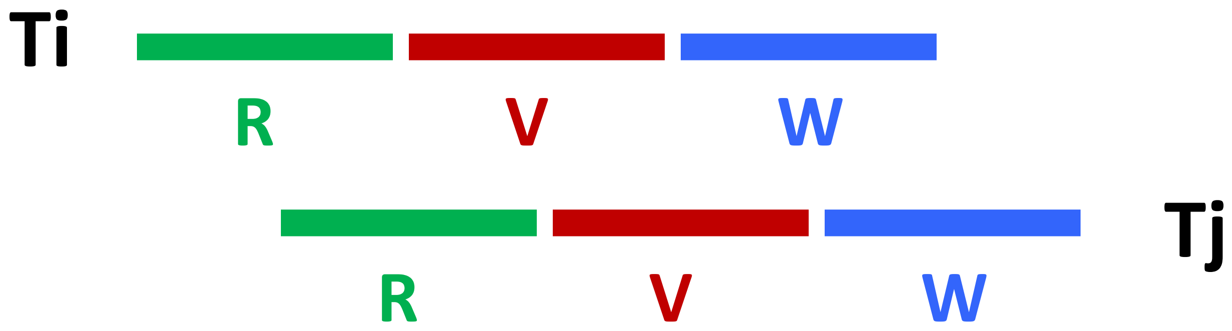
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - <u>And</u> WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty



Ti can write objects which have been written by ***Tj***; hence, to avoid WW conflicts, WriteSet(***Ti***) ∩ WriteSet(***Tj***) should be empty!

The Validation Phase: *Condition 3*

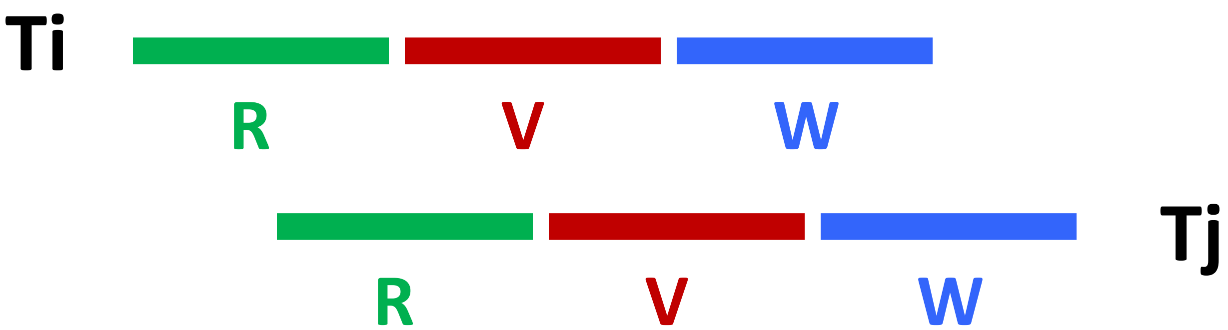
- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - And WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty
  - And WriteSet(***Ti***) ∩ WriteSet(***Tj***) is empty



Ti can write objects which have been written by ***Tj***; hence, to avoid WW conflicts, WriteSet(***Ti***) ∩ WriteSet(***Tj***) should be empty!

The Validation Phase: *Condition 3*

- For all ***i*** and ***j*** such that ***Ti*** < ***Tj***, the validation phase checks that:
  - ***Ti*** completes its Read phase before ***Tj*** does
  - And WriteSet(***Ti***) ∩ ReadSet(***Tj***) is empty
  - And WriteSet(***Ti***) ∩ WriteSet(***Tj***) is empty



Therefore, ***Condition 3*** ensures that no RW, WR or WW will arise!

Summary

- There are several *lock-based* concurrency control schemes (e.g., 2PL & Strict 2PL)
  - The *lock manager* keeps track of the locks issued

- Deadlocks can arise, but they can either be detected and resolved, or initially prevented

- With dynamic databases, naïve locking strategies may expose the *phantom problem*
  - Resolving this problem has to do with the *locking granularity*

Summary

- *Index locking* is common, and affects performance significantly
  - Needed when accessing records via an index
  - Needed for *locking logical sets of records* (index locking/predicate locking)

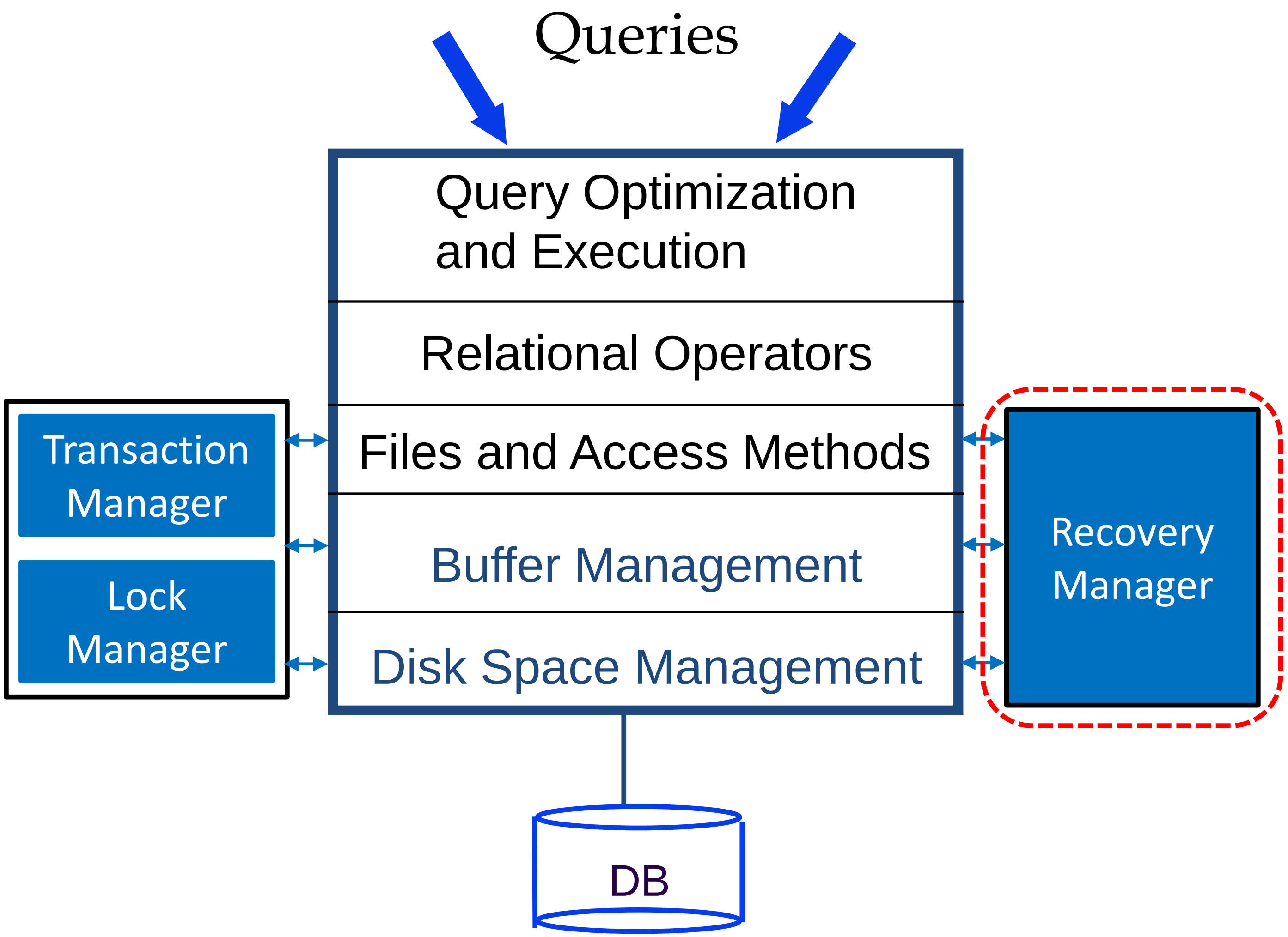
- Tree-structured Indexes:
  - A straightforward use of 2PL is very inefficient
  - Bayer-Schkolnick illustrates a high potential for performance improvement

Summary

- “Pessimistic” Concurrency Control (CC) might limit performance in an environment where reads are common and writes are rare
  - “Optimistic” CC aims at minimizing CC overheads in these kinds of environments

- Most real systems, however, use pessimistic CC

DBMS Layers



Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Outline

- Concurrency Control without Locking
- The ACID Properties ✓
- The Steal, No-Force Approach
- Logging and the WAL Protocol
- The Log

The ACID Properties

- Four properties must be ensured in the face of concurrent accesses and system failures:
  - **A**tomicity: Either all actions of a transaction are carried out or none at all
  - **C**onsistency: Each transaction (run by itself with no concurrent execution) must preserve the consistency of the database
  - **I**solation: Execution of one transaction is isolated (or protected) from the effects of other concurrently running transactions
  - **D**urability: If a transaction commits, its effects persist (even of the system crashes before all its changes are reflected on disk)

The ACID Properties

- Four properties must be ensured in the face of concurrent accesses and system failures:

? Atomicity: The Responsibility of the Recovery Manager

Consistency: The Responsibility of the User

✓ Isolation: The Responsibility of the Transaction Manager

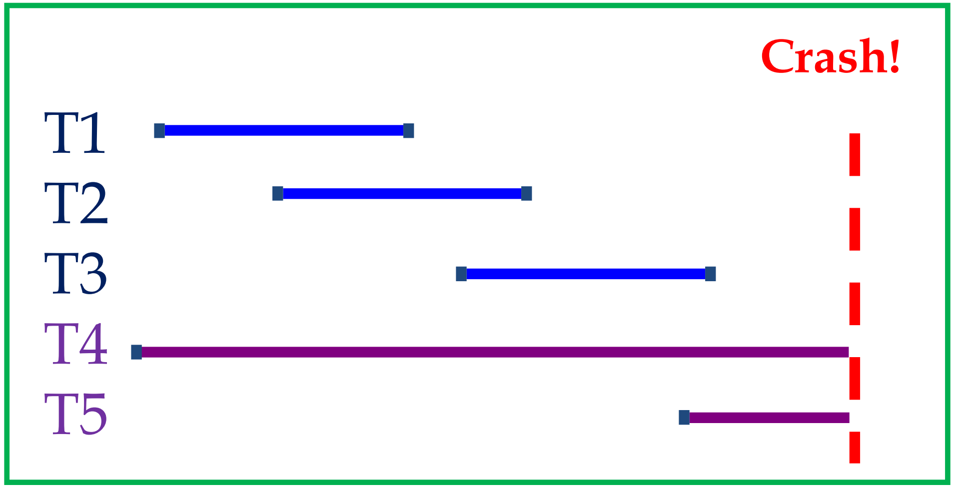
? Durability: The Responsibility of the Recovery Manager

Outline

- Concurrency Control without Locking
- The ACID Properties
- The Steal, No-Force Approach ✓
- Logging and the WAL Protocol
- The Log

Ensuring Atomicity and Durability

- How can the recovery manager ensure atomicity and durability (in case of a failure)?
  - It can ensure atomicity by *undoing* the actions of transactions that did not commit
  - It can ensure durability by *redoing* (all) the actions of committed transactions



- Desired Behavior after the system restarts:
  - T1, T2 & T3 should be durable
  - T4 & T5 should be rolled back

Stealing Frames and Forcing Pages

- To realize what it takes to implement a recovery manager, it is necessary to understand what happens during normal execution
  - Can the changes made to an object ***O*** in the buffer pool by a transaction ***T*** be written to disk before ***T*** commits?
    - Yes, if another transaction *steals* ***O***'s frame (a *steal approach* is said to be in place)
    - No, if stealing is not allowed (a *no-steal approach* is said to be in place)
  - When ***T*** commits, must we ensure that all its changes are immediately *forced* to disk?
    - Yes, if a *force approach* is used
    - No, if a *no-force approach* is used

Steal vs. No-Steal and Force vs. No-Force Approaches

- What if a no-steal approach is used?
  - We do not have to *undo* the changes of an aborted transaction (+)
  - But this assumes that all pages modified by ongoing transactions can be accommodated in the buffer pool (-)

- What if a force approach is used?
  - We do not have to *redo* the changes of a committed transaction (+)
  - But this results in excessive page I/O costs (e.g., when a highly used page is updated in succession by 20 transactions, it would be written to disk 20 times!) (-)

Steal vs. No-Steal and Force vs. No-Force Approaches (*Cont'd*)

- We indeed have four alternatives that we can employ:

	No-Steal	Steal
Force	Trivial, but undesired	High I/O cost, but modified pages need not fit in the buffer pool
No-Force	Low I/O cost, but modified pages need to fit in the buffer pool	Low I/O cost, and modified pages need not fit in the buffer pool ✓

- Most DBMSs use a *steal, no-force approach*

Outline

- Concurrency Control without Locking
- The ACID Properties
- The Steal, No-Force Approach
- Logging and the WAL Protocol ✓
- The Log

Logging and the WAL Property

- In order to recover from failures, the recovery manager maintains a *log* of all modifications to the database on *stable storage* (which should survive crashes)

- After a failure, the DBMS "replays" the log to:
  - Redo committed transactions
  - Undo uncommitted transactions

- Caveat: A log record describing a change must be written to stable storage ***before*** the change is made
  - This is referred to as the *Write-Ahead Log* (*WAL*) *property*

The WAL Protocol

- WAL is the fundamental rule that ensures that a record of every change to the database is available after a crash

- What if a transaction made a change, committed, then a crash occurred (i.e., no log is kept "before" the crash)?
    - The *no-force approach* entails that this change may not have been written to disk before the crash
    - Without a record of this change, there would be no way to ensure that the committed transaction survives the crash
    - Hence, durability cannot be guaranteed!

To guarantee ***durability***, a record for every change must be written to stable storage *before the change is made*

The WAL Protocol (*Cont'd*)

- WAL is the fundamental rule that ensures that a record of every change to the database is available after a crash

- What if a transaction made a change, was progressing, and a crash occurred?
  - The *steal approach* entails that this change may have been written to disk before the crash
  - Without a record of this change, there would be no way to ensure that the transaction can be rolled back (i.e., its effects would be unseen)
  - Hence, atomicity cannot be guaranteed!

To guarantee ***atomicity***, a record for every change must be written to stable storage *before the change is made*

Outline

- Concurrency Control without Locking
- The ACID Properties
- The Steal, No-Force Approach
- Logging and the WAL Protocol
- The Log ✓

The Log

- The log is ***a file of records*** stored in stable storage

- Every log record is given a unique id called the Log Sequence Number (LSN)
  - LSNs are assigned in a monotonically increasing order (this is required by the ARIES recovery algorithm- *later*)

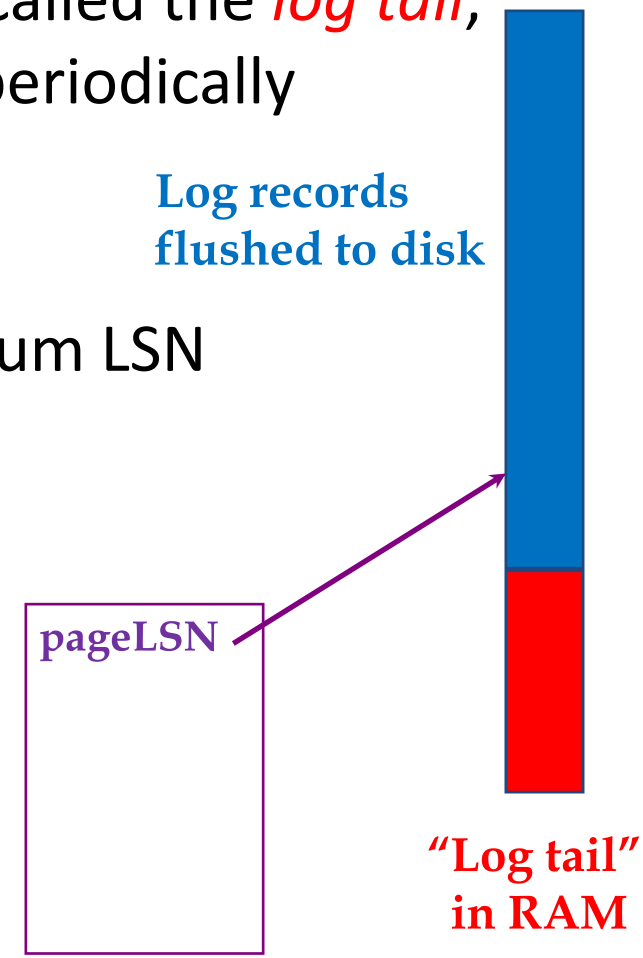
- Every page contains the LSN of the *most recent* log record, which describes a change to this page
  - This is called the pageLSN

The Log (*Cont'd*)

- The most recent portion of the log, called the *log tail*, is kept in main memory and *forced* periodically to disk

- The DBMS keeps track of the maximum LSN flushed to disk so far
  - This is called the flushedLSN

- As per the WAL protocol, before a page is written to disk, pageLSN $\leq$ flushedLSN



When to Write Log Records?

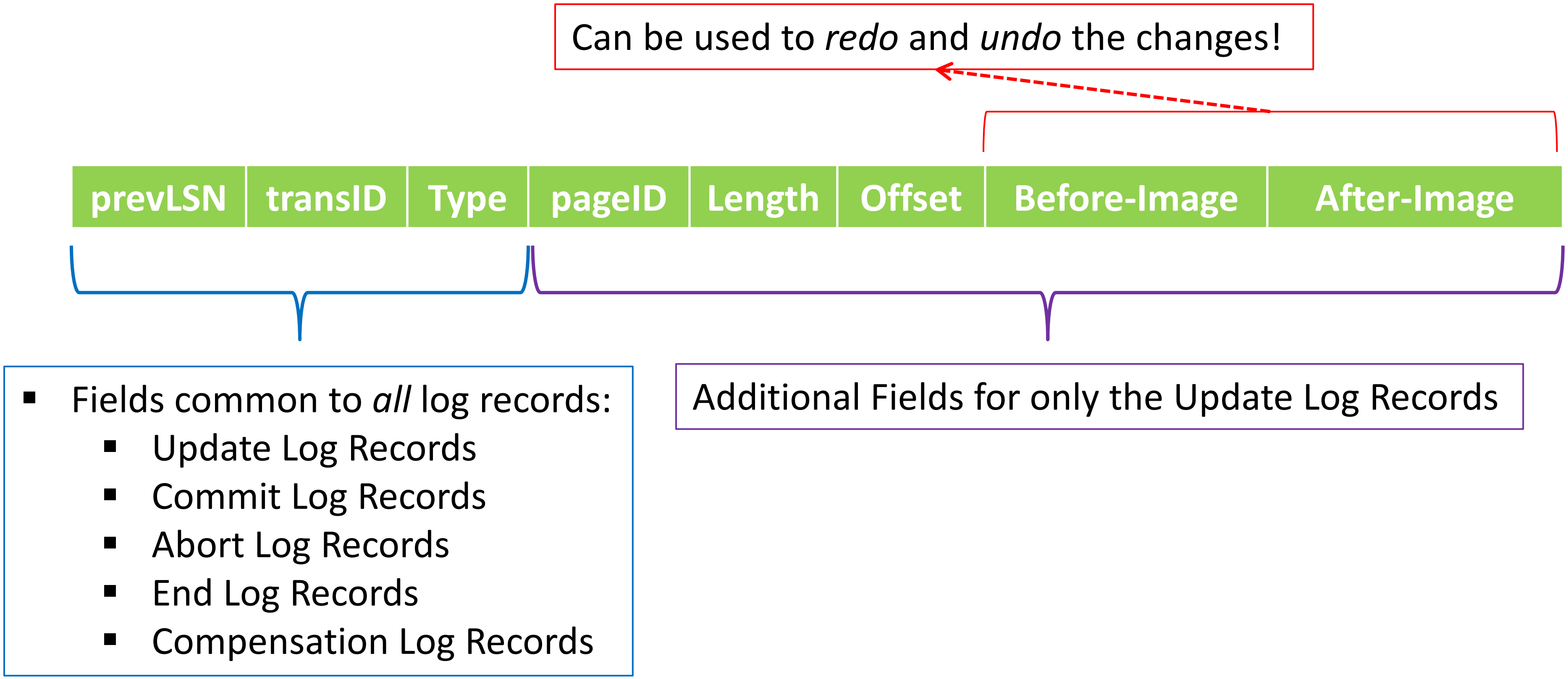
- A log record is written after:
  - Updating a Page
    - An *update log record* is appended to the log tail
    - The pageLSN of the page is set to the LSN of the update log record
  - Committing a Transaction
    - A *commit log record* is appended to the log tail
    - The log tail is written to stable storage, up to and including the commit log record
  - Aborting a Transaction
    - An *abort log record* is appended to the log tail
    - An undo is initiated for this transaction

When to Write Log Records?

- A log record is written after:
  - Ending (After Aborting or Committing) a Transaction:
    - Additional steps are completed (*later*)
    - An *end log record* is appended to the log tail
  - Undoing an Update
    - When the action (described by an update log record) is undone, a *compensation log record* (CLR) is appended to the log tail
    - CLR describes the action taken to undo the action recorded in the corresponding update log record

Log Records

- The fields of a log record are usually as follows:



Other Recovery-Related Structures

- In addition to the log, the following two tables are maintained:
  - The Transaction Table
    - One entry ***E*** for each <u>active</u> transaction
    - ***E*** fields are:
      - *Transaction ID*
      - *Status*, which can be "*Progress*", "*Committed*" or "*Aborted*"
      - *lastLSN*, which is the most recent log record for this transaction
  - The Dirty Page Table
    - One entry ***E'*** for each <u>dirty</u> page in the buffer pool
    - ***E'*** fields are:
      - *Page ID*
      - *recLSN*, which is the LSN of the first log record that caused the page to become dirty

An Example

PageID	recLSN
P500	
P600	

Dirty Page Table

prevLSN	transID	Type	pageID	Length	Offset	Before-Image	After-Image
	T1000	Update	P500	3	21	ABC	DEF
	T2000	Update	P600	3	41	HIJ	KLM
	T2000	Update	P500	3	20	GDE	QRS

LOG

TransID	lastLSN
T1000	
T2000	

Transaction Table

An Example

PageID	recLSN
P500	
P600	

Dirty Page Table

TransID	lastLSN
T1000	
T2000	

Transaction Table

prevLSN	transID	Type	pageID	Length	Offset	Before-Image	After-Image
	T1000	Update	P500	3	21	ABC	DEF
	T2000	Update	P600	3	41	HIJ	KLM
	T2000	Update	P500	3	20	GDE	QRS
	T1000	Update	P505	3	21	TUV	WXY

LOG

An Example

PageID	recLSN
P500	
P600	
P505	

Dirty Page Table

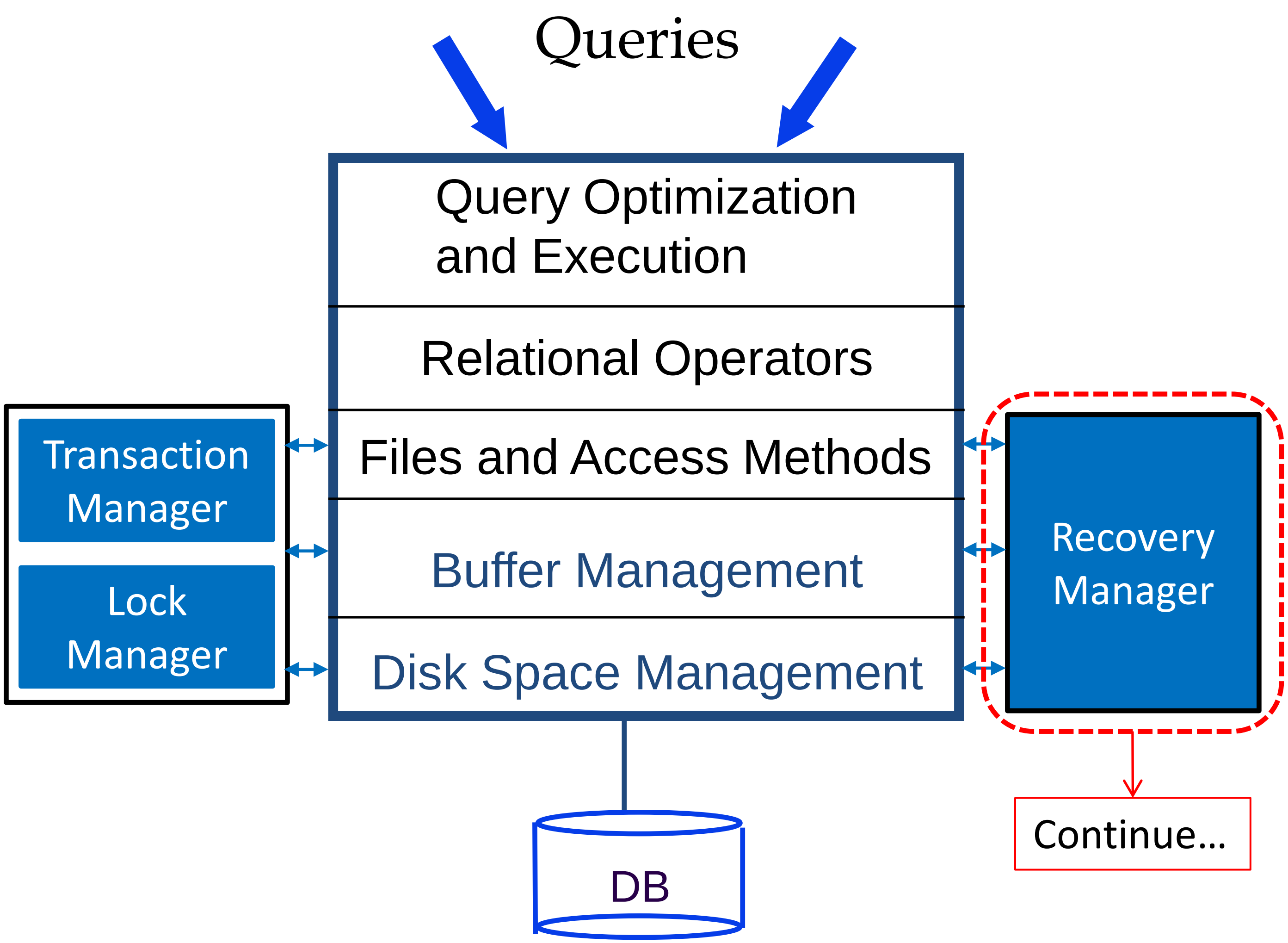
TransID	lastLSN
T1000	
T2000	

Transaction Table

prevLSN	transID	Type	pageID	Length	Offset	Before-Image	After-Image
	T1000	Update	P500	3	21	ABC	DEF
	T2000	Update	P600	3	41	HIJ	KLM
	T2000	Update	P500	3	20	GDE	QRS
	T1000	Update	P505	3	21	TUV	WXY

LOG

Next Class


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
Continue…
DB