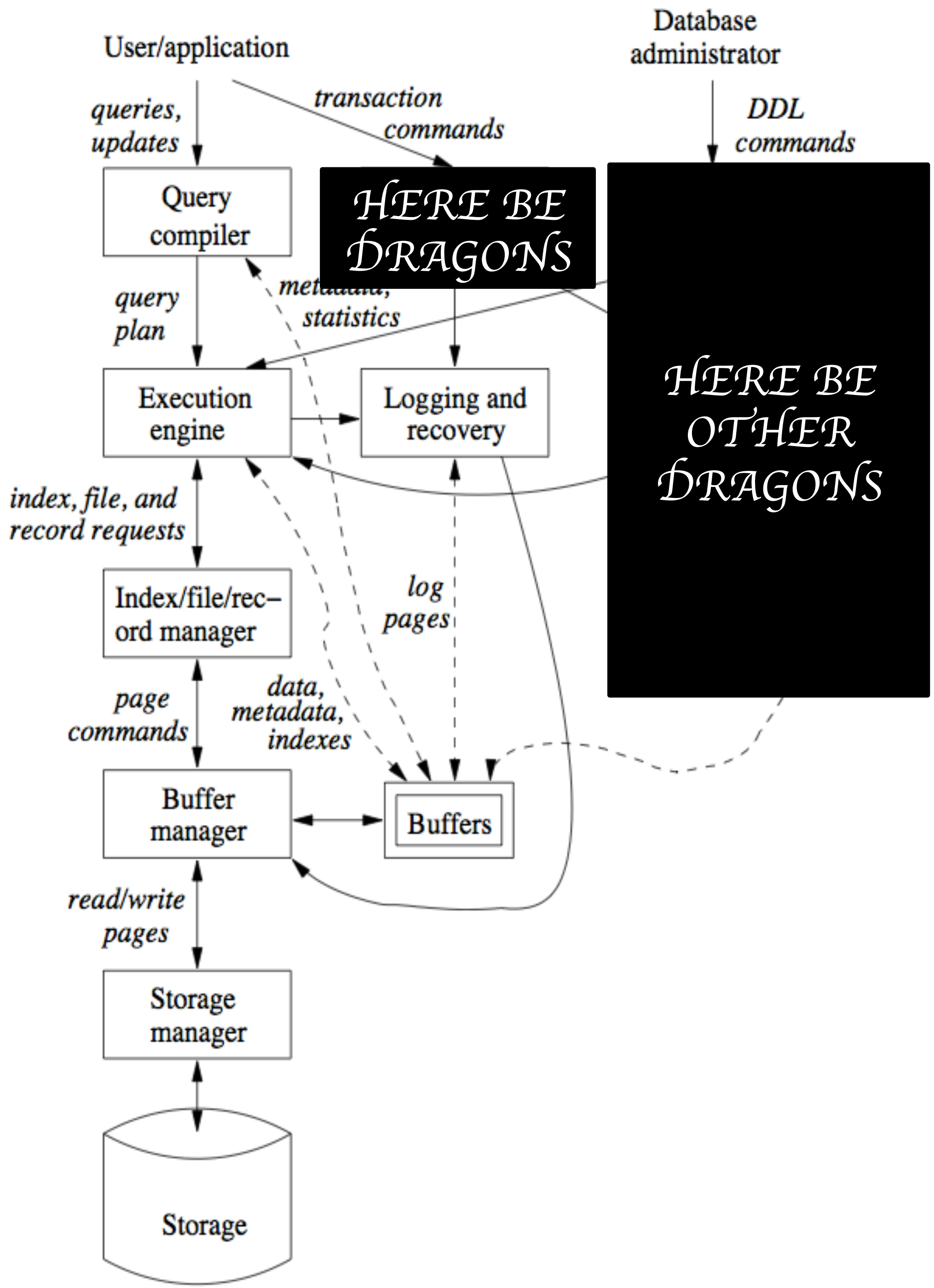
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

CS411 - Logging and Recovery

illinois.edu


User/application
Database administrator
queries, updates
transaction commands
DDL commands
Query compiler
HERE BE DRAGONS
query plan
statistics
Execution engine
Logging and recovery
HERE BE OTHER DRAGONS
index, file, and record requests
Index/file/rec–ord manager
log pages
page commands
data, metadata, indexes
Buffer manager
Buffers
read/write pages
Storage manager
Storage

Review

- Why are we studying logging?
- What does it mean for a transaction to be "durable" or "aborted"?
- What was the name of the logging we learned last time?
- What parts of ACID does logging help ensure?

Example

<START T_1>

<T_1,A,5>

<T_1,B,3>

<START T_2>

<T_1,C,7>

<T_2,D,5>

<COMMIT T_2>

<START T_3>

<T_3,A,2>

<START T_4>

<T_4,A,3>

<T_3,D,5>

<COMMIT T_4>

A=2

B=5

C=6

D=8

Example

<START T_1>
<T_1,A,5>
<T_1,B,3>
<START T_2>
<T_1,C,7>
<T_2,D,5>
<COMMIT T_2>

<START T_3>
<T_3,A,2>
<START T_4>
<T_4,A,3>
<T_3,D,5>
<COMMIT T_4>

A=5
B=3
C=7
D=5

illinois.edu

Undo Logging

- Rules:
  1. Add <T,X,v> to log ***before*** new value of X is written to disk
  2. Add <COMMIT T> ***after all*** results for T have been written to disk

Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8>
READ(B,t)	8	16	8	8	8	
t=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	
						<COMMIT T>
FLUSH LOG						

Recovery

- If we see <START T> without corresponding <COMMIT T>, must undo the transaction
- When the system fails, scan log backward
  - If we see <COMMIT F>, ignore transaction F
  - For every other transaction T, when we see <T,X,v>, write v to X

Undo Logging

- Two problems
  1. If we want to recover, we have to read the entire log file
  2. Must write all changes to disk before committing

Checkpointing

- Periodically:
  - stop accepting new transactions.
  - Wait for all active transactions to commit or abort
  - Flush the log
  - Write <CKPT> into the long
  - Resume transactions
- Recovery only needs to read to <CKPT>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<T_2,C,15>

<T_1,D,20>

<COMMIT T_1>

<COMMIT T_2>

<CKPT>

<START T_3>

<T_3,E,25>

<T_3,F,30>

Checkpointing

- Problem
  - We have to halt all incoming transactions
  - Active transactions might take a long time to complete
  - Our entire DBMS is just waiting!

Nonquiescent Checkpointing

- "*quiescent* - adj. marked by inactivity or repose"
  - Merriam-Webster

- "*quiescent* - adj. being lazy"
  -Ryan Cunnigham

Nonquiescent Checkpointing

1. Write <START CKPT (T_1,T_2,…,T_n)> for all active transactions
2. When all of T_1,T_2,…,T_n commit or abort, write <END CKPT>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<START CKPT(T_1,T_2)>

<T_2,C,15>

<START T_3>

<T_1,D,20>

<COMMIT T_1>

<T_3,E,25>

<COMMIT T_2>

<END CKPT>

<T_3,F,30>

Recovery

- Scan log backward:
  - If we meet <END CKPT> first, continue backward to the next <START CKPT>
  - If we meet <START CKPT (T_1,T_2,…,T_n)> first, continue backward
    - only need to undo T_1,T_2,…,T_n

Example

<START T_1>

<T_1,A,5>

<START T_2>

<START CKPT(T_1,T_2)>

<T_2,C,15>

<START T_3>

<T_1,D,20>

<COMMIT T_1>

<T_3,E,25>

<COMMIT T_2>

<END CKPT>

<T_3,F,30>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<T_1,C,2>

<START CKPT(T_1,T_2)>

<T_2,C,15>

<START T_3>

<T_1,D,20>

<COMMIT T_1>

<T_3,E,25>

<T_3,F,30>

Undo Logging

- Two problems
  1. ~~If we want to recover, we have to read the entire log file~~
  2. Must write all changes to disk before committing

Redo Logging

- Main idea:
  - Log what we're going to change ***before*** we change it
  - Anything not logged was never written
  - After a crash, repeat all the committed transactions

Redo Logging

1. Before modifying X to value v on the disk, log <T,X,v>
2. Before modifying X on the disk, log <COMMIT T>

Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,16>
READ(B,t)	8	16	8	8	8	
t=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,16>
						<COMMIT T>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
OUTPUT(B)	16	16	16	16	16	

Recovery

1. Scan log to identify all committed transactions
2. Scan log from beginning for each <T,X,v>
   – If T is not committed, ignore
   – If T is committed, write v into X

Example

<START T_1>
<T_1,A,5>
<T_1,B,3>
<START T_2>
<T_1,C,7>
<T_2,D,5>
<COMMIT T_2>

<START T_3>
<T_3,A,2>
<START T_4>
<T_4,A,3>
<T_3,D,5>
<COMMIT T_4>

A=2
B=5
C=6
D=8

Example

<START T_1>

<T_1,A,5>

<T_1,B,3>

<START T_2>

<T_1,C,7>

<T_2,D,5>

<COMMIT T_2>

<START T_3>

<T_3,A,2>

<START T_4>

<T_4,A,3>

<T_3,D,5>

<COMMIT T_4>

A=3

B=5

C=6

D=5

Nonquiescent Checkpointing

1. Write <START CKPT (T_1,T_2,…,T_n)> for all active transactions
2. Write all modified elements for committed transactions to the disk
3. Write <END CKPT>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<COMMIT T_1>

<T_2,B,10>

<START CKPT (T_2)>

<T_2,C,15>

<START T_3>

<END CKPT>

<COMMIT T_2>

<COMMIT T_3>

Recovery

- Scan log backward:
  - If we meet <END CKPT> first, continue backward to the next <START CKPT $(T_1, T_2, \ldots, T_{n)}$>
    - Redo all committed transactions in $T_1, T_2, \ldots, T_n$ or started after checkpoint
  - If we meet <START CKPT $(T_1, T_2, \ldots, T_n)$> first, this checkpoint is bunk. Continue looking for <END CKPT>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<COMMIT T_1>

<T_2,B,10>

<START CKPT (T_2)>

<T_2,C,15>

<START T_3>

<T_3,D,20>

<END CKPT>

<COMMIT T_2>

<COMMIT T_3>

Example

<START T_1>

<T_1,A,5>

<START T_2>

<COMMIT T_1>

<T_2,B,10>

<START CKPT (T_2)>

<T_2,C,15>

<START T_3>

<T_3,D,20>

<END CKPT>

<COMMIT T_2>

<COMMIT T_3>

Comparison

Undo Logging

- Data must be written to disk after transaction finishes
- More Disk I/O
- Less Memory

Redo Logging

- Data must remain in memory until transaction finishes
- Less Disk I/O
- More memory

Undo/Redo Logging

- Best of both worlds:
  - Keep logs such that we can either undo or redo transactions
  - When recovering, undo all uncommitted transactions and redo all committed transactions

Undo/Redo Logging

1. Whenever a transaction modifies a value, write <T,X,v,w> to the log where
   – v is the old value
   – w is the new value
   – We don't care whether or not the changes have been made on disk

Example

Action	t	Mem A	Mem B	Disk A	Disk B	Log
						<START T>
READ(A,t)	8	8		8	8	
t=t*2	16	8		8	8	
WRITE(A,t)	16	16		8	8	<T,A,8,16>
READ(B,t)	8	16	8	8	8	
t=t*2	16	16	8	8	8	
WRITE(B,t)	16	16	16	8	8	<T,B,8, 16>
FLUSH LOG						
OUTPUT(A)	16	16	16	16	8	
						<COMMIT T>
OUTPUT(B)	16	16	16	16	16	

Recovery

1. Redo all committed transactions from last to first
2. Undo all uncommitted transactions in reverse

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<START T_3>

<COMMIT T_1>

<T_2,B,9,10>

<T_2,C,14,15>

<START T_4>

<T_3,D,19,20>

<T_4,E,30,31>

<COMMIT T_3>

<T_4,F,43,44>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<START T_3>

<COMMIT T_1>

<T_2,B,9,10>

<T_2,C,14,15>

<START T_4>

<T_3,D,19,20>

<T_4,E,30,31>

<COMMIT T_3>

<T_4,F,43,44>

Nonquiescent Checkpointing

1. Write <START CKPT (T_1,T_2,…,T_n)> for all active transactions
2. Write ***all*** modified elements to the disk
3. Write <END CKPT>

Recovery

- Identify matching <START CKPT> and <END CKPT>
  - Undo all uncommitted transactions that were
    - active at <START CKPT>
    - started after START CKPT
  - Redo all committed transactions that committed after START CKPT

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

<COMMIT T_2>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT ($T_{2)}$>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

<COMMIT T_2>

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

<T_2,C,14,15>

<START T_3>

<T_3,D,19,20>

<END CKPT>

<COMMIT T_2>

illinois.edu

Example

<START T_1>

<T_1,A,4,5>

<START T_2>

<COMMIT T_1>

<T_2,B,9,10>

<START CKPT (T_2)>

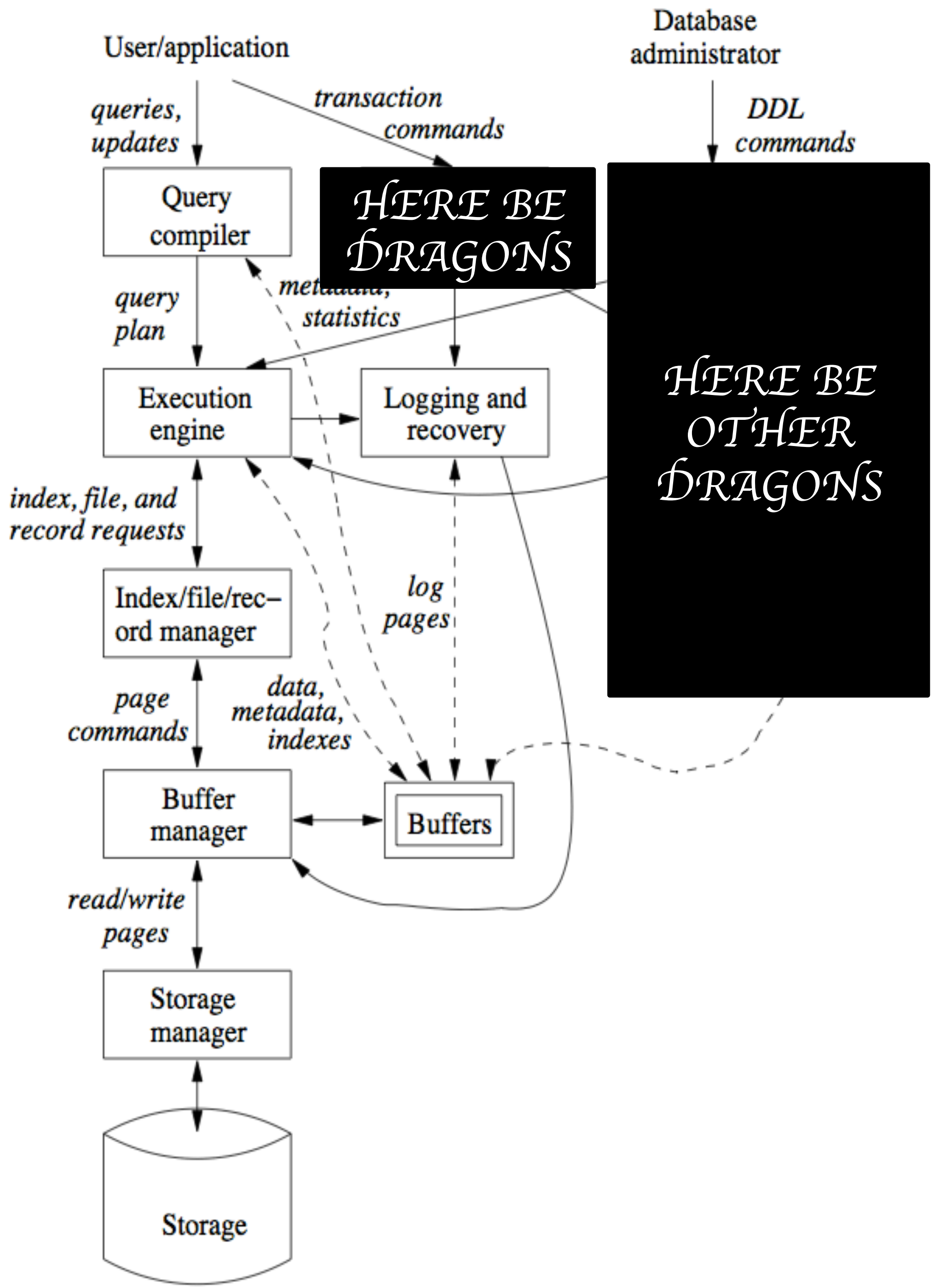
<T_2,C,14,15>

<START T_3>

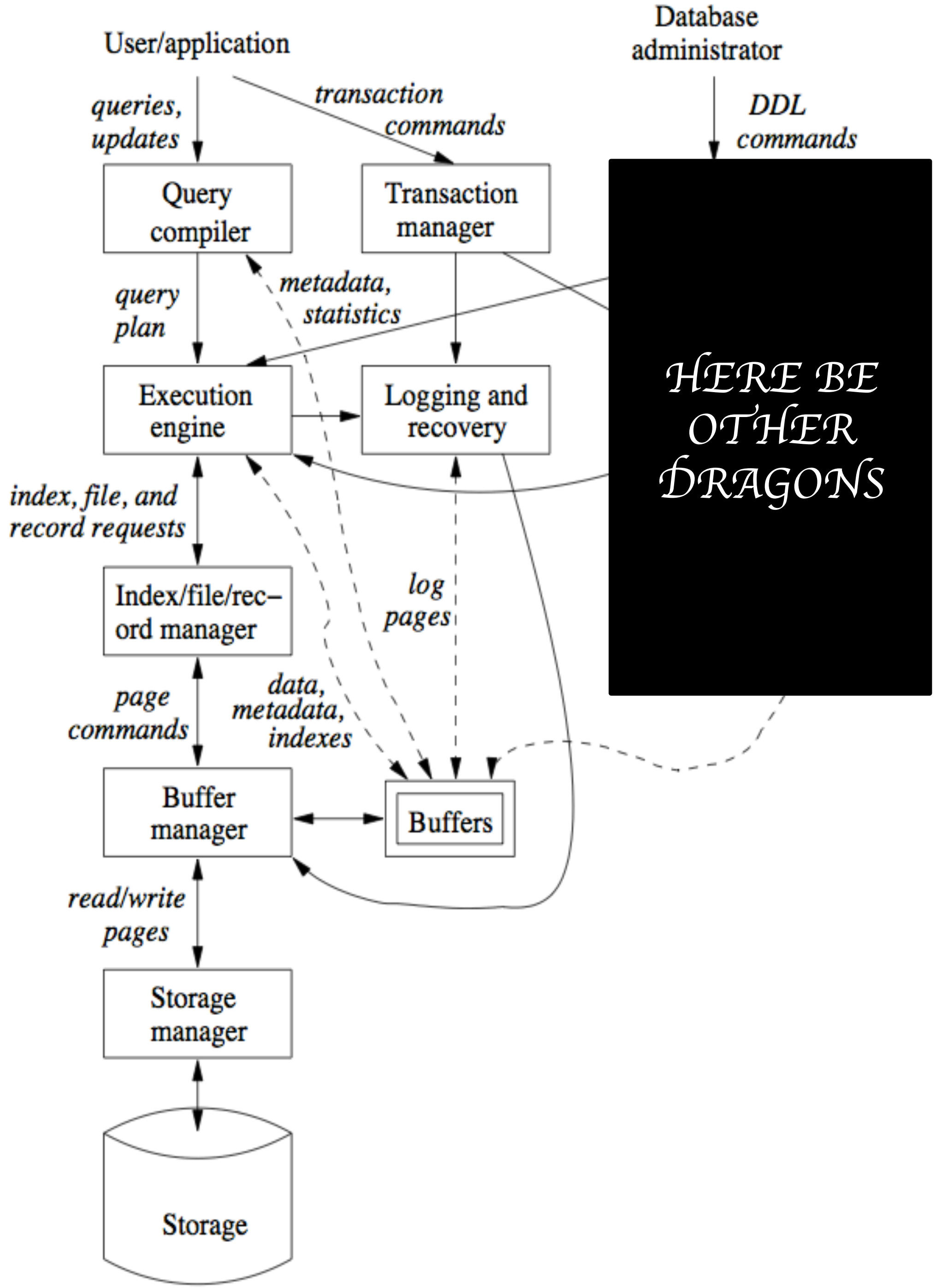
<T_3,D,19,20>

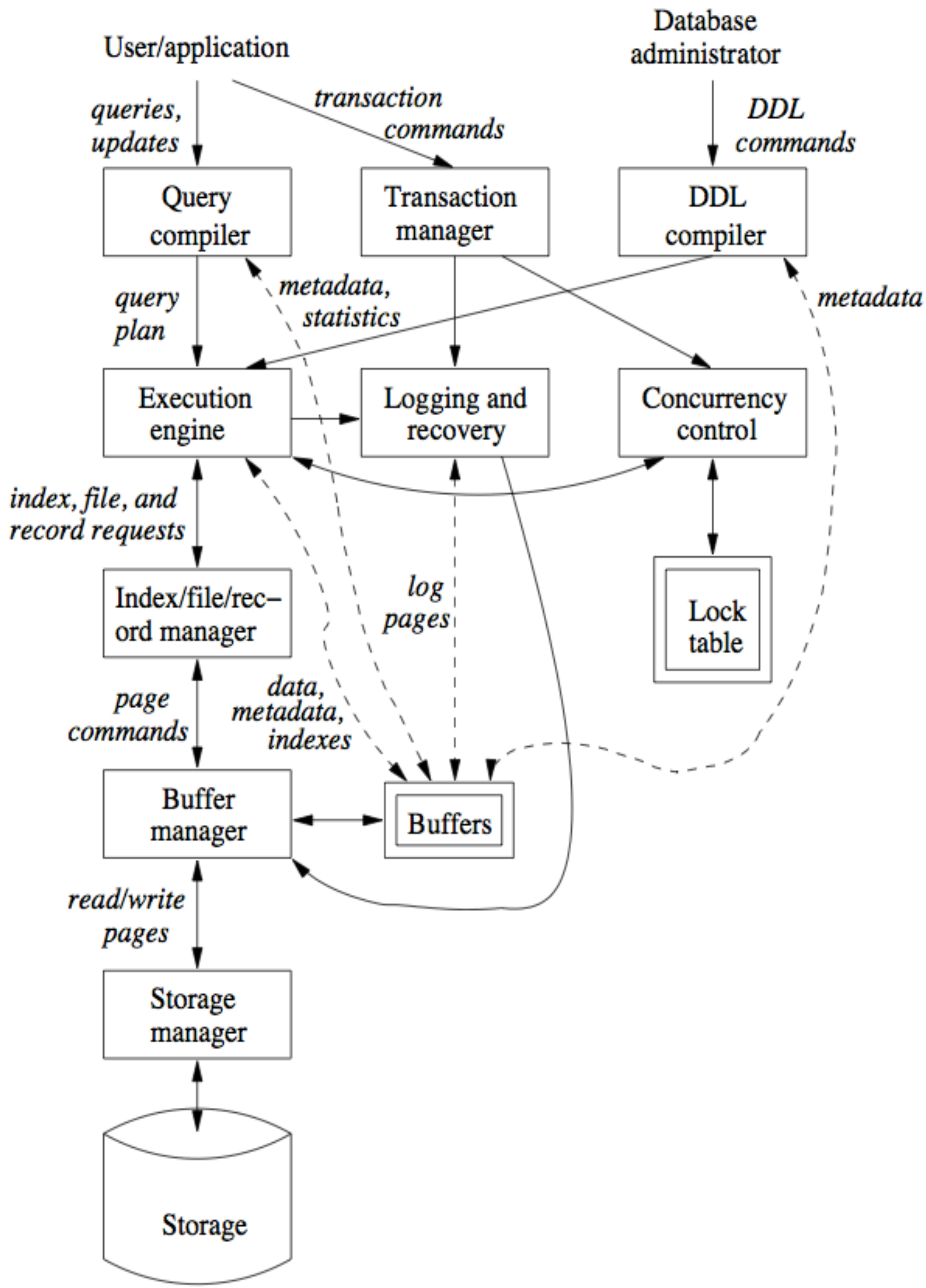
<END CKPT>

<COMMIT T_2>



illinois.edu


User/application
Database administrator
queries, updates
transaction commands
DDL commands
Query compiler
Transaction manager
query plan
metadata, statistics
Execution engine
Logging and recovery
HERE BE OTHER DRAGONS
index, file, and record requests
Index/file/rec–ord manager
log pages
page commands
data, metadata, indexes
Buffer manager
Buffers
read/write pages
Storage manager
Storage



illinois.edu

Concurrency Control

- Interaction among transactions can cause the database state to be inconsistent
- Need to regulate the order of individual steps of transactions
- Assure that transactions preserve consistency when executing simultaneously

Schedule

- Time-ordered sequence of the important actions taken by one or more transactions
- We only care about READ and WRITE actions
  - The same database primitives from logging

Example

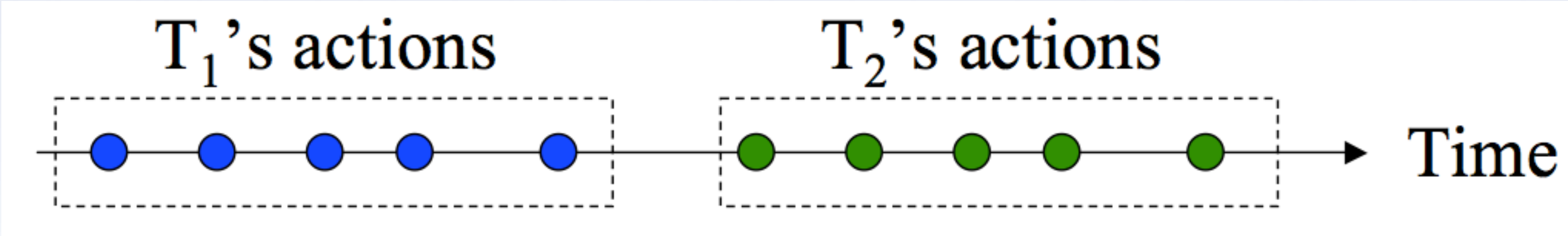
T_1	T_2
READ(A, t)	READ(A, s)
t := t + 100	s := s * 2
WRITE(A, t)	WRITE(A, s)
READ(B, t)	READ(B, s)
t := t + 100	s := s * 2
WRITE(B, t)	WRITE(B, s)

Example

- Our running example will have the constraint that A=B

Serial Schedule

- Execute one transaction first, then the other



illinois.edu

Example

T_1	T_2	A	B
		25	25
READ(A, t)			
t := t + 100			
WRITE(A, t)		125	
READ(B, t)			
t := t + 100			
WRITE(B, t)			125
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	250	
	READ(B, s)		
	s := s * 2		
	WRITE(B, s)		250

Example

T_1	T_2	A	B
		25	25
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	50	
	READ(B, s)		
	s := s * 2		
	WRITE(B, s)		50
READ(A, t)			
t := t + 100			
WRITE(A, t)		150	
READ(B, t)			
t := t + 100			
WRITE(B, t)			150

Concurrent Execution

- Improves throughput
  - CPU actions and I/O actions can happen in parallel
- Reduces average waiting time
  - Short transactions don't have to wait for long transactions to complete
- Just like with operating systems

Serializable Schedule

- A schedule is *serializ**able*** if it has the same effect as some serial schedule

Example

T_1	T_2	A	B
		25	25
READ(A, t)			
t := t + 100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	250	
READ(B, t)			
t := t + 100			
WRITE(B, t)			125
	READ(B, s)		
	s := s * 2		
	WRITE(B, s)		250

Example

T_1	T_2	A	B
		25	25
READ(A, t)			
t := t + 100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s * 2		
	WRITE(A, s)	250	
	READ(B, s)		
	s := s * 2		50
	WRITE(B, s)		
READ(B, t)			
t := t + 100			
WRITE(B, t)			150

Notation

- We don't care about values
- Just ***what*** is accessed, ***who*** accessed it, and ***how*** it is accessed
  - Read: $r_i(X)$ trasaction i reads element X
  - Write: $w_i(X)$ transaction i writes element X

Example

T1: $r_1(A); w_1(A); r_1(B); w_1(B);$

T2: $r_2(A); w_2(A); r_2(B); w_2(B);$

S: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Conflicting swaps

- A pair of consecutive actions conflict if changing order changes the behavior of a transaction:
    1. two actions from the same transaction
    2. $r_i(X), w_j(X)$ or $w_i(X), r_j(X)$
    3. $w_i(X), w_j(X)$

Conflict Serializable

- Making a series of nonconflicting swaps to a schedule, we can produce a serial schedule

Example

r1(A); w1(A); r2(A); w2(A); r1(B); w1(B); r2(B); w2(B);

r1(A); w1(A); r2(A); r1(B); w2(A); w1(B); r2(B); w2(B);

r1(A); w1(A); r1(B); r2(A); w2(A); w1(B); r2(B); w2(B);

r1(A); w1(A); r1(B); r2(A); w1(B); w2(A); r2(B); w2(B);

r1(A); w1(A); r1(B); w1(B); r2(A); w2(A); r2(B); w2(B);

Next Week…

- We’ll investigate these notions more deeply
  - How to tell if a schedule is conflict-serializable
  - Mechanisms for enforcing serializability