Review (Past year question)

- Consider the relations R1(A,B,C), R2(C,D,E) and R3(E,F), with primary keys A, C, E respectively. Assume that R1 has 10000 tuples, R2 has 15000 tuples and R3 has 7500 tuples. For simplicity, assume that all tuples (including the query result) have the same size, and that each page can contain 10 tuples of R. Consider the query: R1 JOIN R2 JOIN R3. Assume that all attributes are of the same size, and any join output will include all attributes of all relations. Further, assume records do not span pages. Assuming the data are uniformly distributed, estimate the result size of the query.
- List all possible plans assuming only left-deep search space is considered (assuming only one join method). You may assume that cross product are to be avoided.
- Compute the cost for each of the above plans you listed to determine the optimal plan. For simplicity, you may assume that only the nested-block join is supported, the buffer size is 100 pages, and all intermediate results are to be stored in secondary storage.

Review (Past year question)

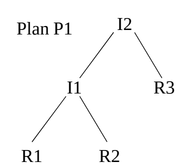
- Consider the relations R1(A,B,C), R2(C,D,E) and R3(E,F), with primary keys A, C, E respectively. Assume that R1 has 10000 tuples, R2 has 15000 tuples and R3 has 7500 tuples. For simplicity, assume that all tuples (including the query result) have the same size, and that each page can contain 10 tuples of R. Consider the query: R1 JOIN R2 JOIN R3. Assume that all attributes are of the same size, and any join output will include all attributes of all relations. Further, assume records do not span pages. Assuming the data are uniformly distributed, estimate the result size of the query.
  - Number of tuples = 10000
  - Number of pages = 10000/10 WRONG!!
  - Number of attributes per page = 30;
  - Number of result tuples per page = 30/8 = 3
  - Number of resultant pages = 10000/3

Review (Past year question)

- Assume Left Deep Tree plans and one join method. In total, there are 6 possible plans, but since cross products are not permitted, we end up with 4 plans
  - (R1 JOIN R2) JOIN R3
  - (R2 JOIN R1) JOIN R3
  - (R2 JOIN R3) JOIN R1
  - (R3 JOIN R2) JOIN R1

Review (Past year question)

- for each plan, compute the cost of each join. there are two points to note: (a) remember to include the cost to write out intermediate results, (b) the number of tuples per page may be different for each intermediate results.



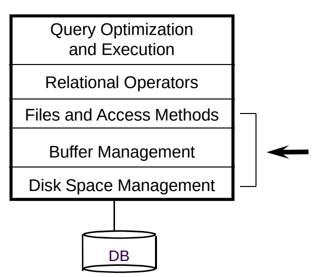
Cost of Plan P1 = Cost (R1 JOIN R2) +
Cost (I1 JOIN R3)

Size (I1) = 10000 tuples; 10000/5 pages
Cost (R1 JOIN R2) = 10000/10 +
1000/98*(15000/15)+
10000/5
Cost(I1 JOIN R3) = join cost + cost to output I2

Transaction Management Overview

There are three side effects of acid. Enhanced long term memory, decreased short term memory, and I forget the third.
- Timothy Leary

Structure of a DBMS



These layers must consider concurrency control and recovery (Transaction, Lock, Recovery Managers)

Transactions

- Transaction ("xact")- DBMS's abstract view of a user program (or activity):
  - A sequence of reads and writes of database objects, e.g., a transaction that transfers $100 from account A to account B can be expressed as:
    - Read Account A;
    - Write Updated Account A ($100 less);
    - Read Account B;
    - Write Updated Account B ($100 more);
  - Unit of work that must commit or abort as an atomic unit
- Transaction Manager controls the execution of transactions.
- User's program logic is invisible to DBMS!
  - Arbitrary computation possible on data fetched from the DB
  - The DBMS only sees data read/written from/to the DB.

ACID properties of Transaction Executions

- Atomicity: All actions in the Xact happen, or none happen.
- Consistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- Isolation: Execution of one Xact is isolated from that of other Xacts.
- Durability: If a Xact commits, its effects persist.

Atomicity and Durability

A.C.I.D.

- A transaction ends in one of two ways:
  - *commit* after completing all its actions
    - "commit" is a contract with the caller of the DB
  - *abort* (or be aborted by the DBMS) after executing some actions.
    - Or *system crash* while the xact is in progress; treat as abort.
- Two important properties for a transaction:
  - *Atomicity* : Either execute all its actions, or none of them
  - *Durability* : The effects of a committed xact must survive failures.
- DBMS ensures the above by *logging* all actions (Recovery):
  - *Undo* the actions of aborted/failed transactions.
  - *Redo* actions of committed transactions not yet propagated to disk when system crashes.

Transaction Consistency

A.C.I.D.

- Transactions preserve DB *consistency*
  - Given a consistent DB state, produce another consistent DB state
- DB Consistency expressed as a set of declarative Integrity Constraints
  - CREATE TABLE/ASSERTION statements
    - E.g. Each CS186 student can only register in one project group. Each group must have 2 students.
  - Application-level
    - E.g. Bank account total of each customer must stay the same during a "transfer" from savings to checking account
- Transactions that violate ICs are aborted
  - That's all the DBMS can automatically check!

Isolation (Concurrency)

A.C.I.D.

- DBMS interleaves actions of many xacts concurrently
  - Actions = reads/writes of DB objects
- DBMS ensures xacts do not "step onto" one another.
- Each xact executes as if it were running by itself.
  - Concurrent accesses have no effect on a Transaction's behavior
  - Net effect *must be* identical to executing all transactions for *some* serial order.
  - Users & programmers think about transactions in isolation
    - Without considering effects of other concurrent transactions!

Concurrency Control & Recovery

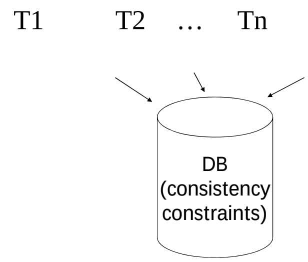
- Concurrency Control
  - Provide correct and highly available data access in the presence of concurrent access by many users
- Recovery
  - Ensures database is fault tolerant, and not corrupted by software, system or media failure
  - 24x7 access to mission critical data
- A boon to application authors!
  - Existence of CC&R allows applications to be written without explicit concern for concurrency and fault tolerance

Concurrency Control

Smile, it is the key that fits the lock of everybody's heart.

Anthony J. D'Angelo,
The College Blue Book

Concurrency Control



Improves latency and throughput

Example:

T1:	Read(A)	T2:	Read(A)
	A ← A+100		A ← A×2
	Write(A)		Write(A)
	Read(B)		Read(B)
	B ← B+100		B ← B×2
	Write(B)		Write(B)

Constraint: A=B

Schedule A: Serial Schedule

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule B

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Schedule D

Same as Schedule C but with new T2'

T1	T2'	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×1;		
	Write(A);	125	
	Read(B);B ← B×1;		
	Write(B);		25
Read(B); B ← B+100;			
Write(B);			125
		125	125

What are good schedules?

- Want schedules that are "good", regardless of
- Only look at order of read and writes

Example:

$Sb=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

What are good schedules?

- Want schedules that are "good", regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

Example:

$Sb=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

Example:

Sb=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

$T_1 \rightarrow T_2$ $T_1 \rightarrow T_2$

$Sb'=r_1(A)w_1(A)\ r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

T_1 T_2

no cycles ⇒ Sb is "equivalent" to a serial schedule (in this case T_1,T_2)

Example (Cont)

$Sd=r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$

Example (Cont)

$Sd=r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$

Example (Cont)

Sd = r_1(A)w_1(A)r_2(A)w_2(A) r_2(B)w_2(B)r_1(B)w_1(B)

$T_1 \rightarrow T_2$

Also, $T_2 \rightarrow T_1$

T1 T2

Sd cannot be rearranged into a serial schedule

Sd is not "equivalent" to any serial schedule

Sd is "bad"

Concepts

Transcation: sequence of r_i(x), w_i(x) actions

Conflicting actions: r1(A) / W2(A); W1(A) / r2(A); W1(A) / W2(A)

Schedule: represents chronological order in which actions are executed

Serial schedule: no interleaving of actions or transactions

Serializable schedule: a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule

Definition

S_1, S_2 are conflict equivalent schedules

if S_1 can be transformed into S_2 by a series of swaps on non-conflicting actions.

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.

Note: (a) Some "serializable" schedules are NOT conflict serializable.
A price we pay to achieve efficient enforcement.
(b) There are alternative (weaker) notions of serializability.

Conflict-Serializability is NOT necessary for Serializability

- S1: w1(Y); w1(X); w2(Y); w2(X); w3(X)
  - Serial schedule
- S2: w1(Y); w2(Y); w2(X); w1(X); w3(X)
  - Serializable schedule since effect is same as S1
- S1, S2 not conflict equivalent

Precedence graph P(S) (S is schedule)

Nodes: transactions in S

Arcs: Ti → Tj whenever

- p_i(A), q_j(A) are actions in S
- $p_i(A) <_S q_j(A)$
- at least one of p_i, q_j is a write

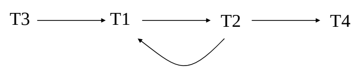
Exercise:

- What is P(S) for
  S = w_3(A) w_2(C) r_1(A) w_1(B) r_1(C) w_2(A) r_4(A) w_4(D)

- Is S serializable?

Exercise:

- What is P(S) for
  S = $w_3(A)\ w_2(C)\ r_1(A)\ w_1(B)\ r_1(C)\ w_2(A)\ r_4(A)\ w_4(D)$



- Is S serializable?

Theorem

$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

S_1, S_2 conflict equivalent $\Rightarrow$ $P(S_1)=P(S_2)$???

$P(S_1)=P(S_2)$ $\Rightarrow$ S_1, S_2 conflict equivalent ???

Theorem

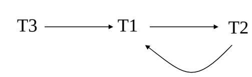
$P(S_1)$ acyclic $\Longleftrightarrow$ S_1 conflict serializable

S_1, S_2 conflict equivalent $\Rightarrow$ $P(S_1)=P(S_2)$???

$P(S_1)=P(S_2)$ $\not\Rightarrow$ S_1, S_2 conflict equivalent ???

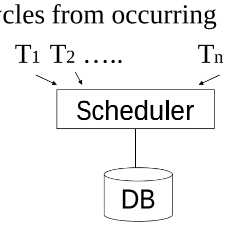
$P(S_1)=P(S_2)$ $\not\Rightarrow$ S_1, S_2 conflict equivalent

- S1 = $w_3(A)\ w_2(C)\ r_1(C)\ r_1(A)\ w_2(B)\ w_1(B)\ w_2(A)$
- S2 = $w_3(A)\ r_1(A)\ r_2(B)\ w_1(B)\ r_1(C)\ w_2(C)\ w_2(A)$



How to enforce serializable schedules?

prevent P(S) cycles from occurring

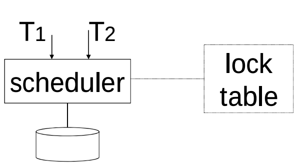


A locking protocol

Two new actions:

lock (exclusive): l_i (A)

unlock: u_i (A)



Rules

Rule #1: Well-formed transactions

Ti: … li(A) … pi(A) … ui(A) ...

Rule #2: Legal scheduler

S = …….. li(A) ………... ui(A) ……...

no lj(A)

Exercise:

- **What schedules are legal?**
  What transactions are well-formed?

  S1 = $l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)$ $r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$ — Not legal

  S2 = $l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)$ — Not well-formed
  $l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)$ — Not legal

  S3 = $l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)$ $l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)$

Schedule F (Schedule C with locking)

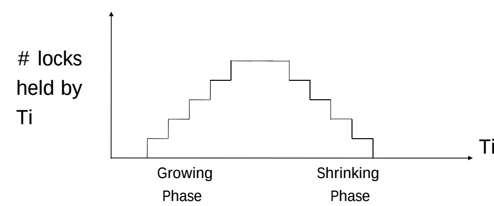
T1	T2	A	B
		25	25
$l_1(A)$;Read(A)			
A←A+100;Write(A);$u_1(A)$		125	
	$l_2(A)$;Read(A)		
		250	
	$l_2(B)$;Read(B)		
	B ← Bx2;Write(B);$u_2(B)$		50
$l_1(B)$;Read(B)			
B ← B+100;Write(B);$u_1(B)$			150
		250	150

Rules 1 & 2 are not enough!

Rule #3 Two phase locking (2PL) for transactions

Ti = ……. li(A) ………... ui(A) ……...

no unlocks — no locks



Schedule G

T1	T2
$l_1(A)$;Read(A)	
A ← A+100;Write(A)	
$l_1(B)$; $u_1(A)$	
	$l_2(A)$;Read(A)
	A ← Ax2;Write(A);$l_2(B)$ (delayed)
Read(B);B ← B+100	
Write(B); $u_1(B)$	
	$l_2(B)$; $u_2(A)$;Read(B)
	B ← Bx2;Write(B);$u_2(B)$;

Schedule H (T2 reversed)

T1	T2
$l_1(A)$; Read(A)	$l_1(B)$;Read(B)
A ← A+100;Write(A)	B ← Bx2;Write(B)
delayed	delayed

Transactions are *deadlocked*

- Some deadlocked transactions are *rolled back* (and all their actions undone)

Theorem

Rules #1,2,3 (2PL) $\Rightarrow$ conflict serializable schedule

What else?

- Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency….
  - Shared locks
  - Multiple granularity
  - Inserts, deletes and phantoms
  - **Other types of CC mechanisms**

Shared locks

So far (exclusive lock):

S1 = ...l_1(A) r_1(A) u_1(A) … l_2(A) r_2(A) u_2(A) …

Do not conflict

Instead:

S2=... ls_1(A) r_1(A) **ls_2(A)** r_2(A) .... us_1(A) us_2(A)

Lock actions

l-t_i(A): lock A in t mode (t is S or X)

u-t_i(A): unlock t mode (t is S or X)

Shorthand:

u_i(A): unlock whatever modes T_i has locked A

Rule #1 Well formed transactions

T_i =... l-S_1(A) … r_1(A) … u_1 (A) …

T_i =... l-X_1(A) … w_1(A) … u_1 (A) …

What about transactions that read and write same object?

Option 1: Request exclusive lock

Ti = ...l-X1(A) … r1(A) ... w1(A) ... u(A) …

Option 2: Upgrade

(E.g., need to read, but don't know if will write...)

T_i=... l-S_1(A) … r_1(A) ... l-X_1(A) …w_1(A) ...u(A)…

Think of
- Get 2nd lock on A, or
- Drop S, get X lock

Rule #2 Legal scheduler

S = ...l-S_i(A) … … u_i(A) …

←→ no l-X_j(A)

S = ... l-X_i(A) … … u_i(A) …

←→ no l-X_j(A)
no l-S_j(A)

A way to summarize Rule #2

Compatibility matrix

Lock already held in	New request: S	New request: X
S	true	false
X	false	false

Rule # 3 2PL transactions

No change except for upgrades:

(I) If upgrade gets more locks
(e.g., S → {S, X}) then no change!

(II) If upgrade releases read (shared)
lock (e.g., S → X)
- can be allowed in growing phase

Theorem

Rules 1,2,3 ⇒ Conf.serializable
for S/X locks schedules

Example

T1	T2
l-S_1(A); r_1(A)	
	l-S_2(A); r_2(A)
	l-S_2(B); r_2(B)
l-X_1(B) (Denied)	
	u_2(A); u_2(B)
l-X_1(B); r_1(B); w_1(B)	
u_2(A); u_2(B)	

Lock types beyond S/X

Examples:

(1) update lock
(2) increment lock

Update locks

A common deadlock problem with upgrades:

T1	T2
$l\text{-}S_1(A)$	
	$l\text{-}S_2(A)$
$l\text{-}X_1(A)$	
	$l\text{-}X_2(A)$

--- Deadlock ---

Solution: Update Locks - If Ti wants to read A and knows it may later want to write A, it requests update lock (not shared)

Comp

New request / Lock already held in:

	S	X	U
S	T	F	T
X	F	F	F
U	TorF?	F	F

Comp

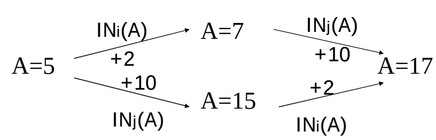
New request / Lock already held in:

	S	X	U
S	T	F	T
X	F	F	F
U	F	F	F

Increment Locks

- Atomic increment action: INi(A)

  {Read(A); A ← A+k; Write(A)}
- INi(A), INj(A) do not conflict!



Comp

New request / Lock already held in:

	S	X	I
S	T	F	
X	F	F	
I			

Comp

New request / Lock already held in:

	S	X	I
S	T	F	F
X	F	F	F
I	F	F	T

Note: object A may be locked in different modes at the same time...

$S1 = ...l\text{-}S_1(A)...l\text{-}S_2(A)... \begin{cases} l\text{-}X_3(A)...? \\ l\text{-}S_3(A)...? \\ l\text{-}U_3(A)...? \end{cases}$

To grant a lock in mode t, mode t must be compatible with all currently held locks on object

Review (Past year exam question)

- Consider a relation databases with the following pages and tuples:
  - Page 1: t1, t2, t3, t4 Page 2: t5, t6, t7
  - Page 3: t8, t9, t10 Page 4: t11, t12, t13, t14
- Suppose the system supports ONLY exclusive lock. We define a "convoy" as a point in time in which one transaction T holds a lock on an object O (O can be a tuple, or page), and at least two other transactions are waiting for the lock on object O. We further define a "deadlock" as a point in time in which there is a sequence of transactions T1, T2, …, Tn, such that for all i such that i < n, Ti waits for T(i+1), and also Tn waits for T1.
- Assume that transactions acquire locks at **page-level**, i.e., to access a tuple, the entire page must be locked. Consider the following three transactions (Note: L = lock, U = unlock, R = Read, W = Write)
  - T1: L(t6), R(t6), L(t9), R(t9), W(t6), U(t6), U(t9)
  - T2: L(t2), R(t2), L(t6), R(t6), W(t2), U(t2), U(t6)
  - T3: L(t8), R(t8), L(t3), R(t3), W(t3), U(t3), U(t8)
- Is there some schedule where a convoy could occur? If so, draw the wait-for graph that shows the convoy. (In a wait-for graph, transactions are nodes and a directed edge from transactions T1 to T2 exists if T1 waits for T2.) If not, explain why not.
- For the same scenario in (I), is there a schedule where a deadlock could occur? If so, draw the wait-for graph that shows the deadlock. Label the edge in the graph with the page that is being waited for. (A wait-for graph shows a deadlock if there is a cycle in the graph.) If not, explain why not.

Review

- Consider a relation databases with the following pages and tuples:
  - Page 1: t1, t2, t3, t4,
  - Page 2: t5, t6, t7
  - Page 3: t8, t9, t10
  - Page 4: t11, t12, t13, t14
- Suppose the system supports ONLY exclusive lock. We define a "convoy" as a point in time in which one transaction T holds a lock on an object O (O can be a tuple, or page), and at least two other transactions are waiting for the lock on object O. We further define a "deadlock" as a point in time in which there is a sequence of transactions T1, T2, …, Tn, such that for all i such that i < n, Ti waits for T(i+1), and also Tn waits for T1.
- Assume that transactions acquire locks at **page-level**, i.e., to access a tuple, the entire page must be locked. Consider the following three transactions (Note: L = lock, U = unlock, R = Read, W = Write)
  - T1: L(t6), R(t6), L(t9), R(t9), W(t6), U(t6), U(t9)
  - T2: L(t2), R(t2), L(t6), R(t6), W(t2), U(t2), U(t6)
  - T3: L(t8), R(t8), L(t3), R(t3), W(t3), U(t3), U(t8)
- Is there some schedule where a convoy could occur? If so, draw the wait-for graph that shows the convoy. (In a wait-for graph, transactions are nodes and a directed edge from transactions T1 to T2 exists if T1 waits for T2.) If not, explain why not.

No convoy, since no more than two transactions accesses the same page

Review

- Consider a relation databases with the following pages and tuples:
  - Page 1: t1, t2, t3, t4, Page 2: t5, t6, t7
  - Page 3: t8, t9, t10 Page 4: t11, t12, t13, t14
- Suppose the system supports ONLY exclusive lock. We define a "convoy" as a point in time in which one transaction T holds a lock on an object O (O can be a tuple, or page), and at least two other transactions are waiting for the lock on object O. We further define a "deadlock" as a point in time in which there is a sequence of transactions ... T1.
- As... Cc... d.

There is a part 2 to this question: What if we are dealing with tuple-level locking?

T3: L(t8), R(t8), L(t3), R(t3), W(t3), U(t3), U(t8)

- For the same scenario in (I), is there a schedule where a deadlock could occur? If so, draw the wait-for graph that shows the deadlock. Label the edge in the graph with the page that is being waited for. (A wait-for graph shows a deadlock if there is a cycle in the graph.) If not, explain why not.

T1 locks Page 2; T2 locks Page 1; T3 locks Page 3; T1 attempts to lock Page 3 (T1 -> T3);
T2 attempts to lock Page 2 (T2 -> T1); T3 attempts to lock Page 1 (T3 -> T1). Deadlock.

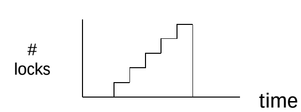
How does locking work in practice?

- Every system is different
  (E.g., may not even provide CONFLICT-SERIALIZABLE schedules)
- But here is one (simplified) way ...

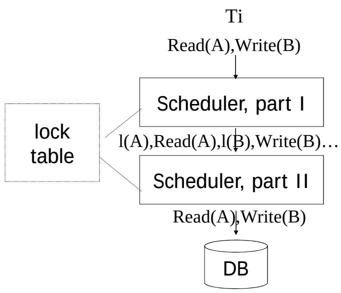
Sample Locking System:

(1) Don't trust transactions to request/release locks

(2) Hold all locks until transaction commits

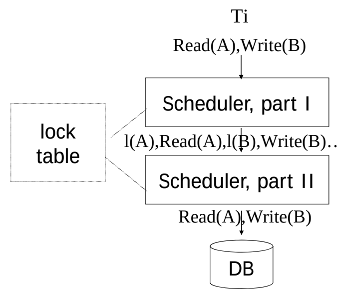


Architecture of a Locking Scheduler



Part I: selects appropriate lock mode, and inserts appropriate lock actions ahead of all database operation

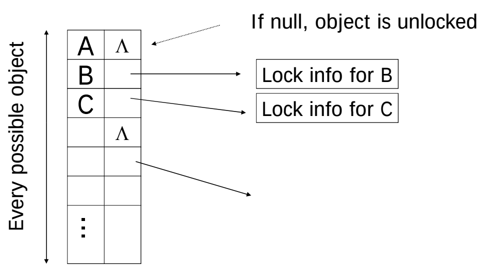
Architecture of a Locking Scheduler



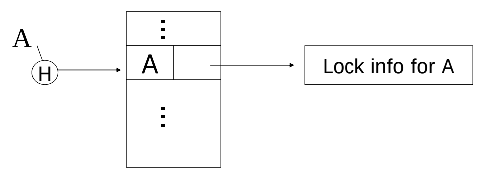
Part II:Executes the operations
a) It determines if lock should be granted; if not, then transaction is delayed.
b) If transaction is not delayed,
- If action is a normal opr, then send it to the dbms
- If action is a lock opr, then check if lock can be granted
  * if so, update lock table
  * if not, delay transaction but update lock table to reflect transaction waiting

c) When a transaction commits/aborts, Part I is notified and releases all locks. Part II will be notified if there are transactions waiting.
d) Part II determines next transactions to be given the released locks. Those that acquired locks can be processed.

Lock table (Conceptually)

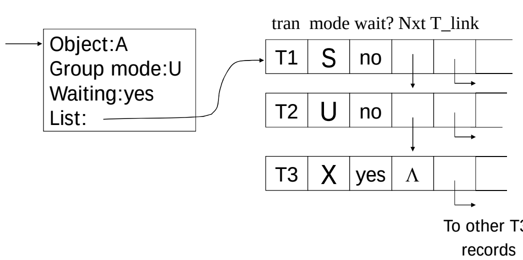


But use hash table:

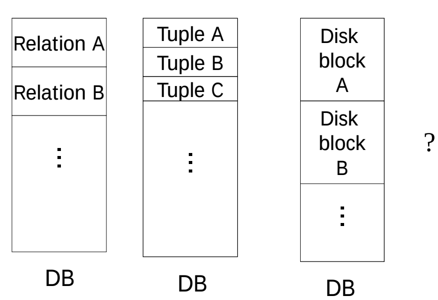


If object not found in hash table, it is unlocked

Lock info for A - example



What are the objects we lock?

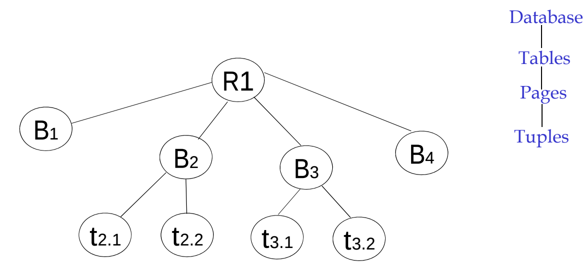


- Locking works in any case, but should we choose <u>small</u> or <u>large objects?</u>

If we lock <u>large</u> objects (e.g., Relations)
    Need few locks
    Low concurrency
If we lock small objects (e.g., tuples,fields)
    Need more locks
    More concurrency

We <u>can</u> have it both ways!!

Managing Hierarchies of Database Elements

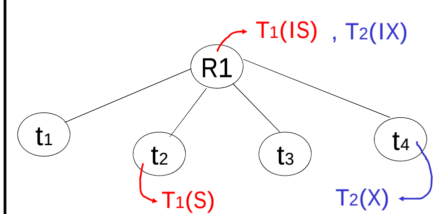


Warning Protocol

Comp

Holder \ Requestor	IS	IX	S	SIX	X
IS	T	T	T	🟩	F
IX	T	T	F	🟩	F
S	T	F	T	🟩	F
SIX	🟩	🟩	🟩	🟩	🟩
X	F	F	F	🟩	F

Multiple Granularity: Warning Protocol



- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

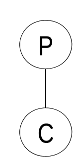
Warning Protocol

Comp

Holder \ Requestor	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Does it make sense to have XIS?

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none



Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q's children are locked by Ti

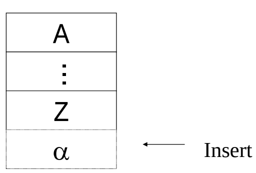
Examples – 2 level hierarchy

Tables
|
Tuples

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
  - Lock escalation dynamically asks for coarser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Insert + delete operations



Modifications to locking rules:

(1) Get exclusive lock on A before deleting A
(2) At insert A operation by Ti,
Ti is given exclusive lock on A

Still have a problem: **Phantoms**

Example: relation R (E#,name,…)
constraint: E# is key
use tuple locking

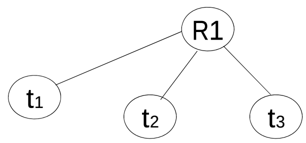
R	E#	Name	….
o1	55	Smith	
o2	75	Jones	

T_1: Insert <99,Gore,…> into R
T_2: Insert <99,Bush,…> into R

T_1	T_2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3[99,Gore,..]	
	Insert o_4[99,Bush,..]

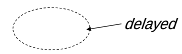
Solution

- Use multiple granularity tree
- Before insert of node Q, lock parent(Q) in X mode

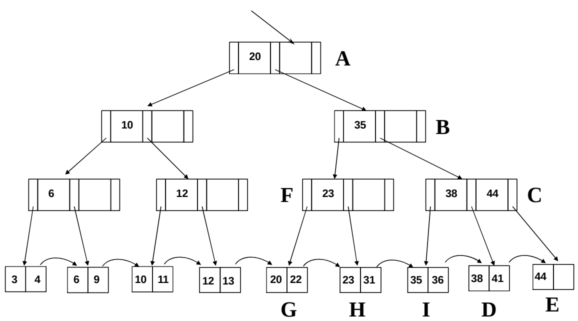


Back to example

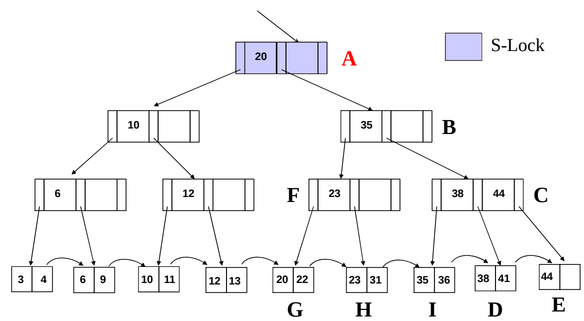
T_1: Insert<99,Gore>	T_2: Insert<99,Bush>
T_1	T_2
$X_1(R)$	
Check constraint Insert<99,Gore> U(R)	
	$X_2(R)$ Check constraint Oops! e# = 99 already in R!



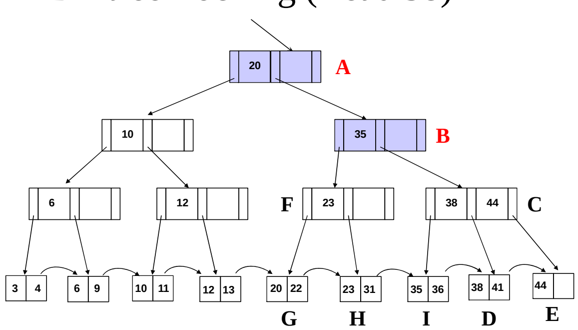
B+-tree (Tree-based) Locking – Crabbing Protocol



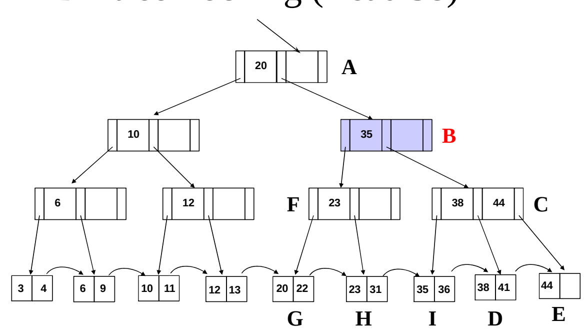
B+-tree Locking (Read 38)



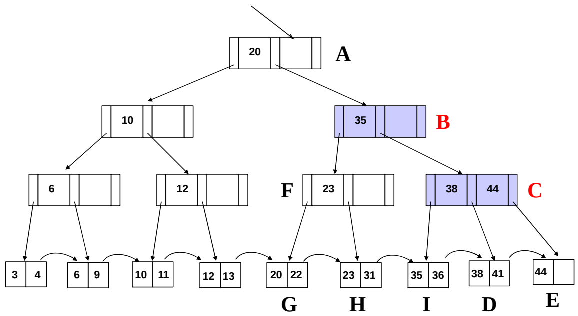
B+-tree Locking (Read 38)



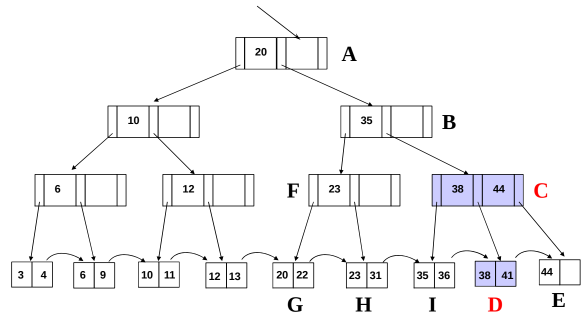
B+-tree Locking (Read 38)



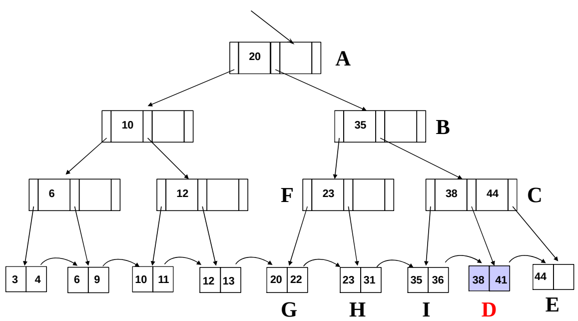
B+-tree Locking (Read 38)



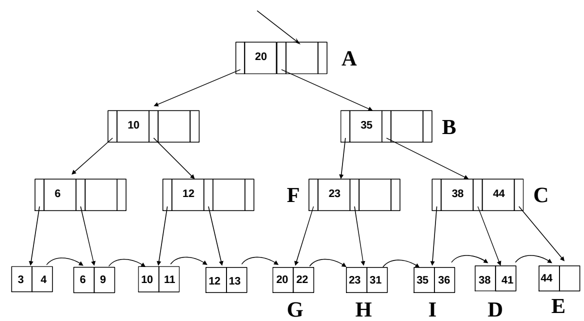
B+-tree Locking (Read 38)



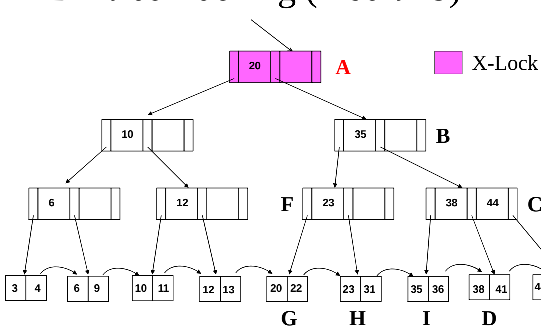
B+-tree Locking (Read 38)



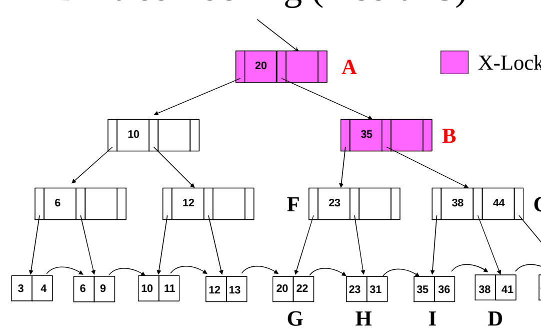
B+-tree Locking (Insert 45)



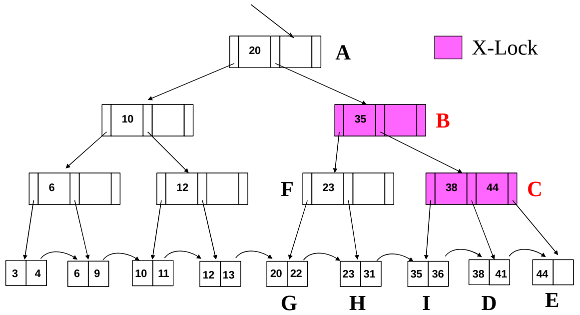
B+-tree Locking (Insert 45)



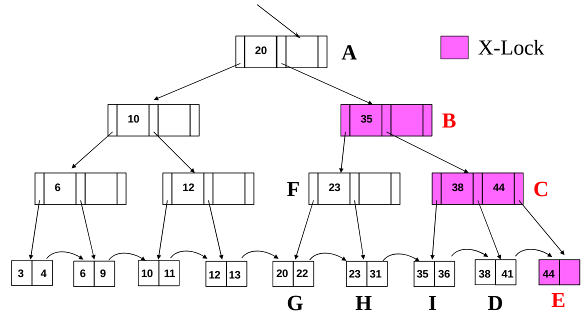
B+-tree Locking (Insert 45)



B+-tree Locking (Insert 45)

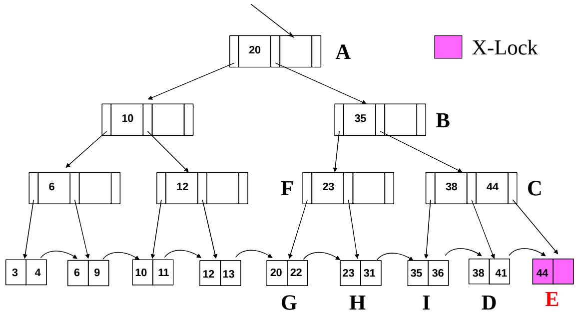


B+-tree Locking (Insert 45)



NOTE: B is not released. Why?

B+-tree Locking (Insert 45)



B+-tree Locking

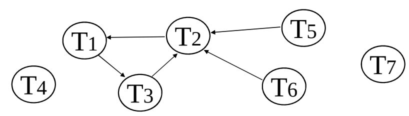
- Can further optimize using S-Lock or Intention Lock for insertion
  - In this case, you may need to upgrade the lock and there is possibility of deadlock arising
- 2PL is *not* used for index locking
- Deletion can be done "efficiently" at the expense of violating the minimum utilization requirement

Deadlocks

- Detection
  - Wait-for graph
- Prevention
  - Wait-die
  - Wound-wait

Deadlock Detection

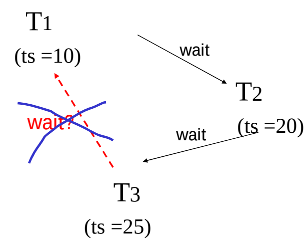
- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim
  - How to determine the victim?



Deadlock Prevention: Wait-die

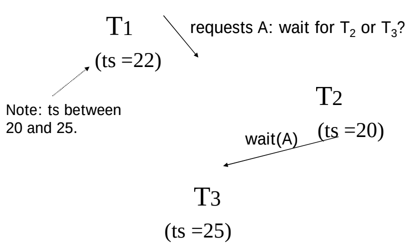
- Transactions given a timestamp when they arrive …. ts(Ti)
- Ti can only wait for Tj if ts(Ti)< ts(Tj)
  ...else die (i.e., abort)

Example:



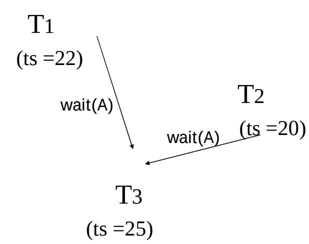
- Important detail: If a transaction re-starts, make sure it gets its original timestamp. Why?

Second Example:



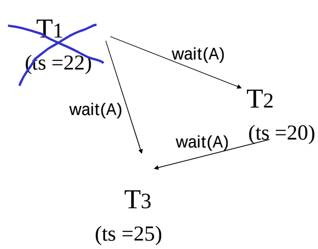
Second Example (continued):

One option: T_1 waits just for T_3, transaction holding lock. But when T_2 gets lock, T_1 will have to die!



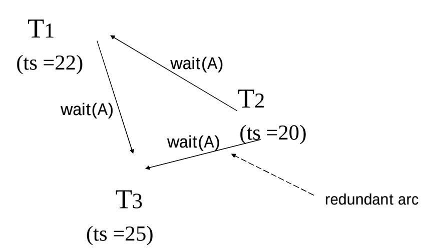
Second Example (continued):

Another option: T_1 only gets A lock after T_2, T_3 complete, so T_1 waits for both T_2, T_3 $\Rightarrow$ T_1 dies right away!



Second Example (continued):

Yet another option: T_1 preempts T_2, so T_1 only waits for T_3; T_2 then waits for T_3 and T_1... $\Rightarrow$ T_2 may starve?

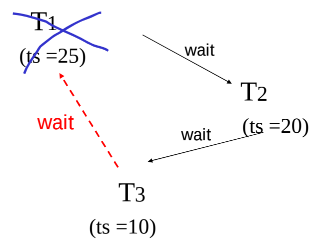


Deadlock Prevention: Wound-wait

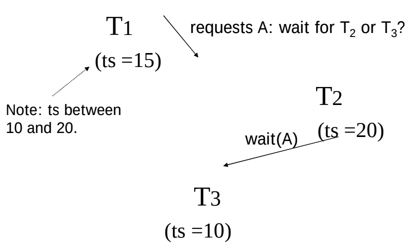
- Transactions given a timestamp when they arrive … ts(T_i)
- T_i wounds T_j if ts(T_i)< ts(T_j)
  else T_i waits

"Wound": T_j rolls back and gives lock to T_i

Example:

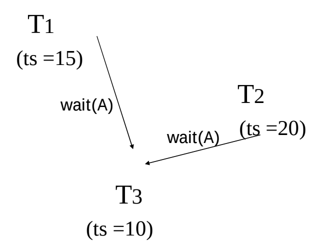


Second Example:



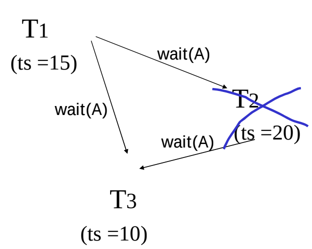
Second Example (continued):

One option: T_1 waits just for T_3, transaction holding lock. But when T_2 gets lock, T_1 waits for T_2 and wounds T_2.



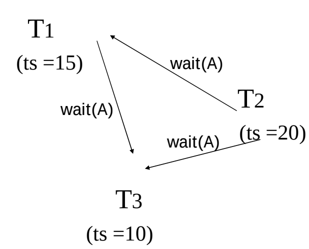
Second Example (continued):

Another option: T_1 only gets A lock after T_2, T_3 complete, so T_1 waits for both T_2, T_3 $\Rightarrow$ T_2 wounded right away!



Second Example (continued):

Yet another option: T_1 preempts T_2, so T_1 only waits for T_3; T_2 then waits for T_3 and T_1… $\Rightarrow$ T_2 is spared!



Summary

- Have studied lock-based CC mechanisms
  - 2 PL
  - Multiple granularity
  - Deadlock
- Did not cover non-locking based CC (timestamp/validation-based) schemes