Advanced (but useful!) Topics in Concurrency Control

Berkeley cs186

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

Multiple Granularity Lock Protocol

Database
Tables
Pages
Tuples

- Each Xact starts from the root of the hierarchy.
- To get S or IS lock on a node, must hold IS or IX on parent node.
  - What if Xact holds S on parent? SIX on parent?
- To get X or IX or SIX on a node, must hold IX or SIX on parent node.
- Must release locks in bottom-up order.

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Lock Compatibility Matrix

Database
Tables
Pages
Tuples

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

- 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.

Lock Compatibility Matrix

Database
Tables
Pages
Tuples

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

- 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.

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

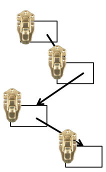
Concurrency Control for Indexes

- **Structural issues**
  - 2PL on B+-tree pages is a rotten idea.
    - Why?
  - Instead, do short locks (latches) in a clever way
  - Idea: Upper levels of B+-tree just need to direct traffic correctly. Don't need to be handled serializably!
- **Phantoms**
  - Even if the B+-tree structure is right, it's incomplete
  - How do you lock what's not there?!
  - This will make more sense shortly :-)

A Note: Latches

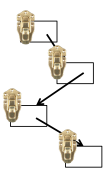
- **There are many data structures inside the DBMS where you want mutual exclusion**
  - E.g. Pin count on buffer page
  - E.g. Wait queue in lock table
  - E.g. B+-tree pages (as we'll see)
- **For these, we acquire a "latch"**
  - **Not** a two-phase lock: we will unlatch almost immediately
  - Usually in-memory next to the data structure in question
    - Fast! Hits in the on-chip cache.
  - S and X latches for reads and writes, respectively
  - Have to latch/unlatch *very carefully.*
    - Only DBMS engineers get access to these APIs.

Simple Ideas for B+-tree Latching



- **Latch Paths**
  - Base case: Get a latch on the root.
  - Induction:
    - assume you have a latch on node N.
    - get a latch on the appropriate child C of N
- **Benefits?**
- **Problems?**

Simple Ideas for B+-tree Latching

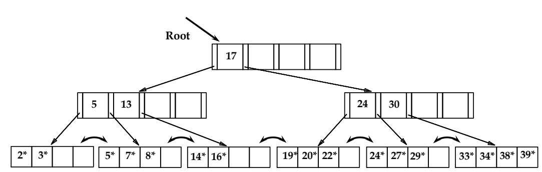


- **Latch Coupling ("crabbing")**
  - Base case: Get a latch on the root.
  - Induction:
    - assume you have a latch on node N.
    - get a latch on the appropriate child C of N
    - release latch on N
- **Benefits?**
- **Problems?**
  - When is it "safe" to release a latch on insert?

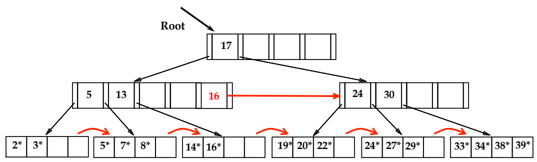
The B-Link Tree

- **A concurrent B+-Tree**
  - Requires *no* latches for readers!
- **Idea: concurrent node split is no problem!**
  - Split nodes are just "twins"
  - No need for latching on read
  - Just detect twin and "go right"
- **Implementation:**
  - Each node stores "high key" -- to help detect if we need to "go right"
  - Each node in the tree has a right-link -- to enable us to "go right"
- **Visibility:**
  - Twins made visible via careful write ordering
  - Twins are "separated" via careful latching
- **Latch coupling is rare**
  - And unlikely to wait on I/O!

A B-Tree from the past



A B-link tree



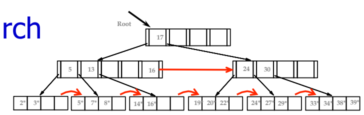
Notation

- **lowercase var: *pointer* to a B-link page**
  - *v, ptr, current*
- **uppercase var: *buffer* containing a B-link page**
  - *A, B*

A subroutine

```
// on-node search, but detect need to move right
page *scannode(value v, buffer A) {
    if (v < A.highkey)
        return child-pointer appropriate for v
    else
        return A.rightlink
}
```

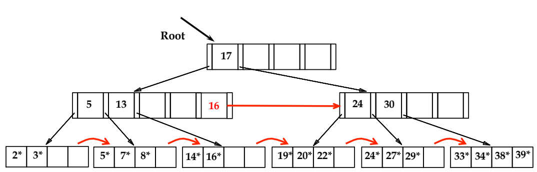
B-link search

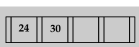


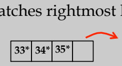
```
bool search(v) {
    current = root;
    A = get(current);
    // traverse down (and right as needed) to leaf level
    while (current is not a leaf) {
        current = scannode(v, A); // may go down or right
        A = get(current);  // atomic!
    }
    // traverse right across leaf level
    while ((t = scannode(v,A))
            == rightlink pointer of A) {
        current = t;
        A = get(current); // atomic!
    }
    // we're at the proper leaf now: just look for v
    if (v is in A)
        return(success);
    else return(failure);
}
```

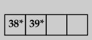
I.e.:
Down and to the right.
No latches.

Example race & move right



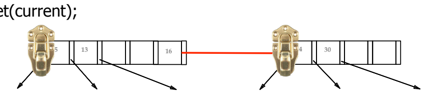
1. T1, searching for 35, reads 

2. T2, inserting 38, latches, splits, and unlatches rightmost leaf
3. T1 arrives at a leaf node that looks like 

4. T1 moves right to find 

A latch-coupling subroutine

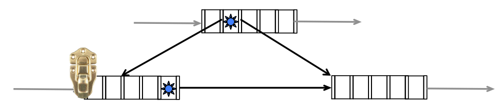
```
page *move_right(value v, page *current)
    // traverse right across leaf level
    latch(current)
    A = get(current)
    while ((t = scannode(v,A))
            == rightlink pointer of A) {
        latch(t)
        unlatch(current)
        current = t;
        A = get(current);
    }
    return A
```



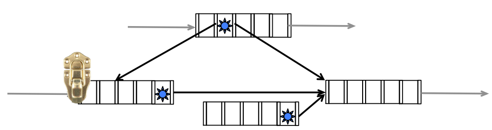
B-link Insert

```
insert(v, ptr) {
    // begin by traversing to leaf without latching
    initialize stack s; // keep a root-to-leaf stack
    current = root;
    A = get(current);
    while (current is not a leaf) {
        t = current;
        current = scannode(v,A);
        if (current not link pointer in A)
                s.push t;
        A = get(current);
    }
    // use latches to isolate the right leaf
    A = move_right(v, current); // latch-couples as needed
    A.doInsertion(v, ptr, s, current) // unlatches after insertion
}
```

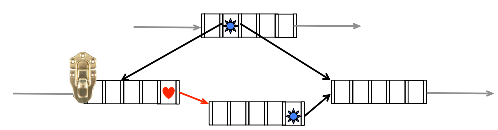
doInsertion: split



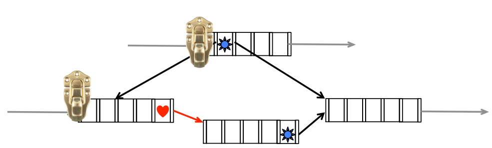
doInsertion: split



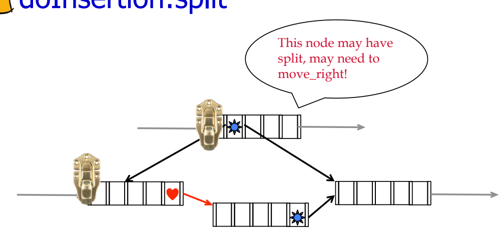
doInsertion:split



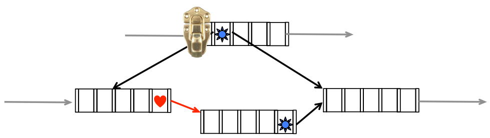
doInsertion:split



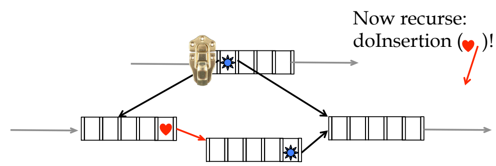
doInsertion:split



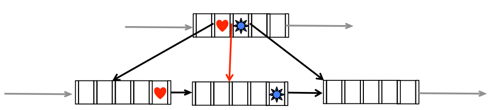
doInsertion:split



doInsertion:split



doInsertion



B-link Insert, cont.

```
doInsertion(v, ptr, s, current) {
  // assumes that current is latched
  A = this
  if A has room {
     insert (v,ptr) on A;
     put(A, current);
     unlatch(current)
  }
  else { // gonna have to split....
     // make new page and redistribute
     u = page.new()
     B = get(u)
     redistribute A over A and B
     y = max value on A
     B.highKey = A.highKey;
     B.rightLink = A.rightLink;
     // create new highKey and rightLink
     A.highkey = y;
     A.rightLink = B;
     // write B then A: appear as "twins"
     put (B, u);
     put (A, current);
     // recurse up-stack to insert <y,u>
     oldnode = current;
     current = s.pop;
     latch(current);
     move_right(v, current);
     unlatch(oldnode);
     A.doinsertion(y, u, s, current);
  }
}
```

Notes

- **How does Deletion work? Easy.**
  - Latch like insertion, delete from leaf node, unlatch
  - No reorganization on underflow
    - simply allow leaves to be under-full
- **How bad is the cost of latch coupling in B-link trees?**
  - Latch-coupling on move_right *can* stall on I/O while latched.
  - Split upwards has at most 3 latches at once
    - 1 for child, 2 for move_right at parent level
    - stack nodes should be in cache
    - again, move-right can stall on I/O while latched
  - Avoiding I/Os during latching:
    - Pre-issue the I/Os speculatively
    - Then latch and re-issue the I/Os
    - Not guaranteed to win: if nodes split, the I/Os could change

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

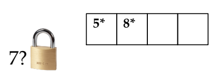
Next Problem: Phantoms

- B-link protects concurrency on *structure*
- Does not address concurrency on the *value domain*

Phantoms Example

- **Suppose you query for sailors with rating between 5 and 8, using a B+-tree**
  - Tuple-level locks in the Heap File
- **I insert a Sailor with rating 7**
- **You do your query again**
  - Yikes! A phantom!
  - Problem: Serializability assumed a static DB!
- **What we want: lock the *logical* range 5-8**
  - Imagine that lock table!
- **What is done: next-key locking in indexes**

Phantom protection: next-key Locking



- On read(x)
  - Use index to find x
  - If x not found, S-lock the *next-higher* item x'
- On insert/update (x)
  - Use index to find x
  - If x not found, X-lock the *next-higher* item x'
- Effectively, each lock on value is a "range lock"
  - $(x_{i-1}, x_i]$
  - conservative, but safe
- No index?
  - Easy answer here for read, update!
  - Treat insert as update.

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

Timestamp Ordered Multi-Version Concurrency Control (TO-MVCC)

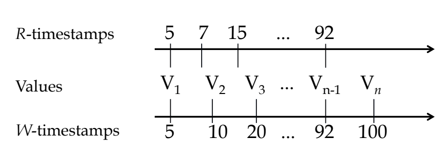
- **Each transaction gets a unique timestemp ts upon entry**
- **For each data item keep:**
  - A set of Read Timestamps, {R-ts}
  - A set of Versions, {<W-ts, value>}

```
Read_ts(x):  // Always succeeds
  Read version of x with largest W-ts < ts
  Add ts to R-ts
```

```
Write_ts(x, v):
  interval = [ts, min(W-ts s.t. W-ts > ts)]
  if exists R-ts in interval
    this transaction must abort
  else
    create new version <ts, v>
```

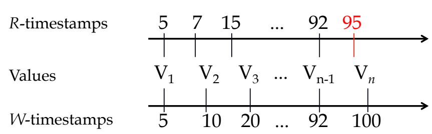
Idea: New writes check to see if they "arrived too late" to affect a previous read.

Value x: "timeline"



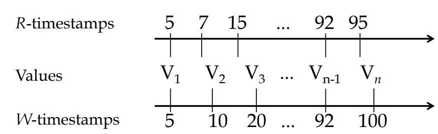
R(X)@95: find biggest W-ts <95, add to R-timestamps

Value x: “timeline”



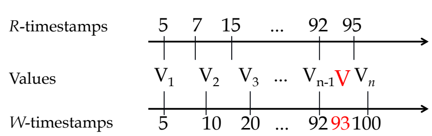
R(X)@95: find biggest W-ts <95, add to R-timestamps

Value x: “timeline”



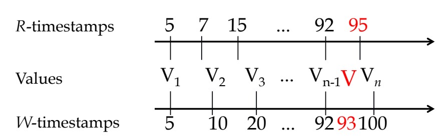
R(X)@95: find biggest W-ts <95, add to R-timestamps
W(X)@93: Interval = [93, 100]

Value x: “timeline”



R(X)@95: find biggest W-ts <95, add to R-timestamps
W(X)@93: Interval = [93, 100]

Value x: “timeline”



R(x)@95: find biggest W-ts <95, add to R-timestamps
W(x)@93: Interval = [93, 100]
Invalidates R(x)@95. Writer must abort.

Pros/cons of MVCC

- Good
  - More concurrency than 2PL
  - Read-only transactions do no special work
    - Optimizes the fast path
  - Writes are append-only: conflict-free
- Mixed
  - No update-in-place
    - Disturbs data locality on disk (e.g. clustering)
    - Maintains past history next to the data
    - Hard to make these kinds of systems efficient
- Bad?
  - Garbage collection problem with versions
  - Restart can be worse than blocking wrt thruput
    - Wasted compute resources

Wind is in MVCC’s sails!

- Flash makes data locality less important
  - No random I/O overhead on reads
- Flash makes CPU overheads more noticeable
  - Locking becomes too slow for read-only workloads
- Cheap storage makes version history worth keeping
  - And even exploiting as a “time travel” or “audit” feature
- Other than ts assignment, each MVCC xact runs *unilaterally*
  - No need to wait for “coordination” with other xacts
  - Very important, esp. in distributed systems
- Some clever use of MVCC schemes recently along these lines
  - E.g. can get coordination-free, restart-free consistency protocols in certain distributed systems contexts
- **Stay tuned, be creative!**

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

Distributed Concurrency Control

- **Consider a parallel or distributed database**
- **One that handles updates**
  - Unlike, say, MapReduce
- **Assume each transaction is assigned to a "coordinator" node**
- **Where is concurrency control state?**
  - locks for 2PL?
  - timestamps/versions for MVCC?
  - Easy! Partition like the data

Partitioned Data, Partitioned CC

- **Every node does its own CC**
- **What could go wrong?**

Partitioned Data, Partitioned CC

- **Every node does its own CC**
- **What could go wrong?**
  - Distributed deadlock: easy enough
    - Gather up all the waits-for graphs, union together.
  - Message failure/delay
  - Node failure/delay
- **How do we decide to commit/abort?**
  - In the face of message/node failure/delay?
  - Hold a vote.
    - How many votes does a commit need to win?
    - How do we implement distributed voting?!
      - In the face of message/node failure/delay?

2-Phase Commit

- **Phase 1:**
  - Coordinator tells participants to "prepare"
  - Participants respond with yes/no votes
    - *Unanimity* required for yes!
- **Phase 2:**
  - Coordinator disseminates result of the vote
- **Need to do some logging for failure handling!**
  - More on this in a couple weeks, time permitting

2-Phase Commit

Like a wedding ceremony!

- **Phase 1:** *"do you take this man/woman..."*
  - Coordinator tells participants to "prepare"
  - Participants respond with yes/no votes
    - Unanimity required for yes!
- **Phase 2:** *"I now pronounce you..."*
  - Coordinator disseminates result of the vote
- **Need to do some logging for failure handling....**
  - More on this in a couple weeks, time permitting

2PC + CC

- **Straightforward for Strict 2PL**
  - If messages are ordered, we have all the locks we'll ever need when we receive commit request.
  - On abort, its safe to abort: no cascade.
- **Somewhat trickier for TO-MVCC**
  - We'll skip the details

Topics for Today

- **Lock Granularity**
  - Maximize concurrency, minimize overhead
- **Index Concurrency:**
  - Structural: B-link trees
  - Logical: Phantoms & next-key locking
- **MultiVersion Concurrency Control (MVCC)**
  - An alternative to 2PL
- **Distributed Concurrency Control**
  - Partitioned state
  - TO-MVCC
  - 2-Phase Commit
- **Weak Isolation**
  - Snapshot Isolation
  - Weaker forms based on locking

Weak Isolation: Motivation

- **Sometimes transactions seem too restrictive**
  - The Chancellor requests the average GPA of all students
  - Various Profs want to make individual updates to grades
  - Can't we all just get along?
- **Sometimes transactions seem too expensive**
  - E.g. 2PC requires computers to wait for each other

Must transactions be "all or nothing"?
Can't we have "loose transactions" or "a little bit of "transactions
Short Answer (tl;dr):
  - Yes, but the API for the programmer is hard to reason about.
  - Still, many people adopt "don't worry be happy" attitude

SQL Isolation Levels (Lock-based)

- **Read Uncommitted**
  - Idea: can read dirty data
  - Implementation: no locks on read
- **Read Committed**
  - Idea: only read committed items
  - Implementation: can unlock immediately after read
- **Cursor Stability**
  - Idea: ensure reads are consistent while app "thinks"
  - Implementation: unlock an object when moving the next
- **Repeatable Read**
  - Idea: if you read an item twice in a transaction, you see the same committed version
  - Implementation: hold read locks until end of transaction
  - No phantom protection
- **Serializable**

Snapshot Isolation (SI)

1. All reads made in a transaction are from the same point in (transactional) time
   - Typically the time when the transaction starts
2. Transaction aborts if its writes conflict with any writes since the snapshot.

When implemented on a MultiVersion system, this can run very efficiently! Oracle pioneered this.

Fact 1: This is not equivalent to serializability.
Fact 2: Oracle calls this "serializable" mode.

SI Problem: Write Skew

- Checking (*C*) and Savings (*S* accounts)
- Constraint: $C_i + S_i >= 0$
- Begin: $C_i = S_i = 100$
  - T1: withdraw \$200 from C_i
  - T2: withdraw \$200 from S_i
- Serial schedules:
  - T1; T2. Outcome:
  - T2; T1. Outcome:
- SI schedule:
  !!

Bad News

- **The lock-based implementations don't exactly match the SQL standards**
  - They do uphold the ANSI standards
  - But the official SQL definitions are (unintentionally) somewhat more general
- **Upshot:**
  - It's *very* hard to reason about the *meaning* of weak isolation
  - Usually people resort to thinking about the *implementation*
  - This provides little help for the app developer!

Worse News!

Database	Default	Maximum
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	RR
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	S
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
RC: read committed, RR: repeatable read, SI: snapshot isolation, S: serializability, CS: cursor stability, CR: consistent read		

Table 2: Default and maximum isolation levels for ACID and NewSQL databases as of January 2013 (from [9]).

Summary

- **There are lots of subtopics in concurrency control!**
  - Different semantics
  - Different protocols
  - Different implementations
  - Different problem settings
- **Lots of renewed innovation**
  - In part due to NoSQL, as we'll discuss shortly