Index Concurrency Control
TODAY'S AGENDA

Phantom Problem
Index Locking
Isolation Levels
Index Crabbing
DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems...
CREATE TABLE people (  
id SERIAL,  
name VARCHAR,  
age INT,  
status VARCHAR  
);

BEGIN  
SELECT MAX(age)  
FROM people  
WHERE status='lit'  
BEGIN  
INSERT INTO people (age=96, status='lit')  
SELECT MAX(age)  
FROM people  
WHERE status='lit'  
COMMIT  
BEGIN  
72  
COMMIT  
96  

THE PHANTOM PROBLEM
WTF?

How did this happen?
→ Because $T_1$ locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.

We will solve this problem in the next class.
PREDICATE LOCKING

Lock records that satisfy a logical predicate:
→ Example: \texttt{status='lit'}

In general, predicate locking has a lot of locking overhead.
Index locking is a special case of predicate locking that is potentially more efficient.
INDEX LOCKING

If there is a dense index on the status field then the txn can lock index page containing the data with status='lit'.

If there are no records with status='lit', the txn must lock the index page where such a data entry would be, if it existed.
LOCKING WITHOUT AN INDEX

If there is no suitable index, then the txn must obtain:

→ A lock on every page in the table to prevent a record’s `status='lit'` from being changed to `lit`.

→ The lock for the table itself to prevent records with `status='lit'` from being added or deleted.
REPEATING SCANS

An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.

→ Have to retain the scan set for every range query in a txn.
→ Andy doesn't know of any commercial system that does this (only just Silo?).
WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.
ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:
→ Dirty Reads
→ Unrepeatable Reads
→ Phantom Reads
ISOLATION LEVELS

**SERIALIZABLE**: No phantoms, all reads repeatable, no dirty reads.

**REPEATABLE READS**: Phantoms may happen.

**READ COMMITTED**: Phantoms and unrepeatable reads may happen.

**READ UNCOMMITTED**: All of them may happen.
<table>
<thead>
<tr>
<th>ISOLATION LEVELS</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
</tbody>
</table>
ISOLATION LEVELS

**SERIALIZABLE**: Obtain all locks first; plus index locks, plus strict 2PL.

**REPEATABLE READS**: Same as above, but no index locks.

**READ COMMITTED**: Same as above, but \( S \) locks are released immediately.

**READ UNCOMMITTED**: Same as above, but allows dirty reads (no \( S \) locks).
SQL-92 ISOLATION LEVELS

Not all DBMS support all isolation levels in all execution scenarios (e.g., replication).

The default depends on implementation...

```
SET TRANSACTION ISOLATION LEVEL <isolation-level>;
```
## ISOLATION LEVELS (2013)

<table>
<thead>
<tr>
<th>Database</th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actian Ingres 10.0/10S</td>
<td><strong>SERIALIZABLE</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>Aerospike</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>Greenplum 4.1</td>
<td>READ COMMITTED</td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>MySQL 5.6</td>
<td>REPEATABLE READS</td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>MemSQL 1b</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>MS SQL Server 2012</td>
<td>READ COMMITTED</td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>Oracle 11g</td>
<td>READ COMMITTED</td>
<td><strong>SNAPSHOT ISOLATION</strong></td>
</tr>
<tr>
<td>Postgres 9.2.2</td>
<td>READ COMMITTED</td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>SAP HANA</td>
<td>READ COMMITTED</td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
<tr>
<td>ScaleDB 1.02</td>
<td>READ COMMITTED</td>
<td>READ COMMITTED</td>
</tr>
<tr>
<td>VoltDB</td>
<td><strong>SERIALIZABLE</strong></td>
<td><strong>SERIALIZABLE</strong></td>
</tr>
</tbody>
</table>

Source: Peter Bailis
SQL-92 ACCESS MODES

You can also provide hints to the DBMS about whether a txn will modify the database.

Only two possible modes:

→ **READ WRITE** (Default)
→ **READ ONLY**

Not all DBMSs will optimize execution if you set a txn to in **READ ONLY** mode.
LOCKING IN B+TREES

What about locking indexes?

They are not quite like other database elements so we can treat them differently:

→ It’s okay to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained.
EXAMPLE

$T_1$ wants to insert entry in node $H$

$T_2$ wants to insert entry in node $I$

Why not use 2PL?
**EXAMPLE**

\(T_1\) wants to insert entry in node \(H\)

\(T_2\) wants to insert entry in node \(I\)

**Why not use 2PL?**

Because txns have to hold on to their locks for too long!
LOCK CRABBING

Protocol to allow multiple threads to access/modify B+Tree at the same time.

Basic Idea:
→ Get lock for parent.
→ Get lock for child
→ Release lock for parent if “safe”.

A **safe node** is one that will not split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)
**LOCK CRABBING**

**Search:** Start at root and go down; repeatedly,
→ Acquire $S$ lock on child
→ Then unlock parent

**Insert/Delete:** Start at root and go down, obtaining $X$ locks as needed.
Once child is locked, check if it is safe:
→ If child is safe, release all locks on ancestors.
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38

It’s safe to release the lock on A.
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #1 – SEARCH 38
EXAMPLE #2 – DELETE 38

The diagram illustrates a deletion operation in a binary search tree. The process involves finding the node with the value 38 and removing it from the tree. The tree is traversed to locate the node, and then the node is replaced with its right child, ensuring the tree remains balanced.

Key steps in the deletion process:
1. Locate the node with the value 38.
2. Replace the node with its right child (either 38 or 44).
3. Adjust the tree structure accordingly.

The diagram visually shows the before and after states of the tree, highlighting the deletion operation and its effects on the tree's structure.
We may need to coalesce B, so we can’t release the lock on A.
We know that C will not need to merge with F, so it’s safe to release A+B.
We know that C will not need to merge with F, so it’s safe to release A+B.
EXAMPLE #2 – DELETE 38

A

B

C

D

E

G

H

I

3 4 6 9 10 11 12 13 20 22 23 31 35 36
EXAMPLE #2 – DELETE 38
EXAMPLE #3 – INSERT 45

We know that if C needs to split, B has room so it’s safe to release A.
EXAMPLE #3 – INSERT 45
E has room so it won’t split, so we can release B+C.
**EXAMPLE #3 – INSERT 45**

E has room so it won’t split, so we can release B+C.
EXAMPLE #4 – INSERT 25

A

B

C

D

E

F

G

H

I


X


X


B20

B35


10


6

12

23

20


3

4

6

9

10

11

12

13

22

23

31

35

36

38

41

44


38


44


6

12

23

38

44


10

12

20
EXAMPLE #4 – INSERT 25

Diagram of a 2-3 tree with nodes A, B, C, D, E, F, G, H, I, and X. The sequence of numbers to be inserted are 25, 6, 12, 23, 35, 38, and 44. The tree structure shows the process of inserting these values.
EXAMPLE #4 – INSERT 25

A

10

20

X

35

B

X

23

C

38 44

F

G

3 4 6 9 10 11 12 13

H

20 22 23 31

I

35 36 38 41

J

44
EXAMPLE #4 – INSERT 25

Diagram of a binary search tree with numbers inserted and labeled nodes. The tree starts with a root node labeled 20, and branches down to nodes labeled 10, 6, 3, 4, 12, 10, 11, 20, 23, 35, 38, 41, 38, 44, and 44.
EXAMPLE #4 – INSERT 25
We need to split H so we need to keep the lock on its parent node.
EXAMPLE #4 – INSERT 25

We need to split H so we need to keep the lock on its parent node.
EXAMPLE #4 – INSERT 25

We need to split H so we need to keep the lock on its parent node.
OBSERVATION

What was the first step that all of the update examples did on the B+Tree?

Delete 38

Insert 45

Insert 25
OBSERVATION

What was the first step that all of the update examples did on the B+Tree? Locking the root every time becomes a bottleneck with higher concurrency.

Can we do better?
BETTER TREE LOCKING ALGORITHM

Assume that the leaf is safe, and use S-locks & crabbing to reach it, and verify. If leaf is not safe, then do previous algorithm.


Concurrence of Operations on B-Trees

R. Bayer* and M. Schkolnick
IBM Research Laboratory, San Jose, CA 95134, USA

Summary: Concurrent operations on B-trees pose the problem of ensuring that each operation is carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these operations are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrence. Another feature required from these protocols is that they be deadlock free. Since the cost to resolve a deadlock may be high.

Recently, there has been some questioning as to whether B-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to B-trees. We present a deadlock-free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as B-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Keogh [9] and Wilhelms [11]. Performance studies of it were performed in the single-user environment. Recently, these structures have been examined for possible use in a multi-user (concurreny) environment. Some initial studies have been made about the feasibility of their use in this type of situation [4, 9] and [11].

An access schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the...
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38
EXAMPLE #2 – DELETE 38

Tree diagram showing the process of deleting the value 38 from a binary search tree.
EXAMPLE #2 – DELETE 38

D will not need to coalesce, so we’re safe!
D will not need to coalesce, so we’re safe!
EXAMPLE #4 – INSERT 25

Diagram showing a binary search tree with nodes containing integers from 3 to 44. The root node is 20, with children 10 and 35. Node 25 is added to the tree, inserting it into the correct position to maintain the tree's order.

- Node 20 (A) has children 10 (B) and 35 (C).
- Node 10 (B) has children 6 and 12.
- Node 12 (B) has children 3, 4, 6, 9, 10, 11, 12, 13.
- Node 35 (C) has children 23, 31, 35, 36, 38, 41, 44.
- Node 25 is inserted into the correct position to maintain the tree's order.
EXAMPLE #4 – INSERT 25

We need to split H so we have to restart and re-execute like before.
BETTER TREE LOCKING ALGORITHM

**Search:** Same as before.

**Insert/Delete:**
- Set locks as if for search, get to leaf, and set *X* lock on leaf.
- If leaf is not safe, release all locks, and restart txn using previous Insert/Delete protocol.

Gambles that only leaf node will be modified; if not, *S* locks set on the first pass to leaf are wasteful.
ADDITIONAL POINTS

Which order to release locks in multiple-granularity locking?
From the bottom up

Which order to release latches in B+Tree latching?
As early as possible to maximize concurrency.
CONCLUSION

Indexes make concurrency control hard because it's a essentially a second copy of the data.

Most applications do not execute with SERIALIZABLE isolation.
NEXT CLASS

More Concurrency Control!!!
Timestamp Ordering!!!