CMSC724: Concurrency

Amol Deshpande

April 1, 2008

1 Overview

Transactions and ACID

- Goal: Balancing *performance* and *correctness*
- Performance: *high concurrency* and *flexible buffer management*
  - STEAL: no waiting for transactions to finish to flush
  - NO FORCE: no disk writes for transactions to commit
- Correctness:
  - Database remains consistent (constraints are satisfied)
    * Many constraints may not be in the database at all
  - Each transaction sees a consistent state
  - System can crash at any time
    * Database, all *external actions* (e.g. printing receipt) remain consistent
  - All while minimizing the programmer burden

Transactions and ACID

- Transaction: A sequence of actions bracketed by begin and end statements.
  - Transactions assumed to be “correct” (programmer burden)
  - Commit or abort: final actions
- ACID:
  - Atomicity: The entire transaction or nothing.
  - Consistency: Each transaction take DB from one consistent state to another.
  - Isolation: Events within a transaction invisible to other transactions.
  - Durability: Once committed, the results of a transaction preserved in spite of failures.
Transactions and ACID

- C & I: Concurrency control
  - Generally locking-based
    - Typically separate protocols for indexes and other high contention objects
  - Optimistic - Let transactions run, check at the end
  - Main principles: serializability and recoverability
    - Independent of the protocol used
  - Terms: schedule, history, serial schedule
  - Serializability: what happened corresponds to a serial execution
    - Conflict graph is acyclic
  - Recoverability: A transaction abort shouldn’t affect other transactions
    - Definitely shouldn’t require aborting committed transactions (not allowed)

Transactions and ACID

- A & D: Recovery mechanisms
  - Generally logging-based
  - Write a log of everything the system did
    - Written sequentially to a separate disk, so fast
  - Main principle: WAL = Write-Ahead Logging
    - Log records go to disk before any permanent actions (like writing to disk, commit)
  - No real alternatives
    - Original Postgres used non-volatile storage in creative ways, but abandoned later

2 Mechanisms: Locks

Locking: Concepts

- Locking: To ensure transactions don’t interfere
  - Share lock (“Read locks”)
  - Exclusive lock (“Write locks”)
- A “well-formed” transaction takes locks on the items it reads or updates.
- Two-phase locking:
  - Growing phase: Take locks, but don’t release
  - Shrinking phase: Release locks, but don’t take
- Well-formed and two-phase guarantees serializability
- Recoverability requires long (till EOT) locks
Granularity of Locks

- Trade-off: Fine granularity → high concurrency, but high overhead (vs coarse granularity)
- Granularities possible: DB, Areas, Files, Pages, Tuples (records), Fields of tuples
- Arrange in a hierarchy
  - Can be a DAG (could also be dynamically changing)
- New types of locks: Intention Shared (IS), Intention Exclusive (IX), Share and Intention Exclusive (SIX)
- Protocol:
  - Get at least intention locks on all ancestors
  - For DAGs: To get an X lock, must lock all paths from root to the node; for S locks, locking one path is enough.

Locks

- Lock tables: Typically a in-memory hash table
- Lock/unlock themselves must be atomic instructions
  - E.g. using semaphores or mutexes
  - Heavyweight: Several hundred instructions per call
- Issues
  - Lock conversion
  - Starvation
  - Deadlocks
    - Detection mechanisms (cycles in the waits-for graphs)
    - Avoidance mechanisms (pre-emptive mechanisms)
    - Time-out based (perhaps more common)

Locks vs Latches

- Latches are short physical locks
  - Like mutexes
- Taken to protect against the DBMS code itself
  - Not transactions
- For instance, to make sure two transactions writing to the same “page” don’t interfere
  - Allowed if we are doing record-level locking
  - Required because updating a page may involve rearrangement inside
- Deadlock with latches: system bug.. shouldn’t have happened
- Deadlock with locks: not a problem
3 Degrees of Consistency

Motivation

• True serializability too restrictive
  – Need more concurrency
  – Some inconsistencies acceptable

• Approach
  – Define “degrees of consistencies”
  – Each transaction/query chooses what it wants
  – The system ensures whatever the transactions chose
    * Can’t be completely arbitrary: e.g. each transaction must hold at least short “write locks”
  – Preferably should not depend on concurrency protocol
    * Most practical ones strongly based on locking (see Generalized Isolation Level Definitions; Adya et al.)

Anomalies

• Easiest to understand using what may go wrong
  – ANSI SQL Isolation levels based on this

• We will follow the terminology in A critique of ANSI SQL Isolation Levels; Berenson et al.

• 0: Lost updates (Dirty Writes)
  – (1) T1 writes X; (2) T2 writes X; (3) T1 aborts and restores back to old value; (4) T2 update has been lost

• 1: Dirty read: Reading the uncommitted data of another transaction
  – (1) T1 writes X; (2) T2 reads X; (3) T1 aborts or commits
  – May be serializable, but not recoverable (if T1 aborts, T2 needs to be aborted)
  – Not always serializable (see later)

• 2: Non-Repeatable read: Transaction repeats a read, reads a different value
  – (1) T1 reads X, (2) T2 writes X, (3) T1 reads X again

• 3: Phantom read:
  – (1) T1 reads all items in range [X, Y]
  – (2) T2 inserts Z in that range
  – (3) T1 re-reads all items in range [X, Y], and finds a new value (“phantom”)

Generalized Isolation Level Definitions; Adya et al.
Anomalies

- English definitions somewhat ambiguous
- E.g. in *Phantom Read*, do we need to have the third step or not?
  - Do we prohibit just the 3rd step or the 2nd step?

- Terminology:
  - w1[X]: Transaction 1 wrote data item X
  - C1, A1: Transaction 1 committed or aborted

- Phenomenon vs Anomalies: Dirty Reads
  - Anomaly: Anomaly1: w1[X] . . . r2[X] . . . (C2 and A1 in some order)
    - Something actually went wrong
  - Phenomenon: P1: w1[X] . . . r2[X] . . . (C1 or A1)
    - Corresponds to possible serial execution (T1, then T2)?

- Berenson et al.: We need “loose” definitions
  - Second defn is looser – it applies to more histories
  - But since we’re using these to restrict, the second defn permits fewer histories

- Example: Implicit constraint: x + y = 100
  - Consider: r1[x=50] w1[x=10] r2[x=10] r2[y=50] c2 r1[y=50] w1[y=90] c1
    - Both transactions obey the constraint
      - But T2 read x=10 and y=50, which violates the constraint
      - Anomaly1 not violated, P1 violated

Anomalies

- Phenomenon: As defined by Berenson et al, 1995
  - P0: garbage writes - w1[X] . . . w2[X] . . . (C1 or A1)
  - P1: dirty reads - w1[X] . . . r2[X] . . . (C1 or A1)
  - P2: non-repeatable read - r1[X] . . . w2[X] . . . (C1 or A1)
  - P4: phantom read - r1[pred] . . w2[Y in pred] . . (C1 or A1)

ANSI SQL Isolation Levels

- NOTE: This is from 1995... may have been changed
- Fuzzy read == non-repeatable read
- Assume loose interpretations of the phenomenon
- Anomaly Serializable ??
  - Turns out disallowing the three phenomenon doesn’t guarantee true serializability
Table 1. ANSI SQL Isolation Levels Defined in terms of the Three Original Phenomena

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>P1 (or A1) Dirty Read</th>
<th>P2 (or A2) Fuzzy Read</th>
<th>P3 (or A3) Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI READ UNCOMMITTED</td>
<td>Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>ANSI READ COMMITTED</td>
<td>Not Possible</td>
<td>Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>ANSI REPEATABLE READ</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Possible</td>
</tr>
<tr>
<td>ANOMALY SERIALIZABLE</td>
<td>Not Possible</td>
<td>Not Possible</td>
<td>Not Possible</td>
</tr>
</tbody>
</table>

Summary so far..

- Anomalies: define formally (Berenson et al. definitions)
- Use them to exclude possible executions
- ANSI SQL Isolation Levels use P1, P2, P3
  - Do not exclude P0.. critical mistake
  - Must be modified to include P0 for all of them
  - That ensures true serializability
- Still locking-based in disguise (Adya et al.)
  - Disallows: w1[X]. r2[X]. C1.. (A2 or C2) (violates P0)
  - Okay if T2 is serialized after T1.
- At some point, you have to ask if the additional concurrency is worth the extra effort
- Next .. how to achieve this

Locking Implementation

- Well-formed reads (writes): Take read (write) lock before the operation
- Short (just for the operation) vs long (till EOT)
- Predicate locks: e.g. lock all tuples with \( R.a \in [10, 20] \)

Locking Implementation

- Each transaction can choose its own degree, as long as every transaction is at least Degree 0
  - If some transaction doesn’t do well-formed write (no locks at all), no other transaction is safe
  - Recoverability requires systemwide degree 1
    * Consider: T1: Degree 0 (short write locks), T2: Degree 3 (long read locks)
    * w1[X]..r2[X]..T1-Abort
    * Allowed even though T2 reads dirty data.
- Cursor Stability ??
  - Commonly used
Table 2. Degrees of Consistency and Locking Isolation Levels defined in terms of locks.

<table>
<thead>
<tr>
<th>Consistency Level = Locking Isolation Level</th>
<th>Read Locks on Data Items and Predicates (the same unless noted)</th>
<th>Write Locks on Data Items and Predicates (always the same)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree 0</td>
<td>none required</td>
<td>Well-formed Writes</td>
</tr>
<tr>
<td>Degree 1 = Locking READ UNCOMMITTED</td>
<td>none required</td>
<td>Well-formed Writes, Long duration Write locks</td>
</tr>
<tr>
<td>Degree 2 = Locking READ COMMITTED</td>
<td>Well-formed Reads, Short duration Read locks (both)</td>
<td>Well-formed Writes, Long duration Write locks</td>
</tr>
<tr>
<td>Cursor Stability (see Section 4.1)</td>
<td>Well-formed Reads, Read locks held on current of cursor, Short duration Read Predicate locks</td>
<td>Well-formed Writes, Long duration Write locks</td>
</tr>
<tr>
<td>Locking REPEATABLE READ</td>
<td>Well-formed Reads, Long duration data-item Read locks, Short duration Read Predicate locks</td>
<td>Well-formed Writes, Long duration Write locks</td>
</tr>
<tr>
<td>Degree 3 = Locking SERIALIZABLE</td>
<td>Well-formed Reads, Long duration Read locks (both)</td>
<td>Well-formed Writes, Long duration Write locks</td>
</tr>
</tbody>
</table>

- Most app programs use a “cursor” to iterate over tuples
- They don’t go back once the cursor moves
- Cursor stability makes sure that no other transaction modifies the data item while the cursor doesn’t move

Other Isolation Levels

- Snapshot isolation (called SERIALIZABLE in ORACLE)
  - Each transaction gets a timestamp
  - “First committer wins”: At commit time, if T1 and T2 have a WW conflict, and T1 commits, T2 is aborted.
  - Implementation: Archive old data and if a transaction asks for a data item, get the data from its start time.
  - A “multi-version” concurrency scheme.

Other Isolation Levels

- Snapshot isolation (SERIALIZABLE in ORACLE)
  - Not truly serializable
  - T1: r(a0), r(b0), w(a1 = a0 - 100), C1
  - T2: r(a0), r(b0), w(b2 = b0 - 100), C2
  - A is checking, B is savings (Constraint: A + B > $0)
  - Initially: A = 70, B = 70
  - Each transaction checks for it, but constraint still not satisfied at the end
  - Need reasoning about “multi-object constraints”
  - Still commonly used (more concurrency).
Other Isolation Levels

- Snapshot isolation (SERIALIZABLE in ORACLE)
  - Also used by SQL Anywhere, Interbase, Firebird, PostgreSQL, Microsoft SQL Server
  - From Wikipedia Entry
    - Quote: 
      "small "SI has also been used to critique the ANSI SQL-92 ... isolation levels, as it exhibits none of the "anomalies" that are prohibited, yet is not serializable (the anomaly-free isolation level defined by ANSI)"
    - Also has a detailed example of the anomaly
  - Although it arose from multi-version schemes, can co-exist with locking

4 Granularity of Locks...; Gray et al.; 1976

Degrees of Consistency

- Definition using locking
  - Degree 0: set short write locks on updated items
  - Degree 1: set long write locks on updated items
  - Degree 2: Degree 1 and set short read locks on read items
  - Degree 3: Degree 1 and set long read locks on read items

- What is "long"?
  - End of the transaction to be safe
  - For serializability, sufficient to hold till the shrinking phase, no recoverability

- Corresponds to the above table

Degrees of Consistency

- Definition using the dirty reads...

  Transaction T “sees” (obeys) degree X consistency if:
  - Degree 0: T does not overwrite dirty data of other transactions
  - Degree 1: Degree 0, and T does not commit any writes before EOT
  - Degree 2: Degree 1, and T does not read dirty data of other transactions
  - Degree 3: Degree 2, and other transactions do not dirty any data read by T before T completes

- Much criticism afterwards – too vague and ambiguous
  - Intended to correspond to the defns using locking
  - Locking is stronger
  - Not exact match: EOT vs shrinking phase
  - No protection against P3: “phantom reads”
5 Optimistic Concurrency Control

Optimistic Concurrency Control

- Simple idea: optimize case where conflict is rare.
  - Think cvs, svn etc.
- Basic idea: all transactions consist of three phases:
  - **Read**: All writes are to private storage (shadow copies).
  - **Validation**: Make sure no conflicts have occurred.
  - **Write**: If Validation successful, make writes public. (If not, abort!)

- Better in large-scale, wide-area distributed systems.
- When useful?
  - All transactions are readers.
  - Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data.
  - Fraction of transaction execution in which conflicts “really take place” is small compared to total pathlength.

The Validation Phase

- Goal: guarantee that only serializable schedules result.
- Technique: find an equivalent serializable schedule.
  - Assign each transaction a TN during execution.
  - Ensure that if you run transactions in order induced by “<” on TNs, you get an equivalent serial schedule.
- Suppose $TN(T_i) < TN(T_j)$. IF:
  - $T_i$ completes its write phase before $T_j$ starts its read phase.
  - $WS(T_i) \cap RS(T_j) = \phi$ and $T_i$ completes its write phase before $T_j$ starts its write phase.
  - $WS(T_i) \cap RS(T_j) = \phi$ and $WS(T_i) \cap WS(T_j) = \phi$ and $T_i$ completes its read phase before $T_j$ completes its read phase.
- THEN serializable (allow $T_j$ writes).

Optimistic Concurrency Control: Details

- Maintain create, read, write sets
- Critical sections:
  - **tbegin**: record the current TN (called *start-tn*)
  - **tend**:

9
∗ Validate against all transactions between start-tn and current transaction number.
∗ If validated, assign the next TN to this transaction
∗ If not validated, abort
– Only one transaction can be in a critical section at any time
– Read queries don’t need to be assigned TNs

**Optimistic Concurrency Control**

- Need to store the read and write sets
  - Could become very large with long transactions
  - Large transactions also result in starvation
  - After enough aborts, lock the whole database (hopefully doesn’t happen often)
- Write Phase:
  - May be too large to be done atomically inside the critical section
  - Also in case of parallel processors, critical sections may become bottlenecks
  - Split the write phase (move large parts outside critical section)
  - Allow interleaving of writes (with additional checks)

**6 Locking vs Optimistic**

**Locking vs Optimistic**

- Many studies comparing the two
- Agrawal/Carey/Livny, ACM TODS 1987 (in redbook)
  - Very detailed performance study comparing blocking, optimistic and immediate restart (when locked, restart)
  - Models different kinds of scenarios (how many resources, processors etc)…
  - Blocking usually wins over the other two
  - Optimistic better with low resource utilization, large think times etc…
- Most systems use locking
- Other issues:
  - Escrow transactions
  - Concurrency in indexes
7 Locking in B-Trees

Concurrency in B+-Trees

- Note: B*-Trees == B+-Trees
  - Record pointers only stored at the leaves
- Recall
  - Search: start at root, traverse down to the leaf
  - Insert:
    * 1. Start at root, traverse down to the leaf.
    * 2. If leaf has space, insert and we are done.
    * 3. If leaf doesn’t have space, split and create a new leaf. Create a new entry for insertion in the parent node.
    * 4. If the parent node has space, insert and we are done. Else continue above.
  - Delete:
    * Same as above, except we merge nodes instead of splitting

Concurrency in B+-Trees

- Discussion from: B+-Tree Concurrency Algorithms; Srinivasan, Carey; VLDB J. 1993
- Safe nodes: has empty space (during inserts), has enough tuples (during deletes)
- Key Observation:
  - During an update (insert/delete), if an interior node is safe, the ancestors of that node will not be needed further
  - The upward propagation will definitely halt at that interior node
- Lock-coupling: get a lock on an index node while holding a lock on the parent, and only then release lock on parent
- Samadi, 1976: Protocol based on safe nodes
  - For updates, (X) lock-couple down the tree and release locks on ancestors when a safe node is encountered
  - For reads, lock-couple to the leaves using IS locks
- Too many X locks
- Bayer, Schkolnick, 1977: improvement on above
  - Searches/reads: same as above
  - Updates:
    * 1. Lock-couple to the leaves using IX locks releasing parents locks immediately.
    * 2. Take X lock on appropriate leaf
    * 3. If leaf safe, insert/delete and be done. Else revert to Samadi, 1976 (start again from top)
• Top-down Algorithms (e.g. Guibas and Sedgewick, 1978)
  – Do preparatory splits/merges on the way down
  – During insert, if an interior node is full, split it
    ∗ A leaf’s parent will always be safe

• B-link Tree Algorithms (Lehman and Yao, 1981)
  – Only one node is locked at any time
  – Uses right-link pointers to “chase” data items
  – Updates are done bottom-up
  – Can cause temporary inconsistencies, but those can be reasoned about

Concurrency in B+-Trees

• Srinivasan, Carey; VLDB J. 1993
• Extensive simulations to compare many of these techniques
  – Similar in spirit to the locking comparison by Agrawal, Carey, Livny
• Results:
  – Lock-coupling with X locks bad for concurrency
  – B-link trees superior in almost all cases
Figure 3. A B-link tree page split

(a) Example B-link tree node

(b) Before half-split

(c) After half-split

(d) After key propagation

Figure 1: B-Link Trees; From: Srinivasan, Carey; VLDB J. 1993