CMSC724: Transactions and ACID properties

Amol Deshpande

University of Maryland, College Park

April 13, 2011
Outline

1 Overview
   - Mechanisms: Locks
2 Optimistic Concurrency Control
   - Locking vs Optimistic
3 Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?
4 Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous
5 Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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 takes the 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: Programmer responsibility
- I: Concurrency control
  - Generally locking-based, but less so with distributed now-a-days
    - Separate protocols for indexes, 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
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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.
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)
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
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Optimistic Concurrency Control

Disadvantages of locking
- Overhead: even read transactions must
- Things like B-Trees require special locking protocols
- Locks may be held for long times
- Locking may only be necessary in worst case

Better to use optimistic protocol if:
- 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.

Especially useful in large-scale, wide-area distributed systems.
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!)
The Validation Phase

- Goal: guarantee that only serializable schedules result.
- Technique: find an equivalent serializable schedule.
  - Assign each transaction a TN during execution.
    - When? Probably best to do it at the start of validation phase
  - Ensure that if you run transactions in order induced by “<” on TNs, you get an equivalent serial schedule.
The Validation Phase

- **Goal:** guarantee that only serializable schedules result.
- **Technique:** find an equivalent serializable schedule.
  - Assign each transaction a TN during execution.
    - **When?** Probably best to do it at the start of validation phase.
  - 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.
- $WS(T_i) \cap RS(T_j) = \emptyset$ and $T_i$ completes its write phase before $T_j$ starts its write phase.
- $WS(T_i) \cap RS(T_j) = \emptyset$ and $WS(T_i) \cap WS(T_j) = \emptyset$ 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 \textit{start-tn})
  - **tend**:
    - Validate against all transactions between \textit{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
thebegin = (
    create set := empty;
    read set := empty;
    write set := empty;
    delete set := empty;
    start tn := tnc)

tend = (  
    (finish tn := tnc;
     valid := true;
     for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
     if valid
       then ((write phase); tnc := tnc + 1; tn := tnc));
    if valid
      then (cleanup)
      else (backup)).
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)
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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, and with large *think times* etc...
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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)
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
Concurrency in B+-Trees

- 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

- 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)
Concurrent access to B+-Trees

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).
Concurrency in B+-Trees

- 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
Concurrency in B+-Trees

- 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
Figure 3. A B-link tree page split

(a) Example B-link tree node

(b) Before half-split

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

- From: Srinivasan, Carey; VLDB J. 1993

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

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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.)
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’s “long”? 
  - End of the transaction to be safe
  - For serializability, sufficient to hold till the shrinking phase, no recoverability
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”
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”
Outline

1 Overview
   - Mechanisms: Locks

2 Optimistic Concurrency Control
   - Locking vs Optimistic

3 Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4 Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5 Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
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
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)
Anomalies

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
Anomalies

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”)
ANSI SQL Isolation Levels

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>

NOTE: This is from 1995... may have been changed
Fuzzy read == non-repeatable read
Anomaly Serializable ??
  Turns out disallowing the three phenomenon doesn’t guarantee true serializability
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?
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:
- \( w_1[X] \): Transaction 1 wrote data item \( X \)
- \( C_1, A_1 \): Transaction 1 committed or aborted
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:
  - $w_1[X]$: Transaction 1 wrote data item $X$
  - $C_1, A_1$: Transaction 1 committed or aborted
- Phenomenon vs Anomalies: Dirty Reads
  - Anomaly: **Anamoly1**: $w_1[X] \ldots r_2[X] \ldots (C_2 \text{ and } A_1 \text{ in some order})$
    - Something actually went wrong
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:**
- \(w_1[X]\): Transaction 1 wrote data item \(X\)
- \(C_1, A_1\): Transaction 1 committed or aborted

**Phenomenon vs Anomalies: Dirty Reads**
- Anomaly: \(\text{Anomaly}_1: w_1[X] \ldots r_2[X] \ldots (C_2 \text{ and } A_1 \text{ in some order})\)
  - Something actually went wrong
- Phenomenon: \(P_1: w_1[X] \ldots r_2[X] \ldots (C_1 \text{ or } A_1)\)
  - Corresponds to possible serial execution (T1, then T2)??
Anomalies

Dirty reads

- Anomaly: **Anomaly1**: \( w_1[X] \ldots r_2[X] \ldots \) (\(C_2\) and \(A_1\) in some order)
- Phenomenon: **P1**: \( w_1[X] \ldots r_2[X] \ldots \) (\(C_1\) or \(A_1\))
Anomalies

- Dirty reads
  - Anomaly: Anomaly1: w1[X] \ldots r2[X] \ldots (C2 and A1 in some order)
  - Phenomenon: P1: w1[X] \ldots r2[X] \ldots (C1 or A1)

- 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
Anomalies

- Dirty reads
  - Anomaly: **Anomaly1**: \( w_1[X] \ldots r_2[X] \ldots \) (C2 and A1 in some order)
  - Phenomenon: **P1**: \( w_1[X] \ldots r_2[X] \ldots \) (C1 or A1)
- 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: \( r_1[x=50] \ w_1[x=10] \ r_2[x=10] \ r_2[y=50] \ c_2 \ r_1[y=50] \ w_1[y=90] \ c_1 \)
  - Both transactions obey the constraint
Anomalies

Dirty reads
- Anomaly: Anomaly1: \( w_1[X] \ldots r_2[X] \ldots \) (C2 and A1 in some order)
- Phenomenon: P1: \( w_1[X] \ldots r_2[X] \ldots \) (C1 or A1)

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: \( r_1[x=50] \, w_1[x=10] \, r_2[x=10] \, r_2[y=50] \, c_2 \) \( r_1[y=50] \, w_1[y=90] \, c_1 \)
  - 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 - $w_1[X] \ldots w_2[X] \ldots (C_1 \text{ or } A_1)$
  - P1: dirty reads - $w_1[X] \ldots r_2[X] \ldots (C_1 \text{ or } A_1)$
  - P2: non-repeatable read - $r_1[X] \ldots w_2[X] \ldots (C_1 \text{ or } A_1)$
  - P4: phantom read - $r_1[\text{pred}] \ldots w_2[Y \text{ in pred}] \ldots (C_1 \text{ or } A_1)$
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
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
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

- 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 \( R.a \in [10, 20] \)

### 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>
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)

\[ w_1[X] \ldots r_2[X] \ldots \text{T1-Abort} \]
Allowed even though T2 reads dirty data.

Cursor Stability ??

Commonly used

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
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.
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)
    - $\text{w1}[X] \ldots r2[X] \ldots T1$-Abort
    - Allowed even though T2 reads dirty data.

- Cursor Stability ??
  - Commonly used
  - 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
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:
    - "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
Outline

1 Overview
   - Mechanisms: Locks

2 Optimistic Concurrency Control
   - Locking vs Optimistic

3 Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4 Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5 Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Snapshot Isolation with some form of locking

If you want transaction-level read consistency, and if your transaction does not require updates, then you can specify a read-only transaction.

... long running read-only transactions sometimes receive a "snapshot too old" error..

.. always performs necessary locking to ensure data concurrency, integrity, and statement-level read consistency..

REPEATABLE READ: .. does not normally support this isolation level, except as provided by SERIALIZABLE..

Also using read-only transaction mode

.. Because Oracle does not use read locks, even in SERIALIZABLE transactions..

Lot more details at the link
Presumably based on locking, but couldn’t find specifics
  Maybe I was not looking in the right place
See DB2 Isolation Levels or this
Supports most standard semantics
**cursor stability (CS):** similar to READ COMMITTED
  takes short read locks and short predicate locks
  .. higher level of integrity with **repeatable read (RR)** .. all
  page locks are held until released by a COMMIT/ROLLBACK
  ... with CS read-only page locks released as soon as another
  page is accessed..
  .. repeatable read is default..

RR appears to prevent Phantoms also – equiv to ANSI SERIALIZABLE
**read stability** between CS and RR – equiv to ANSI REPEATABLE READ
Microsoft SQL Server

- See [Microsoft SQL Server Isolation Levels](#), or [this](#).

- Appears to be locking-based
  - To protect from phantom reads (if required), it will use an INDEX to do predicate locks, and will default to locking the entire table if no appropriate index.

- Nice explanation of phantom reads
  - Especially read the comments and Craig’s answers.

- Details on Predicate Locking – called Key-Range Locks.

- Supports a SNAPSHOT isolation level
  - ... transactions running under snapshot isolation take an optimistic approach to data modification ..
PostgreSQL

- Multi-version concurrency control
- Can do application-level locking if needed

MySQL or InnoDB

See [this](#) for an interesting discussion about how InnoDB and Oracle isolation levels with same names appear to differ

- I didn’t dig deeper, so maybe incorrect
1. Overview
   - Mechanisms: Locks
2. Optimistic Concurrency Control
   - Locking vs Optimistic
3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?
4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous
5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Recovery Mechanism

To guarantee Atomicity and Durability

- Abort/Rollbacks, System Crashes etc..
- Reasons for crashes
  - Transaction failures: logical errors, deadlocks
  - System crash: power failures, operating system bugs etc
  - Disk failure: head crashes
- We will assume **STABLE STORAGE**
  - Data is not lost once its on disk
  - Typically ensured through redundancy (e.g. RAID) and/or wide-area replication
Options

- **ATOMIC:**
  - A transaction’s updates become visible on disk all at once
  - BUT disks only support atomic single-block writes
    - Can be done through “shadow paging”
      - Make a copy of the page being updated and operate on that (System R)
  - Not desirable
    - Storage space, sequentiality lost etc...

- **STEAL:**
  - The buffer manager can steal a memory page for replacement purposes
    - The page might contain dirty writes

- **FORCE:**
  - Before committing a transaction, force its updates to disk
Recovery Mechanism

- Easiest option: NO STEAL, FORCE
  - NO STEAL, so atomicity easier to guarantee
  - No serious durability issues because of FORCE

- Issues:
  - How to force all updates to disk atomically?
    - Can use shadow paging
  - A page might contain updates of two transactions?
    - Use page-level locking etc.
    - For more details, see
      
      Repeating History Beyond Aries; Mohan; VLDB 1991
Recovery Mechanism

- Desired option: STEAL, NO FORCE
- STEAL: Issues
  - Dirty data might be written on disk
  - **Use UNDO logs** so we can rollback that action
  - The UNDO log records must be on disk before the page can be written (Write-Ahead Logging)
    - Otherwise: dirty data on disk, but no UNDO log record
- NO FORCE: Issues
  - Data from committed transactions might not make it to disk
  - **Use REDO logs**
  - The REDO log records must make it disk before the transaction is “committed”
- Either case: log must be on the stable storage
Log Records

- Physical vs Logical logging:
  - Physical: Before and after copies of the data
  - Logical: 100 was added to $t1.a$, diffs

- Must be careful with logical log records
  - More compact, but not idempotent
  - Can’t be applied twice

- Physical more common

- Why okay to write log but not the pages?
  - Logs written sequentially on a separate disk
  - The change, and hence the log record, smaller
Checkpoints

- Don’t want to start from the beginning of LOG everytime.
- Use checkpoints to record the state of the DBMS at any time.
- Simplest option:
  - Stop accepting new transactions, finish all current transactions.
  - Write all memory contents to disk, all log records to LOG disk.
  - Write a checkpoint record.
  - Start processing again.
- Not acceptable.
  - Need to allow checkpoint to happen during normal processing.
  - Typically dirty data written to disk as well.
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Simple Log-based Recovery

- Each action generates a log record (before/after copies)
- **Write Ahead Logging (WAL):** Log records make it to disk before corresponding data page
- **Strict Two-Phase Locking**
  - Locks held till the end of transaction
  - Once a lock is released, not possible to undo
- **Normal Processing: UNDO (rollback)**
  - Go backwards in the log, and restore the updates
  - Locks are already there, so not a problem
- **Normal Processing: Checkpoints**
  - Halt the processing
  - Dump dirty pages to disk
  - Log: *(checkpoint list-of-active-transactions)*
Simple Log-based Recovery: Restart

Analysis:
- Go back into the log till the checkpoint
- Create *undo-list*: \((T_i, \text{Start})\) after the checkpoint but no \((T_i, \text{End})\)
- Create *redo-list*: \((T_i, \text{End})\) after the checkpoint

**Undo before Redo:**
- Undo all transactions on the undo-list one by one
- Redo all transactions on the redo-list one by one
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
     - ARIES
     - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Need for Logical Logging

- Physical logging requires 2PL
  - Not desirable in many scenarios
  - High content objects (e.g., balances)
  - Indexes
    - Recall the B-Link tree concurrency mechanism
    - Transactions can interfere in arbitrary ways
Concurrent with Indexes

- An undesirable scenario

**Figure 1:** Undesirable Interaction Between a Structure Modifying Transaction and an Insert Transaction
Concurrency with Indexes

- Need for logical undo

This scenario shows why the page affected during the rollback of an action may be different from the one affected during forward processing. Such a logical undo may require retraversing the tree from the root to locate the key. This is caused by an intervening page split which moves the originally inserted key to a different page. Writing log records during rollback actions allows the change to the different page to be logged.

Figure 3: Logical Undo Scenario
Fine-Granularity Locking

- Must be careful if doing record-level locking
- Some prior systems did *selective redo*, but can only support page-level locking

![Diagram showing ARIES: A Transaction Recovery Method](image)

**Fig. 15.** Selective redo with WAL—problem-free scenario.

**ARIES:** A Transaction Recovery Method

- **Redo:** Redoes Update 30
- **Undo:** Undoes Update 20

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 10</td>
<td>LSN</td>
</tr>
<tr>
<td>T2 20</td>
<td>Write</td>
</tr>
<tr>
<td>T1 30</td>
<td>Redo</td>
</tr>
<tr>
<td>T1 Commit</td>
<td>Commit</td>
</tr>
</tbody>
</table>

**Notes:**

- **T1** is a Nonloser
- **T2** is a Loser
- **LSN** is a Page_LSN value that is used to determine whether an update operation has been undone.
- **REDO** is used to redo an update that was previously undone.
- **UNDO** is used to undo an update that was previously redone.

---

**ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992.**
In the above case, because of the LSNs, there will be an attempt to undo the effect of T1 even though it is not present in the data

- okay in some cases, but not in many others
Outline

1 Overview
   - Mechanisms: Locks

2 Optimistic Concurrency Control
   - Locking vs Optimistic

3 Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4 Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
     - ARIES
   - Miscellaneous

5 Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
     - Paxos Consensus Commit Protocol
**Log-based Recovery**

- Every database action is logged
- Even actions performed during *undo* (also called *rollback*) are logged

**Log records:**

- (LSN, Type, TransID, PrevLSN, PageID, UndoNextLSN (CLR Only), Data)
- LSN = *Log Sequence Number*
- Type = *Update* / *Compensation Log Record* / *Commit related* / *Non-transaction related* (OS stuff)
- Allows logical logging
  - More compact, allows higher concurrency (*indexes*)
ARIES: Logs

- Physical Undos or Redos (also called page-oriented)
  - Store before and after copies
  - Easier to manage and apply - no need to touch any other pages
  - Requires stricter locking behaviour
    - Hence not used for *indexes*

- Logical Undos
  - More compact, allow higher concurrency
  - May not be idempotent: Shouldn’t undo twice

- Compensation Log Records (CLRs)
  - Redo-only; Typically generated during abort/rollback
  - Contain an UndoNextLSN - can skip already undone records.

- ARIES does “Physiological” logging
  - Physical REDO: Page oriented redo recovery
  - Supports logical UNDO, but allows physical UNDO also
ARIES: Other Data Structures

With each page:
- *page_LSN*: LSN of last log record that updated the page

Dirty pages table: *(PageID, RecLSN)*
- *RecLSN* (recovery LSN): Updates made by log records before RecLSN are definitely on disk
- Min(RecLSN of all dirty pages) → where the REDO Pass starts

Transaction Table: *(TransID, State, LastLSN, UndoNxtLSN)*
- State: Commit state
- UndoNxtLSN: Next record to be processed during rollback
ARIES: Assumptions/Setup

- STEAL, NO FORCE
- In-place updating
- Write-ahead Logging (WAL)
  - Log records go to the stable storage before the corresponding page (at least UNDO log records)
  - May have to flush log records to disk when writing a page to disk
- Log records flushed in order
- Strict 2 Phase Locking
- Latches vs Locks
  - Latches used for physical consistency
  - Latches are shorter duration
ARIES: What it does

- Normal processing:
  - Write log records for each action

- Normal processing: Rollbacks/Partial Rollbacks
  - Supports “savepoints”, and partial rollbacks
  - Write CLRs when undoing
  - Allows logical undos
  - Can release some locks when partial rollback completed

- Normal processing: Checkpoints
  - Store some state to disk
  - Dirty pages table, active transactions etc.
  - No need to write the dirty pages to disk: They are continuously being written in background
  - Checkpoint records the progress of that process
  - Called **fuzzy checkpoint**
Aries: Restart Recovery

- **Redo before Undo**
- Analysis pass
  - Bring dirty pages table, transactions up to date
- **Redo pass (repeating history)**
  - Forward pass
  - Redo everything including transactions to be aborted
  - Otherwise page-oriented redo would be in trouble
- **Undo pass: Undo loser transactions**
  - Backwards pass
  - Undo simultaneously
  - Use CLR to skip already undone actions
- Selective and deferred restart
- Fuzzy image copies
- Media recovery
- High concurrency lock modes (for increment/decrement operations)
- Nested Top Actions:
  - Transactions within transactions
  - E.g. Split a B+-Tree page; Increase the Extent size etc...
  - Use a dummy CLR to skip undoing these if the enclosing transaction is undone.
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Basic ARIES Protocol not that complex
  Fuzzy checkpoints, support for nested transactions etc complicate matters

WAL in PostgreSQL
  Note the line: “.. If fsync is off then this setting is irrelevant, since updates will not be forced out at all.”

Postgres Storage Manager
  Allowed time-travel: kept all copies of the data around
  Used a small NVRAM (flash memory) in interesting ways to simplify recovery
  Eventually taken out of Postgres (because of lack of a proper implementation?)
1. **Overview**
   - Mechanisms: Locks

2. **Optimistic Concurrency Control**
   - Locking vs Optimistic

3. **Locking in B-Trees**
   - Degrees of Consistency
   - Granularity of Locks...
   - ANSI SQL Definition
   - Industry?

4. **Recovery: Concepts**
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. **Distributed Databases**
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Distributed Databases

- Goal: a small-scale (dozens of sites) distributed DBMS
- Desiderata (or Extending \textit{data independence})
  - Location transparency
    - users don’t need know where data is
  - Performance transparency
    - performance should be independent of query submission site
  - Copy transparency
    - allow \textit{replication} for availability
  - Transaction transparency
    - looks like single-site xacts
Distributed Databases

- Goal: a small-scale (dozens of sites) distributed DBMS
- Desiderata (or Extending *data independence*)
  - Fragment transparency
    - tables can be fragmented to different sites
    - like parallel databases
  - Schema change transparency
    - schema updates at a single site should affect global schema
  - Local DBMS transparency
    - arbitrary local DBMSs
- Many research prototypes: R*, SDD-1, Distributed Ingres etc. . .
- Few commercial systems...
Distributed Databases

... vs Data Integration
- Distributed data sources but query processing centralized
- Query processing/performance easy to do
- Harder questions: Schema integration/mapping

... vs Federated
- Federated: A loose federation of autonomous sites
- Emphasis on “autonomous”
- Little control over what happens where
  - Mariposa Project @ Berkeley

... vs Grid/P2P
- Complex query processing hard, not well motivated
- PIER Project @ Berkeley: Allowed relations to be fragmented across DHT/Grid
- Location transparency
- Local privacy & control
- Local performance on local queries
- Site autonomy: No central catalogs, no central scheduler, no central deadlock detector
- Catalogs stored as “soft state”
- Tables could be fragmented, replicated
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Distributed Transactions

- May make updates at many sites
- How do you commit?
  - Everyone must commit or abort
  - 2-Phase, 3-Phase Commit Protocols
- Deadlocks (assuming locking is used)?
  - Distributed deadlock detection
  - Solution 1: Pass around “waits-for” graphs
  - Solution 2: Time-out based (perhaps more practical)
- Next class
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Another important consideration for distributed databases

“Why” replication?

- Better availability
  - Critical in mobile applications, other places where disconnections common
- Better performance
  - Don’t need to go to the “master” site for every update
  - “Caches” are a form of replica
Another important consideration for distributed databases

“Why” replication?
- Better availability
  - Critical in mobile applications, other places where disconnections common
- Better performance
  - Don’t need to go to the “master” site for every update
  - “Caches” are a form of replica

“Why not” replication?
- Keeping the copies up-to-date very tricky
- Systems work well with a few nodes, but larger scale deployments may suffer from "system delusion"
  - System delusion: Inconsistent databases with no obvious way to repair it
  - Scaleup pitfall: systems work well for a few nodes, but don’t at larger scales
Some Examples of "Replicated Databases"

- **Checkbook**
  - Lazy maintenance + Clients deal with reconciliation
  - Use of "commutative" transactions simplifies things

- **Lotus Notes**
  - Used quite widely, especially within IBM
  - Tries to aim for *convergence*: similar to “eventual consistency”

- **Door-to-door sales**
- **Domain Name Servers (DNS)**
  - Lazy approach with reconciliations

- **CODA** is Group+Lazy
  - Everyone can update their copies, reconciliations needed
  - Same with *cvs* or *svn*
Replication Models

- Eager vs Lazy

Figure 1: When replicated, a simple single-node transaction may apply its updates remotely either as part of the same transaction (*eager*) or as separate transactions (*lazy*). In either case, if data is replicated at \( N \) nodes, the transaction does \( N \) times as much work.

<table>
<thead>
<tr>
<th>A single-node Transaction</th>
<th>A three-node Eager Transaction</th>
<th>A three-node Lazy Transaction (actually 3 Transactions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write A Write B Write C Commit</td>
<td>Write A Write A Write B Write B Write C Write C Commit</td>
<td>Write A Write B Write C Commit Write A Write B Write C Commit Write A Commit</td>
</tr>
</tbody>
</table>

Table 1. Variables used in the model and analysis

- Table: Variables
  - DB_Size
  - Object Master
  - Object Group
  - N+1
  - N transactions
  - N object owners
  - one object owner
  - one transaction
  - one object owner
  - one transaction

Gray & Reuter, pp. 149-150
### Replication Models

- **Group vs Master**

<table>
<thead>
<tr>
<th>Table 1:</th>
<th>Variables used in the model and analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>N</strong>: number of distinct objects in the data-set.</td>
<td><strong>DB_Size</strong>: database size.</td>
</tr>
<tr>
<td><strong>N</strong>: number of transactions.</td>
<td><strong>N</strong>: number of object owners.</td>
</tr>
</tbody>
</table>

**Figure 2:** Updates may be controlled in two ways. Either all updates emanate from a master copy of the object, or updates may emanate from any. Group ownership has many more chances for conflicting updates.

**Object Master**

- Each object has a master node. Only the primary copy of the object is updated, and other replicas are read-only. Updates to other replicas propagate asynchronously.

**Object Group (no master)**

- Any node with a copy of a data item can update it. This is often called transactional update-anywhere-anytime-anyway.

**Figure 1:** Shows two ways to propagate updates to replicas:

- **Eager:** One replica is updated by the originating transaction. Updates are applied to all replicas of an object, and all updates emanate from a master copy of the object.
- **Lazy:** Updates may be controlled in two ways. Either updates are applied to all replicas of an object, or updates may emanate from any. Group ownership has many more chances for conflicting updates.
Replication Analysis: Issues

- **Deadlock rate**
  - With large transactions, deadlock frequencies start increasing

- **Reconciliation rate**

- **Time-dilation**
  - As system load and contention increases, time to complete an "action" increases
  - Ignored in this paper

- **With N nodes, N times as many transaction originate**
  - Critical question: Is the database size also increasing?
  - Affects deadlock and reconciliation rates
Eager Replication

Analysis Method

- Model the number of transactions, the running time for each transaction etc.
- Use this to compute the probability of “wait”
- Use this to compute the probability/rate of “deadlocks”
- Turns out: Deadlocks grow as $O(N^3)$ where $N =$ number of nodes
- Lower if you assume the database size also scales
Mobile nodes cannot use this scheme when disconnected.

Rough analysis under simplifying assumptions shows that for *group* replication:

- Deadlocks $\propto N^3$ if database size is constant
- Deadlocks $\propto N$ if database size grows by $N$
- Deadlocks $\propto \text{Transaction-Size}^5$

*Master* replication avoids deadlocks.

Eager replication also has a very high latency.
- Not discussed in the paper as much.
Figure 4: A lazy transaction has a root execution that updates either master or local copies of data. Then subsequent transactions update replicas at remote nodes — one lazy transaction per remote replica node. The lazy updates carry timestamps of each original object. If the local object timestamp does not match, the update may be dangerous and some form of reconciliation is needed.
Group Strategy
- Use timestamps for detecting and reconciling
- Analysis shows:
  - Reconciliation rate $\propto N^3$
- Much worse with mobile nodes
  - Since the update propagation delays would be huge
- System may seem to perform well with few nodes, but much worse as the scale increases

Master Strategy
- No reconciliations required, but much longer waits
- Not suitable for disconnected clients
Examples

- Lotus Notes
  - Supports lazy group replication using timestamps
  - Aims for convergence: if no changes, all replicas should converge to the same state
  - Supported operations
    1. Append: Add data to a file with timestamps
    2. Timestamped replace a value
  - The latter can be problematic because of possibility of lost updates
  - The final version should reflect all changes
  - Could also support:
    3. Commutative updates
Microsoft Access

- Maintains version vectors with replicated records
- Version vectors exchanged on a periodic basis
  - Most recent update wins
  - Rejected updates reported

Question: why doesn’t this result in system delusion?
A Two-Tier Replication Scheme

**Figure 4**: A lazy transaction has a root execution that updates either master or local copies of data. Then subsequent transactions update replicas at remote nodes — one lazy transaction per remote replica node. The lazy updates carry timestamps of each original object. If the local object timestamp does not match, the update may be dangerous and some form of reconciliation is needed.
A Two-Tier Replication Scheme

- Make a distinction between “base” (always-connected) and “mobile” nodes
- Use eager replication among base nodes
- Mobile nodes make “tentative” transactions when disconnected
  - “Submit” these to some base node when connected
  - The transaction may be rejected

Acceptance Criteria:
Submit a criteria (e.g. balance > 0) when submitting a transaction
Even if the results are different, the transaction may be accepted
In other words, use “commutative” transactions
A Two-Tier Replication Scheme

- Make a distinction between “base” (always-connected) and “mobile” nodes
- Use eager replication among base nodes
- Mobile nodes make “tentative” transactions when disconnected
  - “Submit” these to some base node when connected
  - The transaction may be rejected
- Acceptance Criteria:
  - Submit a criteria (e.g. \( \text{balance} > 0 \)) when submitting a transaction
  - Even if the results are different, the transaction may be accepted
- In other words, use “commutative” transactions
Outline

1 Overview
   - Mechanisms: Locks

2 Optimistic Concurrency Control
   - Locking vs Optimistic

3 Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4 Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5 Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
CAP Theorem requires choosing between Consistency, Availability, and Tolerance to Network Partitions

- The three are not equal
- A better restatement: Since network partitions are going to happen, choose between consistency and availability
- Databases usually aim for C
  - If there is partition, system is unavailable
  - Disconnected nodes is an example of this

For always-available systems, must give up Consistency
Client operates on top of a distributed file system, where nodes may fail

- Strong consistency: full serializability
- Eventual consistency: similar to convergence

Special cases of eventual consistency

- Causal consistency: If process A has communicated an update to process B, a subsequent access by B returns the updated value
- Read-your-writes consistency: If process A writes some update, it will see that update
- Session consistency: Same as above, but only within a session
- Monotonic read and Monotonic write consistencies
  - If you read (write) a new version, never read an older version afterwards
Let:
- $N =$ number of replicas
- $W =$ number of replicas that need to acknowledge an update before it is considered completed
- $R =$ number of replicas contacted to access a data object

If $W + R > N$: we have strong consistency
- Similar to *quorum* protocols – more next class
- e.g., $N = 2, W = 2, R = 1$

Weak/eventual consistency arises when $W + R \leq N$
- e.g., $W = 1, R = 1, N > 1$
Consistency Semantics

- Whether we can guarantee read-your-writes, or session consistency depends on "stickiness" of clients to servers
  - If the client connects to the same server every time, easy to guarantee
- Amazon’s Dynamo system (later Cassandra) offers these options to the users/application developers
Are partitions the main thing to worry about?

- Stonebraker’s comments on the above article

Generally LANs within data centers are replicated
- So less chance of partition there

In many cases, we have degenerate partitioning
- A single node gets disconnected
- But we can usually handle that

However partitions can happen on WAN
- Which is the kind of scenario Amazon, Google systems focus on
- How frequent is that scenario?
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
Distributed Transactions in R*

Assumptions
- update in place, WAL
- batched force of log records

Desired Characteristics
- guaranteed xact atomicity
- ability to “forget” outcome of commit ASAP
- minimal log writes & message traffic
- optimized performance in no-failure case
- exploitation of completely or partially R/O xacts
- maximize ability to perform unilateral abort

In order to minimize logging and comm:
- rare failures do not deserve extra overhead in normal processing ("make common case fast")
- hierarchical commit better than 2P
## Normal 2-Phase Commit

<table>
<thead>
<tr>
<th>Coordinator Log</th>
<th>Messages</th>
<th>Subordinate Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREPARE →</td>
<td></td>
<td>prepare*/abort*</td>
</tr>
<tr>
<td></td>
<td>← VOTE YES/NO</td>
<td></td>
</tr>
<tr>
<td>commit*/abort*</td>
<td>COMMIT/ABORT →</td>
<td>commit*/abort*</td>
</tr>
<tr>
<td></td>
<td>← ACK</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- * → forced on log (for durability/atomicity)
- Always log before sending a message
- Total cost:
  - subords: 2 forced log-writes (prepare/commit), 2 messages (YES/ACK)
  - coord: 1 forced log write (commit), 1 async log write (end), 2 messages/subord (prepare/commit)
Normal 2-Phase Commit

- Dealing with failures
  - Recovery process at each site
  - Use logs for handling failures
  - E.g. a subordinate crashes, and wakes up, and sees a prepare* → ask coordinator what happened to the transaction
  - Need to prove correctness carefully

- Hierarchical 2PC
  - Tree-shaped communication hierarchy
  - Non-root, non-leaf nodes behave as both coordinators and subordinates
Presumed Abort/Commit

- **Goal:** Reduce the number of log-writes and messages

- **Presumed abort (PA)**
  - Don’t ACK aborts, don’t force-write logs for aborts
  - After failure and restart, ask the coordinator what happened
  - If coordinator doesn’t find any information, it assumes abort
  - Much fewer messages with read-only transactions

- **Presumed commit (PC)**
  - Don’t ACK commits, don’t force-write logs for commits
  - Some tricky details
  - **Better than PA** if commit is more common than abort

- Can choose on a per-transaction basis
Summary of results

- PA always better than 2P
- If few updating subords, PA is better; if many updating subords, PC is better
- Choice between PA and PC to be made on per-transaction basis
- 2-Phase commit is a blocking protocol
  - Waits on the coordinator/its parent in the hierarchy
  - May wait for a long time if the coordinator goes down
- 3-Phase commit solves this problem
  - Generally considered too expensive and not used

2-Phase Commit in Oracle
Outline

1. Overview
   - Mechanisms: Locks

2. Optimistic Concurrency Control
   - Locking vs Optimistic

3. Locking in B-Trees
   - Degrees of Consistency
   - Granularity of Locks...; Gray et al.; 1976
   - ANSI SQL Definition
   - Industry?

4. Recovery: Concepts
   - Simple Log-based Recovery
   - Why ARIES?
   - ARIES
   - Miscellaneous

5. Distributed Databases
   - Distributed Transactions
   - Dangers of Replication
   - Eventually Consistent
   - Distributed Commit Protocols
   - Paxos Consensus Commit Protocol
The Consensus Problem

- A collection of processes can propose values
- Want to choose a single value
- Safety Requirements:
  - Only a single value among the proposed ones may be chosen
  - A process does not learn false information
- Liveness Requirements:
  - Hard to specify precisely, but some progress should happen
- In presence of:
  - Lost, duplicated messages, that can take arbitrarily long to be delivered
    - i.e., if no message within 1 hour, can’t assume it won’t arrive (e.g., imagine a failed router)
  - Node failures and restarts
    - Nodes must have some limited "memory" (e.g., logs)
The Consensus Problem: Impossibility

- Known to be "impossible" under several different models
  - E.g., Fischer, Lynch, Paterson; 1985
    - Show that impossible to achieve consensus with just one faulty process
- However, the results are pessimistic
  - They don’t rule out consensus
  - Just that it may take infinite time
- Several protocols known to achieve it with probability 1 since the bad case becomes extremely unlikely
- Paxos Consensus can go on indefinitely
We differentiate between: "acceptors", "proposers", "learners"

Focus on the choosing: after choosing, the acceptors can communicate the value to the learners

1. Single acceptor
   - Easy, but no fault tolerance

2. Multiple acceptors
   - A proposer sends a proposed value to a set of acceptors
   - If a majority accepts you are done
   - BUT: What about race conditions?
     - Half the acceptors choose one value, other half chooses another value
See "Paxos Made Simple" by Lamport for the reasoning.

Definitions:

- **ballot number**: a way to disambiguate – each proposer has a unique set of ballot numbers it chooses from.
- A **prepare(ballot number)** request: A request from a proposer to an acceptor asking: is there a chance you accept a proposal with this ballot number?
- A **please-accept(ballot number, value)** request: A request from a proposer to an acceptor to accept a proposed value *value* with this ballot number.
Paxos Consensus Algorithm

Let $P$ be a proposer, and $A$ be one of the acceptors it talks with.

Protocol:

- Phase 1(a): $P$ selects a "ballot number" $n$ and sends `prepare(n)` request to a majority of acceptors.
- Phase 1(b): Acceptor $A$ gets the `prepare(n)` request.
  - If already responded to a ballot number larger than $n$, ignore the request.
  - Otherwise: $A$ promises to $P$ that it won't accept a proposal with smaller ballot number in future.
  - $A$ sends $P$ the value of the highest-numbered proposal it has already accepted, if any.

So if $P$ hears back from a majority of acceptors, it knows:
- Every one of them is willing to accept its proposal.
- It also knows whether a value has already been chosen.
- In that case, it tries to get the same value accepted again.

But it must act fast: some other proposer could send a `prepare()` with a higher number.
Let $P$ be a proposer, and $A$ be one of the acceptors it talks with

Protocol:

- Phase 1(a): $P$ selects a "ballot number" $n$ and sends $prepare(n)$ request to a majority of acceptors.
- Phase 1(b): Acceptor $A$ gets the prepare($n$) request
  - if already responded to a ballot number larger than $n$, ignore the request
  - otherwise: $A$ promises to $P$ that it won't accept a proposal with smaller ballot number in future
  - $A$ sends $P$ the value of the highest-numbered proposal it has already accepted, if any

So if $P$ hears back from a majority of acceptors, it knows:

- Every one of them is willing to accept its proposal
- It also knows whether a value has already been chosen
  - In that case, it tries to get the same value accepted again

But it must act fast: some other proposer could send a prepare() with a higher number
Let $P$ be a proposer, and $A$ be one of the acceptors it talks with.

Protocol:

- Phase 2(a): $P$ sends back $accept(ballotnumber, v)$ where $v$ is:
  - Value of the highest-numbered proposal among the responses (i.e., if there is already an accepted value)
  - Or, any value that it wants

- Phase 2(b): If an acceptor receives an accept request for a proposal number $ballotnumber$, it accepts unless it has already responded to a $prepare$ with a higher ballot number.
Let $P$ be a proposer, and $A$ be one of the acceptors it talks with.

Protocol:

- Phase 2(a): $P$ sends back $\text{accept}(\text{ballotnumber}, v)$ where $v$ is:
  - Value of the highest-numbered proposal among the responses (i.e., if there is already an accepted value)
  - Or, any value that it wants

- Phase 2(b): If an acceptor receives an accept request for a proposal number $\text{ballotnumber}$, it accepts unless it has already responded to a $\text{prepare}$ with a higher ballot number.

Note that: If an acceptor has accepted a value, it cannot accept another value.

- It can accept another proposal with the same value

It is not possible for two different acceptors to accept two different values.

Think about why...
Paxos Consensus Algorithm

- No safety issues
  - A proposer can abandon proposal anytime, acceptors or proposers can crash anytime etc

- Progress?
  - Two different proposers can race each other, getting their prepare()’s accepted, but not their accepts()
  - Typically we assume a single distinguished proposer (like a leader)
    - Multiple leaders don’t compromise correctness, but can impede progress

See Wikipedia page for more details.
Paxos Consensus Algorithm

- No safety issues
  - A proposer can abandon proposal anytime, acceptors or proposers can crash anytime etc

- Progress?
  - Two different proposers can race each other, getting their prepare()’s accepted, but not their accepts()
  - Typically we assume a single distinguished proposer (like a leader)
    - Multiple leaders don’t compromise correctness, but can impede progress

- Typically a single process plays multiple roles
  - Simplest case: each process is a proposer, acceptor, and a learner
Paxos Consensus Algorithm

- No safety issues
  - A proposer can abandon proposal anytime, acceptors or proposers can crash anytime etc

- Progress?
  - Two different proposers can race each other, getting their prepare()’s accepted, but not their accepts()
  - Typically we assume a single distinguished proposer (like a leader)
    - Multiple leaders don’t compromise correctness, but can impede progress

- Typically a single process plays multiple roles
  - Simplest case: each process is a proposer, acceptor, and a learner

- Paxos used in several places, including in Google
  - See Wikipedia page for more details
Recall: Resource Manager == Local Transaction Managers

Each RM decides whether to commit or abort – we must reach consensus on the decision and remember it.
Recall: Resource Manager == Local Transaction Managers

Each RM decides whether to commit or abort – we must reach consensus on the decision and remember it

Briefly:

We run an instance of Paxos for "learning and remembering" the decision of each of the RMs

i.e., whether the RM wants to commit or abort
As usual, the RM writes to stable storage before communicating with the acceptors
Use the same set of \(2F + 1\) acceptors and the same leader for all the paxos instances

Each of the RMs learns the chosen value for each of the other RMs through the acceptors

Acceptors can bundle their messages together
So now, each RM knows what all others want to do and takes appropriate action
2F + 1 acceptors means that we can tolerate upto F failures
If F = 0, then we have only one acceptor == Transaction Manager
   Reduces to 2 Phase Commit
See the
   powerpoint slides by Gray and Lamport for details
   And also helpful animations