Advanced Database Systems:

transactions, database tuning, and advanced topics

Dennis Shasha, shasha@cs.nyu.edu
Main Topics

• Concurrency control — ensuring that each user appears to execute in isolation.

• Recovery — tolerating failures and guaranteeing atomicity.

• Database Tuning — how to make your database run faster.
Concurrency Control

Here is the BALANCES table:

<table>
<thead>
<tr>
<th>Employee Number</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>70</td>
</tr>
<tr>
<td>106</td>
<td>60</td>
</tr>
<tr>
<td>121</td>
<td>80</td>
</tr>
<tr>
<td>132</td>
<td>10</td>
</tr>
</tbody>
</table>
Concurrency Control — continued

Suppose the execution of several transactions overlaps in time. Assume that we are running a transaction, T1, that should compute and print the total of employee balances.

Suppose employee 121 wants to move 40 from account 121 to 101 using transaction T2.

Concurrency control must guarantee that the outcome is as if T1 happened before T2 or T2 happened before T1.

AVOID: debit from account 121, T1, then credit to 101.
Recovery

The database must remain “consistent” despite the fact that hardware fails.

Suppose the above balances are to be credited with their monthly interest. A single transaction might perform the updates.

Every account should be updated exactly once. The recovery subsystem guarantees that this change is all-or-nothing.

Once the transaction “commits,” the update is secure.
Database Tuning

What is it?

The activity of attaining high performance for data-intensive applications.

- Performance $= \text{throughput, usually}$.

- Performance $= \text{response time for real-time applications}$. 
Tuning in Context

Designing an Information System requires:

- An accurate model of the real world — user requirements analysis and specification. Knowledge acquisition + Intuition + CASE tools.

- High performance. The concern of tuning.
Tuning Issues — just about everything

- Conceptual to logical schema mappings — normalization, vertical partitioning, aggregate maintenance.

- Logical to physical schema mappings — indexes, distribution, disk layout.

- Transaction design — query formulation, isolation degree desired, and transaction length.

- Operating system and hardware choices — buffer size, thread priorities, number of disks, processors, and amount of memory.
Target for this Material

- Database administrators and sophisticated application users who already have a database management system.

  Question: How can I improve performance?

- Potential purchasers of a database management system.

  Question: What criteria should I use to choose a system?

- Designers of a database management system.

  Question: What facilities are important?

Also for students....
Tuning Complements Internals

- Internals: teach how to build B-trees.
  Tuning: show relationship between key size, fanout, and depth.
  This in turns motivates discussion of compression.

- Internals: teach clustering indexes, then later concurrency control.
  Tuning: show how clustering indexes can reduce concurrent contention for insert-intensive applications.
  This in turn motivates discussion of the implementation of record-locking.
Why a Principled Approach?

• Portability — you can use techniques on different systems or on different releases of one system.

• Generalizability — principles will lead you to good choices about features that we don’t discuss.

• Comprehensibility — you can keep it all in your head.
Overview of Tuning Part

Unit 1: basic principles of tuning.

Unit 2: concurrency control, logging, operating system, hardware.
theory applied to tuning: transaction chopping

Unit 3: index selection and maintenance.

Unit 4: tuning relational systems.

Unit 5: application-database interactions

Unit 6: data warehouses.

Unit 7+: case studies from consulting.
Advanced Topics

During the semester, we will discuss advanced topics of my current research interest as appropriate. Here are some possibilities:

1. Data structures for decision support.

2. Data mining (why large databases make statistics easy).

3. Buffering algorithms for databases and operating systems that beat LRU (theorists hide a lot in those factors of 2).

4. Farout but very fast approaches to data management.
Principles of Concurrency Control

Goal: Allow many users (perhaps many computers) to use database at same time (concurrently).

Also “correctly.”

Rationale: While one transaction waits for a read or a write, allow another to do some work.

Same as operating system rationale.
Basic Unit of Work: a transaction

Transaction is a program including accesses to the database.

Example: to process a sale, credit (reduce) inventory and debit (increase) cash.

Assumption: If a database starts in a consistent state and transactions are executed serially, then the database remains consistent.

Correctness: Make concurrent execution have same effect as a serial one. Give user illusion of executing alone.
Model for Concurrency Control

Database is a set of data items. (Independent of any data model).

Operations: read(data item); write(data item, value).

begin transaction
    sequence of operations
end transaction
What’s the big deal?

Bank has two automatic teller machines and Bob shares an account with Alice.

Bob wants to transfer 10000 dollars from checking to savings. Alice wants to check total in both (which she believes to be about $20000.)

ATM1: Bob withdraws 10000 from checking.
ATM2: Alice returns checking+savings
ATM1: Bob deposits 10000 into savings.

Alice sees only about $10,000, accuses Bob of gambling, and gets a divorce.

Reason: Execution was not “serializability.”
Serializability

A concurrent execution of a set of transactions is serializable if it produces the same return values from each read and the same result state as a serial execution of the same transactions.

Banking example was not serializable. Can you see why by looking at pattern of reads and writes?

R1(checking) W1(checking) R2(checking) R2(savings) R1(savings) W1(savings)
Criterion for serializability

Two operations conflict if they both access the same data item and one is a write.

Schedule – sequence of interleaved operations from different transactions. Actually, operations on different arguments need not be ordered in time, provided conflicting ones are.

Schedule is serializable if following graph (serialization graph) is acyclic:

Nodes are transactions.

Edge from $T_i$ to $T_j$ if an operation from $T_i$ precedes and conflicts with an operation from $T_j$.

For example, $T1 \rightarrow T2$ because of checking and $T2 \rightarrow T1$ because of savings. Hence schedule is not serializable.
Schedule Equivalence

We say $R_i(x)$ reads-from $W_j(x)$ if $W_j(x)$ precedes $R_i(x)$ and there is no intervening $W_k(x)$.

We say $W_i(x)$ is a final-write if no $W_k(x)$ follows it.

Two schedules are *equivalent* if

1) each read reads-from the same writes in both schedules; and

Condition 1 ensures that each transaction reads the same values from the database in each schedule. So it must issue the same writes (assuming writes depend only on values read and not on something else, such as time).

2) they have the same final writes. Same final state.
Serialization Graph Theorem

Theorem: If the serialization graph of a computation is acyclic, then every transaction reads the same values and writes the same values as it would in a serial execution consistent with the graph.

Proof: Take any topological sort of the graph. The topological sort represents a serial schedule with each transaction’s operations executing all alone. We want to prove first that the serial schedule has the same reads-from as the schedule that produced the graph. We will do that by looking at each item $x$ and showing that each transaction reads the same value of $x$. 
Serialization Graph Theorem Continued

Suppose the actual execution and the serial order have different reads-from. Suppose $R_m(x)$ of $T_m$ reads $x$ from $W_p(x)$ of $T_p$ in the serial order, but $R_m(x)$ of $T_m$ reads $x$ from $W_q(x)$ of $T_q$ in actual execution (implying that both $T_q \rightarrow T_m$ and $T_p \rightarrow T_m$). Since both $T_p$ and $T_q$ contain writes to $x$, they must be connected in the serialization graph, so these three transactions must be ordered in the graph. Two cases:

1) $T_p \rightarrow T_q$. Then $T_q$ must fall between $T_p$ and $T_m$ in serial schedule. So $T_m$ reads-from $T_q$ in serial schedule. Contradiction.

2) $T_q \rightarrow T_p$. Then $T_m$ must read $x$ from $T_p$ in actual execution. Contradiction.

Final write: Your homework.
Guaranteeing Serializability

Predeclaration Locking: Transaction obtains exclusive locks to data it needs, accesses the data, then releases the locks at the end.

Exclusive lock: While T1 holds an exclusive lock on x, no other transaction may hold a lock on x.

Implies: concurrent transactions must access disjoint parts of the database.

Example: L1(x) R1(x) L2(y,z) W2(z) W1(x) UL1(x) W2(y) UL2(y,z)

Non-Example: L1(x,z) R1(x) ?L2(z,y) W2(z) W1(z) UL1(x,z) W2(y)

Even though this is serializable. Transaction 2 cannot get a lock on z.
Two Phase Locking

Problem: Can’t always predict needed data items; even if could, predeclaration locking is often too conservative.

Solution: Get locks as you need them, but don’t release them until the end.

Two phases: acquire locks as you go, release locks at end. Implies acquire all locks before releasing any.

Theorem: If all transactions in an execution are two-phase locked, then the execution is serializable.
Two Phase locking Proof Sketch

Call the lockpoint of a transaction, the earliest moment when that transaction holds all its locks.

Suppose there is a cycle in the serialization graph: $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. Then $T_1$ accesses an item $x_1$ before $T_2$ accesses $x_1$ and the accesses conflict (at least one writes $x_1$). So $T_1$ must release its lock after lockpoint($T_1$) and before $T_2$ acquires its lock on $x_1$, which is before lockpoint($T_2$). Therefore lockpoint($T_1$) precedes lockpoint($T_2$). Similarly, there is an $x_2$ such that $T_2$ accesses $x_2$ before $T_3$ and the accesses conflict. So, lockpoint($T_2$) precedes lockpoint($T_3$). By transitivity, lockpoint($T_1$) precedes lockpoint($T_1$). Obviously absurd.
Read locks

Two phase locking based on exclusive locks is too conservative. Consider the schedule:

\[ R_1(x) \ R_2(x) \ R_2(y) \ R_1(y). \]

Since none of these write, every order is equivalent. So, exclusive (write) locks are too strong. Introduce notion of shared (read) locks.
Read Lock Rules

• While T holds a read lock on x, no other transaction may acquire an exclusive lock on x.

While T holds an exclusive lock on x, no other transaction may acquire any lock on x.

• T acquires a read lock on item x if it only wants to read x.

T acquires an exclusive lock on x if it may want to write x.
Deadlock Detection

Construct a blocking \textit{(waits-for)} graph. $T \rightarrow T'$ if $T'$ has a lock on an item $x$ and $T$ needs a lock on $x$.

If system discovers a cycle, it aborts some transaction in cycle (perhaps lowest priority one or most recently started one).

Example: $T_1 \rightarrow T_2$, $T_2 \rightarrow T_4$, $T_4 \rightarrow T_5$, $T_4 \rightarrow T_3$, $T_3 \rightarrow T_1$

Abort one of $T_1$, $T_2$, $T_3$, $T_4$.

Cycle detection need not happen frequently. Deadlock doesn’t go away.
Database Concurrency Control and Operating System Concurrency Control

Similarities: notion of mutual exclusion. Concern with deadlock.

Difference: Database concurrency control concerned with accesses to multiple data items. Operating Systems generally only concerned about synchronized access to single resources.

When multiple accesses are required, operating system will get a coarse lock that covers both.

That would be like locking the whole database or a whole relation — unacceptable in database setting.
Performance of Two Phase Locking


Assumptions for model:

All transactions require the same number of locks.

All data items accessed with equal probability.

All locks are write locks.

Transactions are strict 2PL (release locks upon commit or abort).
Resource and Data contention

Resource contention – contention over memory space, processor time, I/O channels.

Data contention – delays due to lock conflicts.

Thrashing occurs when throughput drops as more processes are added. In operating systems, thrashing occurs due to resource contention over memory – processes waste their time performing I/O.

In database systems, thrashing can happen due to data contention alone.
Why is There Data Contention Thrashing?

1. Deadlocks – transactions must repeatedly restart.

2. Blocking – transactions sit waiting in queues, causing other transactions to wait in queues.

The second effect is more pronounced than the first according to simulation and experiment. Even at data contention thrashing point, only 2% of transactions are deadlocked. (By far, the great majority are deadlocks among two transactions.)

Rule of thumb: if half the transactions are blocked, system is thrashing.
Definitions

N — multiprogramming level.

k — number of locks a transaction requires

D — number of data items (i.e. the unit of locking granularity).

Intuitively, the bigger D, the fewer the conflicts, and the bigger N, the more. But k plays two roles: the probability of conflict per request increases with \( kN \) and the likelihood of some conflict increases with k.
Derivation

Assuming deadlocks are rare (we are not in data contention thrashing region), each transaction has $k/2$ locks on average. By assumption, all are write locks and are uniformly distributed.

Probability of conflict per request $= (kN/2)/D$. Call it $p$.

Probability of a conflict sometime during the transaction $1 - (1 - (p))^k$. If $p$ ($kN/2D$) is small, the exponential term can be approximated by $1 - kp$.

Thus, the probability of a conflict sometime during the transaction can be calculated assuming conflicts are mutually exclusive at different times in the transaction $= kp$. 
Response Time

Let $R$ be the response time for a given load (to be calculated).

Let $T$ be the response time with concurrency control switched off (resource contention only).

Assume that when a transaction is blocked, no other transaction waits for the same lock, so waiting time is $R/2$ on average. Also, assume at most one wait per transaction.

Then $R = T + (R/2)kp$.

So, $R = T/(1 - (kp/2))$.

The throughput is just $N/R$, the number of processes divided by the response time.

So throughput $= (N/T) \cdot (1 - k^2N/4D)$.
Qualitative conclusions

Resource contention acts as a dilation factor (increases $T$).

Only refine granularity in such a way that $k$ increases slower than $D^{1/2}$. Otherwise throughput decreases.

If a fraction $s$ of the lock requests are reads, then it is as if $D$ increases to $D/(1 - s^2)$.

If a certain portion of the data has a higher traffic rate than others, then it is as if the number of granules is reduced.
Multi-granularity Locks

What if your system has some short transactions (update a seat) and some long ones (write a report)?

It would be nice to use fine (record) level granularity for the first and coarse (file) level granularity for the second.

Solution: use intention locks.
What are Intention Locks?

In basic scheme, there are intention read and intention write locks. No intention lock conflicts with any other.

However, intention read locks conflict with write locks. Intention write locks conflict with read and write locks.
Using Intention Locks

Intention lock down to the level of granularity above what you need, e.g. intention write the database and file, then write lock the record.

The report writer would intention read the database, then read lock the file.

Thus, it would conflict with writes of the file, but not with writes to another file.
Non-locking Non-deadlock Protocols

Give unique timestamps to transactions (TS). Give each data item a read timestamp (RTS) and a write timestamp (WTS).

T tries to read an item x:

if $\text{TS}(T) \geq \text{WTS}(x)$
then T reads x, $\text{RTS}(x) := \max(\text{RTS}(x), \text{TS}(T))$
else T aborts

T tries to write an item x:

if $(\text{TS}(T) \geq \text{WTS}(x))$ and $(\text{TS}(T) \geq \text{RTS}(x))$
then T writes x, $\text{WTS}(x) := \text{TS}(T)$
else T aborts
Non-locking Non-deadlock Protocols — properties

No deadlock since no waiting.

Serializable since $T \rightarrow T'$ implies $\text{TS}(T) < \text{TS}(T')$. Verify this in the homework.

Can adapt 2PL schedulers to use timestamping to prevent repeated aborts.
Locking non-deadlock protocols I

Kedem and Silberschatz JACM 1983, vol. 30, no. 4, October 787-804

Assign an ordering to the data items and insist that the transactions acquire locks in this order.

Natural: When items are logical records in a database structured according to a hierarchy.

Example: IMS (hierarchical data model).

Items are nodes of a B tree.

Nesting: entire database, relation, each block, each tuple.
Rules of Hierarchical Locking

1. Except for the first item locked, no transaction can lock an item unless lock is currently held on item’s parents.

2. No item is ever locked twice by one transaction.

Tree Locking protocol is serializable (assume all write-locks).

Assume, pessimistically, that we modify every node we lock. (This can only add to conflicts.)

Proof is involved. Here are some hints.
Main ideas of Tree Locking Proof

First(T) is the first node locked by transaction T.

If two transactions T and T’ conflict, then either First(T) is an ancestor of First(T’) or vice versa. The two transactions conflict on the "lower" of of first(T) and first(T’).

If the serialization graph contains the path $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ then there is an i ($1 \leq i \leq n$) such that first($T_i$) is an ancestor of first($T_j$) for every j ($1 \leq j \leq n$). (Proof by induction.)

Hint: Let $T_k$ be ancestor up to first m. Then Tm+1 is either an ancestor or descendant of Tm and hence of $T_k$. 
Tree Locking Hints Continued

Define a minimal path from $T_1$ to $T_n$ to be one of minimum length in the serialization graph.

If the serialization graph contains the minimal path $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ and first($T_1$) is an ancestor of first($T_j$) for all $1 \leq j \leq n$, then first($T_1$) is an ancestor of first($T_2$) is an ancestor of first($T_3$), etc.

Proof by induction. Must show that the path would not be minimal otherwise. That is, if first($T_{i-1}$) is an ancestor of first($T_i$) and first($T_{i+1}$) is an ancestor of first($T_i$), then $T_{i-1}$ conflicts with $T_{i+1}$. 
Tree Locking Theorem

Theorem: The tree protocol is serializable.

Proof sketch: find a minimal cycle. $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ Suppose first($T_1$) is the ancestor of all other first nodes. Then they must all have the same first node. Hence $T_1$ locks first($T_1$) before it locks first($T_1$), contradiction.
Directed Acyclic Graph locking

Consider a directed acyclic graph with a root. A dag protocol works as in the tree case with the modification:

Unless \( x \) is the root, to obtain a lock on \( x \), transaction must be holding a lock on some parent of \( x \), and there must have been a time at which the transactions held locks on all parents of \( x \).
Deadlock Avoidance Protocols

Give each transaction a unique timestamp. Require that numbers given to transactions always increase.

Use two phase locking.


Desired Effect: Older transactions eventually make it, because no one aborts them. Proviso: no transaction is allowed to stop trying to make progress.
Wait-Die and Kill-Wait

- **Wait-die**: If $T$ tries to access a lock held by an older transaction (one with a lesser timestamp), then $T$ aborts and restarts. Otherwise $T$ waits for the other transaction to complete.

- **Kill-wait**: If $T$ tries to access a lock held by an older transaction (one with a lesser timestamp), then $T$ waits. Otherwise $T$ aborts the other transaction.
Deadlock

Summary of deadlock considerations

Goals: 1. Every transaction will eventually terminate.

2. Want to use as few messages as possible.

3. Want there to be the possibility of interactive use of the database

4. Want strategy to work on LAN, preferably without increasing the cost as the system gets bigger.
Issue: Victim selection

current blocker – The one you find right away.

random blocker – any one at all.

min locks – one that holds fewest locks.

youngest – one with the most recent initial startup time.

min work – pick the transaction that has consumed the least amount of physical resources.

min locks has best performance, though it doesn’t guarantee termination.
Performance Conclusion

Use a locking strategy such as two phase locking.

Get highest throughput by using continuous detection and min locks strategy. Picking the youngest is within 10% of best throughout. This assumes that deadlock detection itself requires no work.

Kill-wait strategy comes in second, but is better if deadlock detection itself is expensive. Also, in interactive environments where think times are long, deadlock detection can cause excessive blocking (even in absence of deadlock, a person may be very slow).

May try to optimize kill-wait as follows: wait for some time before killing.
Optimistic Protocols (certifying)


Do not delay any operation, but don’t change the permanent database either.

At commit time, decide whether to commit or abort the transaction.
Optimistic Protocols at Commit

$T_i$'s readset, $RS(i) = \{x \text{ st } T_i \text{ has read } x\}$

$WS(i) = \{x \text{ st } T_i \text{ has written } x \text{ to a workspace}\}$

When receiving endi, certifier does the following:

$RS(\text{active}) = \bigcup RS(j) \text{ for } T_j \text{ active but } j \neq i$

$WS(\text{active}) = \bigcup WS(j) \text{ for } T_j \text{ active but } j \neq i$

if $RS(i) \cap WS(\text{active}) = \phi$

and $WS(i) \cap (RS(\text{active}) \cup WS(\text{active})) = \phi$

then certify (commit)

else abort
Verifying Optimistic Protocols

Theorem: Certifier produces serializable schedules.

Proof: Only care about certified transactions. If $T_i \rightarrow T_j$, then $T_i$ must have been certified before $T_j$ accessed the item on which they conflict. (Two cases: $T_i$ writes $x$ before $T_j$ accesses $x$ and $T_i$ reads $x$ before $T_j$ writes $x$.) This relation must be acyclic.
Applications for optimistic concurrency control

source: "The datacycle architecture for very high throughput database systems" Gary Herman, Gita Gopal, K. C. Lee, and Abel Weinrib, ACM SIGMOD 1987

In the Datacycle architecture proposed at Bell Communications Research, there is a central site that broadcasts the entire database repeatedly.

Updates to the database may come from any site. These come across separate wires from the broadcast wires from the central site.
Datacycle Architecture

Because the entire database is broadcast periodically, read and write locks would be absurd.

Read-only transactions commit provided they read data from a single broadcast cycle.

An update transaction commits if the versions of the data it reads are still the current versions. Updates are effected between broadcast cycles.

Why does this work? See homework.
Multiversion Read Consistency

A much used protocol in practice.

A read-only transaction obtains no locks. Instead, it appears to read all data items that have committed at the time the read transaction begins.

Implementation: As concurrent updates take place, save old copies.

Why does this work? See homework.
Available Copies Algorithm

Replicated data can enhance fault tolerance by allowing a copy of an item to be read even when another copy is down.

Basic scheme:

Read from one copy;

Write to all copies

On read, if a client cannot read a copy of $x$ from a server (e.g. if client times out), then read $x$ from another server.

On write, if a client cannot write a copy of $x$ (e.g. again, timeout), then write to all other copies, provided there is at least one.
Available Copies Continued

- **At Commit time:** For two phase locked transactions, ensure that all servers that you accessed (read or write) have been up since the first time they were accessed. Otherwise, abort. (Note: Read-only transactions need not abort in this case.)

- **When a new site is introduced or a site recovers:** The unreplicated data is available immediately for reading. Copies of replicated data items should not respond to a read on \( x \) until a committed copy of \( x \) has been written to them.
Site recovery

0. Commit transactions that should be committed and abort the others.

1. All non-replicated items are available to read and write.

2. Replicated parts:

   Option 1: Quiesce the system (allow all transactions to end, but don’t start new ones) and have this site read copies of its items from any other up site. When done, this service has recovered.

   Option 2: The site is up immediately. Allow writes to copies. Reject reads to x until a write to x has occurred.
Reason for Abort on Failure

Suppose we don't abort when sites that we have read from fail. Assume that lock managers are local to sites.

Suppose we have sites A, B, C, and D. A and B have copies of x (xA and xB) and C and D have copies of y (yC and yD).

$T_1$: R(x) W(y)

$T_2$: R(y) W(x)

Suppose we have R1(xA) R2(yD) site A fails; site D fails; W1(yC) W2(xB)

Here we have an apparently serializable computation, yet transaction 2 reads y before transaction 1 writes it and transaction 1 reads x before transaction 2 writes it.
Available Copies Problems

Suppose T1 believes site A is down but T2 reads from it. This could easily give us a non-serializable execution, since T2 might not see T1’s write to a variable that A holds.

So, when T1 believes a site is down, all sites must agree. This implies no network partitions.

If network partitions are possible, then use a quorum. In simplest form: all reads and all writes must go to a majority of all sites.
Weikum’s technique

- Research group at ETH Zurich led by Gerhard Weikum has been looking for automatic load control techniques to maximize throughput. They observe that higher multiprogramming levels may make better use of resources but may also increase data contention.

- Key parameter: Fraction of locks that are held by blocked transactions. If that fraction is 0, then there is no waiting. If it is .23 or higher, then there is likely to be excessive data contention.
Escrow Method

- Work by Pat O’Neil. Consider an aggregate data item, e.g. account balance.

- Observe that the operations are commutative, e.g. additions and subtractions. Want to avoid holding locks until the end of the transaction, but a subtraction can’t go through if there are insufficient funds.

- Basic idea: perform the operation on the balance, release the lock, but keep the update in an escrow account. If the transaction commits, then update on balance becomes permanent. Otherwise undone is performed based on Escrow account.
SAGAS

- Ken Salem and Hector Garcia-Molina: long lived activities consisting of multiple steps that are independent transactions. Example: making a multi-city airline reservation for a large number of individuals.

- Each step is an independent transaction and has a compensating counter-transaction. Counter-transactions attempt to undo the affect of the transaction, e.g. cancel a reservation that a reservation transaction made.

- A compensating transaction commits only if the corresponding component transaction commits but the saga aborts.
Workflow Systems

- I have an organization with many closed database systems, e.g. a telephone company or a sales and manufacturing organization.

- An activity might be a sale. This touches many databases. The results of one database may affect the steps taken by the next ones.
Taxonomy of Workflow Applications

- route work among people in a good way (e.g. Lotus Notes).

Main Memory Database — Dali approach

- A Bell Labs research database, its applications are call routing, number translation, and real-time control.

- Special data structure: T-trees (binary tree with many records in the middle of the node). Balanced using rotations as in an AVL tree. Low fanout works well for in-memory, provided a node fits within a cache line.
Concurrency Control

- Concurrency control uses wait-free style concurrency control so readers never have to wait: build subtree on the side and link in when all done.

- Each reader is given a timestamp and free nodes only when all readers that began before the nodes became unreachable have completed. (drain technique)
Multi-level Recovery

- Nested top level actions that are beneficial side effects. (This is for splits and merges.) They commit even if the entire transaction doesn’t.

- post-commit action — something that happens only after a commit has occurred (though it is locked), e.g. freeing a node after it has been deleted shouldn’t be done before commit.
Fuzzy Checkpointing

- Establish mark at head of the redo log. Write database image, then write undo log.

- Rebuild current state by start with database image, apply undo, then apply the redo log starting at mark.

- Does this work?
Throw Away Concurrency Control and Two Phase Commit

Arthur Whitney
KX Systems, a@kx.com

Dennis Shasha
New York University, shasha@cs.nyu.edu
Our Biases

• Lots of experience helping people speed up applications.
  Arthur: design a better programming language.
  Dennis: tune the databases and fix bugs.

• Modern database systems seem to require many escapes, e.g. for Wall Street analytics, graphical user interfaces. We prefer a monolingual environment that embodies a programming language.

• Our customers are impatient and very nervous.
What’s Wrong With Concurrency Control and 2PC

- **Concurrency Control:**
  Nasty bugs: if aborts aren’t handled properly by the application, effects of aborted transactions leave effects on program variables. These bugs are not discovered until production.

  Performance: bottlenecks on shared resources are frequent. They require radical schema changes, e.g. hash the processid into a special field, version fields.

- **Two Phase Commit:**
  Customers don’t like the message overhead of two phase commit, particularly in WAN environments, so they settle for replication servers which give warm, not hot, backups.
An Example Alternative

- Main memory database, logically single threaded, operation logging, full replication. Net result: a replicated state machine whose commits are uncoordinated.

- Operations are logged and acked in batches, so much less than one overhead message per transaction. Each site dumps its state in a round robin fashion.
Large Databases: what is needed

• Since the algorithm executes transactions sequentially, big databases (i.e., too big for RAM) can’t take advantage of disk bandwidth. Like life before operating systems.

• Executing in order of arrival is too slow, but want to appear to do so.

• NB: Serializability at each replicated site is not enough. Do you see why?
Algorithm OBEYORDER

1. Construct a predicate called CONFLICT that takes two transaction instances and determines whether they would conflict.

2. If t CONFLICTS with t’ and t has an earlier arrival number than t’, then form a directed edge (t, t’). This produces a graph $G = (T, E)$ where $T$ is the set of all the transactions in the batch (or batches) and $E$ is the set of directed edges formed as described.

3. Execute $T$ in parallel, respecting the order implied by $E$. 
Issues with OBEYORDER

- Need a CONFLICT predicate. (Can be difficult to write.)

- If there are many conflicts, must do more. Observe: can always prefetch data provided it is globally visible.

- The old standby: if data can be partitioned, buy another processor.
Summary

• In-memory databases can use a different approach from on-disk databases (just ask TimesTen). No concurrency control, operation recovery, and hot backups.

• If data spills over to disk, then you need to invent a new concurrency control scheme.

• You can get transactional guarantees plus hot backup, all with low overhead.
Principles of Logging and Recovery

Motivation: Hardware and software sometimes fail. Normal programs just restart. But data may be corrupted.

Example: Money transaction fails after adding money to cash, but before subtracting value of item from inventory. Accounts unbalanced.

Recovery avoids incorrect states by ensuring that the system can produce a database that reflects only successfully completed transactions.
Assumptions

What is a failure?

Innocent question, huge effect on algorithms.

Traitorous failure – failed components continue to run, but perform incorrect (perhaps malicious) actions.

Clean failure – when a site fails, it stops running. (Mimics hardware error on fail-stop processors.)

Soft clean failure – contents of main memory are lost, but secondary memory (disks and tapes) remain. Secondary memory called stable storage.
Our Assumptions

• Soft clean failure.  
  Protect secondary memory using disk mirroring or redundant arrays of disks. 
  Paranoia must be no deeper than your pocket!

• Atomic write – Can write a single page to disk in an all or nothing manner. 
  Use checksums to see whether a write succeeded.
Database Model

Database — set of data items in stable storage. Each data item is a page.

Audit trail — set of data items in stable storage. Each data item is a page. (Scratch space)

Operations:

Read — read a page from database.

Write — write a page to stable storage.

Commit — indicate that transaction has terminated and all updated pages should be permanently reflected in the database.

Abort — indicate that no updates done by transaction should be reflected in the database.
States of a Transaction

Active — issued neither abort nor commit.

Aborted — issued abort.

Committed — issued commit.

Note: a transaction may not be both committed and aborted.

Objective of recovery: Ensure that after a failure, can reconstruct the database so it has updates from committed transactions only.
Strategy: Keep around redundant information

The *before-image* of \( x \) with respect to a transaction \( T \) is the value of \( x \) just before the first write of \( T \) on \( x \) occurs.

The *after-image* of \( x \) with respect to a transaction \( T \) is the value of \( x \) just after the last write of \( T \) on \( x \) occurs.

Want to keep around before-images until commit and want after-images in stable storage by commit.
Logging Rules

1. Log-ahead rule: Ensure that before-images of all pages updated by T are in stable storage at least until T commits. Allows system to recreate state before T began.

2. Commit rule: When T commits, have after-images of all pages updated by T somewhere in stable storage. Allows system to create a database state reflecting T's updates.

Rules + atomicity of commit = recoverability
Algorithmic Taxonomy

Undo strategy — Some of T’s writes may go to the database before T commits. If T aborts, the system restores the database state to one excluding the updates caused by T.

Redo strategy — Some of T’s writes may not go to the database before T commits. Sometime after T commits, the system transfers T’s updates from the audit trail or the buffer to the database.

Possibilities:

1. No Undo, No Redo
2. No Undo, Redo
3. Undo, No Redo
4. Undo, Redo

All four have been implemented.
No Undo, Redo

Description: Each after-image for $T$ is written to audit trail (i.e. log) sometime before commit time. Satisfies commit rule.

Before-images not touched in database. Satisfies log-ahead rule.

Commit step consists of writing “commit $T$” in the audit trail atomically (i.e., on a single page).

Recovery from System failure — transfer pages from audit trail to database of committed transactions.

Abort a transaction — erase its audit trail record.
No Undo, Redo — Issues

Question: When exactly to write committed pages to database? (At commit or later.)

Issues: size of physical memory; disk head movement.
Undo, No Redo

Description: Transaction first transfers before-image of each page to audit trail, then puts the after-image in the database.

Commit — write a commit page to audit trail.

Abort a transaction — write all before-images of pages updated by the transaction into database.

Recovery from System failure — abort all uncommitted transactions.

Issue: Requires forcing pages to database disks while transaction executes. Bad for performance.
Redo and Undo (Writeahead log)

Description: T doesn’t touch the database disks for short transactions, but may write after-images to the database for long update transactions. In such a case, it writes the before-image to the audit trail first.

Commit — write commit record to audit trail.

Abort — Transfer necessary before-images back to the database.

Recovery from System failure — abort all uncommitted transactions.

Evaluation: requires more I/O than the above schemes, but that I/O is all to the audit trail. Most freedom for buffer manager.
No Redo, No Undo

Assume we have a directory on one page that points to every data item. (Otherwise, create a data structure with one page at the root.)

Description: Duplicate the directory. Call one real and the other shadow. Record transaction T’s updates on the shadow directory, but don’t change the items pointed to by the real directory.

Commit consists of making the shadow directory the real one and making the real one the shadow directory.

Abort and recovery require no work.

Recovery and Concurrency Control

The two are conceptually independent.

Reason: When scheduling algorithm allows a transaction to commit, it is saying that T’s updates may become permanent. Recovery merely ensures that the updates become permanent in a safe way.

Example: Two Phase Locking + Redo, No undo.
2PL + Redo, No Undo

Transaction acquires locks as it needs them and doesn’t release them until commit (in primitive versions, until writes are written to the database disks). All writes during transaction are to audit trail.

If transaction aborts, system releases locks and frees pages of audit trail.

Try Optimistic + Undo, No Redo?
Distributed Commit Protocols

Scenario: Transaction manager (representing user) communicates with several database servers.

Main problem is to make the commit atomic.

Naive approach: Transaction manager asks first server whether it can commit. It says yes. Transaction manager tells it to commit. Transaction manager asks next server whether it can commit. It says no. Transaction manager tells it to abort.

Result: partially committed transaction. Should have aborted.
Solution

Two phase commit:

1) Transaction manager asks all servers whether they can commit.

1b) Upon receipt, each able server saves all updates to stable storage and responds yes.

If server cannot say yes (e.g. because of a concurrency control problem), then it says no. In that case, it can immediately forget the data.

2) If all say yes then transaction manager tells them all to commit. Otherwise, (some say no or don’t respond after a time) transaction manager tells them all to abort.

2b) Upon receipt, the server writes the commit record.
Failure Scenarios

If a database server fails during first step, all abort.

If a database server fails during second step, it can see when it recovers that it must commit the transaction.

If transaction manager fails during second step, then the servers who haven’t received commit either must wait or must ask other servers what they should do.
Performance Enhancements

1. Read-only transaction optimization. Suppose a given server has only done reads (no updates) for a transaction. Instead of responding to the transaction manager that it can commit, it responds READ-only.

The transaction manager can thereby avoid sending that server a commit message.

2. Eager server optimization. If the protocol dictates that each server will receive at most one message from each transaction, then the server can precommit after completing its work and inform the transaction manager that it is prepared to commit. The transaction manager then does the second phase.
On Recovery

A site can play the role of a transaction manager and a server.

Transaction manager role: Site checks its commit log (this is a stable log of all the ”commit T” commands that it has written). It then checks all the server logs and resends commits for any transaction that has been committed at the site but has not been committed at the server.

Server role: Site checks its logs for transactions that have been precommitted only. It then asks the manager of that transaction what to do. It can also ask the other servers what to do, if the other servers don’t forget their transaction information.
Simple, Rational Guidance for Chopping Up Transactions

or

How to Get Serializability Without Paying For It

Dennis Shasha
Eric Simon
Patrick Valduriez
Outline

• Motivation

• Critical Assumption

• Example

• Sufficient Conditions

• Chopping Optimization

• Algorithm and Analysis

• Applications
Motivation

- Many proposals for concurrency control methods.
  Aimed at designers.

- Practitioners are stuck with two phase locking. Their only tuning knobs are
  - chop transactions into smaller pieces
  - choose degrees 1 or degree 2 isolation.
Critical Assumption

Environment in which we know the transaction mix (e.g., real-time or on-line transaction processing).

That is, no unpredictable, ad hoc queries.
Purchase Transaction — 1

Purchase:
add value of item to inventory;
subtract money from cash.

Constraint: cash should never be negative.
Purchase Transaction — 2

Application programmers chop as follows:

1. First transaction checks to see whether there is enough cash. If so, add value of item to inventory. Otherwise, abort the purchase.

2. The second transaction subtracts the value of the item from cash.

Cash sometimes becomes negative. Why?
Purchase Application — 3

By contrast, if each numbered statement is a transaction, then following bad execution can occur. Cash is $100 initially.

1. P1 checks that cash > 50. It is.

2. P2 checks that cash > 75. It is.


4. P2 completes. Cash = −25
No surprise to your university professor, who says something like:

You Idiot! You should never have chopped this transaction up!

Why did I pass you from my course anyway?
Purchase Transaction — 4

Surprise: Simple variant guarantees that cash will never become negative.

1. First transaction checks to see whether there is enough cash. If so, subtract cash. Otherwise, abort the purchase.

2. The second transaction adds value of item to inventory.

Goal of research: Find out why this works!
Special Recovery Hacks

Must keep track of which transaction piece has completed in case of a failure.

Suppose each user X has a table UserX.

- As part of first piece, perform insert into UserX (i, p, 'piece 1'), where i is the inventory item and p is the price.

- As part of second piece, perform insert into UserX(i, p, 'piece 2').

Recovery includes reexecuting the second pieces of inventory transactions whose first pieces have finished.
Assumptions

• Possible to characterize all transactions during some interval.

• Want serializability for original transactions.

• On failure, possible to determine which transactions completed and which did not.
Chopping

For simplicity, assume sequential transactions.

A chopping is a partition of the transaction into pieces such that the first piece has all rollback statements.

Each piece will execute using two phase locking (if it aborts, execute again).
Graphical Characterization

Chopping graph — Undirected graph whose nodes are pieces. Two kinds of labeled edges.

1. Conflicts: C edge between p and p’ if the two pieces come from different transactions and issue conflicting instructions.

2. Siblings: S edge between p and p’ if they come from the same original transaction.

Note: no edge can have both an S and a C label.
Correctness

A chopping of a set of transactions is *correct* if any execution of the chopping is equivalent to some serial execution of the original transactions.

Equivalent = every read returns same value in two executions and writes write same value.
Sufficient Conditions for Correctness

SC-cycle — a simple cycle that includes at least one S edge and at least one C edge.

Theorem 1: A chopping is correct if its chopping graph contains no SC-cycle.
Proof of theorem 1

Suppose there were a cycle in the serialization graph of original transactions. 
\[ T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1. \]
\( T_i \rightarrow T_j \) means \( T_i \) issues op that conflicts with and precedes an op in \( T_j \).

Identify pieces associated with each transaction that are involved in this cycle:
\[ p \rightarrow p' \rightarrow \ldots \rightarrow p'' \]
Both \( p \) and \( p'' \) belong to transaction \( T_1 \).

If every arrow corresponded to a C edge, then \( p \neq p'' \) since each piece uses two phase locking so serialization graph on pieces is acyclic. Otherwise \( p = p'' \) is possible, but then every other edge would be either a C edge of an S edge. So, cycle among original transactions implies SC-cycle. Contradiction.
Purchase Example

Original transactions:

\( P(i,p) : \)
1. if \( \text{cash} > p \) then \( \text{invent}(i) += p \);
2. \( \text{cash} -= p \);

If we chop \( P(i,p) \) into two transactions, we’ll get an SC-cycle.
Purchase Variant

Original transactions:

P(i,p):
1. if cash > p then cash -= p;
2. invent(i) += p;

Chopping P(i,p) does not introduce an SC-cycle.
CHOPPING EXAMPLE

Purchase transaction (price, item)

1. read(cash); If cash > price then
   read(inventory[item]); inventory[item] += price;
   write(inventory[item]);

2. read(cash); cash -= price;
   write(cash);

The chopping graph has SC-cycles

Fig.C.1
CHOPPING EXAMPLE

MODIFIED PURCHASE TRANSACTION (price, item)

1. read (cash); if cash > price then
   read (cash); cash -= price;

2. read(inventory[item]; inventory[item] += price;
   write(inventory[item]);

There is no SC-cycle in the chopping graph

Fig. C.2
Example 1

Original three transactions:

T1: R(x) W(x) R(y) W(y)
T2: R(x) W(x)
T3: R(y) W(y)
Example 1 — Chop T1

T11: \( R(x) \ W(x) \)
T12: \( R(y) \ W(y) \)

No cycle.
Example 1 — too much chopping

Break up T11 further into

T111: R(x)
T112: W(x)

will result in an SC-cycle.
Optimization

Question: Does a finest chopping exist?

Answer: yes.

Key Observation: If \( T \) is chopped and is in an SC-cycle with respect to \( T' \), then chopping \( T' \) further or gluing the pieces of \( T' \) together will not eliminate that cycle.

Moral: if chopping \( T \) causes a cycle, then nothing you do to other transactions can help.
Reasons

Suppose we break $p$ of $T'$ into $p_1$ and $p_2$. If $p$ is in a cycle, then $p_1$ will have an $S$ edge to $p_2$, so at most the cycle will be lengthened.

Suppose we combine two pieces $p_1$ and $p_2$ of $T'$ into piece $p$. If $p_1$ alone had been in a cycle, then so will $p$. If cycle went through $S$ edge between $p_1$ and $p_2$, then cycle will just be shorter.
Systematic Method to Obtain Finest Chopping

Original set of transactions: $T_1, T_2, ..., T_n$.

- For each transaction $T_i$, $F_i = \text{chop } T_i$ as finely as possible with respect to the other (unchopped) transactions.

- Finest chopping is $F_1, F_2, ..., F_n$.

Algorithm is connected components algorithm in $C$ graph for each $T_i$. Complexity is $O(n \times (e + m))$, where $e$ is the number of $C$ edges in the transaction graph and $m$ is max number of database accesses.
Putting the three pieces of T3 into one will not make the chopping of T1 OK. Nor will chopping T3 further.
Application to Typical Database Systems

SQL system with bind variables, e.g. update salary of employee :x.

Determining conflicts is difficult. Can use predicate locking,
AND name LIKE 'T%'
vs.
AND name LIKE 'S%

Our new idea: number of conflicts is significant.
Example Application

Suppose a single query of form:

SELECT ...
FROM account

is concurrent with updates of the form:

Update ...
FROM account
WHERE acctnum = :x

If acctnum is a key, then conflict on only one record.

Can run at degree 2 isolation. (Or could chop if all updates in first query were modifies.)
Reason: no SC-cycle because Replace is on a key.

Fig. C.4
Related Work

A lot of work that seeks to reduce concurrency control constraints, mostly through new algorithms or weakening of isolation guarantees.

Aimed at implementors of DBMS’s.

Our work aims at users.

Farrag and Ozsu — application knowledge of application, e.g. hotel reservations.

Garcia-Molina — partition transactions into classes. In same class run concurrently, whereas synchronize among classes. Weaker notions of consistency by using counterstep transactions if something bad happens.

Lynch — nested classes. Ensure a specific order among conflict steps.
Related Work — 2

Bayer — new concurrency control and recovery mechanism to allow a single batch transaction to run among many short transactions.

Oracle and Gemstone use similar scheme for long readers.

Hsu and Chan — special concurrency control algorithms for situations in which data is divided into raw data and derived data. Consistency of the raw data is not important.

O’Neil — exploits commutativity of increments to release write locks early.
Related Work — 3

Wolfson — looks at early release of locks when user has complete control over acquisition and release of locks.

Bernstein, Shipman and Rothnie — introduced conflict graph in SDD-1 context.

Casanova — generalized that notion to include program flow.

Shasha and Snir — further generalized to include atomicity constraints in access of parallel shared memory. SC-graphs are a special case.
Future Work

Have:

simple, efficient algorithm to partition transactions into the smallest pieces possible when transaction mix is known.

Open:

• How to extend to sagas (undo transactions), tree-locking, multi-level concurrency control?

• Suppose a given set of transactions do not chop well. Can one partition the set into several subsets, execute each subset in its own subinterval, and thereby achieve a good chopping?