Advanced Database Systems:

transactions, database tuning, and advanced topics

Dennis Shasha, shasha@cs.nyu.edu

©2023 Dennis Shasha

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:

Employee Number	Balance
101	70
106	60
121	80
132	10

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 = throughput, usually.
- Performance = 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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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 $30000.)

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

Alice sees only about $20,000, accuses Bob of gambling, and gets a divorce.

Reason: Execution was not "serializable."

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 at least 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, $T1 \longrightarrow T2$ because of checking and $T2 \longrightarrow T1$ because of savings. Hence schedule is not serializable.

©2023 Dennis Shasha

Schedule Equivalence

We say Ri(x) reads-from Wj(x) if Wj(x) precedes Ri(x) and there is no intervening Wk(x).

We say Wi(x) is a final-write if no Wk(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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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: $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$. Then T_1 accesses an item x1 before T_2 accesses x1 and the accesses conflict (at least one writes x1). So T_1 must release its lock after lockpoint(T_1) and before T_2 acquires its lock on x1, which is before lockpoint(T_2). Therefore lockpoint(T_1) precedes lockpoint(T_2). Similarly, there is an x2 such that T_2 accesses x2 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.

©2023 Dennis Shasha

Read locks

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

R1(x) R2(x) R2(y) R1(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.

©2023 Dennis Shasha

Deadlock Detection

Construct a *blocking (waits-for)* graph. T $\longrightarrow$ T' if T needs a lock on item x, T' has a conflicting lock on x or T' is ahead of T on the wait queue for x and T' seeks a conflicting lock on x.

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

Example: T1 $\longrightarrow$ T2, T2 $\longrightarrow$ T4, T4 $\longrightarrow$ T5, T4 $\longrightarrow$ T3, T3 $\longrightarrow$ T1

Abort one of T1, T2, T3, T4.

Cycle detection need not happen frequently. Deadlock doesn't go away.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

Deadlock Avoidance Protocols

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

Use two phase locking.

Rosenkrantz, Stearns, Lewis, ACM Transactions on Database Systems 1978 (nice shelf life)

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 keep deadlocks as short a time as possible.

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

Optimistic Protocols (certifying)

Kung and Robinson, "On optimistic methods for concurrency control," proc. 1979 international conference on very large databases.

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 st T_i has read x}

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

When receiving endi, certifier does the following:

RS(active) = $\cup$ RS(j) for T_j active but j $\neq$ i

WS(active) = $\cup$ WS(j) for T_j active but j $\neq$ i

if RS(i) $\cap$ WS(active) = ϕ
and WS(i) $\cap$ (RS(active) $\cup$ WS(active)) = ϕ
then certify (commit)
else abort

©2023 Dennis Shasha

Verifying Optimisitic 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: Ti writes x before Tj accesses x and Ti reads x before Tj writes x.) This relation must be acyclic.

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.

Benefits of Multiversion Protocol

Suppose initial values are x0 and y0.

R1(x) W2(x) W2(y) R1(y)

is clearly not serializable on a single copy. But what if T1 read the values of x and y present when T1 began (i.e. read x0 and y0). Then it appears as if T1 went completely before T2 without any read locks.

Multiversion read consistency applies only to read-only transactions. Read-write transactions must obtain read locks and read the latest version of each data item.

Why Can't We Apply Multiversion Read Consistency to Read-Write Transactions?

Any reads on data item x by transaction T (even when T includes writes) reads the latest version of x that was committed when T began

Writes acquire (exclusive) locks in a normal two phased manner.

T1: x := y

T2: y := x

x0 = 3

y0 = 17

R1(y) R2(x) W1(x, 17) W2(y, 3)

But what would happen in any serial execution?

Very similar to Snapshot Isolation.

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 transaction 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 a two phase locked transaction T, T tests whether all servers that T accessed (read or write) have been up since the first time T accessed them. If not, T aborts. (Note: Read-only transactions using multiversion read consistency 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.

In practice, a mixture of the two.

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) // e.g., y:= x

T_2: R(y) W(x) // e.g., x:= y

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

Here we have a computation that uses two phase locking and executes till the end, 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.

©2023 Dennis Shasha

Weikum's technique

- Research group at ETH Zurich led by Gerhard Weikum looked for automatic load control techniques to maximize throughput.

  They observe that higher multiprogramming levels may make better use of resources but may also increase 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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

Taxonomy of Workflow Applications

- Route work among people in a good way, but have a way to back out of sagas.

- The general category is business process management. Here is an example of a process (multi-step and possibly multiple databases):

  https://www.stakeholdermap.com/bpm/busines
  process-model-payment-process.html

Simple, Rational Guidance for Chopping Up Transactions

or

How to Get Serializability Without Paying For It

Dennis Shasha
Eric Simon
Patrick Valduriez

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.

©2023 Dennis Shasha

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.

3. P1 completes. Cash $= 50$.

4. P2 completes. Cash $= -25$

©2023 Dennis Shasha

Purchase Transaction — 3

No surprise to you, because you know which choppings are correct and which aren't. You say something like:

You fool! You should never have chopped this transaction up in that way!

How did you never learn about chopping correctly?

©2023 Dennis Shasha

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!

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

Proof of theorem 1

Suppose there were a cycle in the serialization graph of original transactions.
T1 $\rightarrow$ T2 $\rightarrow$ ... $\rightarrow$ Tn $\rightarrow$ T1.
Ti $\rightarrow$ Tj means Ti issues op that conflicts with and precedes an op in Tj.

Identify pieces associated with each transaction that are involved in this cycle:
p $\rightarrow$ p' $\rightarrow$ ... $\rightarrow$ p''
Both p and p'' belong to transaction T1. The edges correspond to either S or C edges.

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 or an S edge and there must be at least one S edge. So, cycle among original transactions implies SC-cycle. Contradiction.

©2023 Dennis Shasha

Purchase Example

Original transactions:

P(i,p):
1. if cash $>$ p then invent(i) $+=$ p;
2. 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, because while there are conflict edges between the 1 pieces, there are none between the 2 pieces (increments are commutative).

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 p1 and p2.
If p is in a cycle, then p1 will have an S edge to p2, so at most the cycle will be lengthened.

Suppose we combine two pieces p1 and p2 of T' into piece p.
If p1 alone had been in a cycle, then so will p. If cycle went through S edge between p1 and p2, then cycle will just be shorter.

©2023 Dennis Shasha

Systematic Method to Obtain Finest Chopping

Original set of transactions: T1, T2, ..., Tn.

- For each transaction Ti, Fi = chop Ti as finely as possible with respect to the other (unchopped) transactions.
- Finest chopping is F1, F2, ..., Fn.

Algorithm is connected components algorithm in C graph for each Ti. 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.

©2023 Dennis Shasha

Why Connected Components?

Consider Ti. When chopping Ti with respect all other transactions, we might have something like:

Ri(x) Wi(y) Ri(z) Wi(x) Wi(q) Wi(z)

Now, say that other transactions conflict on y and x. If the connected component algorithm connects Wi(y) and Wi(x), then they must be in the same piece. Otherwise, there would be an SC cycle (S between Wi(y) and Wi(x) and C edges along the rest of the connected components algorithm with the unchopped other transactions).

©2023 Dennis Shasha

Example Application

Suppose a single query of form:

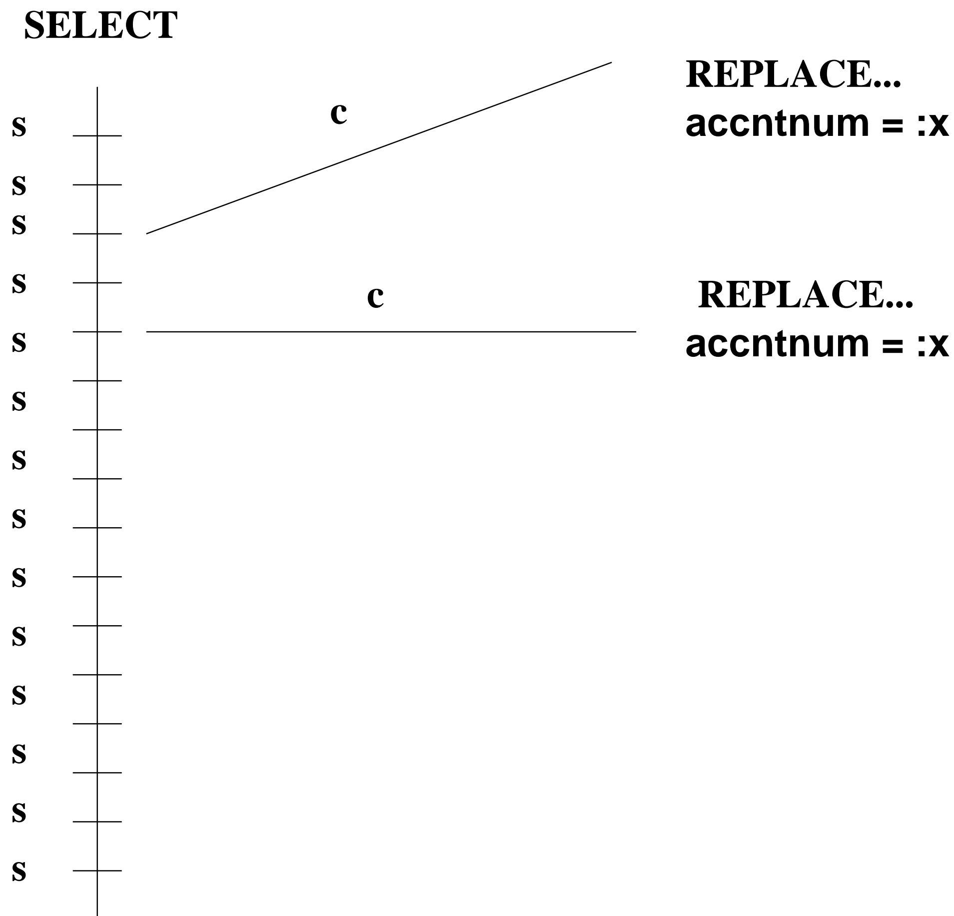
SELECT ...
FROM account

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.)



Degree 2 isolation has the effect of chopping the scan so each record access is a single transaction.
In this case, degree 2 isolation is as good as degree 3.

Reason: no SC-cycle because Replace is on a key.

Fig. C.4

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?

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

Evaluation: disrupts secondary memory layout. Very fast recovery.

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

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.

Throw Away Concurrency Control and Two Phase Commit

Arthur Whitney

KX Systems, arthur@kx.com

Dennis Shasha

New York University, shasha@cs.nyu.edu

Steve Apter

©2023 Dennis Shasha

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.

©2023 Dennis Shasha

Summary

- In-memory databases can use a different approach from on-disk databases. 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.

Lessons from Wall Street: case studies in configuration, tuning, and distribution

Dennis Shasha

Courant Institute of Mathematical Sciences

Department of Computer Science

New York University

shasha@cs.nyu.edu

http://cs.nyu.edu/cs/faculty/shasha/index.html

occasional database tuning consultant on Wall Street

©2023 Dennis Shasha

Wall Street Social Environment

- Very secretive, but everyone knows everything anyway (because people move and brag).

- Computers are cheap compared to people. e.g., terrabytes of RAM is a common configuration for a server and will grow.

- Two currencies: money and fury.

Wall Street Technical Environment

- Analytical groups use APL (or q or kdb) or Matlab or Excel with extensions to value financial instruments: bonds, derivatives, and so on. These are the "rocket scientists" because they use continuous mathematics and probability (e.g. Wiener processes).

- Mid-office (trading blotter) systems use relational systems. These maintain positions and prices. Must be fast to satisfy highly charged traders and to avoid arbitrage (delays can result in inconsistencies).

- Backoffice databases handle final clearance.

©2023 Dennis Shasha

Overview

- Configuration — disaster-proof systems, interoperability among different languages and different databases.

- Global Systems — semantic replication, rotating ownership, chopping batches.

- Tuning — clustering, concurrency, and hashing; forcing plans.

©2023 Dennis Shasha

Preparing for Disaster

- Far from the simple model of stable storage that we sometimes teach, though the principles still apply.

- Memory fails, disks fail (in batches), fires happen (Credit Lyonnais, our little terrorist incident), and power grids fail. If your system is still alive, you have a big advantage.

- You can even let your competitors use your facilities ... for a price.

©2023 Dennis Shasha

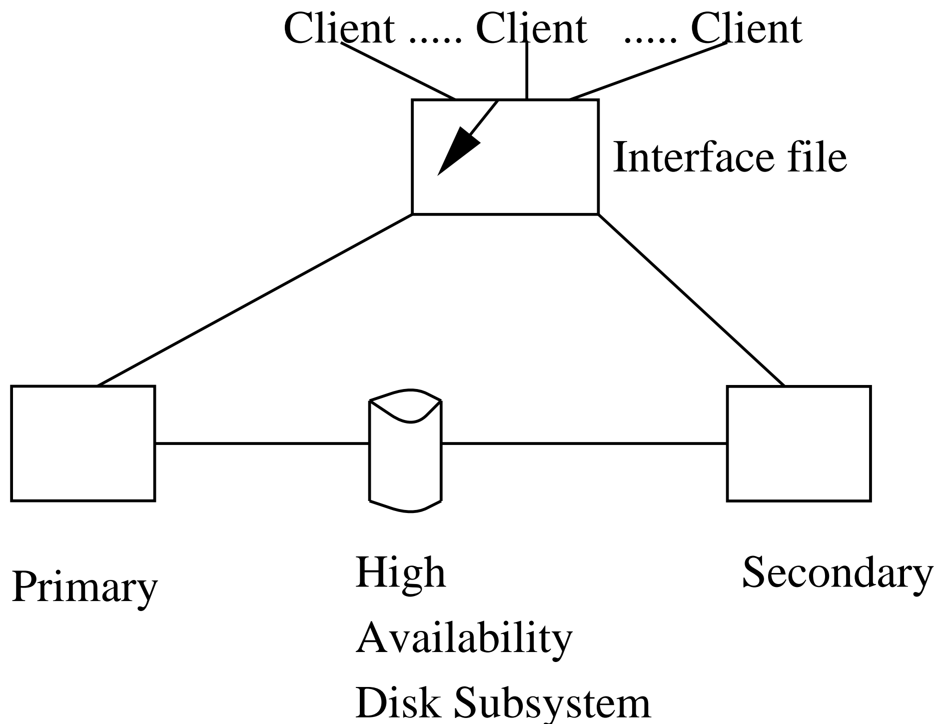
Case: Bond Futures

- Server for trading bond futures having to do with home mortgages.
- Application used only a few days per month, but the load is heavy. During a weekend batch run, 11 out of 12 disks from a single vendor-batch failed.

High Availability Servers

- A pair of shared memory multiprocessors attached to RAID disks.
- If the primary multiprocessor fails, the backup does a warm start from the disks.
- If a disk fails, RAID masks it.
- Does not survive disasters or correlated failures.

©2023 Dennis Shasha



Writes go to the primary and into the high availability
disk subsystem. This subsystem is normally a RAID device,
so can survive one or more disk failures.
If the primary fails, the secondary works off the same
disk image (warm start recovery).

Vulnerability: High availability disk subsystem fails
entirely.

©2023 Dennis Shasha

Dump and Load

- Full dump at night. Incremental dumps every three minutes.

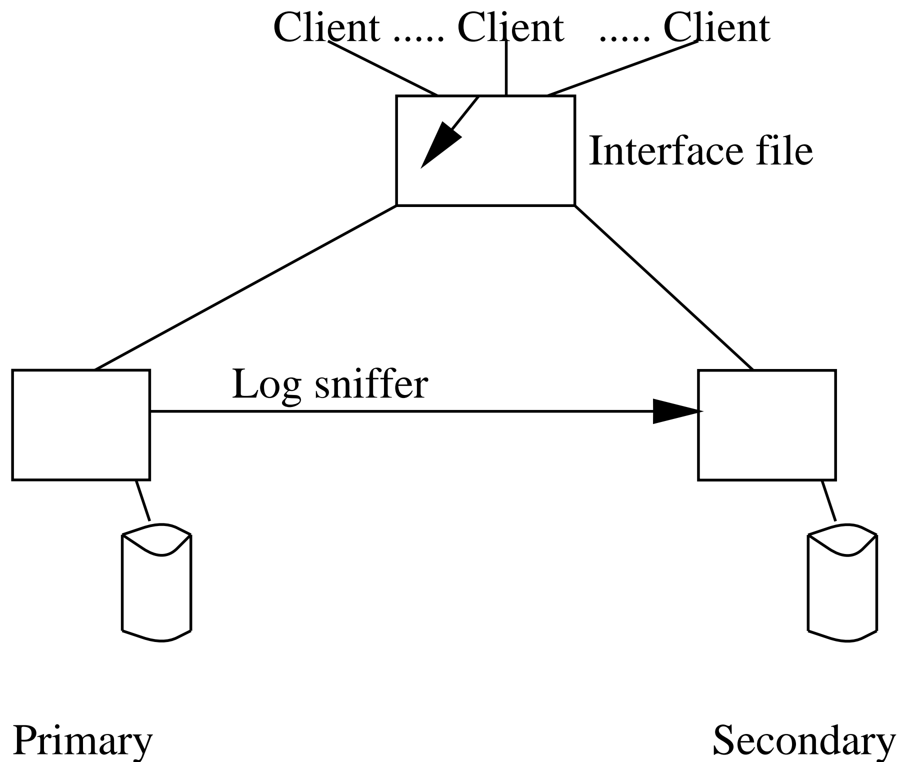
- Can lose committed transactions, but there is usually a paper trail.

- Backup can be far away.

Replication Server

- Full dump nightly. All operations at the primary are sent to the secondary after commit on the primary.

- May lose a few seconds of committed transactions.



Basic architecture of a replication server.
The backup reads operations after they are committed
on the primary. Upon failure, the secondary becomes the
primary by changing the interface file configuration variables.

Vulnerability: if there is a failure of the primary after commit
at the primary but before the data reaches the secondary, we
have trouble.

Remote Mirroring

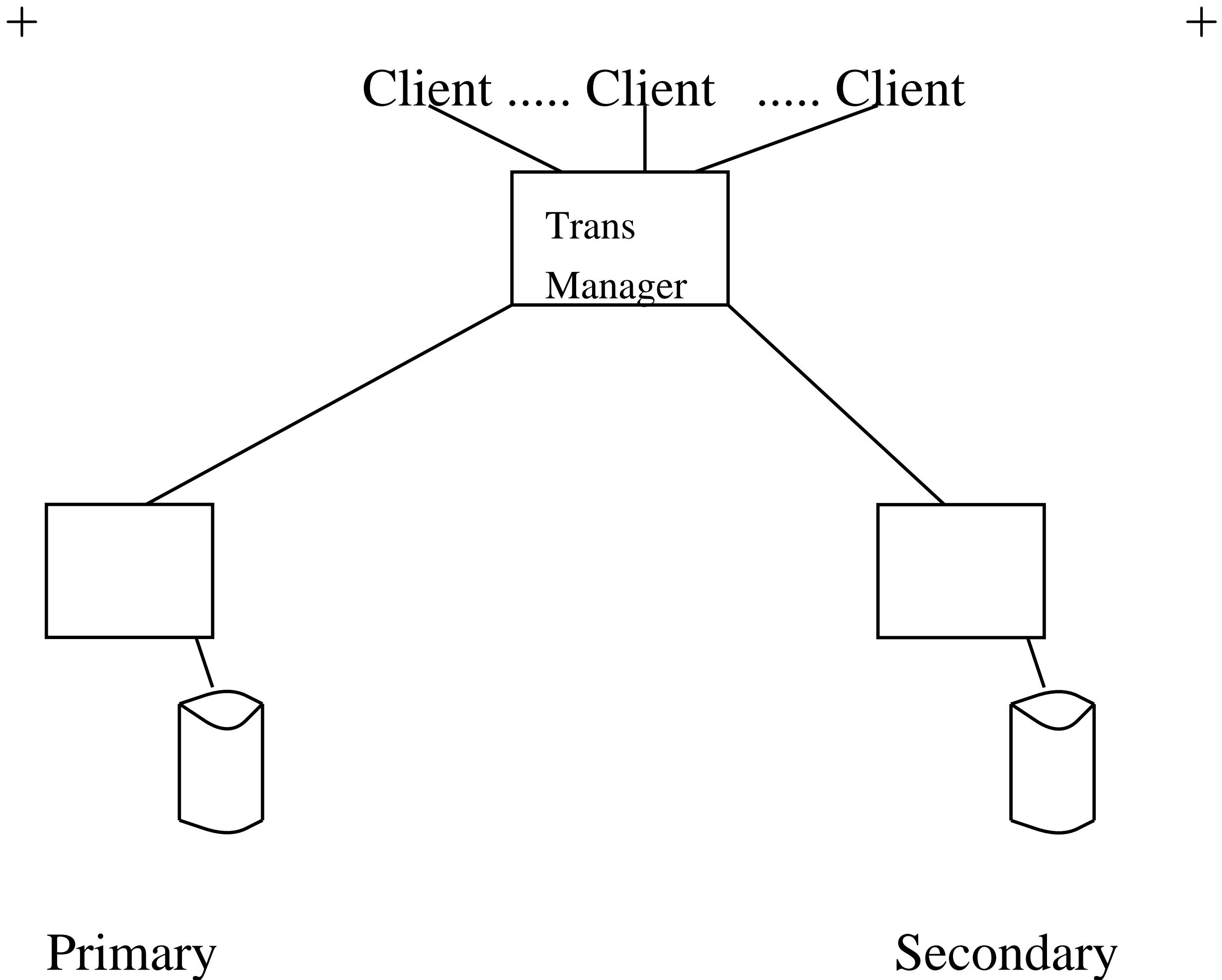
- Writes to local disks are mirrored to disks on a remote site. The commit at the local machine is delayed until the remote disks respond.

- Backup problems may cause primary to halt.

- Reliable buffering can be used (e.g. Qualix), but the net result is rep server without the ability to query the backup.

Two Phase Commit

- Commits are coordinated between the primary and backup.
- Blocking can occur if the transaction monitor fails. Delays occur if backup is slow.
- Wall Street is scared of this though less and less.



Two phase commit: transaction manager ensures that updates on the primary and secondary are commit-consistent. This ensures that the two sides are in synchrony.

Vulnerability: blocking or long delays may occur at the primary either due to delays at the secondary (in voting) or failure of the transaction manager.

Quorum Approach

- Servers are co-equal and are interconnected via a highly redundant wide area cable.

- Clients can be connected to any server. Servers coordinate via a distributed lock manager.

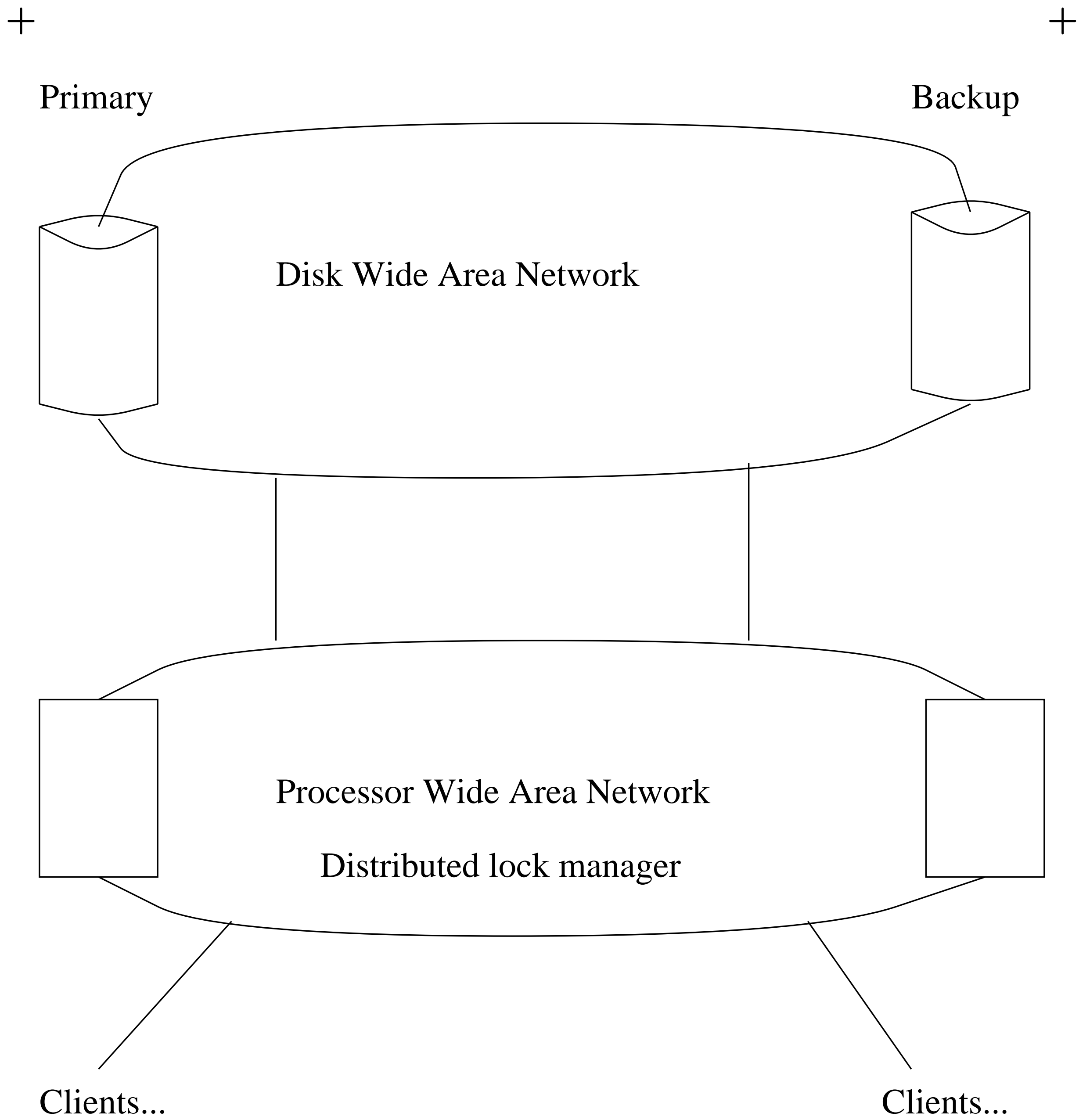
- Disks are connected with the servers at several points and to one another by a second wide area link.

Heartbeats

- Heartbeats monitor the connectivity among the various disks and processors.

- If a break is detected, one partition holding a majority of votes continues to execute.

- Any single failure of a processor, disk, site, or network is invisible to the end users (except for a loss in performance).



Quorum Approach as Used in most Stock and Currency Exchanges.
Survives Processor, Disk, and Site failures.

Quorum approach used in most exchanges.

The Need for Globalization

- Stocks, bonds, and currencies are traded nearly 24 hours per day (there is a small window between the time New York closes and Tokyo opens).

- Solution 1: centralized database that traders can access from anywhere in the world via a high-speed interconnect.

- Works well across the Atlantic, but is very expensive across the Pacific. Need local writes everywhere in case of network partition.

Distributed Solution

- Two phase commit worries users because of blocking and delays. Replication can result in race condition/anomalies. (e.g. Gray et al. Sigmod 96).
- Sometimes, application semantics helps.

Case: Options traders

- A trading group has traders in 8 locations accessing 6 Sybase servers. Access is 90% local.

- Exchange rate data, however, is stored centrally in London. Rate data is read frequently but updated seldom (about 100 updates per day).

- For traders outside of London, getting exchange rates is slow.
  Can we replicate the rates?

Consistency Requirements

- If a trader in city X changes a rate and then runs a calculation, the calculation should reflect the new rate (So, can't update London and wait for replication.)

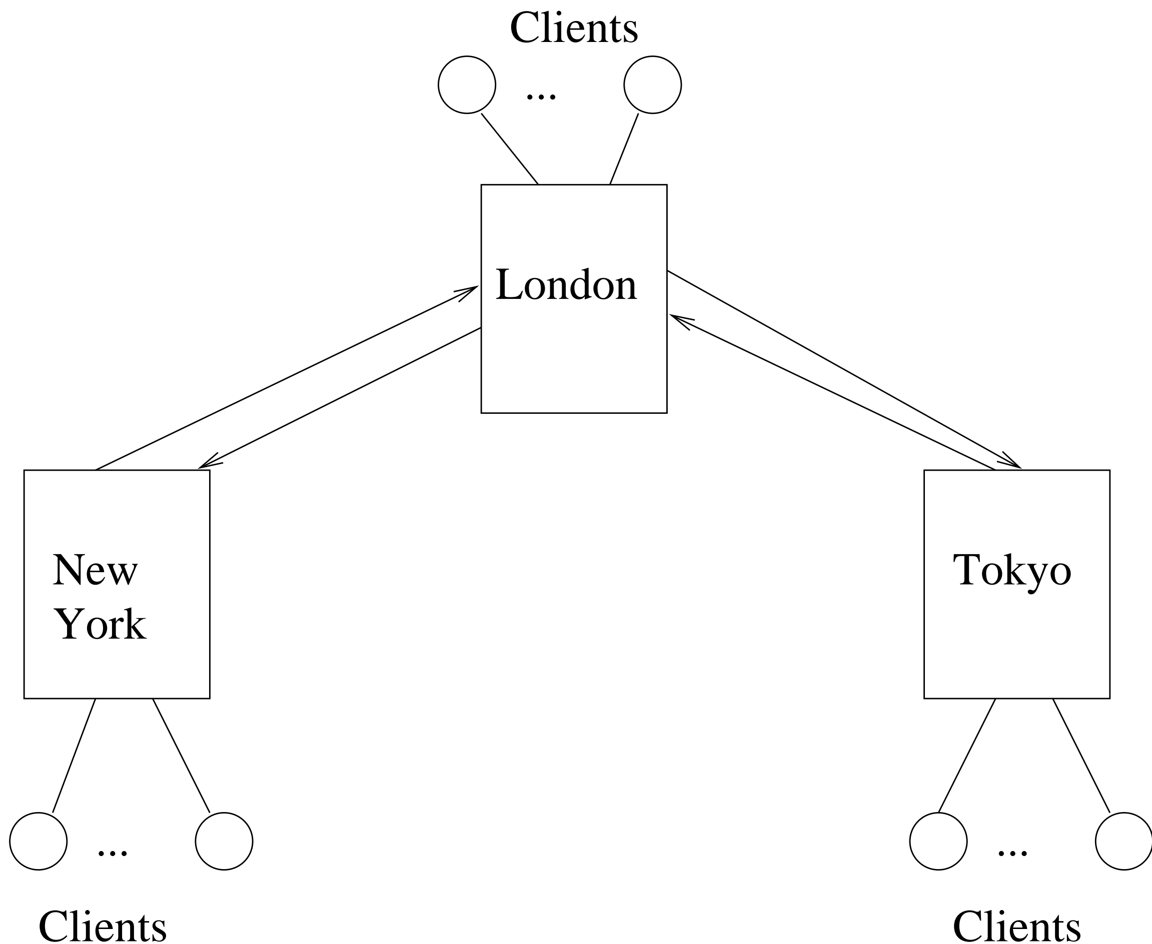
- All sites must agree on a new exchange rate after a short time (must converge). (So, can't use vanilla replication server.)

Clock-based Replication

- Synchronize the clocks at the different sites. (Use a common time server.)

- Attach a timestamp to each update of an exchange rate.

- Put a database of exchange rates at each site. An update will be accepted at a database if and only if the timestamp of the update is greater than the timestamp of the exchange rate in that database.



Clients send rates to local machines where they take immediate effect.
Rates and timestamps flow from one server to the other.
Latest timestamp does the update.
Ensures: convergence and primacy of latest knowledge.

Timestamped Replication

©2023 Dennis Shasha

Case: Security Baskets

- Trade data is mostly local, but periodically traders collect baskets of securities from multiple sites.
- The quantity available of each security must be known with precision.
- The current implementation consists of an index that maps each security to its home database. Each site retrieves necessary data from the home site.

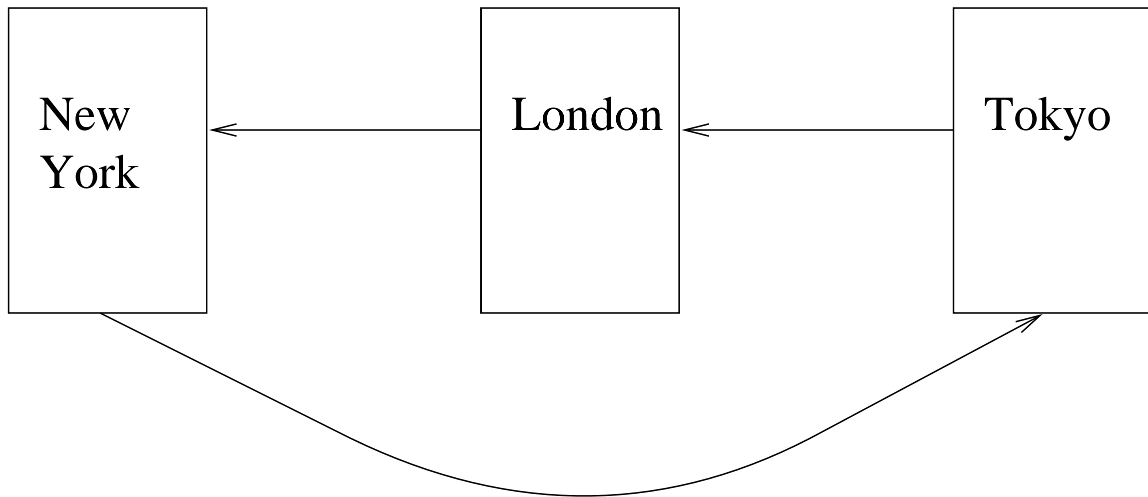
Rotating Ownership

- Maintain a full copy of all data at all sites.
- Not all of this data will be up-to-date ("valid") at all times however. Can be used for approximate baskets.
- When a market closes, all its trades for the day will be sent to all other sites. When receiving these updates, a site will apply them to its local database and declare the securities concerned to be "valid."

Rotation Issues

- Receiving ownership must be trigger-driven rather than time-driven.

- Suppose New York assumes it inherits ownership from London at 11 AM New York time. If the connection is down when London loses its ownership, then some updates that London did might be lost.



Ownership travels from east to west as exchanges close.
A given exchange should assert ownership only after
it is sure that the previous exchange has processed all
trades.

Rotating Ownership

Case: Batch and Global Trading

- When the trading day is over, there are many operations that must be done to move trades to the backoffice, to clear out positions that have fallen to zero and so on. Call it “rollover.”

- Straightforward provided no trades are hitting the database at the same time.

- In a global trading situation, however, rollover in New York may interfere with trading in Tokyo.

©2023 Dennis Shasha

Chop the batch

- "Chop" the rollover transaction into smaller ones.

- The conditions for chopping are that the ongoing trades should not create cycles with the rollover pieces.

- New trades don't conflict with rollover. Lock conflicts are due to the fact that rollover uses scans.

Good Candidates for Chopping

- Batch operations that don't logically conflict with ongoing operations. (Index conflicts are not a problem).
- Chopping means take each batch operation and break it into independent pieces, e.g., delete zero-valued positions, update profit and loss.
- If batch operations are not idempotent, it is necessary to use a "breadcrumb" table that keeps track of which batch operations a process has completed.

©2023 Dennis Shasha

Tuning Case: Interest Rate Clustering

- Bond is clustered on interestRate and has a non-clustered index on `dealid`. Deal has a clustered index on `dealid` and a non-clustered index on `date`.

- Many optimizers will use a clustering index for a selection rather than a non-clustering index for a join. Often good. The trouble is that if a system doesn't have bit vectors, it can use only one index per table.

©2023 Dennis Shasha

Query to be Tuned

```
select bond.id
from bond, deal
where bond.interestRate = 2.6
and bond.dealid = deal.dealid
and deal.date = '7/7/2011'
```

What Optimizer Might Do

- Pick the clustered index on interestRate.
- May not be selective because most bonds have the same interest rate.
- This prevents the optimizer from using the index on **`bond.dealid`**. That in turn forces the optimizer to use the clustered index on **`deal.dealid`**.

©2023 Dennis Shasha

Alternative

- Make deal use the non-clustering index on `date` (it might be more useful to cluster on `date` in fact) and the non-clustering index on *bond.dealid*.

- Logical IOs decrease by a factor of 40 (170,000 to 4,000).

Case: Temporal Table Partitioning

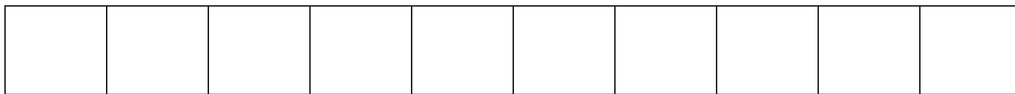
- `Position` and `trade` were growing without bound. Management made the decision to split each table by time (recent for the current year and historical for older stuff). Most queries concern the current year so should be run faster.

- What happened: a query involving an equality selection on `date` goes from 1 second with the old data setup to 35 seconds in the new one. Examining the query plan showed that it was no longer using the non-clustered index on `date`. Why?

Use of Histogram

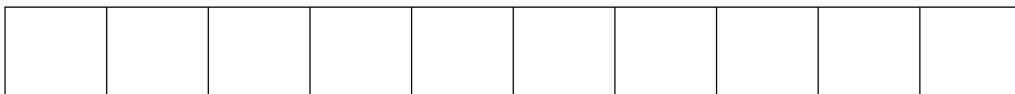
- Optimizer uses a histogram to determine usefulness of a non-clustering index.
- Histogram holds 500 cells, each of which stores a range of `date` values.
- Each cell is associated with the same number of rows (those in the cell's `date` range).

Initially, each cell was associated with several days' worth of rows.



Many days

After reducing the size of the table, each cell was associated with less than a day's worth of rows. So, a single day query spills on several cells.



Single day

Non-clustering index is not used if more than one cell contains the searched-for data.

Heuristic Brittleness

- The optimizer's rule is that a non-clustering index may be used only if the value searched fits entirely in one cell.

- When the tables became small, an equality query on `date` spread across multiple cells. The query optimizer decided to scan.

- A warning might help.