Unit 2: Tuning the Guts

- Concurrency Control — how to minimize lock contention

- Recovery — how to minimize logging overhead

- Operating System — tune buffer size, thread scheduling, etc.

- Hardware — when to allocate more memory, more disks, more processors

CONCURRENCY CONTROL

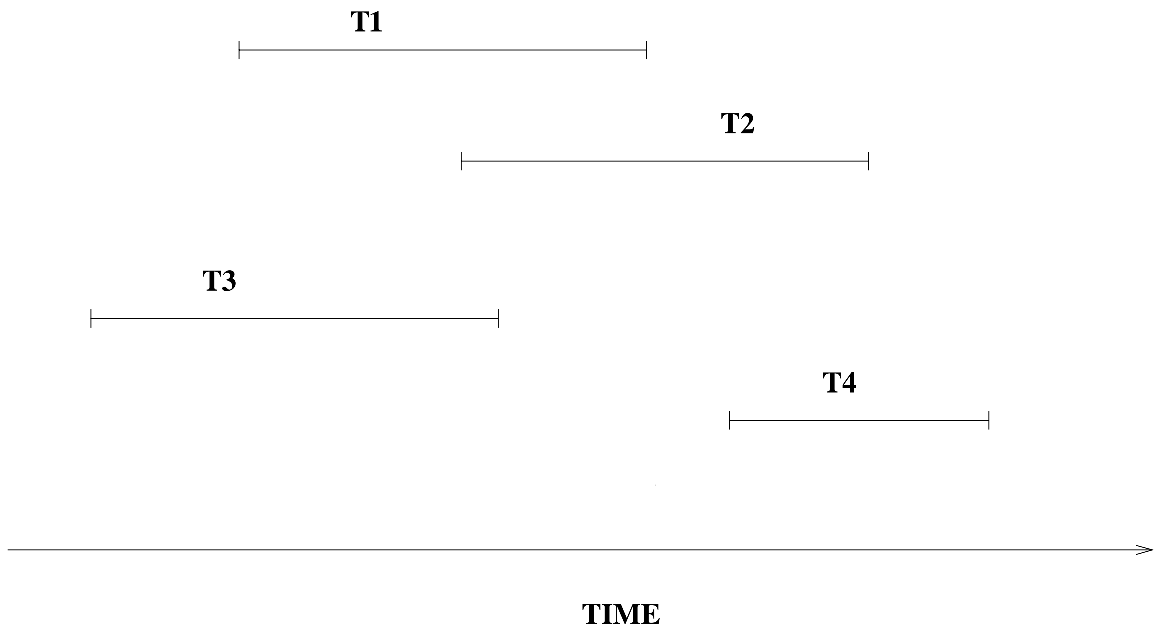
RECOVERY SUBSYSTEM

OPERATING SYSTEM

PROCESSOR (S), DISK(S), MEMORY

COMMON UNDERLYING COMPONENTS OF ALL DATABASE SYSTEMS

Fig. 2.1



T1 is concurrent with T2 and T3 only.
T2 is concurrent with T1, T3 and T4.

FIGURE: Concurrency

Fig. 2.2

Concurrency Control Goals

Performance Goal: Reduce

- blocking — one transaction waits for another to release its locks (should be occasional at most).
- deadlock — a set of transactions in which each transaction is waiting for another one in the set to release its locks (should be rare at most).

Implications of Performance Goal

Ideal transactions (units of work)

- Acquire few locks and prefer read locks over write locks (reduce conflicts).
- Locks acquired have "fine granularity" (i.e. lock few bytes to reduce conflicts).
- Held for little time (reduce waiting).

Correctness Goal

- Goal: Guarantee that each transaction appears to execute in isolation (degree 3 isolation).

- Underlying assumption: programmer guarantees that if each transaction appears to execute alone, then execution will be correct.

Implication: tradeoff between performance and correctness.

Example: Simple Purchases

Consider the code for a purchase application of item i for price p.

1. if cash $<$ p then roll back transaction

2. inventory(i) := inventory(i) + p

3. cash := cash − p

Purchase Example — two applications

Purchase application program executions are P1 and P2.

- P1 has item i with price 50.
- P2 has item j with price 75.
- Cash is 100.

If purchase application is a single transaction, one of P1 or P2 will roll back.

Purchase Application — concurrent anomalies

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

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 $= -25$

Purchase Application — solutions

- Orthodox solution — make whole program a single transaction.

  Implication: cash becomes a bottleneck.

- Chopping solution — find a way to rearrange and then chop up the programs without violating degree 3 isolation properties.

Purchase Application — chopping solution

Rewritten purchase application. Each bullet is a separate transaction.

- If cash $<$ p then roll back application.

  cash := cash - p

- inventory(i) := inventory(i) + p

Cash no longer a bottleneck.

When can chopping work? See appendix of tuning book for details.

Recovery Hacks for Chopping

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 requires reexecuting the second pieces of inventory transactions whose first pieces have finished.

Sometimes Degree 3 Isolation is Not Necessary

Statistical sampling of a raw data feed usually does not require full isolation.

Example: survey application counts the number of depositor balances over $1,000 one day. Doesn't need exact number.

Degree 2 isolation — reads hold locks only during accesses (default in Sybase — called level 1)

Degree 1 isolation — reads never obtain locks.

Multiversion Read Consistency

Certain systems, e.g. Oracle, RDB and Gemstone, offer facility whereby a long read will appear to see a snapshot of the data as it appeared when the read began.

- Good news: Gives degree 3 isolation without adding lock contention.
- Bad news: Costs space (e.g. if page p is updated after read begins, must keep old copy of p).

©Dennis Shasha

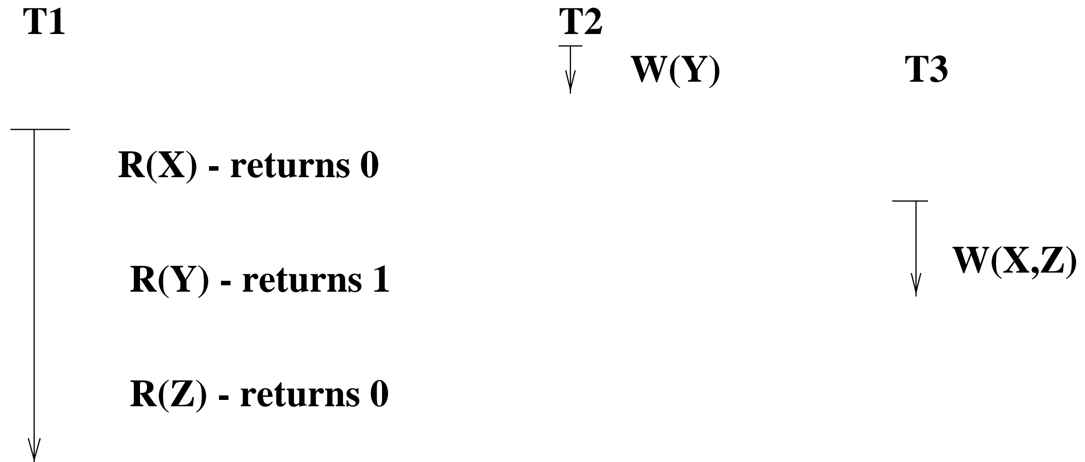
Multiversion Read Consistency

X=Y=Z=0

T1: R(X), R(Y), R(Z)

T2: Y:=1

T3: Z:=2, X:=3



Thus, reads return values at beginning of T1

Fig. 2.3

Sacrificing Isolation for Performance

A transaction that holds locks during a screen interaction is an invitation to bottlenecks.

Airline reservation

1. Retrieve list of seats available.
2. Talk with customer regarding availability.
3. Secure seat.

Performance would be intolerably slow if this were a single transaction, because each customer would hold a lock on seats available.

Solution

Make first and third steps transactions.

Keep user interaction outside a transactional context.

Problem: ask for a seat, but then find it's unavailable. Possible but tolerable.

Low Isolation Counter Facility

Consider an application that assigns sequential keys from a global counter, e.g. each new customer must contain a new identifier.

The counter can become a bottleneck, since every transaction locks the counter, increments it, and retains the lock until the transaction ends.

ORACLE offers facility (called SEQUENCE) to release counter lock immediately after increment.

Reduces lock contention, but a transaction abort may cause certain counter numbers to be unused.

©Dennis Shasha

Effect of Caching

The ORACLE counter facility can still be a bottleneck if every update is logged to disk.

So, ORACLE offers the possibility to cache values of the counter. If 100 values are cached for instance, then the on-disk copy of the counter number is updated only once in every 100 times.

In experiments of bulk loads where each inserted record was to get a different counter value, the load time to load 500,000 records went from 12.5 minutes to 6.3 minutes when I increased the cache from 20 to 1000.

Select Proper Granularity

- Use record level locking if few records are accessed (unlikely to be on same page).

- Use page or table level locking for long update transactions. Fewer deadlocks and less locking overhead.

  Example: In some systems, it is possible to choose between page and table locks with respect to a given table within a program or DB startup file.

Avoid Catalog Updates

The catalog can easily become a concurrency control bottleneck.

- It is small.
- Compiles of embedded queries and ad hoc queries must read it.

So, in production setting, discourage such updates.

Prototype application developers, keep away!

Load Control

Allowing too many processes to access memory concurrently can result in memory thrashing in normal time-sharing systems.

Analogous problem in database systems if there are too many transactions and many are blocked due to locks.

Gerhard Weikum and his group at ETH have discovered the following rule of thumb: if more than 23% of the locks are held by blocked transactions at peak conditions, then the number of transactions allowed in the system is too high.

Recovery Subsystem

Logging and recovery are pure overhead from point of view of performance, but necessary for applications that update data and require fault tolerance.

Tuning the recovery subsystem entails:

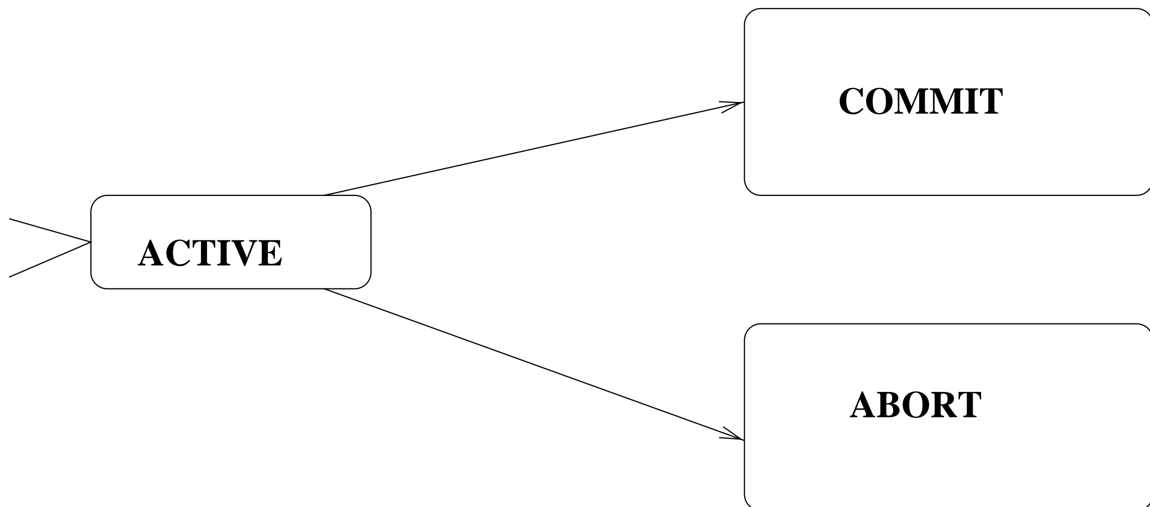
1. Managing the log.
2. Choosing a data buffering policy.
3. Setting checkpoint and database dump intervals.

Recovery Concepts

Every transaction either *commits* or *aborts*. It cannot change its mind.

Goal of recovery: Even in the face of failures, the effect of committed transactions should be permanent; aborted transactions should leave no trace.

Two kinds of failures: main memory failures, disk failures.



ONCE A TRANSACTION ENTERS COMMIT OR ABORT, IT CANNOT CHANGE ITS MIND

THE GOAL OF THE RECOVERY SUBSYSTEM IS TO IMPLEMENT THIS FINITE STATE AUTOMATON

Fig.2.4

What About Software?

Trouble is that most system stoppages these days are due to software. Tandem reports under 10% remaining hardware failures.

Fortunately, nearly 99% of software bugs are "Heisenbugs:"
one sees them once and then never again (study on IBM/IMS).

Heisenbugs stop the system without damaging the data.

Hardware-oriented recovery facilities are also useful for Heisenbugs.

Logging Principles

log — informally, a record of the updates caused by transactions. Must be held on durable media (e.g. disks).

- A transaction must write its updates to a log before committing. That is, before transaction ends.

  Result: holds committed pages even if main memory fails.

- Sometime later, those updates must be written to the database disks.

  How much later is a tuning knob.

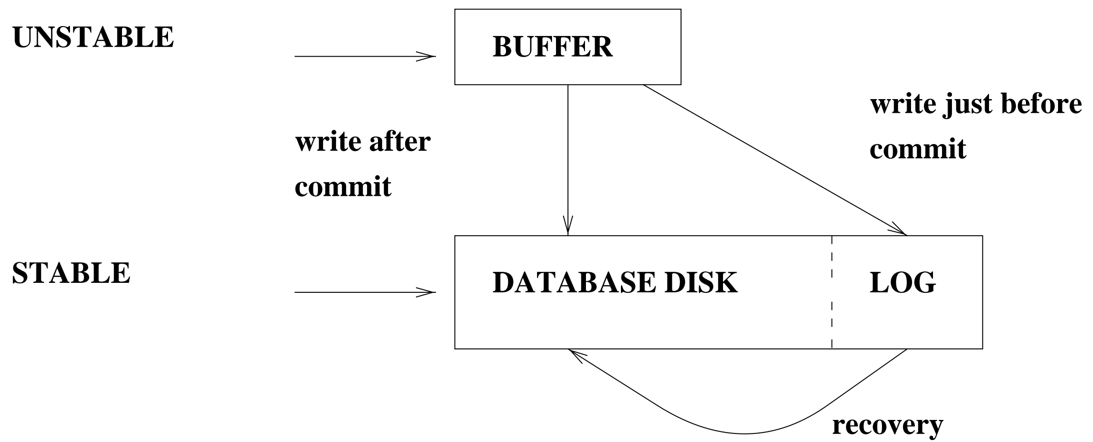


Fig. 2.5

Managing the Log

- Writes to the log occur sequentially.
- Writes to disk occur (at least) 10 times faster when they occur sequentially than when they occur randomly.

Conclusion: a disk that has the log should have no other data.

Better reliability: separation makes log failures independent of database disk failures.

©Dennis Shasha

Buffered Commit

- Writes of after images to database disks will be random.

- They are not really necessary from point of view of recovery.

Conclusion: good tuning option is to do "buffered commit." (Net effect: wait until writing an after image to the database costs no seeks.)

Buffered Commit — tuning considerations

- Must specify buffered commit, e.g., Fast_Commit option in INGRES. Default in most systems.

- Buffer and log both retain all items that have been written to the log but not yet to the database.

  This can consume more free pages in buffer. Regulate frequency of writes from buffer to database by DBWR parameters in ORACLE.

- Costs: buffer and log space.

Group Commits

If the update volume is EXTREMELY high, then performing a write to the log for every transaction may be too expensive, even in the absence of seeks.

One way to reduce the number of writes is to write the updates of several transactions together in one disk write.

This reduces the number of writes at the cost of increasing the response time (since the first transaction in a group will not commit until the last transaction of that group).

Database Dump Intervals

A database dump is a full copy of the database at some point in time.

Bad point: Creates overhead during on-line processing. (Should not be done more often than once or twice a day.)

Good point: Permits recovery from crash of database disks.

(i) current DB state = log + dump

Checkpoint Intervals

A checkpoint is a partial flush of the updates of the log onto the database disk.

Bad point: Creates overhead during on-line processing though less than a dump. Should not be done more often than every twenty minutes or so.

Good points:

- Reduces time to recover from a crash of buffer.
- Reduces size of log unless needed for the database dump.

(ii) current DB state = log + database disks.

Batch to Mini-Batch

Consider an update-intensive batch transaction.

If concurrent contention is not an issue, then it can be broken up into short transactions known as mini-batch transactions.

Example: Transaction that updates, in sorted order, all accounts that had activity on them in a given day.

Break up to mini-transactions each of which accesses 10,000 accounts and then updates a global counter. Easy to recover. Doesn't overfill the buffer.

Example

Suppose there are 1 million account records and 100 transactions, each of which takes care of 10,000 records.

- Transaction 1:
  Update account records 1 to 10,000.
  global.count := 10,000

- Transaction 2:
  Update account records 10,001 to 20,000
  global.count := 20,000

and so on....

Operating System Considerations

- Scheduling and priorities of threads of control.

- Size of virtual memory for database shared buffers.

- Lay out of files on disk.

Threads

- Switching control from one thread to another is expensive on some systems (about 1,000 instructions).

  Run non-preemptively or with long, e.g., 1 second long, timeslice.

- Watch priority:

  – Database system should not run below priority of other applications.

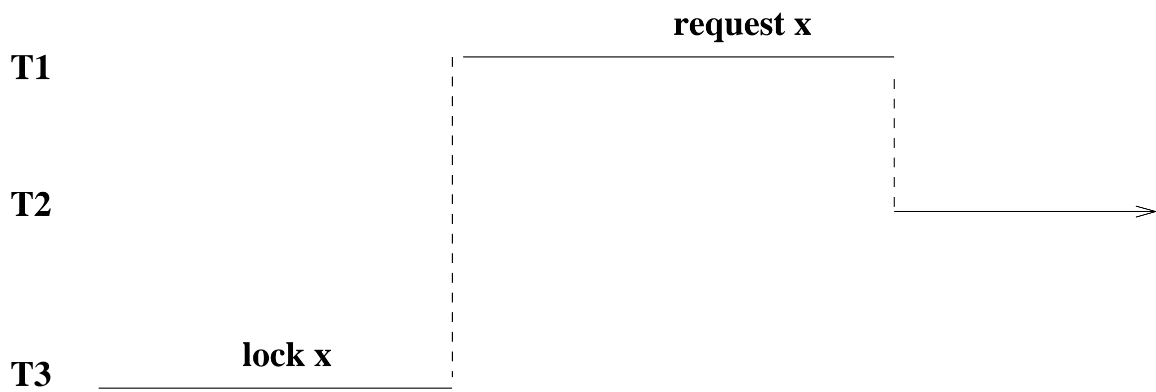
  – Avoid *priority inversion*

Example of Priority Inversion

Three transactions: T1, T2, T3 in priority order (high to low).

1. T3 obtains lock on x and is preempted.
2. T1 blocks on x lock, so is descheduled.
3. T2 does not access x and runs for a long time.

Net effect: T1 waits for T2.



T1 waits for a lock that only T3 can release. But T2 runs instead.

PRIORITY INVERSION

Fig. 2.8

©Dennis Shasha

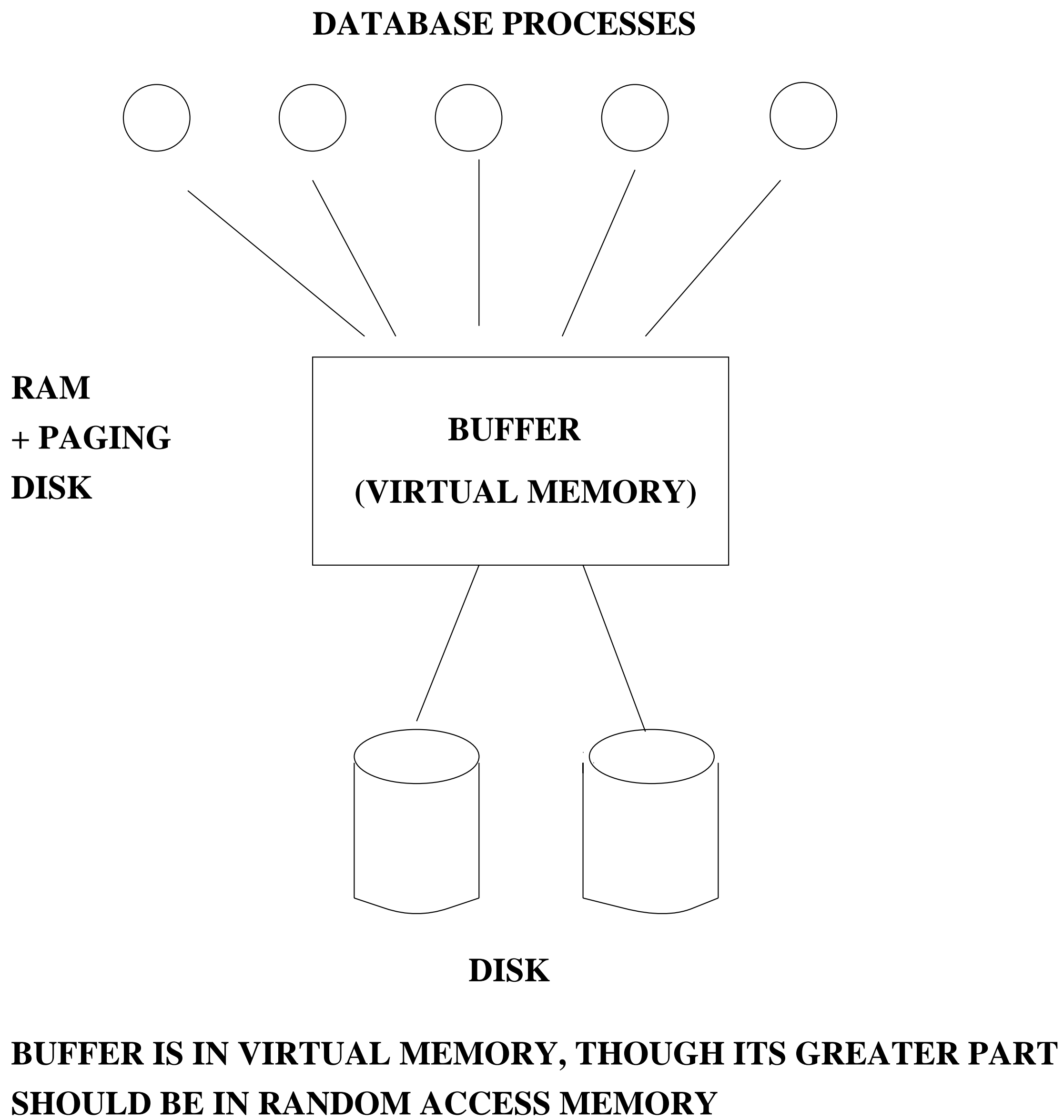
Avoiding Priority Inversion

- Give all transactions the same priority (recommended by many systems, e.g., ORACLE). Avoids the inversion problem on lock, but can lead to undesirable fairness.

- Specify dynamic priorities (either of operating system threads or of database system threads) that allow holder of lock to inherit priority of highest priority waiter of lock (e.g., new versions of SYBASE).

Buffer Memory — definitions

- buffer memory — shared virtual memory (RAM + disk).
- logical access — a database management process read or write call.
- physical access — a logical access that is not served by the buffer.
- hit ratio — portion of logical accesses satisfied by buffer.



BUFFER IS IN VIRTUAL MEMORY, THOUGH ITS GREATER PART SHOULD BE IN RANDOM ACCESS MEMORY

Fig. 2.6

Buffer Memory — tuning principles

- Buffer too small, then hit ratio too small.

  Some systems offer facility to see what hit ratio would be if buffer were larger, e.g. X$KCBRBH table in ORACLE. Disable in normal operation to avoid overhead.

- Buffer is too large, then hit ratio may be high, but virtual memory may be larger than RAM allocation resulting in paging.

Recommended strategy:

Increase the buffer size until the hit ratio flattens out. If paging, then buy memory.

Disk Layout

Whether or not a database management system uses the local operating system's file system (most do not), the following issues are important.

- Allocate long sequential slices of disk to files that tend to be scanned. Adjust pre-fetching parameter.

- Control utilization factor of disk pages depending on scan/update ratio

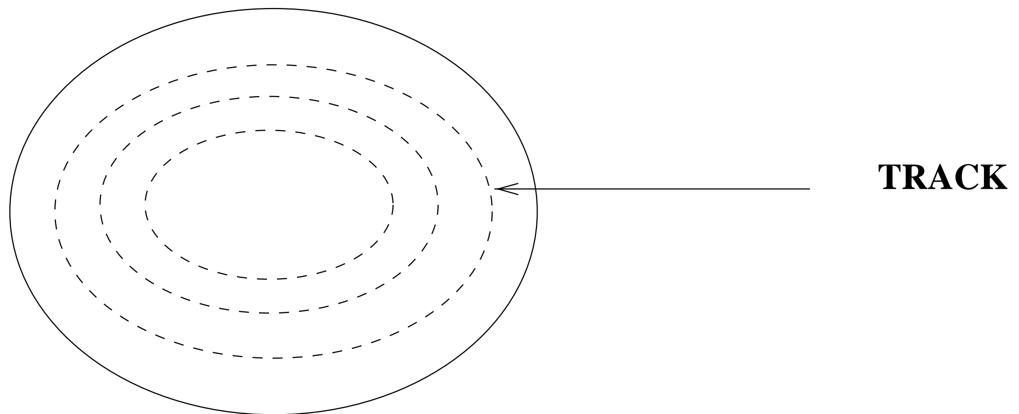
- Frequently accessed data in middle of magnetic disks.
  Outside of middle on CLV optical disks.

Sequential Slices

Allocating long sequential slices of disk helps in the following situations.

- Inserts to a history file or log.
  Any insert-intensive file, in fact. Because consecutive inserts can occur without seeks (movements from one track to another).

- Any scan-intensive file (e.g. where indexes are not useful). Make sure that your system allows pre-fetching (e.g., DB_FILE_MULTIBLOCK_READ_COUNT in ORACLE).



SURFACE OF A PLATTER ON A DISK. THE CONCENTRIC DASHED CIRCLES ARE CALLED TRACKS.

IN CLV FORMAT, OUTER TRACKS HAVE MORE DATA THAN INNER TRACKS

Fig.2.7

Utilization Considerations

Utilization = percentage of a page that can have data and yet still receive an insert (e.g., PCTFREE parameter in ORACLE).

- High utilization helps scans, because fewer pages need to be accessed, provided there are no overflows.

- Low utilization reduces likelihood of overflows when there are many updates that may change the size of a record (e.g. string fields that have NULL values when first inserted).

Hardware Tuning

- Add memory — enables you to increase database buffer size without increasing paging.

- Buy disks —

  - to ensure log is on a separate disk

  - to mirror a frequently read file

  - or to partition a large file.

©Dennis Shasha

Trading Memory Against Disk Costs

Gray and Putzolu offer a "5 minute rule" for trading memory against disk. Details in book (p. 42), but intuition is following:

- Price of keeping a page in memory is cost of RAM and supporting circuitry, e.g. $0.05 per kilobyte.

- Price of keeping a page on disk is cost of periodic disk accesses.
  For example, if a disk gives 50 accesses per second and costs $10,000, then an access every 100 seconds costs $2 (actually, a bit more because of controller costs).

Tradeoff based on frequency of access and page size.

Add Processors

- To offload non-database applications onto other processors

- To offload data mining applications to old database copy

- To increase throughput to shared data — use a shared memory architecture or a shared disk architecture.

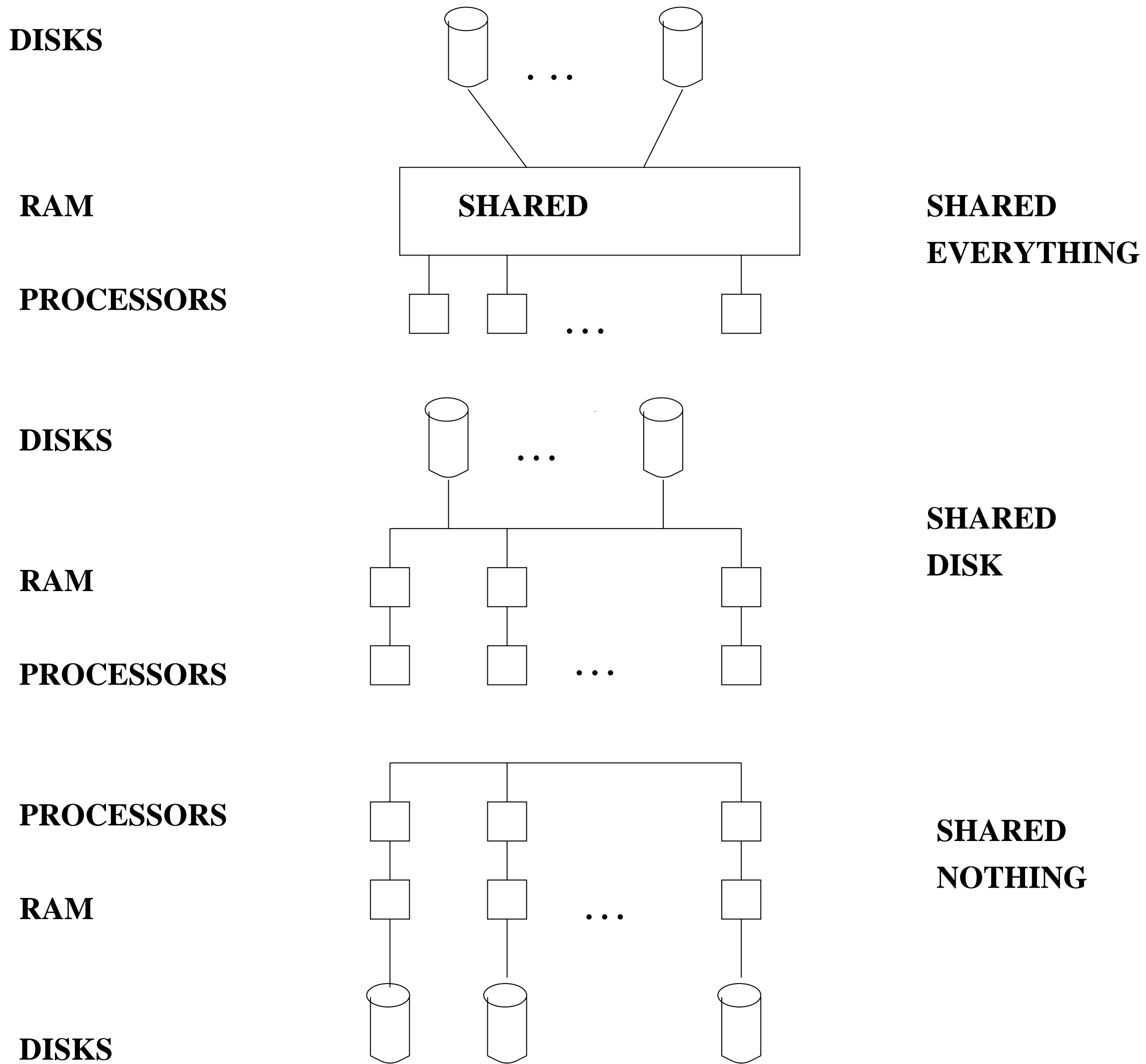


Fig. 2.9: POSSIBLE PROCESSOR-DISK-RAM CONFIGURATIONS

Scenario 1

- Many scans are performed.
- Disk utilization is high (long disk access queues).
- Processor and network utilization is low.
- Execution is too slow, but management refuses to buy disks.

Scenario 1: What's Wrong?

Clearly, I/O is the bottleneck. Possible reasons:

- Load is intrinsically heavy. Buy a disk with your own money.
- Data is badly distributed across the disks, entailing unnecessary seeks.
- Disk accesses fetch too little data.
- Pages are underutilized.

Scenario 1: an approach

Since processor and network utilization is low, we conclude that the system is I/O-bound.

- Reorganize files to occupy contiguous portions of disk.

- Raise pre-fetching level.

- Increase utilization.

Scenario 2

A new credit card offers large lines of credit at low interest rates.

Set up transaction has three steps:

1. Obtain a new customer number from a global counter.

2. Ask the customer for certain information, e.g., income, mailing address.

3. Install the customer into the customer table.

The transaction rate cannot support the large insert traffic.

Scenario 2: What's Wrong?

Lock contention is high on global counter.

In fact, while one customer is entered, no other customer can even be interviewed.

Scenario 2: Action

- Conduct interview outside a transactional context.

- Obtain lock on global counter as late as possible.

  Or use special increment facility if available (lock held on counter only while it is incremented).

Scenario 3

Accounting department wants average salary by department.

Arbitrary updates may occur concurrently.

Slow first implementation:

```
begin transaction
 SELECT dept, avg(salary) as avgsalary
 FROM employee
 GROUP BY dept
end transaction
```

Scenario 3: What's Wrong?

Lock contention on employee tuples either causes the updates to block or the scan to abort (check deadlock rate and lock queues).

Buffer contention causes the sort that is part of the group by to be too slow (check I/O needs of grouping statement).

Scenario 3: Action

Partition in time or space. In descending order of preference:

1. Pose this query when there is little update activity, e.g., at night.

2. Execute this query on a slightly out-of-date copy of the data.

3. Use multiversion read consistency if available.

4. Use degree 2 isolation; get approximate result.

Scenario 4

Three new pieces of knowledge:

1. The only updates will be updates to individual salaries. No transaction will update more than one record.

2. The answer must be at degree 3 isolation.

3. The query must execute on up-to-date data.

What does this change?

Scenario 4: Action

Degree 2 isolation will give equivalent of degree 3 isolation in this case.

The reason is that each concurrent update transaction accesses only a single record, so the degree 2 grouping query will appear to execute serializably with each update.

Scenario 5

Tickerplant application — distributes trade quantities and prices (ticks) to traders.

- Ticks enter at 500-1,000 per second.
- Each broker follows only certain stocks and bonds.

System is not keeping up.

Scenario 5: What's Wrong?

Some possibilities to check:

- Network to brokers is overloaded.
- Disks may be overloaded due to inserts of ticks.

Scenario 5: Approach

- Disk accesses should be limited to history file.

- Use disk extents.

- Processing to traders should be distributed. Partition traders into groups and dedicate one processor of a shared memory multiprocessor to each group as well as one network cable attached directly to that processor.

Scenario 6: Get One and Hold It

Application consists of many workstations, each of which has following algorithm: get next unprocessed tuple, process it in a few seconds and return it with processed flag set.

Application is slow because once one workstation gets the tuple to process, it locks out the 80 others who are scanning behind it.

Scenario 6: Partitioning Approach

Ask whether the tuples really must be processed in a particular order. If order may be approximate, then try following.

- Create 20 (or so) values of a new field A. Assign a number between 1 and 20 to A field of each tuple as it comes in. Index tuples with a clustering index on A. Partition workstations so workstation i handles tuples whose A field is (i mod 20) + 1.

  For better load balancing, have some workstation randomly choose A values of tuples to work on. That is, pick x between 1 and 20, try to get a tuple with A=x. If none, pick another x.

Case Study: Sales and Marketing Application

- Application had a 3 gigabyte RAID disk. Database size about 2 gigabytes.
- Also, 4 internal disks of 200 megabytes each.
- Expect 50 active users

Case Study: Basic Decisions

- The mirrored logs should go on two disks off of different controllers of two internal disks.

- The rollback segments and system table should go on the third internal disk.

  The rest of the database should be on the Raids.

Oracle Parameter Settings (INIT.ORA)

- db_block_size — 4k (page size)
- DB_BLOCK_BUFFERS —- 20,000 (basic buffer)
- LOG_BUFFER — 32 K (for group commits)
- SHARED_POOL_SIZE — 10 Meg (for compiled queries)

More Oracle Parameters (INIT.ORA

- DB_FILE_MULTIBLOCK_READ_COUNT = 16 (for readahead, related to track size)

- PROCESSES = 50 at least (users and daemons that can connect)

  SEMMNS, SEMMNI, SEMMSL should be set appropriately to avoid running out of semaphores (default setting of 400 is fine for this number of PROCESSES).

- LOG_CHECKPOINT_INTERVAL — 50,000 (number of blocks)

  A Checkpoint will occur when above limit is reached or when a single logfile is full. Logfile size is 10M.

Unit 3: Indexes

- query types
- key types
- data structures
- clustering indexes
- non-clustering indexes
- composite indexes
- care and feeding of indexes

LANGUAGE INTERFACE
(RELATIONAL OR OBJECT-ORIENTED)

INDEXES

CONCURRENCY CONTROL

RECOVERY

OPERATING SYSTEM

HARDWARE

INDEXES IN DATABASE SYSTEM ARCHITECTURE

Fig. 3.1

Query Types

- point query — returns at most one record, e.g. equality selection on social security number.
- multi-point query — may return several records, e.g. equality selection on department.
- range query — may return many records, e.g. all employees who earn between $50,000 and $90,000.

More Query Types

- prefix query — based on prefix of an attribute, e.g., name = 'Sm%'.
- extremal query — may return many records based on a maximum or minimum, e.g. employee(s) who earn the most.
- join query — results from a join condition, e.g. employee.name = student.name

Key Types

- sequential key — attribute whose value is monotonic with time of insertion, e.g. a timestamp.

- non-sequential key — the opposite

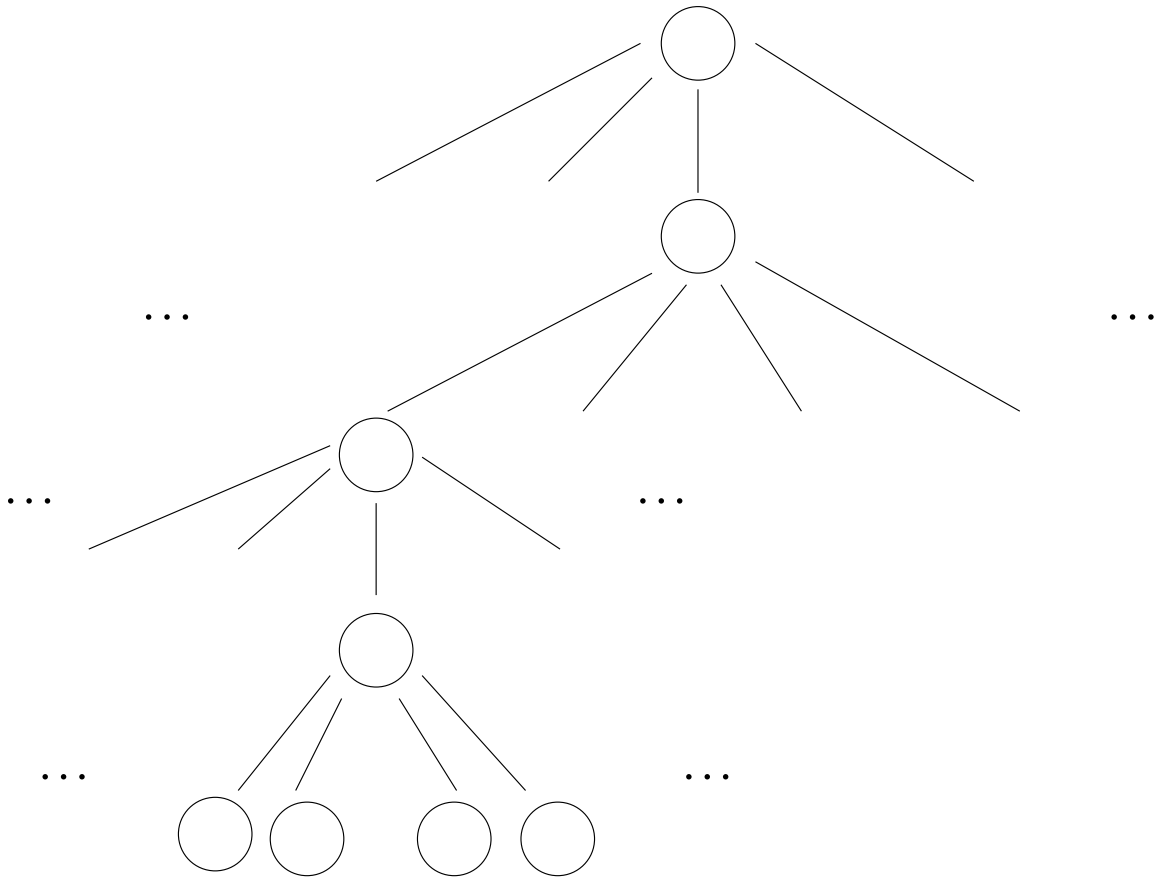
On certain data structures, sequential key may cause lock contention.

Can you see why?

Data Structure

Maps a key value to the location(s) where the record(s) having that key value can be found.

Different structures are good for different query types.



Tree with fanout of four and five levels

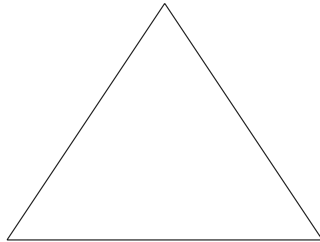
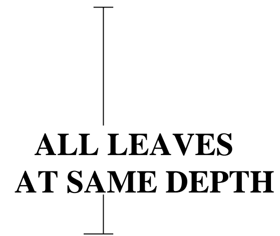
LEVELS AND BRANCHING FACTOR

Fig. 3.2

B-tree

Balanced tree. Often with links from leaf to neighboring leaf.

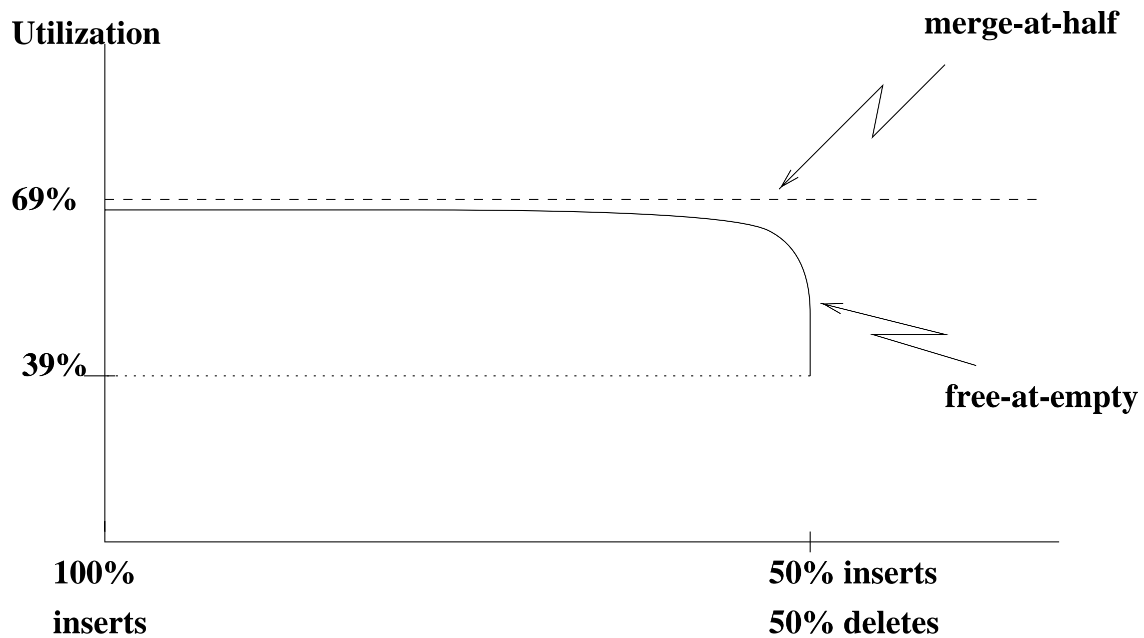
- Good for range, prefix, and extremal queries.
- Self-regulating — reorganization is rarely necessary.
- Lock contention on last page if there are many inserts and key is sequential.
- Bad if key is very large, e.g. if it is a string. In that case, use compression to avoid too many levels.

KEYS ARE SORTED AT LEAVES

A B-TREE IS A MULTI-ARY TREE SUCH THAT EVERY LEAF IS AT THE SAME DISTANCE FROM THE ROOT AND THE KEYS (ATTRIBUTE(S) BEING INDEXED) ARE SORTED AT THE LEAVES.

Fig. 3.3



Utilization if free-at-empty.
B-trees stays near 69% until the percentage of inserts fall to 52%.

E.g., Ingres does not even free empty nodes. Periodically, administrator reorganizes the index.

Fig. 3.4

Key Length and Fanout Example

- Pointer — 6 bytes.
- Page — 4 kilobyte
- Number of leaf key-pointer pairs — 42 million.
- Average node utilization — 69%.

With 4 byte keys: 3 levels.

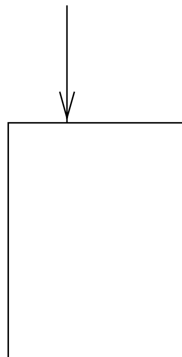
With 94 byte keys: 5 levels.

COMPRESSION TYPES

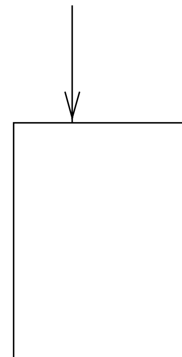
COMPRESSION EXAMPLES WITH: Robert, Robin

PREFIX COMPRESSION

Robe

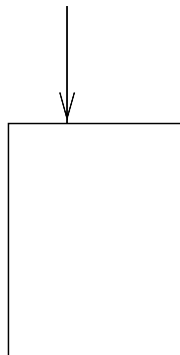


Robi



INFIX COMPRESSION (expensive in processor time)

Robe



3i

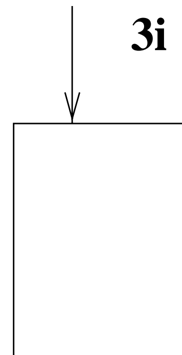


Fig. 3.4a

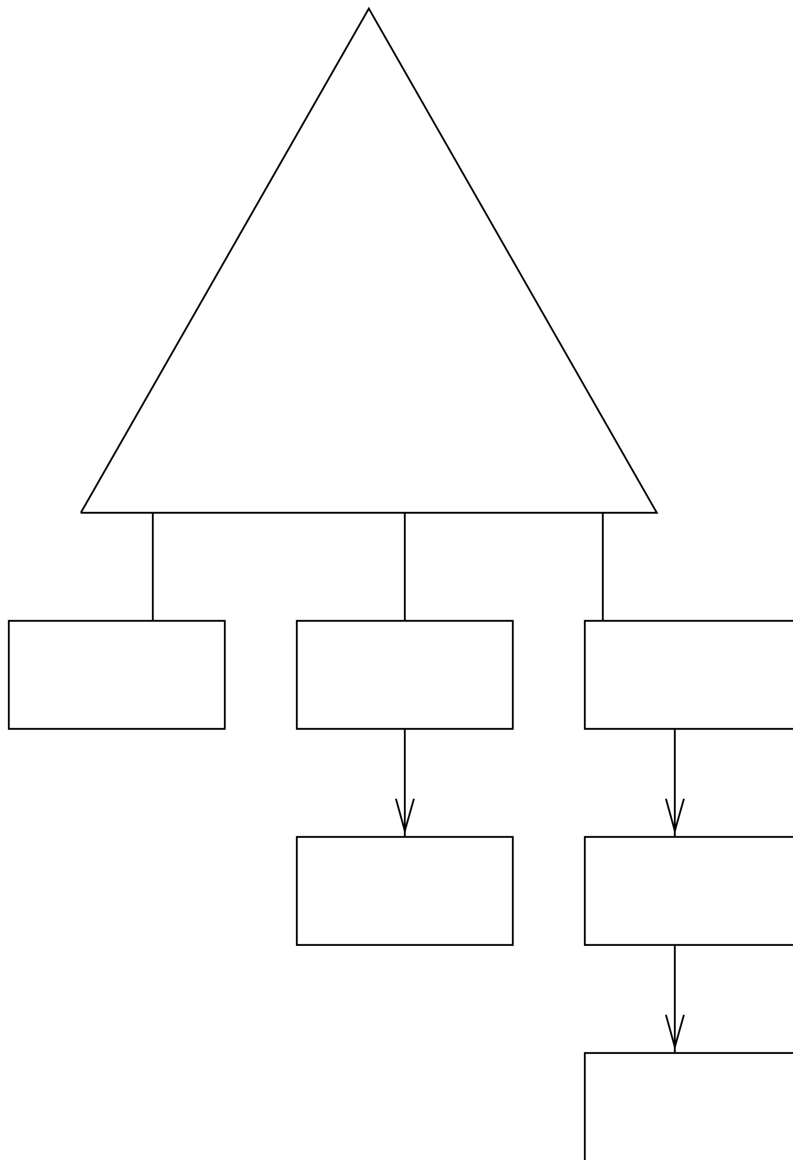
ISAM Structure

Index sequential access method — like B-tree except that tree is never modified. Instead leaves overflow.

- Better than B-trees for range, prefix, and extremal queries, provided there are few overflows, because utilization can approach 100%.

- No locking overhead on interior tree nodes.

- Bad when there are insertions, because overflows quickly destroy structure.



In ISAM structures, the index is fixed, but the leaves may overflow forming chains.

Hash Structures

Hash function = given a key, returns the location(s) of the record(s) containing the key (possibly through an overflow chain).

- Great for point queries, provided hash table is 50% full (or less).
- Little extra overhead for large keys.
- Useless for range, prefix, or extremal queries, because close keys may be mapped to locations that are far apart.
- Requires reorganization if becomes too full.

Comparison of Data Structures

- Point/multipoint queries — Hash > ISAM > Btree, if no overflows.
- Few inserts on sequential key — Hash > Btree > ISAM, because of concurrency control problems (especially with page locking).
- Heavy inserts — Btree > Hash > ISAM — second two must be reorganized periodically.

Sparse vs. Dense Indexes

Index = data structure with pointers to a table of records.

Sparse index = the data structure has one pointer to each page of records.

Dense index = the data structure has one pointer to each record.

- For small records, a sparse index may be a level shorter than a dense one.
- In some systems (e.g., SYBASE), dense index is better for queries for which search criteria and select list are all in index. No need to visit data.

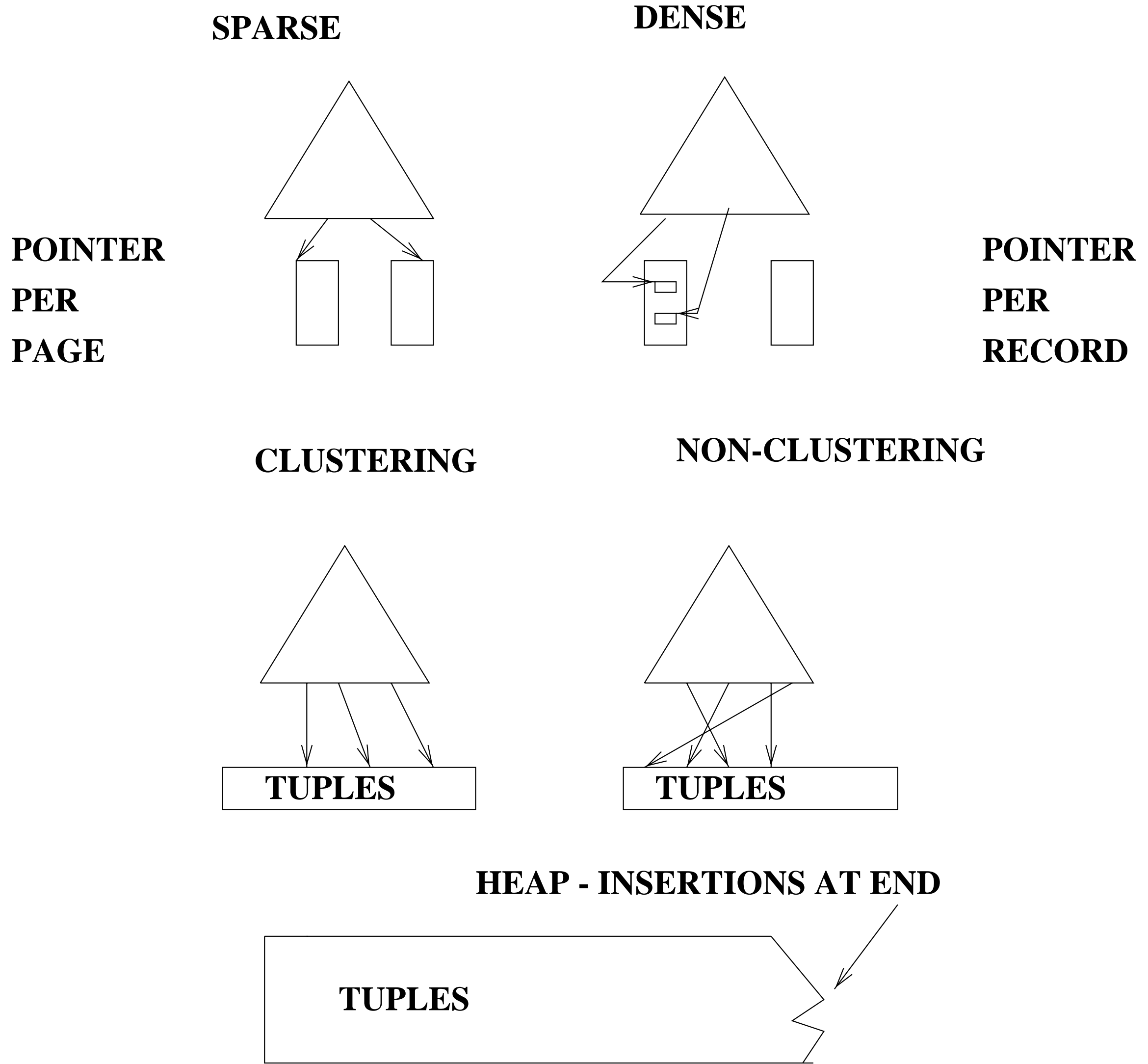
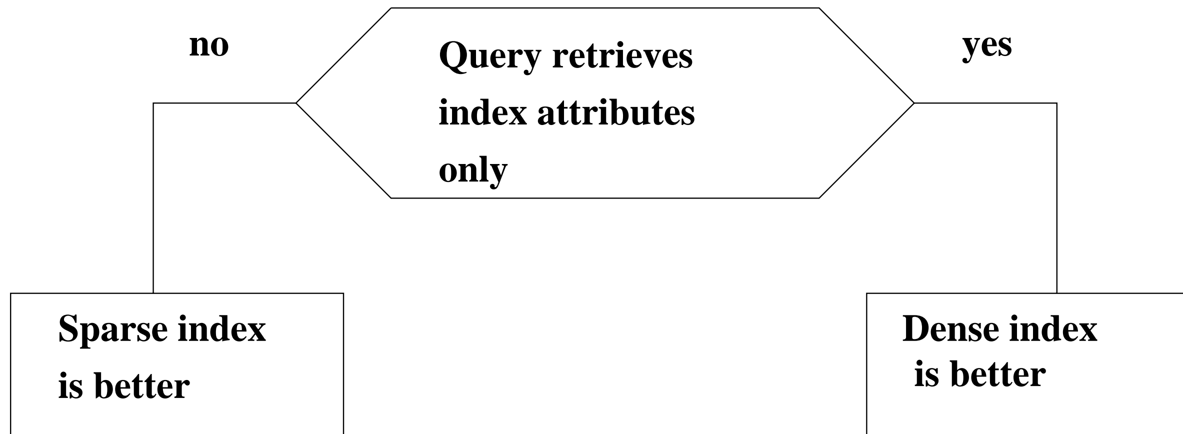


Fig. 3.6



ALL ELSE BEING EQUAL, IT IS ALMOST ALWAYS BETTER TO HAVE A SPARSE INDEX: ALL UPDATES, MOST QUERIES.

Fig.3.7

©Dennis Shasha

Clustering Index

Clustering index = data structure that implies a table organization.

That is, the table is organized (e.g., sorted) based on the clustering attribute(s).

Ex: Phone book is clustered on last name, first name.

Review of Index Terminology

Index = data structure (Btree, Hash, ISAM) + pointers to data

Sparse index = one pointer per data page

Dense index = one pointer per data record

Clustering index = may be sparse or dense and implies a data organization, e.g. Btree on A, data records sorted on A.

Non-clustering index = enforces nothing about data organization, so must be dense.

Clustering vs. Non-clustering

- Clustering index may be sparse (up to the DBMS implementor).

- Good for range (e.g., R.A between 5 and 15) and prefix queries (e.g. R.Name = 'Sm%').

  Near key values in data structure correspond to near tuples.

- Good for concurrency — usually.

Clustering Indexes and Concurrency Control

- If no clustering index, then insertions occur at end of a heap (file organized by time of insertion).

  Concurrent insertions will then conflict on last page of heap.

- If there is a clustering index, e.g. on SocSecNum, then consecutive insertions will likely be far apart. Low contention.

- However — if key value proportional to time of insertion, then clustering index based on B-tree causes a problem.
  Do you see why?
  Would hashing make any difference?

Clustering Indexes: bad news

- Inserts tend to be placed in the middle of the table. This can cause overflows, destroying the benefits of clustering.
- Similarly for updates.

So, may be a good idea to use fairly low page utilization when using a clustering index.

Non-Clustering Index

Non-clustering index = data structure but no imposition on structure of table. May have several per table.

- Dense, so some queries can be answered without access to table. For example, assume a non-clustering index on attributes A, B and C of R.

  SELECT B, C FROM R WHERE A=5

- Good for point queries and for selective multi-point, and for extremal queries. May or may not be useful for join queries.

Non-clustering Indexes and Selectivity — Example 1

- Pages are 4 kilobytes.
- Attribute A takes on 20 different values.
- Query is multi-point on A.
- There is a non-clustering index on A.

If record is 50 bytes long, there are 80 records per page. Nearly every page will have a matching record.

Don't create non-clustering index.

Non-clustering Indexes and Selectivity — Example 2

- Pages are 4 kilobytes.
- Attribute A takes on 20 different values.
- Query is multi-point on A.
- There is a non-clustering index on A.

If record is 2000 bytes long, there are 2 records per page. Only 1 in ten pages will have a matching record.

Create non-clustering index.

Quantitative Conclusions

F — number of records per page.

D — number of different values of attribute.

P — number of pages prefetched when performing a scan.

- if $D < FP$, then no non-clustering index.
- otherwise, create non-clustering index provided multipoint queries on attribute are frequent.

Practice: derive this.

Composite Indexes

Composite index = index on multiple attributes, e.g. last name, first name; latitude, longitude; or supplier, part

Good way to support uniqueness of multiple attributes.

Ex: onorder(supplier, part, quantity).

Each supplier may be in several records; similarly for parts. No single combination, however, will be in more than one record.

Composite Indexes — 2

Some geographical queries.

```
SELECT name FROM city
WHERE population ≥ 10000
AND latitude = 20
AND longitude ≥ 5 AND longitude ≤ 15
```

Good if index is on latitude, longitude as opposed to longitude, latitude. Most specific goes first.

Bad if key size is a problem.

©Dennis Shasha

Joins and Indexes

In general considerations are similar to those for selections.

- A clustering index shines when the join must access the table data.

- A dense non-clustering index shines when the join does not need to access the table data. (Semi-join condition.)

Join — Examples

Which makes which index type shine?

```
SELECT R.A, R.D
FROM R, S
WHERE R.B = S.C
```

```
SELECT R.A, R.D, S.E
FROM R, S
WHERE R.B = S.C
```

Join — How Clustering Helps

When semi-join condition does not hold, clustering helps in three ways.

1. If clustering index sparse, then fewer levels.

2. If many S records may hold a given R.B value (S.C not a key) then clustering on C clusters those S records. In this case, a non-clustering index may be worse than building a hash join from scratch according to experiments at Tandem.

Clustering vs. Nonclustering

- Clustering —
  sparse is possible
  strong on point queries
  strong on multipoint queries
  strong on range/prefix queries (Btree/ISAM)

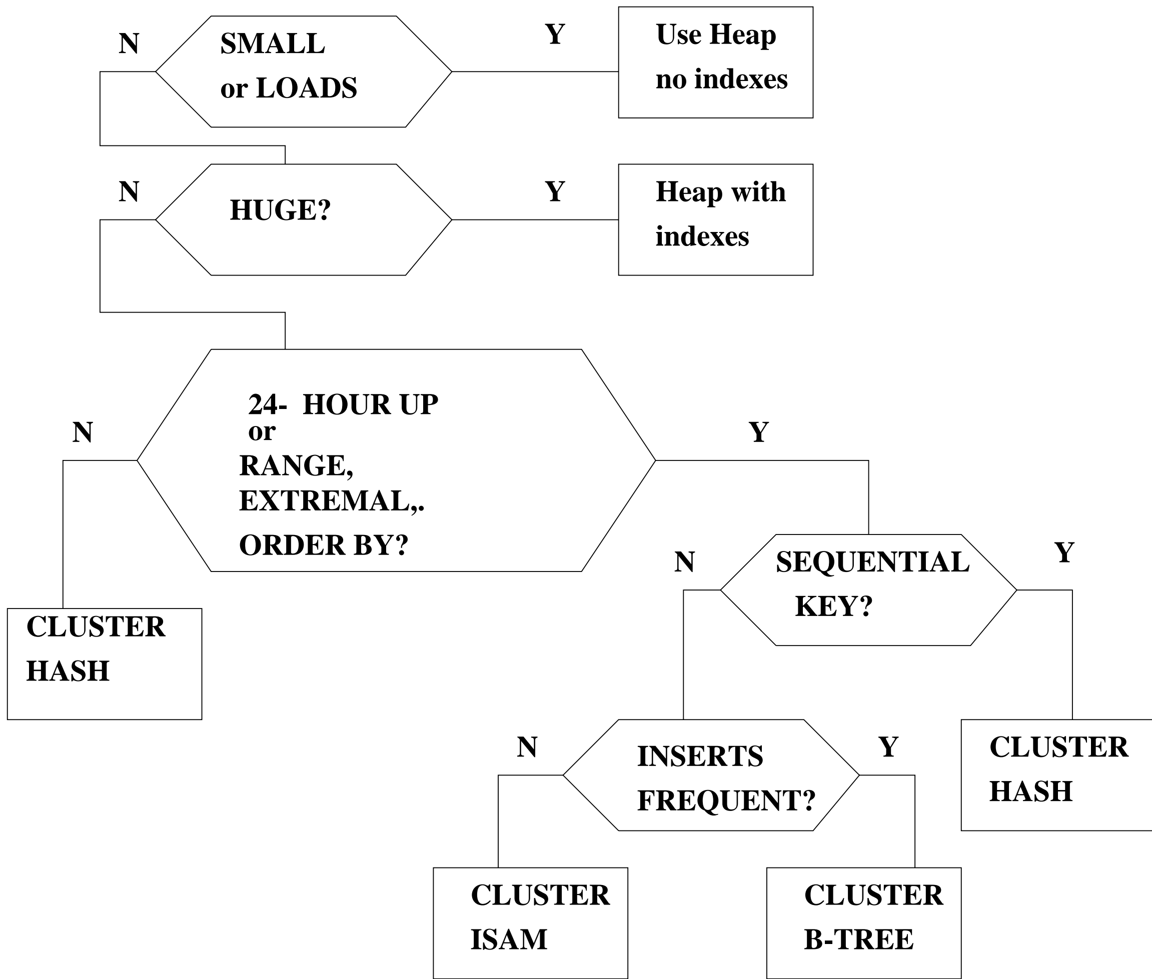
- Nonclustering —
  must be dense
  strong on point queries
  strong on multipoint queries if selective.

Indexes and Size

Don't index a small table (unless read-only).

- An index search may read at least one index page and one data page. On the other hand, if the entire relation is held on a single track and you can pre-fetch that, you need only one disk access.

- If many inserts execute on a table with a small index, then the index itself may become a concurrency control bottleneck.



BASIC INDEX SELECTION

Fig. 3.8

How to Distribute Hot Table

If a table is hot, then partition your index across several disks.

- If insert-intensive, then put non-clustering indexes on separate disks from clustering and table data (e.g., put in different Tablespace in ORACLE).

  Reason: Each insert will update all non-clustering indexes, but will use the clustering index to do its search. This organization will spread the load.

- If read-mostly, partition indexes and table over all disks. Balances load for random reads (e.g., use STRIPEDTABSPACE in ORACLE).

General Care and Feeding

Here are some maintenance tips on indexes:

- Give them a face lift: eliminate overflow chains on index or table.

- Drop indexes when they hurt performance. (Rebuild clustering indexes after sorting first.)

  Ex: during a batch insertion, you may be better off having no secondary indexes.

- Check query plan.

Reasons System Might Not Use an Index

- Catalog may not be up to date.

  Optimizer may believe table is too small.

- Query may be badly specified. For example, in some systems (e.g. ORACLE v6 and earlier):

  SELECT * FROM employee
  WHERE salary/12 $\geq$ 4000

  would not use salary index whereas following would:

  SELECT * FROM employee
  WHERE salary $\geq$ 4000 * 12

Scenario 1

Suppose there are 30 employee records per page.

Each employee belongs to one of 50 departments.

Should you put a non-clustering index on department

Scenario 1: Action

If such an index were used, performance would be worse, not better. The reason is that approximately 3/5 of the pages would have employee records from any given department.

Using the index, the database system would access 3/5 of the pages in random order. A table scan would likely be faster.

So, index is likely to be write-only.

Scenario 2

There are no updates. Here are queries. Which indexes should be established?

1. Count all the employees that have a certain salary. (frequent)

2. Find the employees that have the maximum (or minimum) salary within a particular department. (rare)

3. Find the employee with a certain social security number. (frequent)

Scenario 2: Action

- Non-clustering index on salary, since the first query can be answered solely based on the non-clustering index on salary.

- Sparse clustering index on social security number if the employee tuples are small, because a sparse index may be a level shorter than a dense one.

- Non-clustering composite index on (dept, salary) using a B-tree for second query, should it become more important.

Scenario 3

The employee table has no clustering indexes.

Performs poorly when there are bursts of inserts.

Locking is page-based.

Scenario 3: What's Wrong?

Employee is organized as a heap, so all inserts to employee occur on the last page.

Last page is a locking contention hot spot. Use monitoring facilities to check lock statistics (e.g. MONITOR LOCK in ORACLE).

Scenario 3: Action

- Find some way to smooth out the bursts of inserts.

- Use record-level locking.

- Create a clustering index based on hashing on customer number or social security number. Alternatively, create a B-tree-based clustering index on social security number.

Scenario 4

A credit card company tries to support the following:

- Insert new client records. (very frequent)
- Locate a client by Social Security number. (very frequent)
- Locate a client by customer number, a sequential key. (frequent)
- Scan the table for customer billing purposes. (rare)

Clustering hash index on Social Security number. Nonclustering B-tree index on customer number. Locking is page level.

Scenario 4: What's Wrong?

All inserts modify the last page of the B-tree index.

Observe that the lock contention does not occur on data pages because the data pages are clustered by Social Security number rather than by order of insertion.

Scenario 4: Action

- If record locking is available, try that first.

- Use a hash nonclustering index on customer number instead of a B-tree index.

  Hashing will tend to place new distinct records randomly in the hash index, thus avoiding the creation of a contention bottleneck.

Scenario 5

Ellis Island is a small island south of Manhattan through which flowed some 17 million immigrants to the United States between the late 1800's and the mid-1900's.

Immigration workers filled in some 200 fields on each immigrant, e.g. last name, first name, city of origin, ship taken, nationality, religion, arrival date and so on.

Want pure retrieval database for this data.

Querier is assumed to know last name of immigrant at least.

Most queriers will know the last name and either year of arrival or first name.

What is a good structure?

Scenario 5: Action

ISAM structures can be used since no updates.

If not available, then use B-tree.

- Clustering composite index on (last name, first name).
- Composite indexes on (last name, year of arrival) and whatever other combination of attributes queriers are likely to know.
- Only cost to indexes is space.

Unit 4: Relational Systems

- Comparative advantage of relational systems.
- Tuning normalization, clustering and denormaliza
- Query rewriting as tuning technique.
- Importance of triggers.
- Connections to applications.

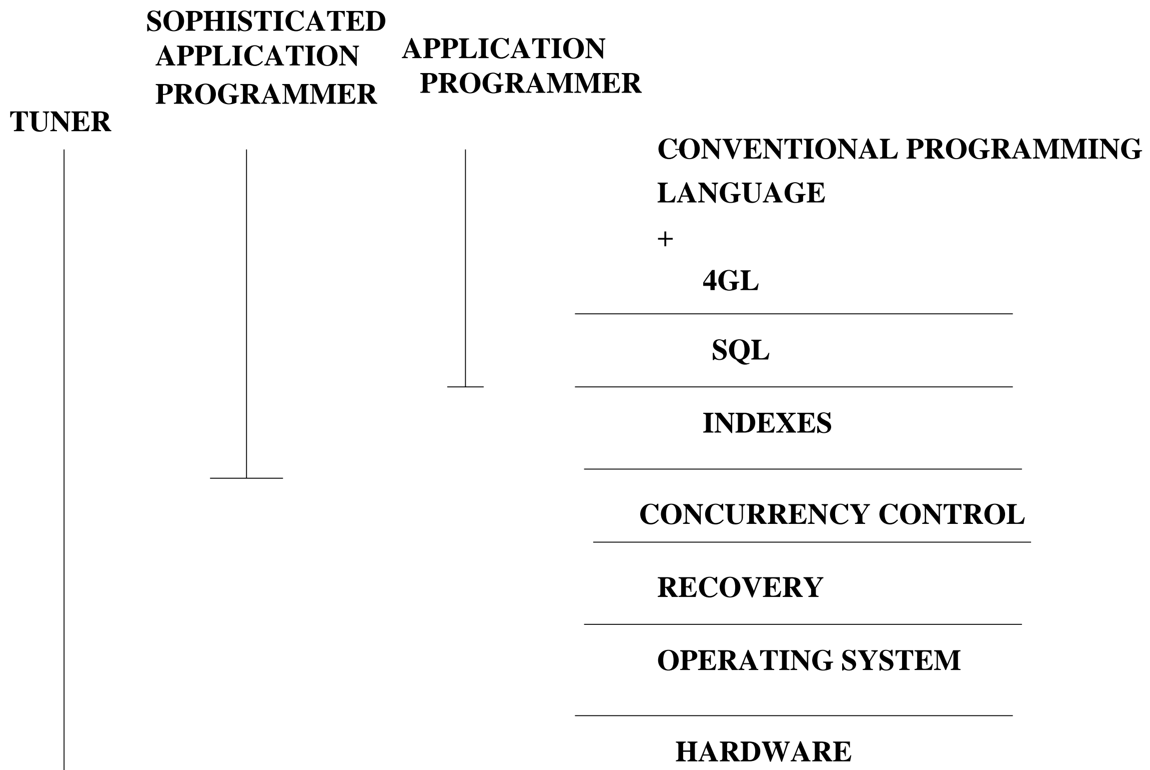
Comparative Advantage of Different Data Models

- Hierarchical and network data models — stable, simple applications where high performanc is critical. Trillion dollars worth of software for such systems.

- Object-oriented data model — expressive, complex applications that do not now use databases.

  Performance advantage: in-memory graph traversal.

- Relational (and object-relational) systems capture everything else — from accounting to decision support to genetics.



Architecture of Relational Database Systems
Responsibilities of people with different skills.

fig. 4.1

Normalization: motivating example

Application about suppliers and parts on order.

Schema design I (unnormalized):

- Onorder1(supplier_ID, part_ID, quantity, supplier

Schema design II (normalized):

- Onorder2(supplier_ID, part_ID, quantity)
- Supplier(supplier_ID, supplier_address)

Normalization: definitions 1

- The *key* of a relation is a minimal subset of attributes such that no two distinct records of the relation have the same values on those attributes.

  For Onorder1 and Onorder2, supplier_ID and part_ID together constitute a key. For Supplier, supplier_ID alone is a key.

- X *determines* A means that if any two tuples have the same X values, then they have the same A values.

  Keys always determine all other attributes.

Normalization: definitions 2

Relation is normalized if, whenever X determines A and A is not an attribute of X, X is a key or a superset of a key.

- Onorder1 is not normalized because supplier determines supplier_address, but supplier is not a superset of the key of Onorder1.
- By contrast, Onorder2 is normalized because supplier_ID and part_ID together determine quantity and they constitute a key. For Supplier, supplier_ID determines supplier_address so is the key.

Comparison: Space

Example: 100,000 orders outstanding and 2,000 suppliers.

Supplier_ID is an eight byte integer.

Supplier_address requires 50 bytes.

- Space – The second schema will use extra space for the redundant supplier_ID.
  Total: $2000 \times 8 = 16{,}000$ bytes.

  First schema stores 100,000 supplier addresses compared with 2000 for second.
  Total: $98000 \times 50 = 4{,}950{,}000$ bytes.

  Net savings for second schema: 4,934,000 bytes

Comparison: Performance

- Suppose one wants address of the supplier from where a given part has been ordered.

  First schema (unnormalized) is better — requires merely a selection versus a join.

- On insertion, first schema requires adding the supplier address to every supplier-part-quantity triple.

  Either extra data entry effort or extra lookup.

  Second schema (normalized) is better.

Is Normalization Good?

In unnormalized schema, relationship between supplier_ID and supplier_address is repeated for every part on order.

- This wastes space.
- It may or may not be good for performance.
  - Good for queries that correlate parts with supplier addresses.
  - Bad for inserts.

  More details to come.

Tuning Normalization

Consider a bank whose Account relation has the schema:

- (account_ID, name, street, postal_code, balance)

When is it worthwhile to adopt the following schema?

- (account_ID, balance)

- (account_ID, name, street, postal_code)

Both schemas are normalized. Second one results from *vertical partitioning*.

Value of Vertical Partitioning

Second schema has following benefits for simple account update transactions that access only the ID and the balance:

- Sparse clustering index on account_ID of (account_ID, balance) relation may be a level shorter than it would be for the full relation.

- More account_ID-balance pairs will fit in memory, thus increasing the hit ratio.

In Moderation....

Consider the further decomposition:

- (account_ID, balance)

- (account_ID, name, street)

- (account_ID, postal_code)

Still normalized, but not good since queries (e.g. monthly statements, account update) require either both street and postal_code or neither.

Vertical Partitioning: rule of thumb

If XYZ is normalized and XY and XZ are also normalized, then use XYZ unless both of following hold:

- User accesses rarely require X, Y, and Z, but often access XY or XZ alone (80% of time if sparse clustering, 90% if dense).

- Attribute Y or Z values are large.

World of Bonds

Brokers base their bond-buying decisions on price trends.

Database holds the closing price for the last 3,000 trading days.

Prices regarding the 10 most recent trading days are frequently accessed.

Basic schema:

- (bond_ID, issue_date, maturity, ...) — about 500 bytes per record.
- (bond_ID, date, price) — about 12 bytes per record.

Vertical Anti-Partitioning

- (bond_ID, issue_date, maturity, today_price, yesterday_price, ... 10dayago_price) — 544 bytes per record.

- (bond_ID, date, price)

Avoids a join when retrieving information about a bond including statistics about the last 10 days' prices.

Typical Queries for Mail Orders

- Correlate the sale of some item, e.g. safari hat, with the city where the customer lives.
- Find a correlation between different items bought at different times, e.g. suitcases and safari hats.
- Find address of a promising customer.

How should Oldsale be organized?

Denormalization Has its Place

Denormalized — Add customer address to Oldsale.

This design is attractive for the query that correlates customer city with items purchased as well as for the query that finds the address of the promising customer.

Neither of the other designs is as good.

Organization Lessons

1. Insert- and update-intensive applications should use a standard, normalized design.

2. Read-only activities that would require many joins on a normalized schema are the best candidates for denormalization.

Convenience Stores

The accounting department of a convenience store chain issues queries every twenty minutes to discover:

- the total dollar amount on order from a particular vendor and
- the total dollar amount on order by a particular store outlet.

These take a long time on original schema.

Original Schema

- order(ordernum, itemnum, quantity, purchaser, vendor)

- item(itemnum, price)

Order and item each has a clustering index on itemnum.

Can you see why the total dollar queries will be expensive?

Query Maintenance

Add:

- VendorOustanding(vendor, amount), where amount is the dollar value of goods on order to the vendor, and

- StoreOutstanding(purchaser, amount), where amount is the dollar value of goods on order by the purchasing store, with a clustering index on purchaser.

Each update to order causes an update to these two redundant relations (triggers would make this fast). Worthwhile if lookup savings greater than update overhead.

Query Rewriting

General principle:
The first tuning method to try is the one whose effect is purely local.

Query rewriting has this property.

Two ways to see that a query is running too slowly.

1. It issues far too many disk accesses, e.g. a point query scans an entire table.

2. Its *query plan*, i.e. the plan the optimizer has chosen to execute the query, fails to use a promising index.

©Dennis Shasha

Running Examples

- Employee(ssnum, name, manager, dept, salary, numfriends).

  Clustering index on ssnum; non-clustering indexes on name and dept each. Ssnum and name each is a key.

- Student(ssnum, name, degree_sought, year).

  Clustering index on ssnum; non-clustering index on name; keys are ssnum and name each.

- Tech(dept, manager, location)

  Clustering index on dept; key is dept.

Eliminate Unneeded DISTINCTs

Query: Find employees who work in the information systems department. There should be no duplicates.

```
SELECT DISTINCT ssnum
FROM Employee
WHERE dept = 'information systems'
```

DISTINCT is unnecessary, since ssnum is a key of Employee so certainly is a key of a subset of Employee. (Note: On Sybase 4.9, I've seen the elimination of a distinct reduce the query time by a factor of 20).

Subqueries

Query: Find employee social security numbers of employees in the technical departments. There should be no duplicates.

```
SELECT ssnum
FROM Employee
WHERE dept IN (SELECT dept FROM Tech)
```

might not use the index on Employee dept in some systems. However, equivalent to:

```
SELECT DISTINCT ssnum
FROM Employee, Tech
WHERE Employee.dept = Tech.dept
```

Is DISTINCT needed?

©Dennis Shasha

DISTINCT Unnecessary Here Too

In the nested query, there were no duplicate ssnum's.

Will there be in the rewritten query?

Since dept is a key of Tech, each Employee record will join with at most one Tech tuple. So, DISTINCT is unnecessary.

Reaching

The relationship among DISTINCT, keys, and joins can be generalized.

- Call a table T *privileged* if the fields returned by the select contain a key of T.
- Let R be an unprivileged table. Suppose that R is joined on equality by its key field to some other table S, then we say that R *reaches* S.
- Now, define *reaches* to be transitive. So, if R1 reaches R2 and R2 reaches R3, then say that R1 reaches R3.

Reaches: Main Theorem

There will be no duplicates among the records returned by a selection, even in the absence of DISTINCT, if one of the following two conditions hold:

- Every table mentioned in the from line is privileged.

- Every unprivileged table reaches at least one privileged one.

Reaches: proof sketch

- If every relation is privileged, then there are no duplicates even without any qualification.

- Suppose some relation T is not privileged but reaches at least one privileged one, say R. Then the qualifications linking T with R ensure that each distinct combination of privileged records is joined with at most one record of T.

Reaches: example 1

```
SELECT ssnum
FROM Employee, Tech
WHERE Employee.manager = Tech.manager
```

The same Employee record may match several Tech records (because manager is not a key of Tech), so the Social Security number of that Employee record may appear several times.

Tech does not reach privileged relation Employee.

Reaches: example 2

```
SELECT ssnum, Tech.dept
FROM Employee, Tech
WHERE Employee.manager = Tech.manager
```

Each repetition of a given ssnum value would be accompanied by a new Tech.dept, since Tech.dept is the key of Tech.

Both relations are privileged.

Reaches: example 3

```
SELECT Student.ssnum
FROM Student, Employee, Tech
WHERE Student.name = Employee.name
AND Employee.dept = Tech.dept
```

Both Employee and Tech reach Student, though Tech does so indirectly.

Tech $\rightarrow$ Employee $\rightarrow$ Student

So no duplicates.

Correlated Subqueries

Query: Find the highest paid employees per department.

```
SELECT ssnum
FROM Employee e1
WHERE salary =
(SELECT MAX(salary)
FROM Employee e2
WHERE e2.dept = e1.dept
)
```

May search all of e2 (or all records having department value e1.dept) for each e1.

Use of Temporaries

```
SELECT MAX(salary) as bigsalary, dept
INTO temp
FROM Employee
GROUP BY dept

SELECT ssnum
FROM Employee, temp
WHERE salary = bigsalary
AND Employee.dept = temp.dept
```

Again, no need for DISTINCT, because dept is key of temp.

Abuse of Temporaries

Query: Find all information department employees with their locations who earn at least $40,000.

```
SELECT * INTO temp
FROM Employee
WHERE salary ≥ 40000

SELECT ssnum, location
FROM temp
WHERE temp.dept = 'information'
```

Selections should have been done in reverse order. Temporary relation blinded optimizer.

Better without Temporaries

```
SELECT ssnum
FROM Employee
WHERE Employee.dept = 'information'
AND salary ≥ 40000
```

Procedural Extensions to SQL

- Interactions between a conventional programming language and the database management system are expensive.

- Good to package a number of SQL statements into one interaction.

- The embedded procedural language that many systems offer includes control flow facilities such as if statements, while loops, goto's, and exceptions.

Inner Loop of Genealogical Query

```
WHILE EXISTS(SELECT * FROM Temp1)
BEGIN
  INSERT Ancestor
  SELECT * FROM Temp1;

  INSERT Temp2
  SELECT * FROM Temp1;

  DELETE Temp1 FROM Temp1;

  INSERT Temp1
  SELECT Parental.parent
  FROM Parental, Temp2
  WHERE Parental.child = Temp2.parent;

  DELETE Temp2 FROM Temp2;

END
```

Triggers

A *trigger* is a stored procedure that executes as the result of an event.

In relational systems, the event is usually a modification (insert, delete, or update) or a timing event (it is now 6 A.M.).

The trigger executes as part of the transaction containing the enabling event.

Reasons to Use Triggers

- A trigger will fire regardless of the application that enables it.

  This makes triggers particularly valuable for auditing purposes or to reverse suspicious actions, e.g. changing salary on Saturday.

- Triggers can also maintain integrity constraints e.g. referential integrity or aggregate maintenanc

- A trigger can respond to events generated by a collection of applications. May help performance.

Life without Triggers

Application which must display the latest data inserted into a table.

Without triggers, must *poll* data repeatedly.

```
SELECT *
FROM interestingtable
WHERE inserttime ≥ lasttimeIlooked + 1
```

Update lasttimeIlooked based on current time.

Poll too often and you will cause lock conflicts with input.

Poll too seldom and you will miss updates.

Triggers Can Help

An *interrupt-driven* approach is to use a trigger to send the data directly to the display application when a modification occurs.

```
CREATE TRIGGER todisplay
ON interestingtable
FOR insert AS
SELECT *
FROM inserted
```

This trigger will avoid concurrency conflicts since it will execute within the same transaction that inserts into interestingtable.

The trigger will provide new data to the display exactly when produced.

Tuning the Application Interface

Application interacts with database system via programming languages or fourth generation languages.
Examples of considerations:

- If transaction updates most of records in a table, then obtain a table lock.

  Avoids deadlocks and overhead of escalation.

- Retrieve only needed columns.

  1. Save data transfer cost.

  2. May be able to answer certain queries within an index.

Summary of Relational Tuning

1. Tune queries first:
   check query plan
   rewrite queries without changing index or table structure to avoid bad subqueries, tune temporaries, and so on

2. Establish the proper indexes (previous unit).

3. Cluster tables.

4. Consider using redundancy.

5. Revisit normalization decisions — views hide this from user.

Scenario 1

Oldsale(customernum, customercity, itemnum, quantity, date, price).

To serve the data mining needs, there are indexes on customernum, customercity and item.

Updates to Oldsale take place as a bulk load at night. Load times are very slow and the daytime performance is degenerating.

Scenario 1: What's Wrong?

The indexes are slowing down the bulk load.

The bulk load is causing overflows in the indexes.

Scenario 1: Action

Drop the indexes at night while modifying the table.

Recreate them after the load has finished. This will eliminate overflows and empty nodes.

The load should lock the entire table.

Scenario 2

Suppose you are given the following relation

Purchase(purchasenum, item, price, quantity, supplier, date)

with a clustering index on purchasenum.

You want to compute the cost of items based on a first-in first-out ordering. That is, the cost of the first purchase of an item should be accounted for before the cost of a later item.

We want to do this for all the 10,000 data items.

Processing is slow.

Scenario 2: current implementation

For each such data item :x, we return the data sorted by date. (Bind variable :x is rebound 10,000 times.)

```
SELECT *
FROM Purchase
WHERE item = :x
ORDER BY date
```

The application runs too slowly.

Scenario 2: What's Wrong?

For each data item, there is a separate sort command.

This creates significant disk accesses unless the whole table fits into main memory.

Scenario 2: Action

```
SELECT *
INTO temp
FROM Purchase
ORDER BY item, date
```

This will require only one scan and sort of Purchase instead of 10,000 scans.

Then go through temp sequentially using a 4GL or programming language.

Another possibility is to cluster by (item, date) if queries on purchasenum are infrequent.

Scenario 3

Want to audit the event that a depositor's account balance increases over $50,000. Exact amount is unimportant.

```
CREATE TRIGGER nouveauriche
ON Account
FOR update
AS BEGIN
  INSERT Richdepositor
  FROM inserted
  WHERE inserted.balance > 50000
END
```

Trigger consumes excessive resources.

Scenario 3: What's Wrong?

- Trigger will fire even if the only records affected by a modification belonged to poor depositors.

- On an update of a depositor's balance from $53,000 to $54,000, will write a depositor record into Richdepositor.

  But it is already there.

Scenario 3: Action

```
CREATE TRIGGER nouveauriche
ON Account
FOR update
AS
IF update(balance)
BEGIN
  INSERT Richdepositor
  FROM inserted, deleted
  WHERE inserted.balance >= 50000
  AND deleted.balance < 50000
  AND deleted.account_ID =
        inserted.account_ID
END
```

Tuning an Object-Oriented Database System (Optional)

Unit 5

Dennis Shasha

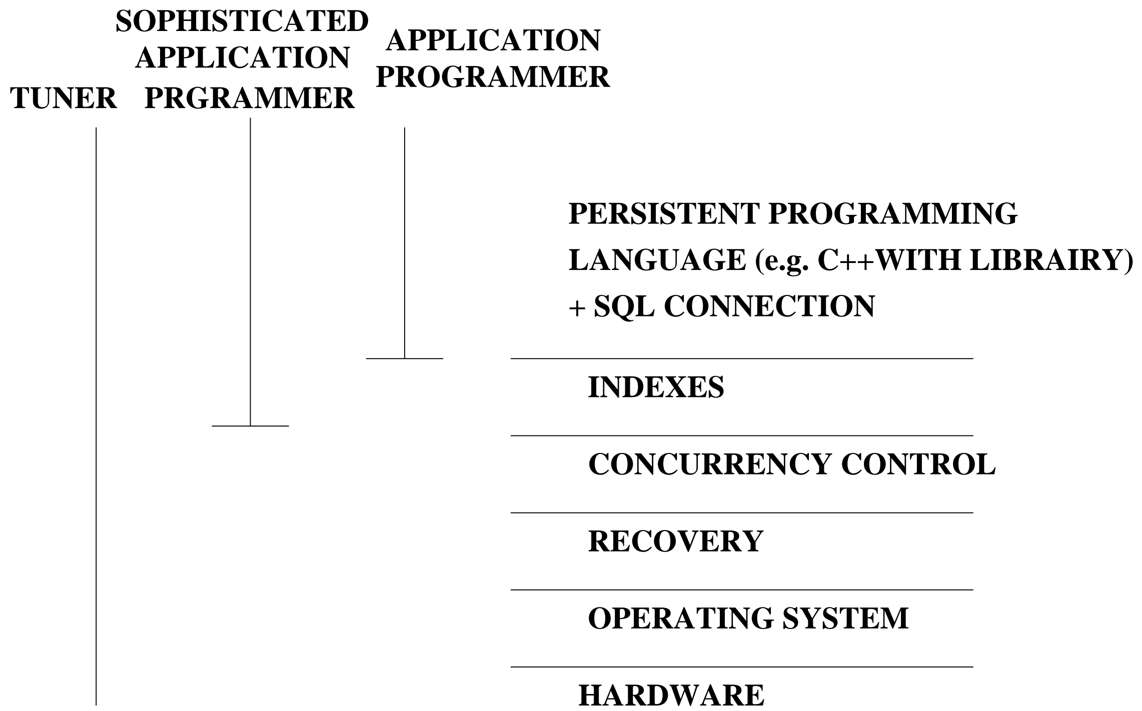
Goals

1. Identify good applications for object-oriented database systems.

2. Suggest ways to speed up object-oriented applications.

3. Study the response of relational vendors.

4. Speculate on the prospectus for such systems.



Architecture of Object-Oriented Database Systems with Responsibilities

Fig. 5.1

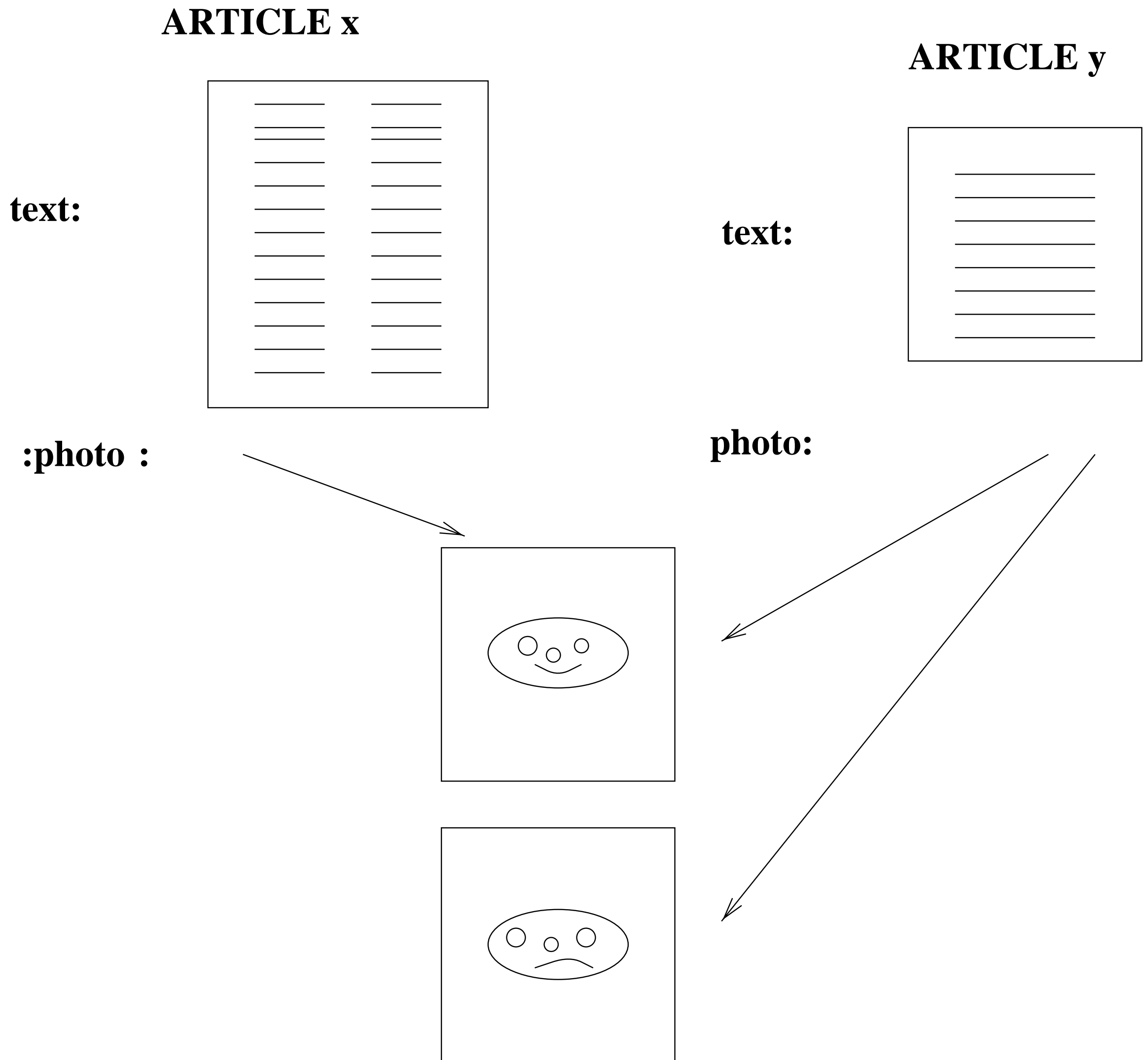
Basic Concepts

Object is a collection of data attributes, an identity and a set of operations, sometimes called *methods*.

Ex: a newspaper article object may contain zero or more photographs as constituent objects and a text value.

Operations will be to edit the article and display it at a certain width: **edit(), display(width)**.

Operations are defined by the *class* to which object belongs.



Article objects with text subvalues and photograph contituent objects.

Fig. 5.2

Languages for OODB's

In research literature, about 20.

In commercial world, C++ and Smalltalk.

(My guess is that C++ will win — cynics may say by inertia.)

Any language should:

- Allow run-time addition of classes (like addition of relational schema at runtime).

- Be compiled, at least mostly.

Claimed Benefits of OODB's

- Better platform for software engineering — all good software will change. Objects can be reused (provided they are maintained). Analogy with new model of cars (objects are fuel tanks, suspension, etc.)

- Relational type model is too poor, e.g. against sorted sequences and pointers.

- Relational operations are too weak, e.g. can't do display(width) easily.

- Impedance mismatch in relational systems — set-oriented and record-oriented languages must communicate in ugly ways, e.g. cursors.

Application Properties that Are Good for OODB's

- Not already in a database.

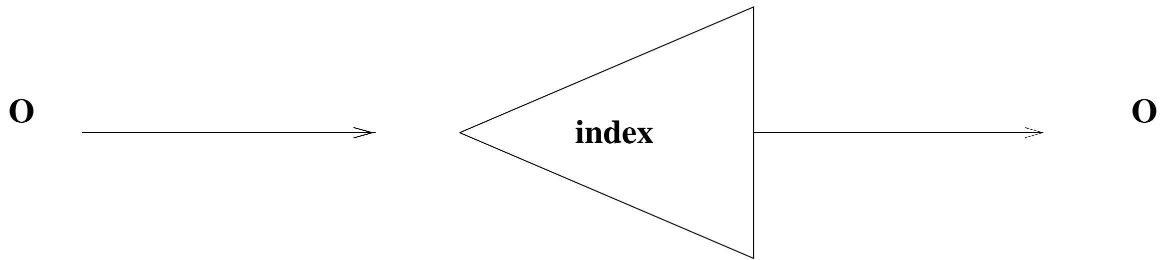
  Only 10% of computerized data is in databases.

- Application data structure is a graph. Pointer dereferencing predominates and data fits in main memory.

  Ex: design checking programs for circuits

  Reason: OODB's dereference pointers in around 10 instructions; relational systems take 1000 using foreign keys through indexes.

OODB : direct pointers in virtued access memory



RELATIONAL : indirection through indexes

Fig.5.3

A Design Applications

- A server for hypermedia documentation supporting Large machinery — for example, an airplane. The documentation integrates text, graphs, video, and voice. Typical queries:
  - what is attached to drive train?
  - what is path from X to Y?

Existing Applications Continued

- A database for a part-component explosion for an airplane manufacturer. Typical queries include:
  - Is there a path between mechanical linkage X and control panel Y going through linkage Z?
  - Change all sensors of type A in any circuit responsible for flight control to type B. (Note that this is an update that requires graph traversal. Such updates are sometimes called *complex updates*.)

Performance Features to Look For

- Standard relational-style index possibilities. Usual concurrency control and logging options (e.g. buffered commit).

- Index structures for ordered data types such as lists and sequences e.g. positional B-tree (keys are positions of members in ordering). Efficient pointer-following of data already in memory.

- Efficient on-line garbage collection in memory and on disk (because of pointers).

- User may bring data buffers and computation into client workstation. (Independent of model.)

©Dennis Shasha

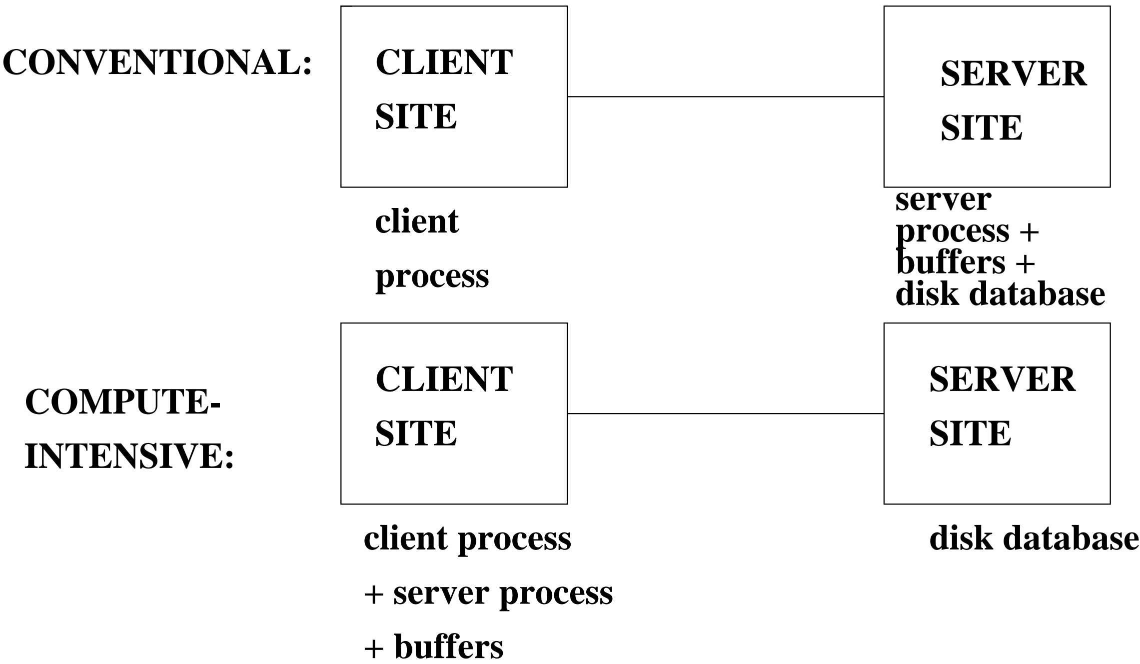


Fig. 5.4

Performance Features — 2

- Concurrency control options.
  - Check-out — hold locks over long time.
  - Gentle Aborts — create new version from updates of aborted transactions
- Pre-fetching options.
  - fetch objects as needed;
  - pre-fetch all objects reachable from an accessed object; or
  - pre-fetch a user-defined cluster upon access to any member of that cluster.
- Control layout of objects on disk.

©Dennis Shasha

Controversies Over References

- Should an object identifier = location on disk?
  - Avoids indirection.
  - Hurts ability to perform storage compaction.
- Should an object reference in virtual memory = virtual memory pointer?
  - Faster, can use standard libraries without changes.
  - Need large virtual memories.

Empirical Results in a 1993 Benchmark

On a single site benchmark done at Wisconsin, Object Store (for whom object references are virtual memory pointers) took 101.6 seconds on first invocation ("cold"), but only 6.8 seconds on second invocation.

For Ontos (for whom object references are location-independent), the cold invocation took 45.6 seconds and the warm one 12.6.

Explanation: Object Store encounters substantial overhead because of memory mapping overhead. On the other hand, using virtual memory pointers is faster than Ontos's "test if virtual memory pointer is valid, then use it else bring in target" approach.

Tuning Object Layout

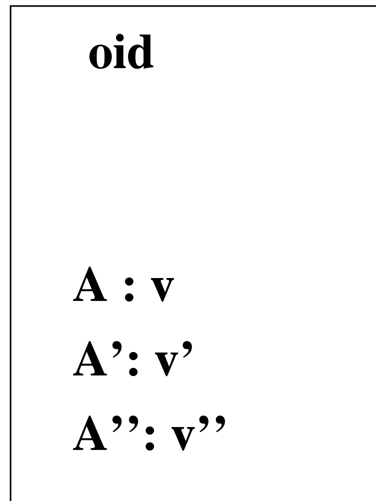
- *object attribute* = attribute whose contents (logically) are other objects.

  The object contents may be logically shared.

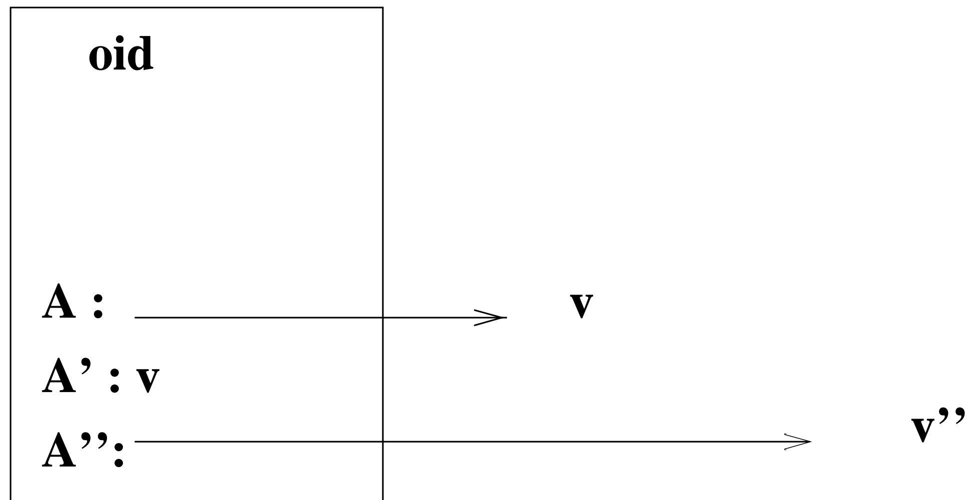
  The objects may be accessed independently.

- *value attribute* = attributes whose contents are simple values or arrays of simple values.

Question: What should be clustered (colocated) with an object and what should not be?



CLUSTERED



UNCLUSTERED FOR v AND v’’
(OCTOPUS LAYOUT)

Fig. 5.5

Rules for Object Layout

- ```
  if A is small value attribute then
     cluster the A value with o
  end if
  ```

- ```
  if A is large value attribute
   and is USUALLY accessed
     whenever o is accessed, then
           cluster
      else
           uncluster
  end if
  ```

Object Layout — subobjects

- ```
  if A is an object attribute and either
    large or
    often accessed
        independently of class C or
    widely shared then
          uncluster
      else
          cluster
  end if
  ```

  A system that loads objects by time of insertion will often cluster objects with their subobjects in any case.

  Eliminates need for explicit clustering.
  ```

Computation Layout

- If repeatedly access some set of objects X with few concurrent updates by other users of X,
  then put X in client workstation buffer

- Otherwise, (e.g. small transaction, high contention transactions, or computation-poor transactions)
  leave X in server.

Performance Results of Client Caching

Client caching between transactions can make running times of multi-transaction applications on single data approximately equal to a single transaction doing the same work. Without client caching, each new transaction must fetch pages from the server. This can hurt performance on hot starts.

On a multi-site benchmark at University of Wisconsin, Object Store (which does client caching) had multi-transaction performance within factor of 1.1 of single transaction doing the same work (no concurrency).

For Ontos (no client caching), factor was 3.5.

Caveat: all systems are under development and are constantly improving.

Revenge of the Relational Systems

Argument of IBM, Oracle, Illustra.

- SQL is widely spoken. Very often, interoperability is more important than object orientation.

- SQL can be supported efficiently.

- Can be extended with special data types each having special operations that can be accessed through select clauses.

Object-Relational Systems

- Long Fields — for image, voice and text data.

- User-defined operations — for rotating images, enlarging them. Be able to include these into select statements (Illustra, Oracle 8, etc.)

- Recursion — (too much ado) find all ancestors of Jeffrey Green.

  Can use WHILE loop of procedural extensions. New efforts to maintain transitive closures.

Illustra/Informix/UniSQL/Oracle

- Developer identifies a new type as a C structure. These can participate in a hierarchy. For example, JPEG move clip built as a sub-type of movie clip. Types can be composites of other types.

- Developer specifies methods for these types. Can inform system about costs of these methods.

- Zero or more columns in zero or more tables can be defined to have a specific type. Selects can refer to these columns with the special methods as defined.

- One can also add new index types such as R Trees.

Optimization Tricks

- Defer final select if it involves a user-defined function. e.g. select showvideo(name) where ....
- Remember expensive calculations, e.g. where mrivalue(scan) ...
- Force the plan.

Prospectus

Believer:

- Present database management systems are too slow for computationally-expensive low concurrency applications, like electrical computer design. Pointer dereferencing is essential.

- Impedance mismatch forces unnatural implement

- Objects are right for software maintenance.

My View

If OODB's

- standardize on some language, e.g. C++ or Java or some XML dialect.

- achieve high performance on several applications

- offer SQL access

they will find markets.

Users of relational systems will evolve to object-relati systems without making the leap to C++-based object-oriented systems.

Scenario 1

An electronic archival service uses a C++-based object-oriented database system to store its data.

Article objects consist of text, date, author and subject keyword value attributes, and zero or more photographs as object attributes.

Articles may be accessed by date, author, subject keywords, or (rarely) by performing a string search on the text.

What clustering strategy would you choose for the text with respect to the article objects?

Scenario 2

- Stock market application requires extensive time series analysis of financial instruments.
- Relational systems don't give order.
- Does object-relational or object-oriented make more sense?

Benchmarking (Optional)

Hazards of Choosing a System

- Insufficient transaction throughput.

- Too much transaction throughput at too high a price. (TPS on mainframe is 30 times more expensive than on a personal computer.)

- Starts off well, but becomes overwhelmed, e.g. Minitel, most intranets.

Measurement Approaches

There are lies, damn lies, and then there are benchmarks — folklore

- Application exists, then bring in equipment and/or software and try application. Simple in principle provided software is compatibl
- Application doesn't exist, so must use indirect measures. Unfortunately, database performance depends on many factors:
  - disk subsystem
  - operating system
  - algorithms

Create Your Own Benchmark

Must model: data and transactions.

For data, size is important. On toy database:

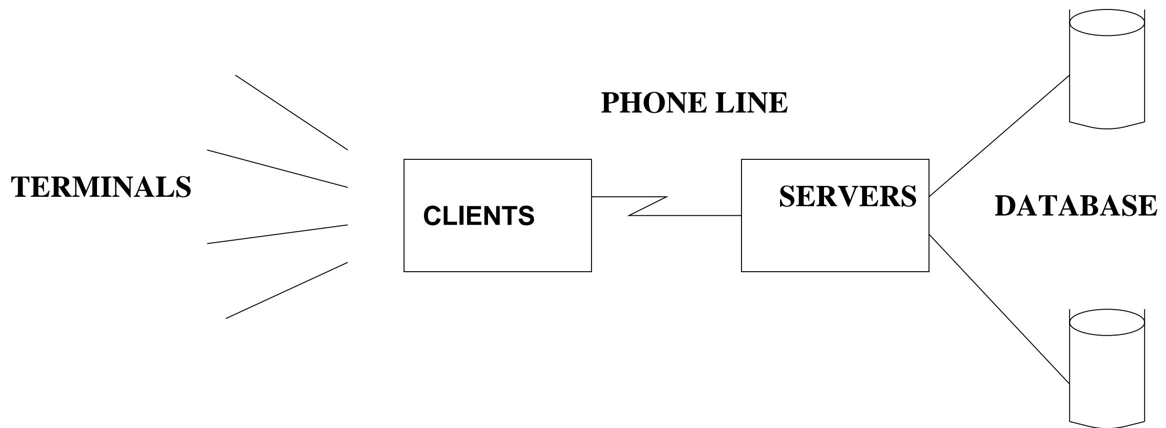
- Query optimizer may scan instead of using index.
- Buffer will give unrealistically high hit ratio.

Your Own Benchmark — transactions

For transactions, three critical features:

- Terminals — do terminals buffer characters? If not, load will be high.
- Application programming — can you safely ignore it? Yes for transaction processing, no for design checkers.
- Database accesses — how are indexes used? which accesses are updates?

Difficult and expensive to do well.



Architecture of Enterprise Client-Server

Transaction Processing System.

Fig. 6.1

Standard Benchmark

Main dimensions of choice:

- transaction length: short or long.
- transaction characteristics: update or read-only.
- interaction with application logic: none or frequent.

On-Line Transaction Processing

On-line transaction = under 10 disk accesses with little computation.

Low-end: 100 on-line transactions per second (TPS) against sub-gigabyte database.

High-end: 50,000 TPS against terabyte database.

Applications: telecommunications, airline reservatior banking, transportation, brokerage, process control, and so on.

OLTP is big business

Hardware and software for OLTP represents approximately $50 billion per year.

Standards organization: Transaction Processing Performance Council. www.tpc.org

Most mature benchmarks: TPC/A and TPC/B (now outlawed). New ones are TPC/C, TPC/D, and TPC/H.

TPC/B Benchmark Comes from Banking Deposit

1. Read 100 bytes including account id, teller id, branch id and quantity information from the terminal.
2. update a random account balance
3. write to a history file
4. update a teller balance
5. update a branch balance
6. commit the transaction
7. write 200 bytes to the terminal

Tightly Specified

Benchmark attempts to prevent unrealistic assumptions that would inflate results. Examples:

- Think time should not be constant, but based on a truncated negative exponential distribution.
- Degree 3 isolation (serializability) and tolerate failure of any disk.
- 95% of the transactions must complete within two seconds.
- Independent auditor.

Typical Cheating Techniques

- Disable statistical optimizer.
- Do redo logging only (as in Oracle 7 discrete transaction.)
- Turn off archiver so log is overwritten. Don't checkpoint.

Order Entry (TPC/C)

Proposed benchmark to simulate transactions of geographically distributed sales districts and associated warehouses.

- payment — record receipt of payment.
- order-status — determine status of an order.
- delivery batch — process 10 orders by deferred execution.
- stock-level transaction — returns information about the quantity remaining of items recently sold. Entails several joins and returns about 8,000 records.

©Dennis Shasha

Simple Scan Benchmark

To test your disk and simple scan performance, Jim Gray suggests trying a simple count query on a 1 million record table
SELECT count(*) FROM T WHERE x between (-999999, 999999)

Assuming a 10 MB/second scan rate and 100 byte records, this should take about 10 seconds. If more then either disk or disk controller is old, the operating system and database are not prefetching, the data is not clustered on disk, the record movement software is inefficient.

Parallel Scan Benchmark

n disks,
n controllers (or more),
n processors
n times as many records uniformly partitioned

same query should have same elapsed time.

Parallel Modification Benchmarks

Insert from one table to another, then replicate both tables four times.
See if log becomes a bottleneck.
See if rollback is fast.

Other Benchmarks

- How are referential integrity constraints checked?
- How do I guarantee that a secondary key is unique?
- Are triggers parallelized?
- Can I reorganize my 10 terabyte database in an hour?

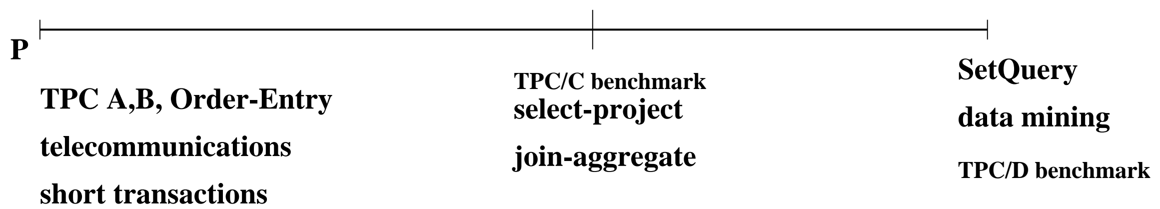
Summary

1. Application runs, then test products directly.

2. Application is on-line transaction processing, then use TPC/A or TPC/B.

3. Application is general relational, then use ASAP or TPC/C.

4. Application is decision support, then use Set Query or TPC/D or TPC/H.



Range of Applications on Relational Systems and corresponding benchmarks. For graph traversal applications, use the 007 benchmark.

Fig. 6.2

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

Wall Street Social Environment

- Very secretive, but everyone knows everything anyway (because people move and brag).
- Computers are cheap compared to people. e.g., 2 gigs of RAM is a common configuration for a server and will grow once 64 bit addressing comes in.
- Two currencies: money and fury.

Wall Street Technical Environment

- Analytical groups use APL or FAME or object-oriented systems 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 Sybase. 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.

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.

- Complaints, Kudos, and a Request

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, NY Stock Exchange), 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.

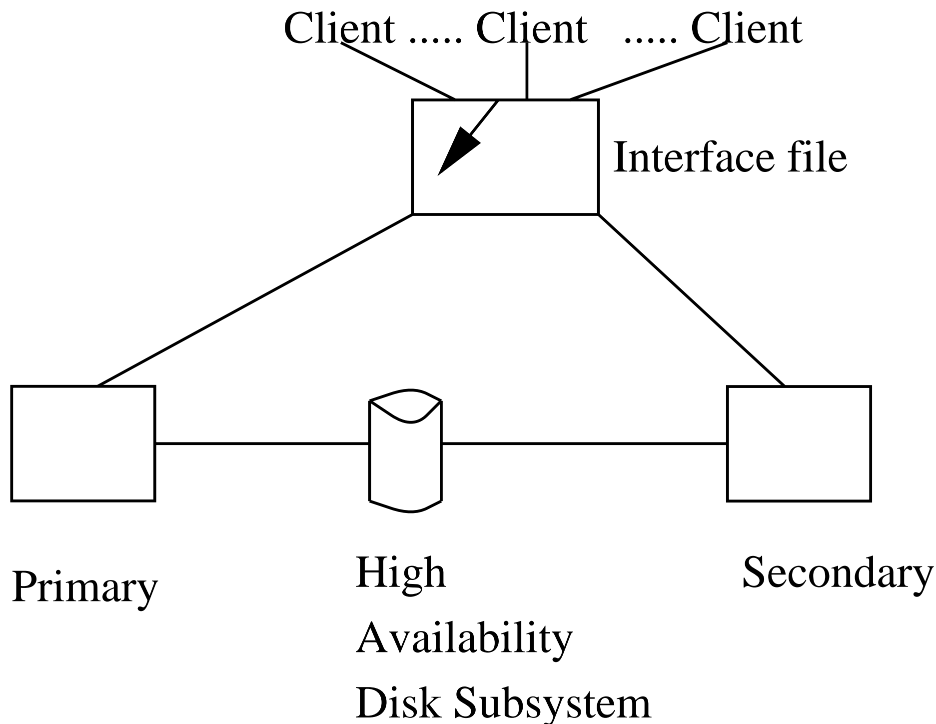
Case: Bond Futures

- 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 two-gigabyte 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.



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).

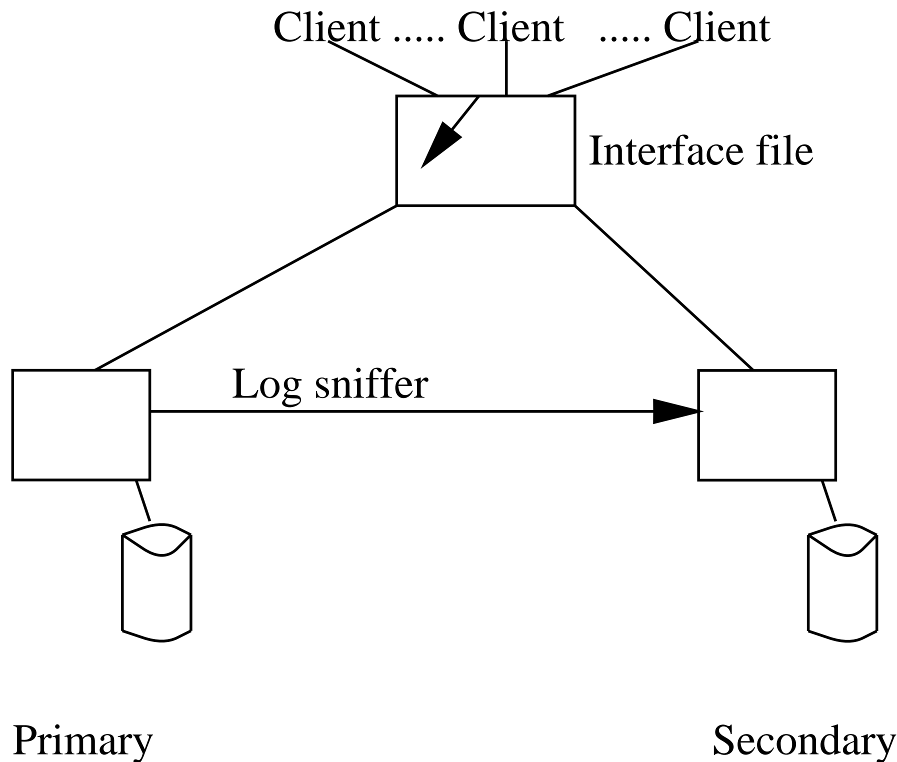
Vulnerability: High availability disk subsystem fails entirely.

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.
- Slight pain to administer, e.g. schemas, triggers, log size.



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.

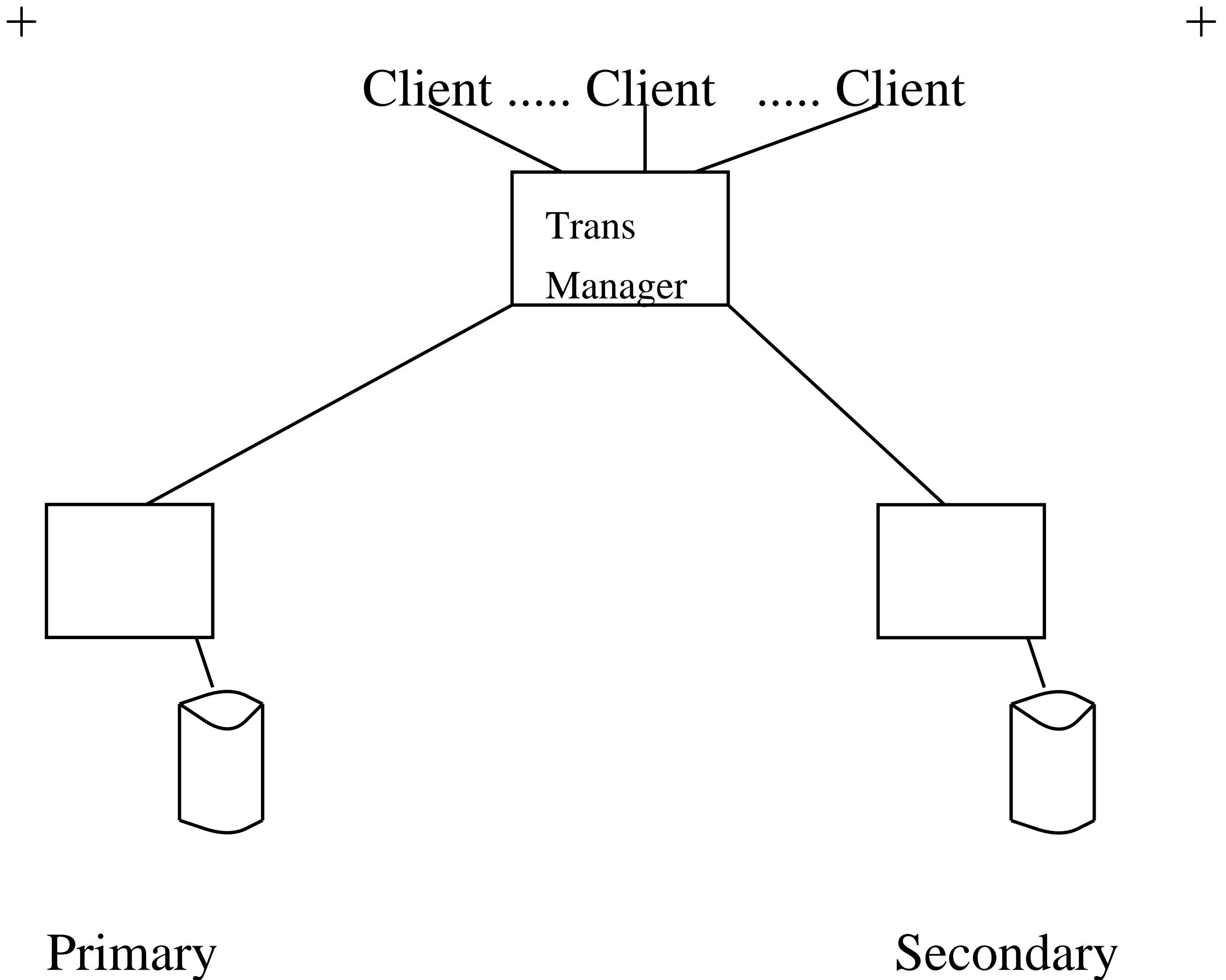
- Reliable buffering can be used (e.g. Qualix), but the net result is rep server without the ability to query the backup.

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.



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 (e.g., DEC, HP, IBM, ISIS....)

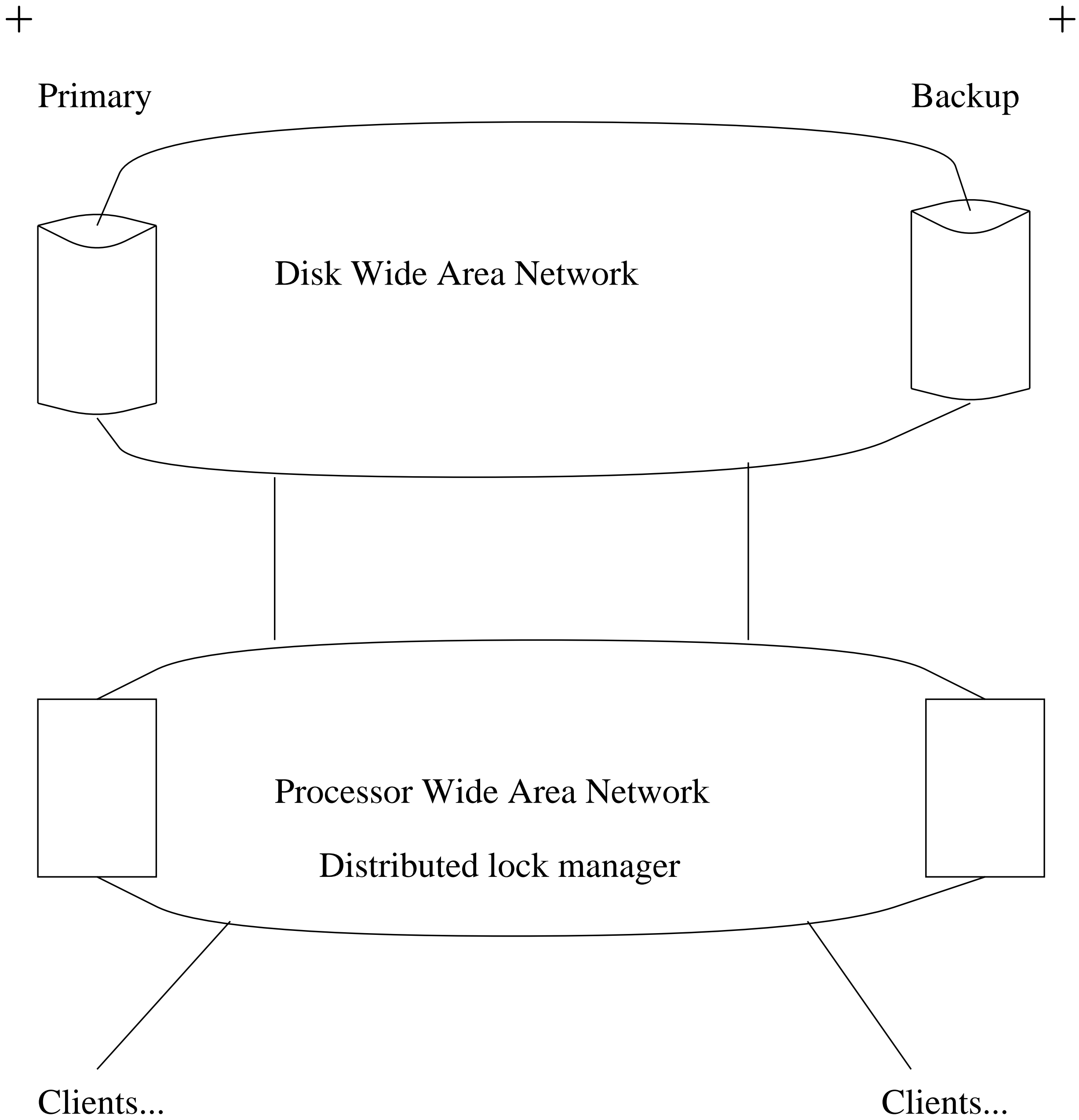
- Servers are co-equal and are interconnected via a highly redundant wide area cable.
- Clients can be connected to any server. Servers coordinate via a distributed lock manager.
- Disks are connected with the servers at several points and to one another by a second wide area link.

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.

Which to Use

- Stock exchanges use the quorum approach.

- Midoffice database servers often use dump and load or rep server. Symmetric approaches that may cause the primary to delay are too scary.

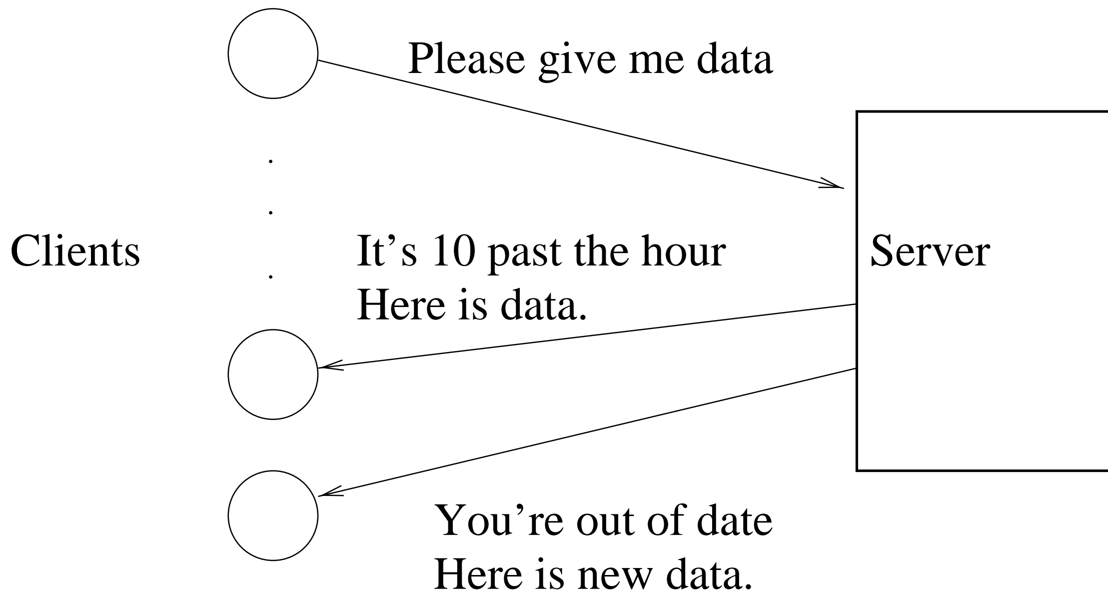
- Don't buy batches from one vendor.

Case: Indicative Data Display

- Indicative data is data that doesn't change much, e.g. payment schedules of bonds, customer information.

- Must be at a trader's fingertips.

- Relational connections to personal computers are too slow. So data is held outside the database.



The fashion is for the serves to be stateless,
but this implies that clients may have out-of-date data.
Stateful Servers are better.

What happens if two clients update concurrently?

How to Handle Updates?

- Ignore updates until the next day (used all too often).

- Program clients to request refresh at certain intervals.

- Have the server hold the state of the clients. Send messages to each client when an update might invalidate a client copy or simply refresh the screens.

Question for Vendors

- Consider data structures that are kept outside the database because it is too computationally or graphically intensive to put in the database.

- How best should you keep the data used by that application up-to-date?

- What facilities should you give to handle concurrent client updates?

Case: Interdatabase Communication

- As in many industries, financial database systems grow up as islands and then discover — surprise — they must interconnect. Source sends data to destination which then sends some confirmation.

- Replication server is a possible solution to this problem, but
(i) Commits at the source may not make it.
(ii) Responses from the destination imply two-way replication. Known to be hazardous.
(iii) Hard to determine where the data is.

Use Tables as Buffers

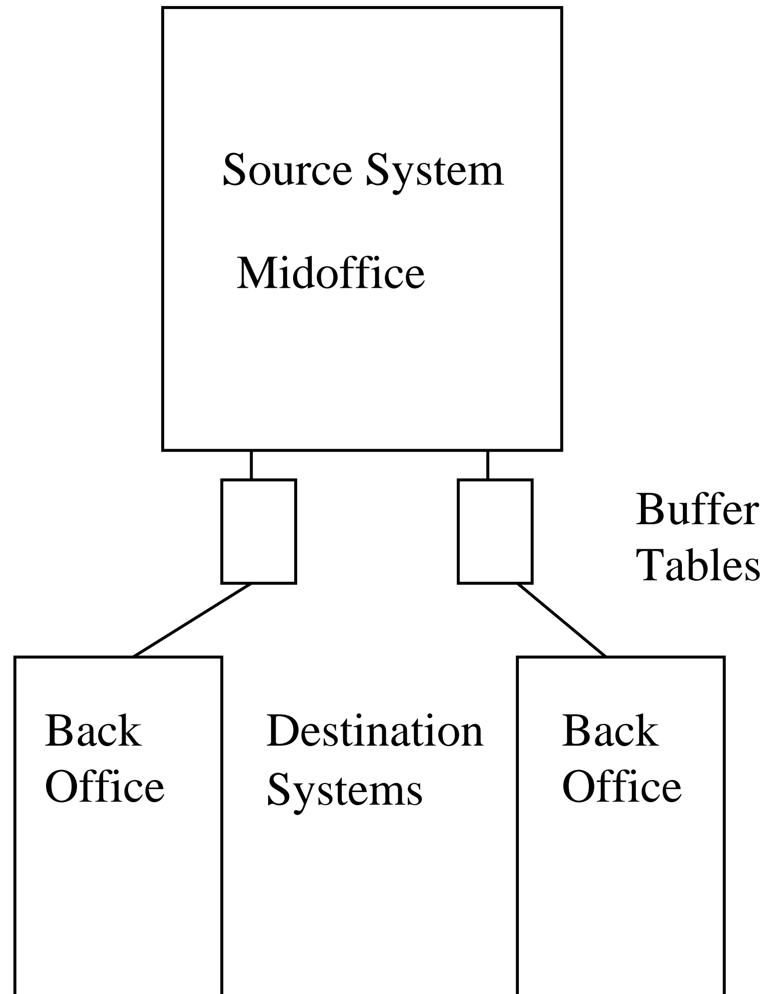
- Implement a buffer table on the source system side that holds information in the denormalized form required by the destination side.

- The destination database reads a new tuple t from the buffer table.

- After processing t, the destination database flags t as deletable or deletes t itself in the buffer table.

Overcoming Blocking

- Get blocking if source and destination scan the buffer table for update purposes.
- Approach 1: Destination database puts responses in a different table to avoid update-upd conflicts on the buffer table.
- Approach 2: Use clustering in the buffer to avoid blocking.



Source system transactions write to
buffer tables and back office systems
read from them.
If back office systems must respond,
then either cluster the buffer tables
or use a second response table written
by the back office system.

Buffer Tables

Clustering Solution Expanded

- Cluster on some hashing of a key value.

- Source writes new records to one cluster at a time.

- Destination updates records in one cluster at a time in round-robin fashion.
  No FIFO guarantee, but every sent tuple will be received.

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?

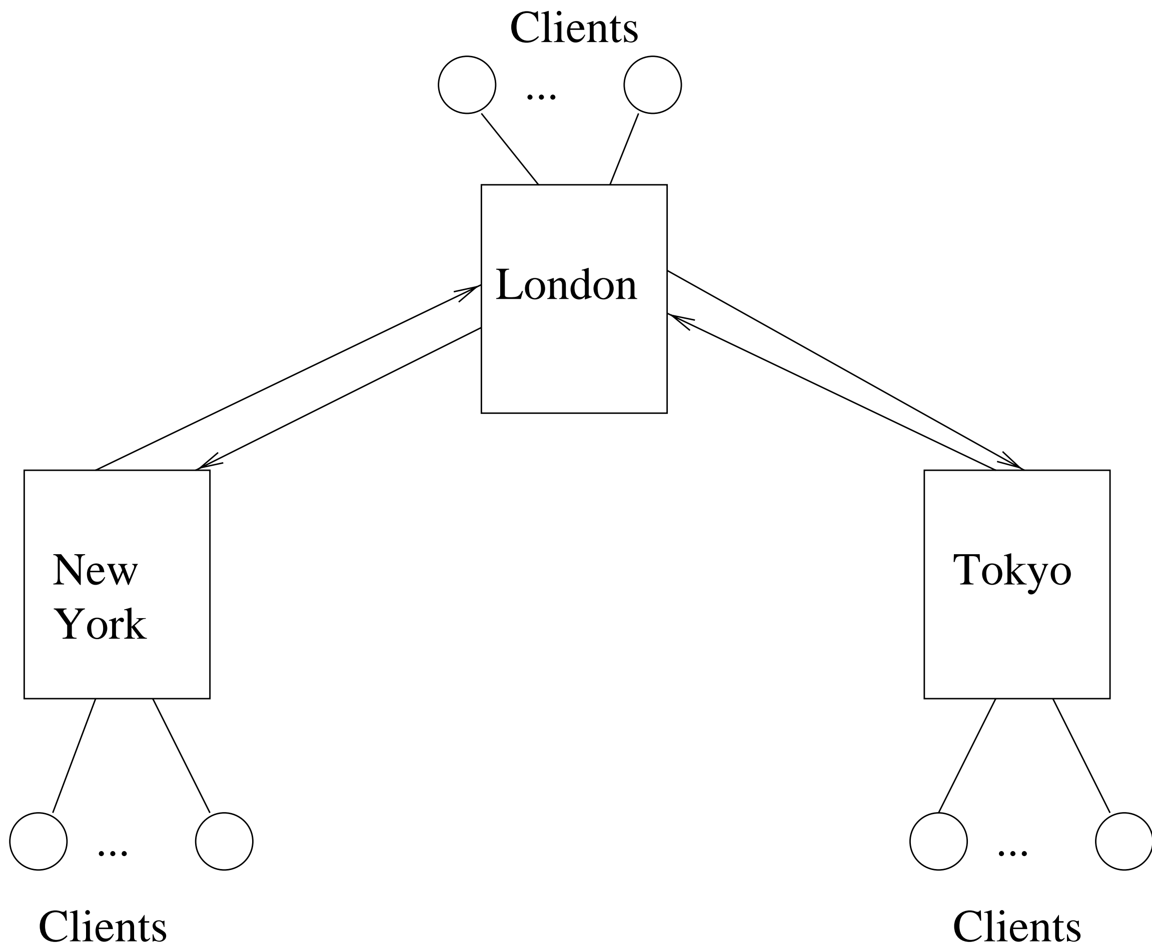
Consistency Requirements

- 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

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.

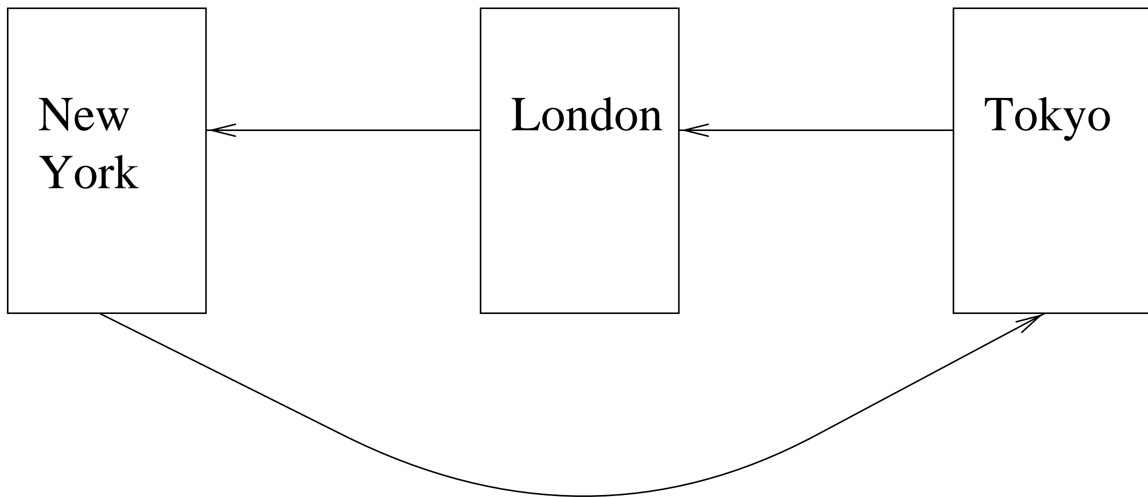
- Not all of this data will be up-to-date ("valid") at all times however. Can be used for approximate baskets.

- 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.

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.

Tuning Case: Sequential keys, clustering and blocking

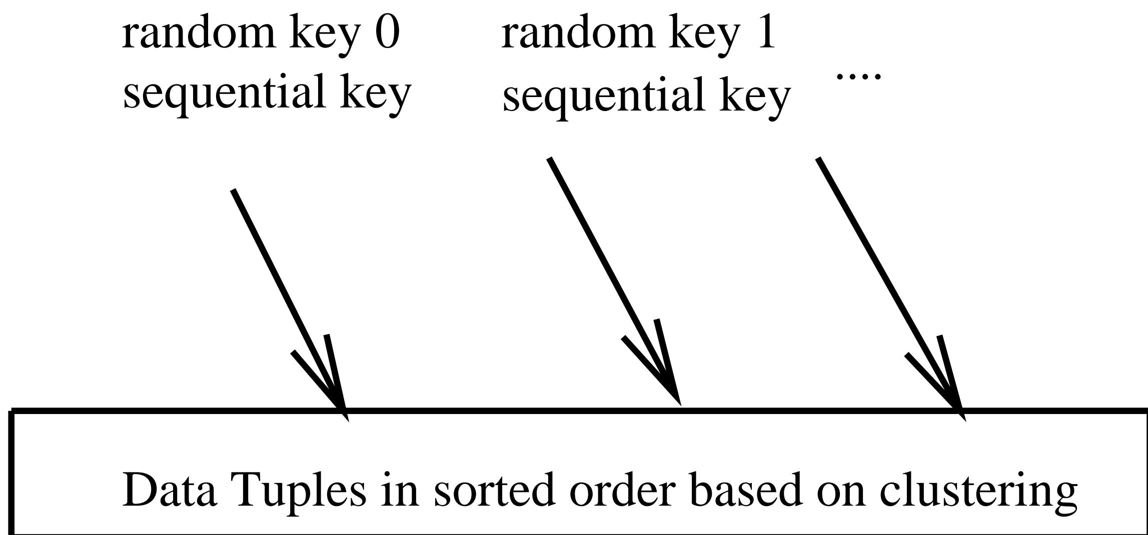
- Sequential keys (i.e., keys whose values are monotonic in time) are used to identify rows in trade and position tables uniquely. Suppose the table is clustered on a sequential key.
- Buffer behavior is good since all inserts hit the same few pages.
- Multiple concurrent inserts will conflict on the last page of a data structure or of a data page. Especially bad for page-level locking systems.

Hash Clusters

- Create a key:
  concat(hash(process id), sequential key).

- inserts cluster at as many locations as there are possible hash values.

- Good clustering without concurrency loss.



Different random key, sequential key concatenations will not conflict with one another.
They will still however give good buffering behavior since only one page per random key need be in the database cache.

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.

Query to be Tuned

```
select bond.id
from bond, deal
where bond.interestRate = 5.6
and bond.dealid = deal.dealid
and deal.date = '7/7/1996'
```

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`**.

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).

Complaints and Kudos

- It's important to know what your system does badly. For Sybase, the NOT IN subquery is particularly bad. Rewriting queries to get rid of them can reduce the number of logical IOs by a factor of 6 in cases I've seen.

- Removing DISTINCTs when they are unnecessary can improve a query's performance by 25%.

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?

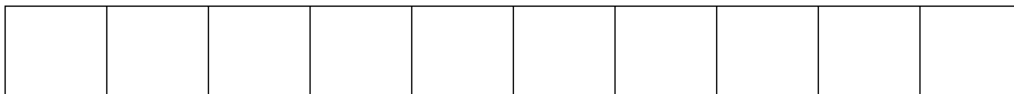
Use of Histogram

- 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.

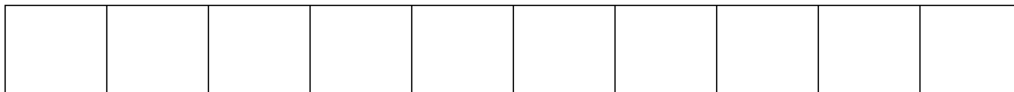
- Each cell is associated with the same number of rows (those in the cell's `date` range).

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.

RAID disks

- Raid 5 discs seem to be much faster for load applications than Raid 0, giving approximately a 60% improvement (14 minutes to 5 minutes)

- Raid 5: each disk has a portion of a sector. there are n subsector portion and a parity subsector that make up a sector. A small update will write a single subsector.

RAIDs on online transaction processing

- How Raid 5 works when updating a subsector: it must read the old version of that subsector `Sold`, read the parity subsector `Pold`, `Pnew := (Sold xor Snew) xor Pold`

- This is two reads and two writes for one write.

- Savage and Wilkes (Usenix ref) have a nice solution to this problem that involves delaying the write to the parity disk. Ted Johnson and I have a technique for making this safe.

Kudos: sample of new monitoring tools

- Average utilization of packets (if high, then bigger network packets might help).
- Why the log buffer is flushed (if before transaction completes, then perhaps make it bigger).
- Reason for task context switches. (Tells you if there is too much locking.)

My Requests to Industry

- Database systems on Wall Street require (i) an engine, (ii) a user interface, (iii) a third generation language for math functions and interprocess communication. Getting these to work together is hard.

- Most of the updatable data fits in a few gigabytes however.

- Perhaps a programming language approach is better.

Shape of this Programming Language

- Array based for time series, top ten queries etc.

- Integrated GUI, e.g. negative values of some variable turn red, newly updated values blink. This happens by defining an attribute on the variable. (Ravi Krishnamurthy proposed something like this in Sigmod 1996).

- Include interprocess communication.

Transaction Processing with a Programming Language

- Operation logging. Recovery by replaying the log from the last dump.

- Eliminate concurrency control by single threading or run-time conflict detection. Deadlocks and blocking require too much development time.

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

Summary: the main challenges

- Wall Street is different from you and me, it has more money... Also, more demands.
- High availability and reliability: hot remote backup with low probability of blocking.
- Global: must worry about distribution across WANs, where delays are significant and breakdowns

Research and Products: db system issues

- Batch cycle overlaps online activity. This results in significant blocking and requires concurrent maintenance operations (e.g. tear down and build up of indexes).

- Need a science of tuning in the spirit of Schek and Weikum's Comfort project.

- Would really like a good sizing tool: given a distributed application, what hardware and interconnection bandwidth should I buy?

Research and Products: language issues

- SQL 92 is complicated and too weak. SQL 93 and object-relational systems may fill the void.
- Bulk operations on arrays would be really useful however.
- There is a whole class of applications that would be better off without concurrency control.

References

1. The Dangers of Replication and a Solution. Jim Gray, Pat Helland, Pat O'Neil, and Dennis Shasha, Sigmod 1996. pp. 173-182

2. Is GUI Programming a Database Research Problem? Nita Goyal, Charles Hoch, Ravi Krishnamurthy, Brian Meckler, and Michael Suckow. Sigmod 1996, p. 517-528.

3. The COMFORT Automatic Tuning Project Gerhard Weikum, Christof Hasse, Axel Moenkeberg, and Peter Zabback. Information Systems, vol. 19 no. 5, July 1994.

4. Transaction Chopping: Algorithms and Performance Studies. Dennis Shasha, Francois Llirbat, Eric Simon, Patrick Valduriez *ACM Transactions on Database Systems*, October 1995, pp. 325-363.

References – Continued

5. "AFRAID — A frequently redundant array of independent disks" Stefan Savage and John Wilkes 1996 USENIX Technical Conference, January 22-26, 1996

6. *Database Tuning: a principled approach* Prentice-Hall, 1992. (Dennis Shasha) The book that got me all these gigs.