Database Tuning (part 1)

Philippe Bonnet, DIKU
bonnet@diku.dk

Joint work with Dennis Shasha, NYU
and Alberto Lerner, Google Inc.

What is Database Tuning?

Activity of making a database application run faster:

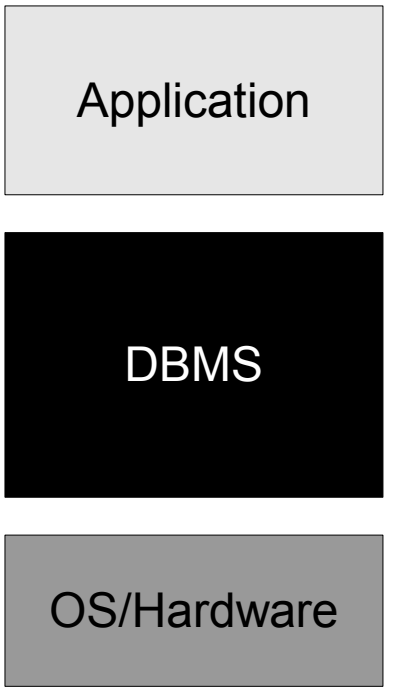
– Faster means higher throughput (or response time)

– Avoiding transactions that create bottlenecks or avoiding queries that run for hours unnecessarily is a must.

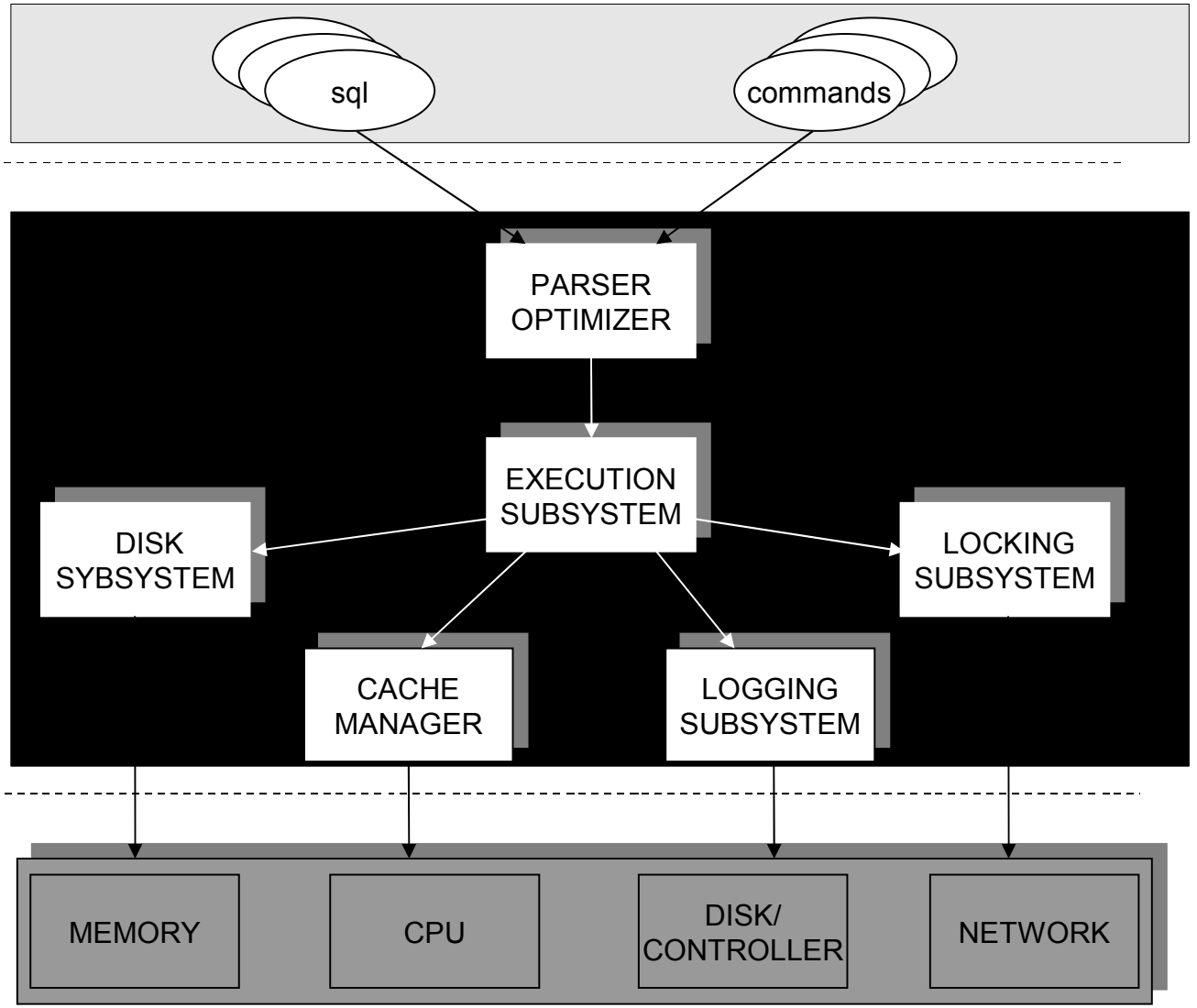
.

Why Database Tuning?

- Troubleshooting:
  - Make managers and users happy given an application and a DBMS
- Capacity Sizing:
  - Buy the right DBMS given application requirements and OS/Hardware
- Application Programming:
  - Coding your application for performance



Why is Database Tuning hard?



The following query runs too slowly

```
select *
from R
where R.a > 5;
```

What do you do?

Tutorial Objectives

1. Tuning Principles
   - Backed by experiments : How do tuning principles impact performances on my system?
2. Troubleshooting Methodology:
   - Troubleshooting (what is happening?)
   - Hypothesis formulation
     - What is the cause of the problem?
     - Apply tuning principles to propose a fix
   - Hypothesis verification (experiments)

Concurrency Control

- The CC scheduler cannot see the entire schedule:
  - Sees one request at a time and decides whether to allow it to be serviced
  - Makes conflicts explicit
    - Request lock on item before it is accessed (S,X, ...)
  - Delays operations that are conflicting with non committed operations
    - Locking protocol: 2PL

Check the following web site for documentation About IBM DB2 locks
http://publib.boulder.ibm.com/infocenter/db2help/topic/com.ibm.db2.udb.doc/admin/c0005270.htm

Tuning Course Overview

1. Tuning the guts
   a) Lock Tuning
   b) Log Tuning
   c) Storage Tuning
   d) OS Tuning
2. Schema Tuning
3. Index Tuning
4. Query Tuning
5. API Tuning
6. Troubleshooting

Tutorial
Part 1: Introduction and Lock Tuning
Part 2: Index Tuning and Troubleshooting

Concurrency Control Goals

- Performance goals
  - Reduce blocking
    - One transaction waits for another to release its locks
  - Avoid deadlocks
    - Transactions are waiting for each other to release their locks
- Correctness goals
  - Serializability: each transaction appears to execute in isolation
  - The programmer ensures that serial execution is correct.

Trade-off between correctness and concurrency

Ideal Transaction

- Acquires few locks and favors shared locks over exclusive locks
  - Reduce the number of conflicts -- conflicts are due to exclusive locks
- Acquires locks with *fine granularity*
  - Reduce the scope of each conflict
- Holds locks for a short time
  - Reduce waiting

Lock Tuning

- Transaction Chopping
  - Rewriting applications to obtain best locking performance
- Isolation Levels
  - Relaxing correctness to improve performance
- Bottlenecks
  - Using system features to circumvent bottlenecks

Example: Simple Purchases

- Purchase item I for price P
  1. If cash < P then roll back transaction (constraint)
  2. Inventory(I) := inventory(I)+P
  3. Cash := Cash – P
- Two purchase transaction P1 and P2
  - P1 has item I for price 50
  - P2 has item I for price 75
  - Cash is 100

Example: Simple Purchases

- If 1-2-3 as one transaction then one of P1, P2 rolls back.
- If 1, 2, 3 as three distinct transactions:
  - P1 checks that cash > 50. It is.
  - P2 checks that cash > 75. It is.
  - P1 completes. Cash = 50.
  - P2 completes. Cash = - 25.

Example: Simple Purchases

- Orthodox solution
  - Make whole program a single transaction
    - Cash becomes a bottleneck!
- Chopping solution
  - Find a way to rearrange and then chop up the transactions without violating serializability.

Example: Simple Purchases

- Chopping solution:
  1. If Cash < P then roll back.
     Cash := Cash – P.
  2. Inventory(I) := inventory(I) + P
- Chopping execution:
  - P11: 100 > 50. Cash := 50.
  - P21: 75 > 50. Rollback.
  - P12: inventory := inventory + 50.

Transaction Chopping

- Execution rules:
  - When pieces execute, they follow the partial order defined by the transactions.
  - If a piece is aborted because of a conflict, it will be resubmitted until it commits
  - If a piece is aborted because of an abort, no other pieces for that transaction will execute.

Transaction Chopping

- Let T1, T2, …, Tn be a set of transactions. A chopping partitions each Ti into pieces ci1, ci2, …, cik.
- A chopping of T is rollback-safe if (a)T does not contain any abort commands or (b) if the abort commands are in the first piece.

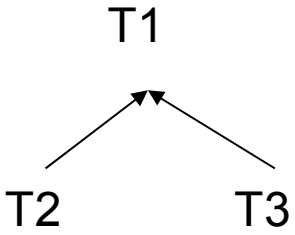
Correct Chopping

- Chopping graph (variation of the serialization graph):
  - Nodes are pieces
  - Edges:
    - C-edges: C stands for conflict. There is a C-edge between two pieces from different transactions if they contain operations that access the same data item and one operation is a write.
    - S-edges: S stands for siblings. There is an S-edge between two pieces, iff they come from the same transaction.
- A chopping graph contains an S-C cycle if it contains a cycle that includes at least one S-edge and one C-edge.

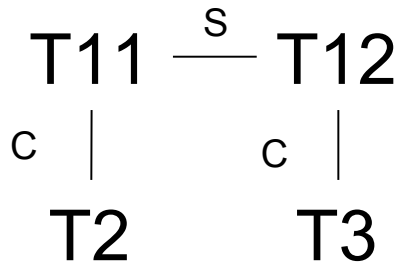
Correct Chopping

- A chopping is correct if it is rollback safe and its chopping graph contains no SC-cycle.

T1: r(x) w(x) r(y) w(y)
T2: r(x) w(x)
T3: r(y) w(y)

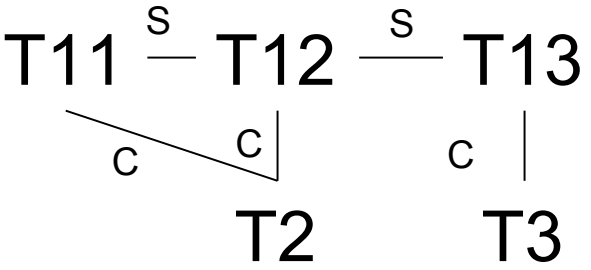


T11: r(x) w(x)
T12: r(y) w(y)



CORRECT

T11: r(x)
T12: w(x)
T13: r(y) w(y)



NOT CORRECT

Chopping Example

T1: RW(A) RW (B)

T2: RW(D) RW(B)

T3: RW(E) RW(C)

T4: R(F)

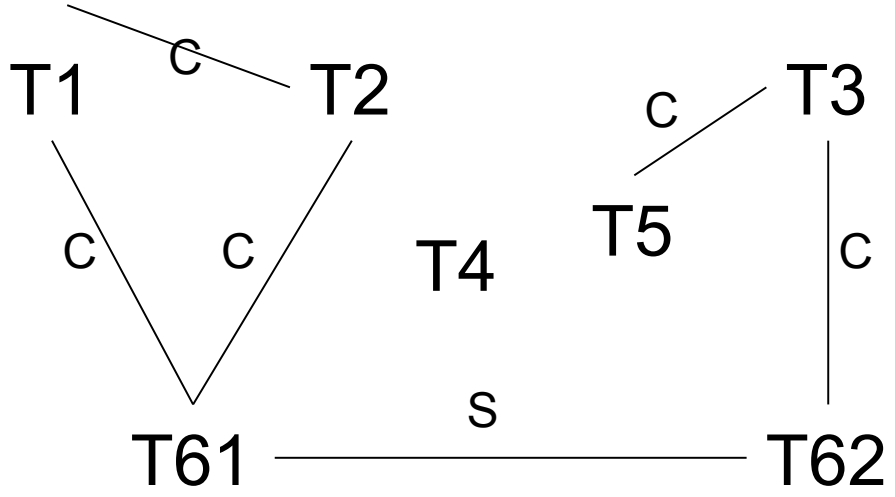
T5: R(E)

T6: R(A) R(F) R(D) R(B) R(E) R(G) R(C)

Chopping Example

T61: R(A) R(F) R(D) R(B)

T62: R(E) R(G) R(C)



Finest Chopping

- A private chopping of transaction Ti, denoted private(Ti) is a set of pieces {ci1, ci2, …, cik} such that:
  - {ci1, ci2, …, cik} is a rollback safe chopping
  - There is no SC-cycle in the graph whose nodes are {T1, …, Ti-1, ci1, ci2, …, cik, Ti+1, … Tn}
- The chopping consisting of {private(T1), private(T2), …, private(T2)} is rollback-safe and has no SC-cycles.

Finest Chopping

- In: T, {T1, .. Tn-1}
- Initialization
  - If there are abort commands
    - then p_1 := all writes of T (and all non swappable reads)that may occur before or concurrently with any abort command in T
    - else p_1 := first database access
  - P := {x | x is a database operation not in p_1}
  - P := P U {p_1}

Finest Chopping

- Merging pieces
  - Construct the connected components of the graph induced by C edges alone on all transactions {T1, …, Tn-1} and on the pieces in P.
  - Update P based on the following rule:
    - If p_j and p_k are in the same connected component and j < k, then
      - add the accesses from p_k to p_j
      - delete p_k from P

Lock Tuning

- Transaction Chopping
  - Rewriting applications to obtain best locking performance
- Isolation Levels
  - Relaxing correctness to improve performance
- Bottlenecks
  - Using system features to circumvent bottlenecks

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
- Single transaction is intolerable, because each customer would hold lock on seats available.
- Keep user interaction outside a transactional context

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

Isolation Levels

- Read Uncomitted (No lost update)
  - Exclusive locks for write operations are held for the duration of the transactions
  - Lock for writes until commit time. No locks for reads
- Read Committed (No inconsistent retrieval)
  - Lock for writes until commit time.
  - Shared locks are released as soon as the read operation terminates.
- Repeatable Read (no unrepeatable reads)
  - Strict two phase locking: lock for writes and reads until commit time.
- Serializable (no phantoms)
  - Table locking or index locking to avoid phantoms

Phantom Problem

Example: relation R (E#,name,…)

constraint: E# is key

use tuple locking

R

	E#	Name	….
o1	55	Smith	
o2	75	Jones	

T_1: Insert <99,Povl,…> into R

T_2: Insert <99,Eric,…> into R

T_1	T_2
$S_1(o_1)$	$S_2(o_1)$
$S_1(o_2)$	$S_2(o_2)$
Check Constraint	Check Constraint
⋮	⋮
Insert o_3 [99,Povl,..]	
	Insert o_4[99,Eric,..]

Solutions to the Phantom Problem

- Table locks
  - No problems
  - No concurrency
- Index locks
  - Range locking (using next Key locking)
  - More complex
  - More concurrency

Check out Mohan's ARIES KVL paper:
http://www.vldb.org/conf/1990/P392.PDF

MySQL Locking Case

```
Create table rate_type(id int, name varchar(20), rate int) type = InnoDB;
create table counter(id int primary key) type=InnoDB;
insert into counter values (1, 1);
```

T: start transaction;

```
select @id := nextkey, @name := concat('rate', nextkey), @rate := FLOOR(1000 +
RAND() * 4000) from counter  for update;
insert into rate_type values (@id, @name, @rate);
update counter set nextkey = @id + 1;
commit;
```

Using a few concurrent threads running T at read committed level, students observed the following::

```
mysql> Failed to execute:  select @id := nextkey, @name := concat('rate', nextkey),
@rate := floor(1000 + rand() * 4000) from counter for update
...
mysql> Deadlock found when trying to get lock; try restarting transaction
```

MySQL Locking Case

```
------------------------
LATEST DETECTED DEADLOCK
------------------------
061005 15:16:14
*** (1) TRANSACTION:
TRANSACTION 0 82502, ACTIVE 0 sec, OS thread id 812 starting index read
mysql tables in use 1, locked 1
LOCK WAIT 2 lock struct(s), heap size 320
MySQL thread id 24, query id 632 localhost 127.0.0.1 root Sending data
select @id := nextkey, @name := concat('rate', nextkey), @rate := FLOOR(1000 + R
AND() * 4000) from counter for update
*** (1) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 3525 n bits 72 index `GEN_CLUST_INDEX` of table
`tuning/counter` trx id 0 82502 lock_mode X locks rec but not gap waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 6; hex 0000017e4800; asc    ~H ;; 1: len 6; hex 000000014241; asc     BA
;; 2: len 7; hex 0000000de11ad2; asc     ? ?;; 3: len 4; hex 8000000a; asc     ;
;
*** (2) TRANSACTION:
TRANSACTION 0 82499, ACTIVE 0 sec, OS thread id 1644 starting index read, thread
declared inside InnoDB 500
mysql tables in use 1, locked 1
4 lock struct(s), heap size 320, undo log entries 1
MySQL thread id 3, query id 633 localhost 127.0.0.1 root Updating
update counter set nextkey = @id + 1
*** (2) HOLDS THE LOCK(S):
```

MySQL Locking Case

*** (2) WAITING FOR THIS LOCK TO BE GRANTED:
RECORD LOCKS space id 0 page no 3525 n bits 72 index `GEN_CLUST_INDEX` of table `tuning/counter` trx id 0 82499 lock_mode X waiting
Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
0: len 6; hex 0000017e4800; asc ~H ;; 1: len 6; hex 000000014241; asc BA ;; 2: len 7; hex 0000000de11ad2; asc ? ?;; 3: len 4; hex 8000000a; asc ;;

*** WE ROLL BACK TRANSACTION (1)

In other words:

- T1's select for update waits to acquire an exclusive row lock on counter.nextkey
- T2 already has an exclusive row lock on counter.nextkey and wants to acquire a table lock (update).

That is called a lock upgrade.

This causes a deadlocks because T2 cannot get the table lock while T1 waits for a row lock and T1 cannot get its lock while T2 has it.

MySQL Locking Case

The reason the updates wants a table lock is that there is no other solution (the update statement on counter has no where clause).

Two solutions:

(1) use serializable isolation level to force table locks for both select for update and update

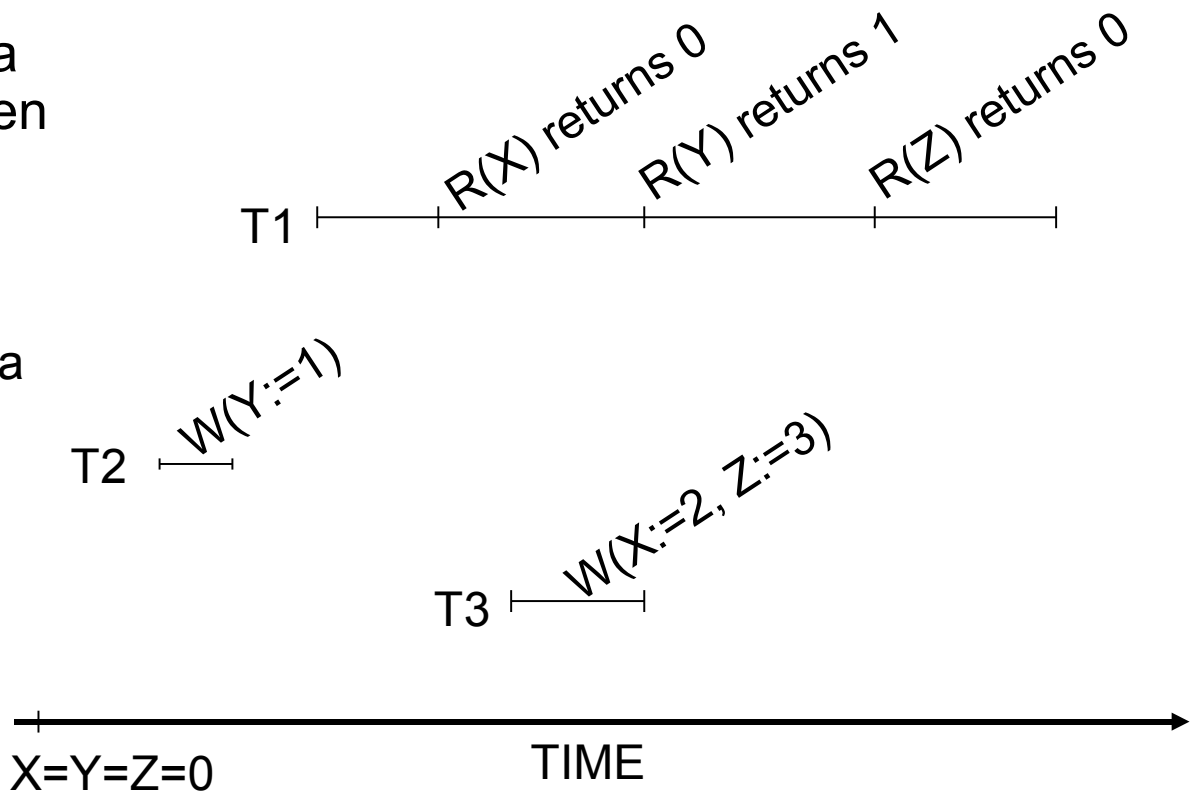
(2) use a solution that forces select for update and update to use row locks:

```
create table counter(id int not null primary key, nextkey int) type=InnoDB;
insert into counter values (1, 1);

start transaction;
select @id := nextkey, @name := concat('rate', nextkey), @rate := FLOOR(1000 + RAND() *
4000) from counter where id = 1 for update;
insert into RATE_TYPE_ADHOC values (@id, @name, @rate);
update counter set nextkey = @id + 1 where id = 1;
commit;
```

Snapshot isolation

- Each transaction executes against the version of the data items that was committed when the transaction started:
  - No locks for read
  - Locks for writes
  - Costs space (old copy of data must be kept)
- Almost serializable level:
  - T1: x:=y
  - T2: y:= x
  - Initially x=3 and y =17
  - Serial execution: x,y=17 or x,y=3
  - Snapshot isolation: x=17, y=3 if both transactions start at the same time.



Value of Serializability -- Data

Settings:

```
accounts( number, branchnum,  balance);
create clustered index c on accounts(number);
```

- 100000 rows
- Cold buffer; same buffer size on all systems.
- Row level locking
- Isolation level (SERIALIZABLE or READ COMMITTED)
- SQL Server 7, DB2 v7.1 and Oracle 8i on Windows 2000
- Dual Xeon (550MHz,512Kb), 1Gb RAM, Internal RAID controller from Adaptec (80Mb), 4x18Gb drives (10000RPM), Windows 2000.

Value of Serializability -- transactions

Concurrent Transactions:

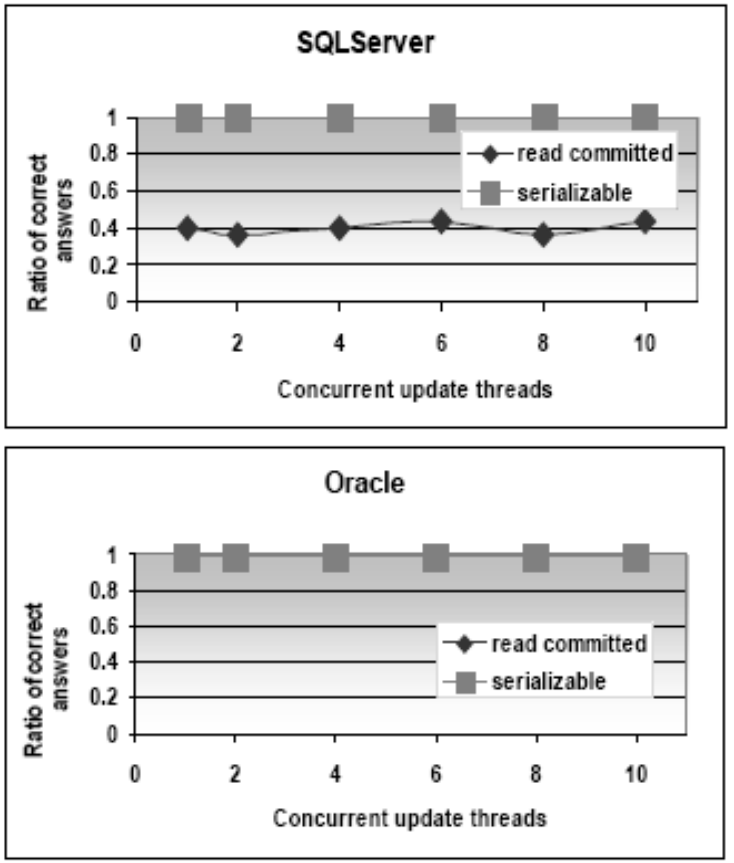
- T1: summation query [1 thread]

```
select sum(balance) from accounts;
```

- T2: swap balance between two account numbers (in order of scan to avoid deadlocks) [N threads]

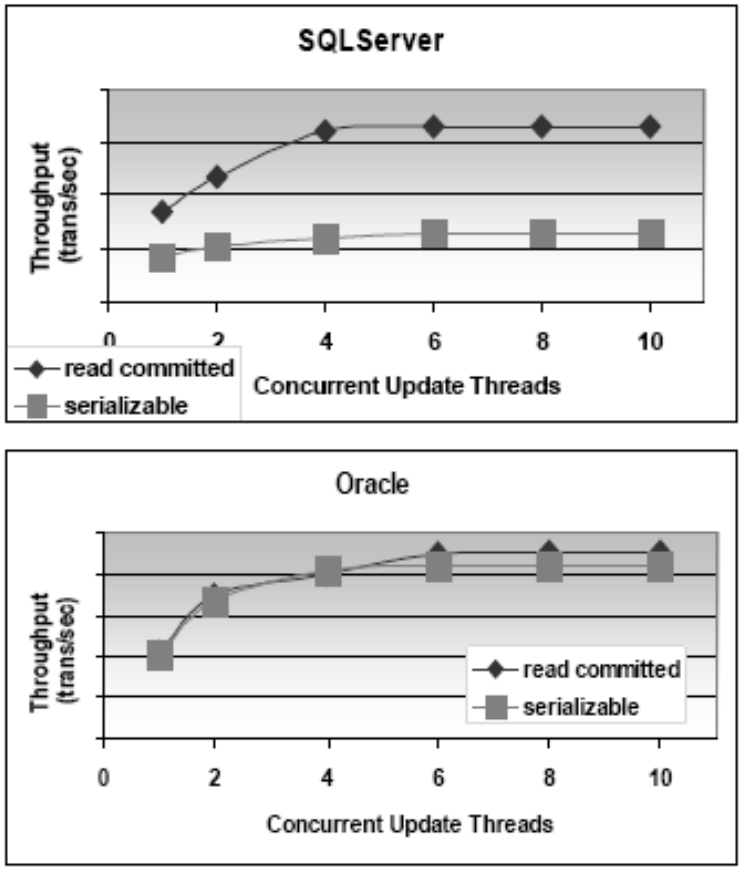
```
valX:=select balance from accounts where number=X;
valY:=select balance from accounts where number=Y;
update accounts set balance=valX where number=Y;
update accounts set balance=valY where number=X;
```

Value of Serializability -- results



- With SQL Server and DB2 the scan returns incorrect answers if the read committed isolation level is used (default setting)
- With Oracle correct answers are returned (snapshot isolation), but beware of swapping

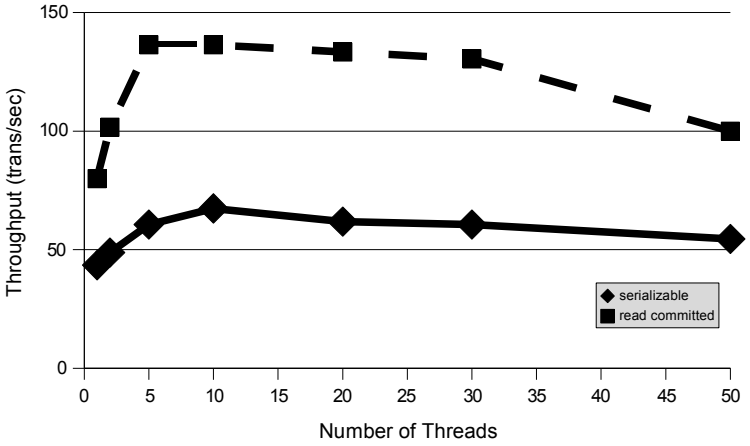
Cost of Serializability



Because the update conflicts with the scan, correct answers are obtained at the cost of decreased concurrency and thus decreased throughput.

Value of Serializability - MySQL

Read committed yields a higher throughput than serializable. This is because at the serializable isolation level, the *sum* transaction sets next-key locks while scanning the table, whereas at the read committed isolation level, the sum transaction relies on snapshot isolation, i.e., no read lock are used.



Lock Tuning

- Transaction Chopping
  - Rewriting applications to obtain best locking performance
- Isolation Levels
  - Relaxing correctness to improve performance
- Bottlenecks
  - Using system features to circumvent bottlenecks

Latches and Locks

- Locks are used for concurrency control
  - Requests for locks are queued
    - Lock and wait set associated to each data item
    - All lock elements that belong to a Transaction are associated
    - Entries from wait set promoted to lock set when possible
  - Lock data structure
    - Item id, mode, granularity,
- Latches are used for mutual exclusion
  - Requests for latch succeeds or fails
    - Active wait (spinning) on latches on multiple CPU.
  - Single location in memory
    - Test and set for latch manipulation

Locking in SQL Server 7

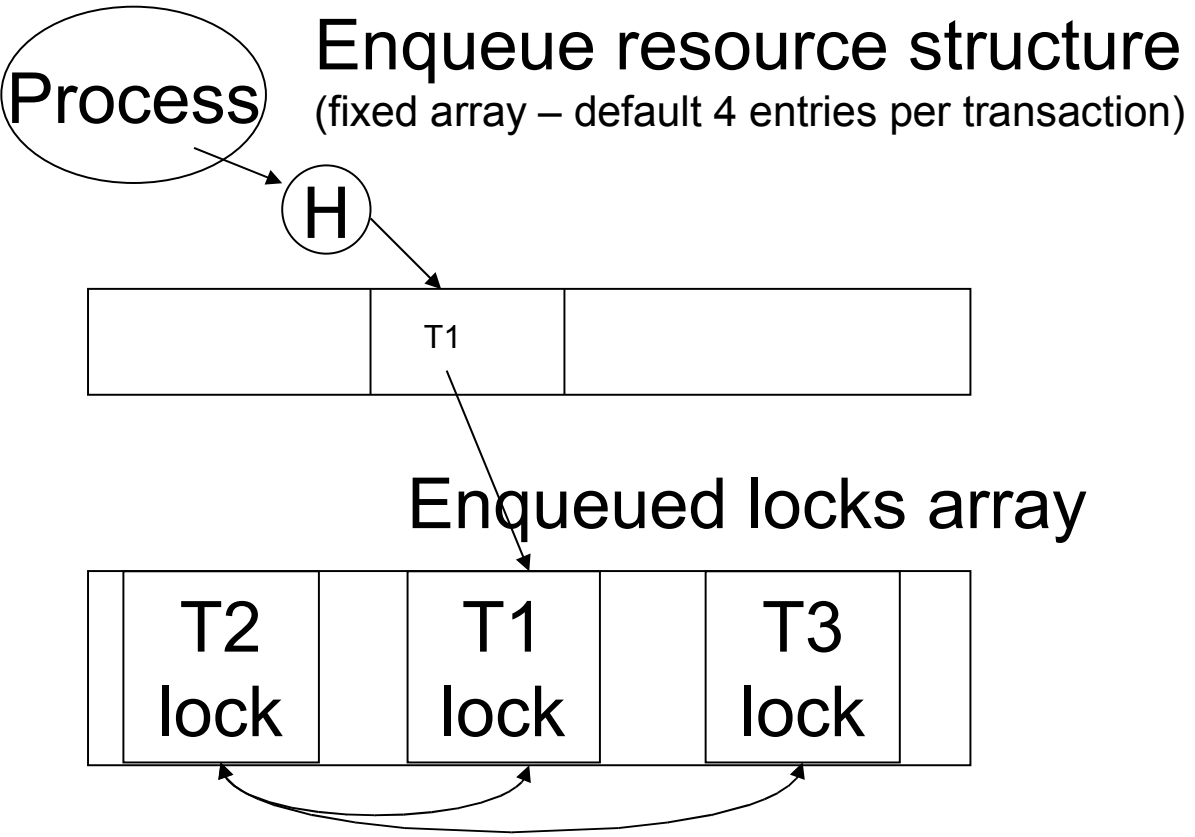
syslockinfo

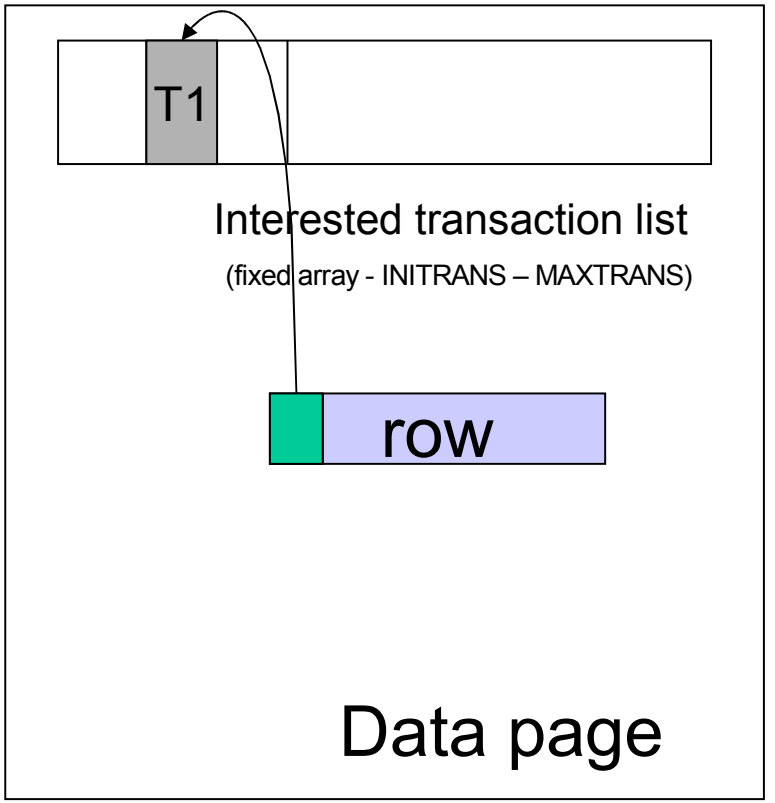
spid	dbid	objid	lock granularity	lock mode	lock owner	lock waiter
10	1	117	RID	X	LO1	LW1, LW4
10	1	117	PAG	IX	LO1	
10	1	117	TAB	IX	LO1	LW3
10	1	118	RID	S	LO2, LO3	LW2

Lock – 32 bytes

Lock owner block – 32 bytes
Lock waiter block – 32 bytes

Locking in Oracle 8i





Enqueue wait
(time out ~ 3sec)

Deadlock detection

Logical Bottleneck: Sequential Key generation

- Consider an application in which one needs a sequential number to act as a key in a table, e.g. invoice numbers for bills.
- Ad hoc approach: a separate table holding the last invoice number. Fetch and update that number on each insert transaction.
- Counter approach: use facility such as Sequence (Oracle)/Identity(SQL Server).

Counter Facility -- data

Settings:

```
accounts( number, branchnum, balance);
create clustered index c on accounts(number);

counter ( nextkey );
insert into counter values (1);
```

- default isolation level: READ COMMITTED; Empty tables
- Dual Xeon (550MHz,512Kb), 1Gb RAM, Internal RAID controller from Adaptec (80Mb), 4x18Gb drives (10000RPM), Windows 2000.

Counter Facility -- transactions

No Concurrent Transactions:

- System [100 000 inserts, N threads]
  - SQL Server 7 (uses Identity column)

```
insert into accounts values (94496,2789);
```

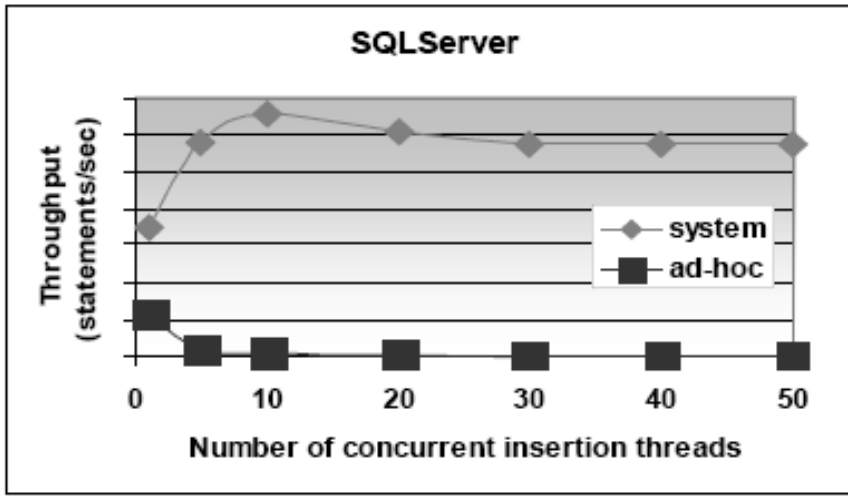
  - Oracle 8i

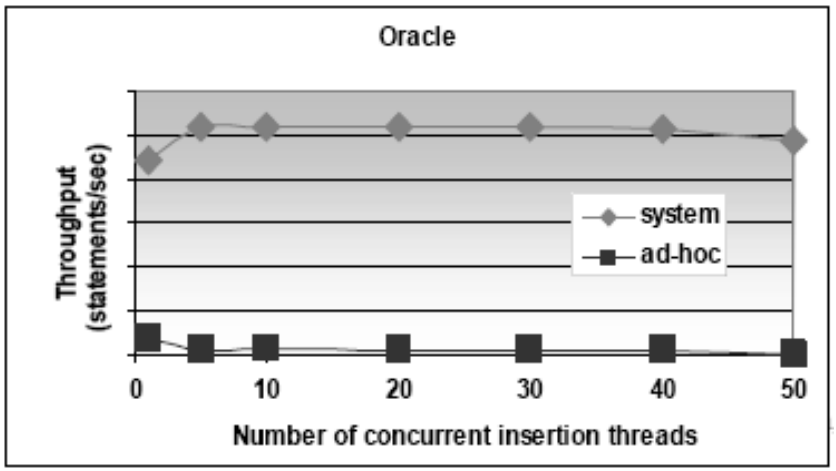
```
insert into accounts values (seq.nextval,94496,2789);
```

- Ad-hoc [100 000 inserts, N threads]

```
begin transaction
  NextKey:=select nextkey from counter;
  update counter set nextkey = NextKey+1;
commit transaction
begin transaction
  insert into accounts values(NextKey,?,?);
commit transaction
```

Avoid Bottlenecks: Counters





- System generated counter (system) much better than a counter managed as an attribute value within a table (ad hoc).
- The Oracle counter can become a bottleneck if every update is logged to disk, but caching many counter numbers is possible.
- Counters may miss ids.

Database Tuning (part 2)

Philippe Bonnet, DIKU
bonnet@diku.dk

Joint work with Dennis Shasha, NYU

EDBT Summer School 2007

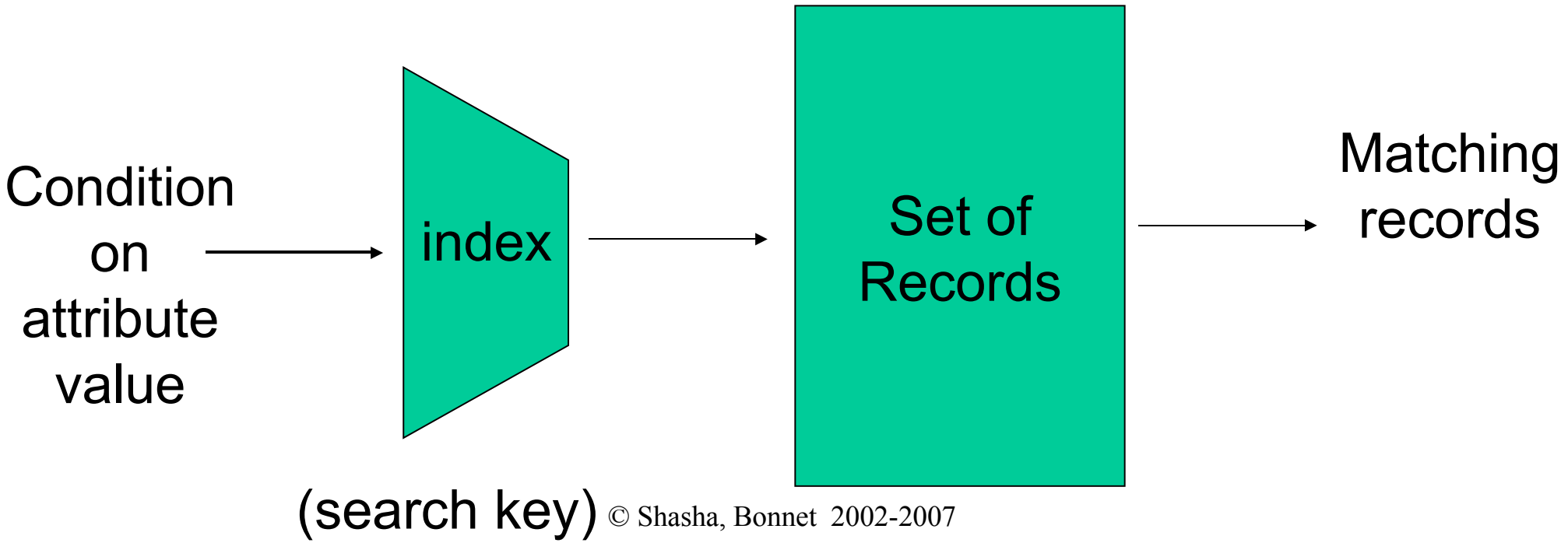
Tuning Course Overview

1. Tuning the guts
   a) Lock Tuning
   b) Log Tuning
   c) Storage Tuning
   d) OS Tuning
2. Schema Tuning
3. Index Tuning
4. Query Tuning
5. API Tuning
6. Troubleshooting

Tutorial
Part 1: Introduction and Lock Tuning
Part 2: Index Tuning and Troubleshooting

Index

An index is a data structure that supports efficient access to data



Search Keys

- A (search) key is a sequence of attributes.

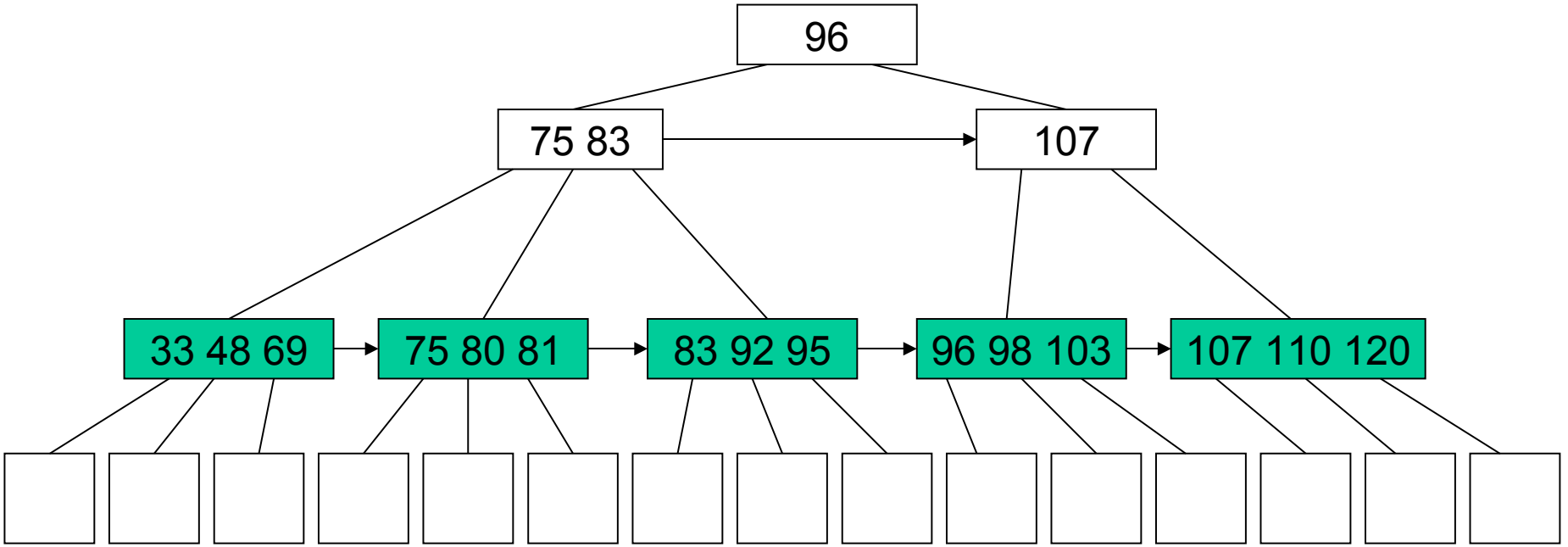
  create index i1 on accounts(branchnum, balance);
- Types of keys
  - Sequential: the value of the key is monotonic with the insertion order (e.g., counter or timestamp)
  - Non sequential: the value of the key is unrelated to the insertion order (e.g., social security number)

Data Structures

- Most index data structures can be viewed as trees.
- In general, the root of this tree will always be in main memory, while the leaves will be located on disk.
  - The performance of a data structure depends on the number of nodes in the average path from the root to the leaf.
  - Data structure with high fan-out (maximum number of children of an internal node) are thus preferred.

B+-Tree

- A B+-Tree is a balanced tree whose leaves contain a sequence of key-pointer pairs.



B+-Tree Performance

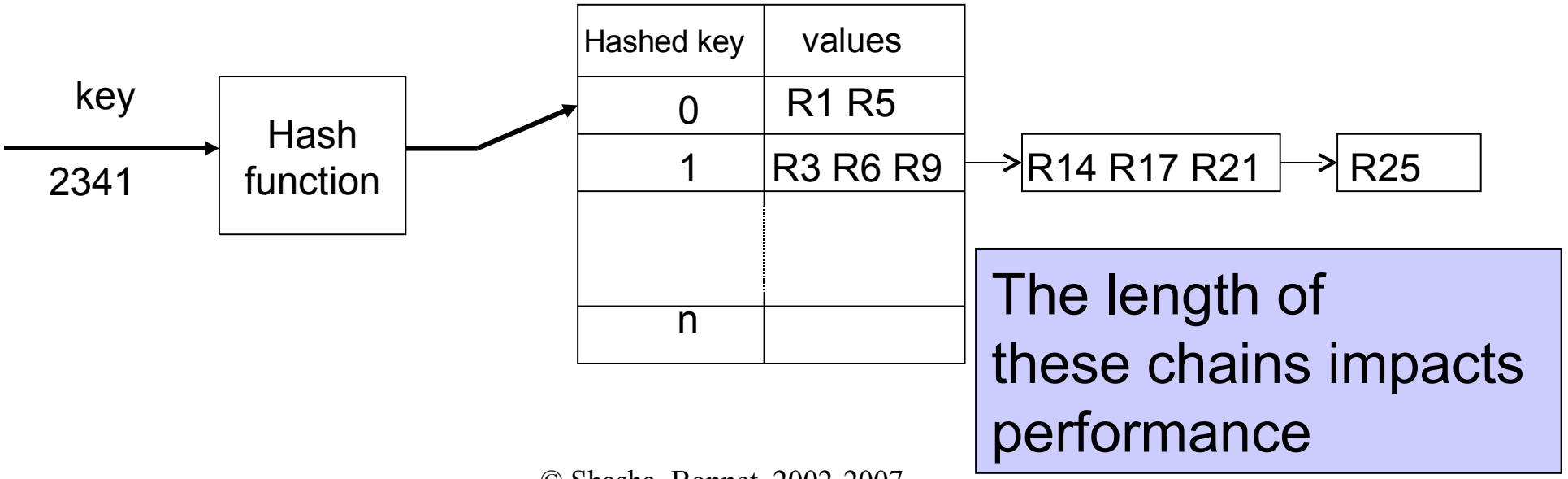
- Tree levels
  - Tree Fanout
    - Size of key
    - Page utilization
- Tree maintenance
  - Online
    - On inserts
    - On deletes
  - Offline
- Tree locking
- Tree root in main memory

B+-Tree Performance

- Key length influences fanout
  - Choose small key when creating an index
  - Key compression
    - Prefix compression (Oracle 8, MySQL): only store that part of the key that is needed to distinguish it from its neighbors: Smi, Smo, Smy for Smith, Smoot, Smythe.
    - Front compression (Oracle 5): adjacent keys have their front portion factored out: Smi, (2)o, (2)y. There are problems with this approach:
      - Processor overhead for maintenance
      - Locking Smoot requires locking Smith too.

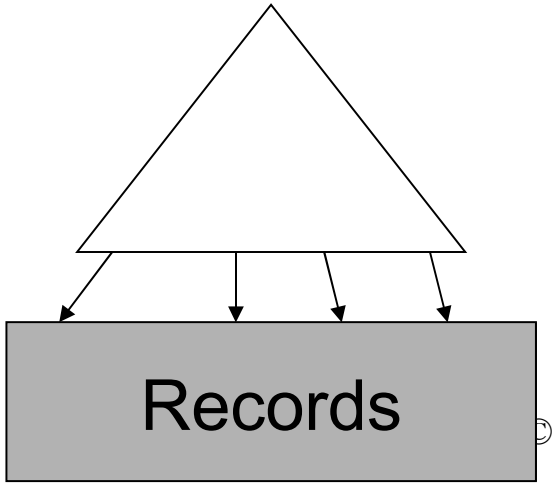
Hash Index

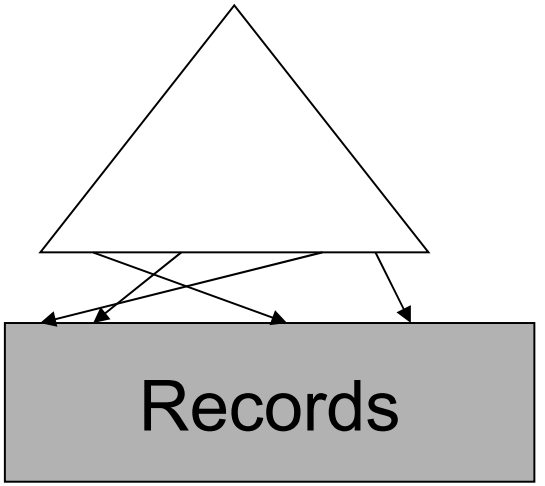
- A hash index stores key-value pairs based on a pseudo-randomizing function called a *hash function*.



Clustered / Non clustered index

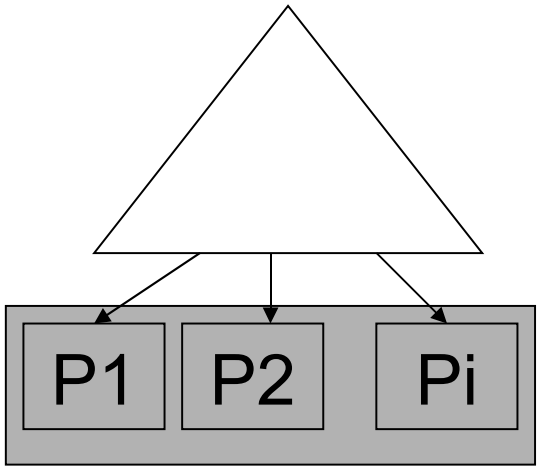
- Clustered index (primary index)
  - A clustered index on attribute X co-locates records whose X values are *near* to one another.
- Non-clustered index (secondary index)
  - A non clustered index does not constrain table organization.
  - There might be several non-clustered indexes per table.

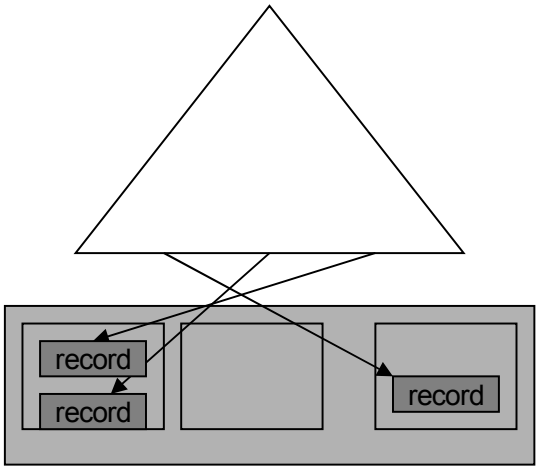




Dense / Sparse Index

- Sparse index
  - Pointers are associated to pages
- Dense index
  - Pointers are associated to records
  - Non clustered indexes are dense





Clustered Index

- Because there is only one clustered index per table, it might be a good idea to replicate a table in order to use a clustered index on two different attributes
  - Yellow and white pages in a paper telephone book
  - Low insertion/update rate

Clustered Index

Benefits of a clustered index:

1. A sparse clustered index stores fewer pointers than a dense index.
   - This might save up to one level in the B-tree index.
2. A clustered index is good for multipoint queries
   - White pages in a paper telephone book
3. A clustered index based on a B-Tree supports range, prefix, extremal and ordering queries well.

Clustered Index

4. A clustered index (on attribute X) can reduce lock contention:

   Retrieval of records or update operations using an equality, a prefix match or a range condition based on X will access and lock only a few consecutive pages of data

<u>Cost of a clustered index</u>

1. Cost of overflow pages
   - Due to insertions
   - Due to updates (e.g., a NULL value by a long string)

Non-Clustered Index

Benefits of non-clustered indexes

2. A dense index can eliminate the need to access the underlying table through covering.
   - It might be worth creating several indexes to increase the likelihood that the optimizer can find a covering index

1. A non-clustered index is good if each query retrieves significantly fewer records than there are pages in the table.
   - Point queries
   - Multipoint queries:

*number of distinct key values > c * number of records per page*

Where c is the number of pages retrieved in each prefetch

Index Implementations in some major DBMS

- SQL Server
  - B+-Tree data structure
  - Clustered indexes are sparse
  - Indexes maintained as updates/insertions/deletes are performed
- DB2
  - B+-Tree data structure, spatial extender for R-tree
  - Clustered indexes are dense
  - Explicit command for index reorganization
- Oracle
  - B+-tree, hash, bitmap, spatial extender for R-Tree
  - No clustered index until 10g
    - Index organized table (unique/clustered)
    - Clusters used when creating tables.
- MySQL/InnoDB
  - B+-Tree organized tables, R-Tree (geometry and pairs of integers)
  - Indexes maintained as updates/insertions/deletes are performed

Constraints and Indexes

- Primary Key, Unique
  - A non-clustered index is constructed on the attribute(s) that compose the primary key with the constraint that values are unique.
- Foreign Key
  - By default, no index is created to enforce a foreign key constraint.

Index Tuning

- Index data structure
- Search key
- Size of key
- Clustered/Non-clustered/No index
- Covering

Types of Queries

1. Point Query

```
SELECT balance
FROM accounts
WHERE number = 1023;
```

2. Multipoint Query

```
SELECT balance
FROM accounts
WHERE branchnum = 100;
```

1. Range Query

```
SELECT number
FROM accounts
WHERE balance > 10000;
```

2. Prefix Match Query

```
SELECT *
FROM employees
WHERE  name = 'Jensen'
       and firstname = 'Carl'
       and age < 30;
```

Types of Queries

1. Extremal Query

```
SELECT *
FROM accounts
WHERE balance =
  max(select balance from accounts)
```

2. Ordering Query

```
SELECT *
FROM accounts
ORDER BY balance;
```

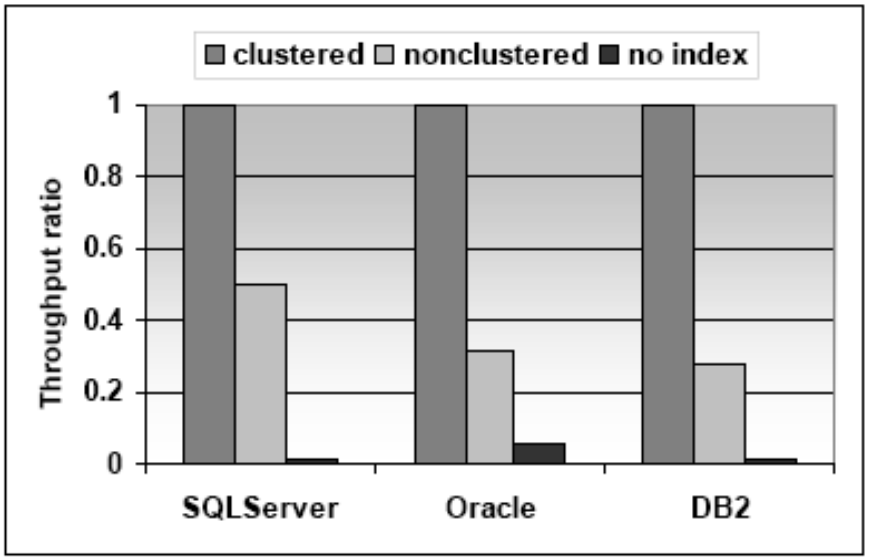
1. Grouping Query

```
SELECT branchnum, avg(balance)
FROM accounts
GROUP BY branchnum;
```

2. Join Query

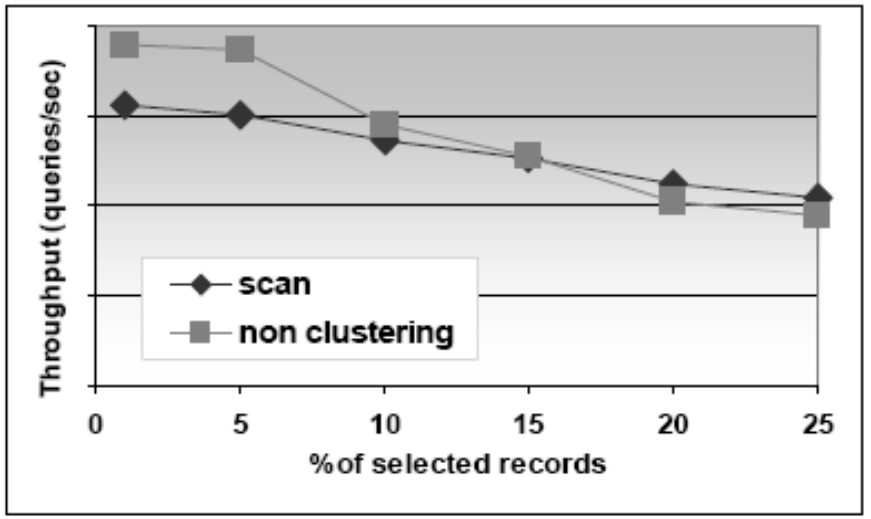
```
SELECT distinct branch.adresse
FROM accounts, branch
WHERE
  accounts.branchnum =
                branch.number
and accounts.balance > 10000;
```

Clustered Index

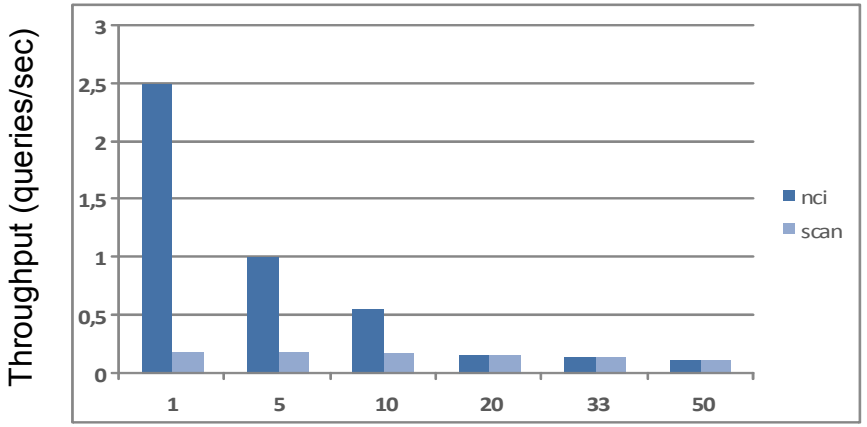


- Multipoint query that returns 100 records out of 1000000.
- Cold buffer
- Clustered index is twice as fast as non-clustered index and orders of magnitude faster than a scan.

Scan Wins More Often than Not



DB2, 10000 RPM HDD



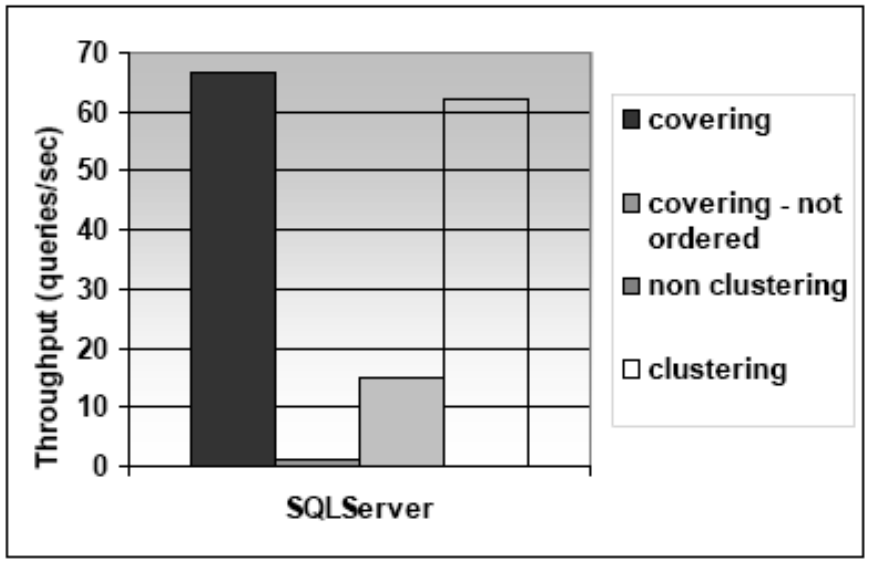
MySQL, 7500 RPM HDD

- Range Query
- We observe that if a query retrieves 10% of the records or more, scanning is often better than using a non-clustering non-covering index.
- MySQL hardcoded this rule in their optimizer: scan is used if query selectivity > 10%
- However, crossover depends on record size, table organization and HDD characteristics!

Covering Index - defined

- Select name from employee where department = “marketing”
- Good covering index would be on (department, name)
- Index on (name, department) less useful.
- Index on department alone moderately useful.

Covering Index - impact

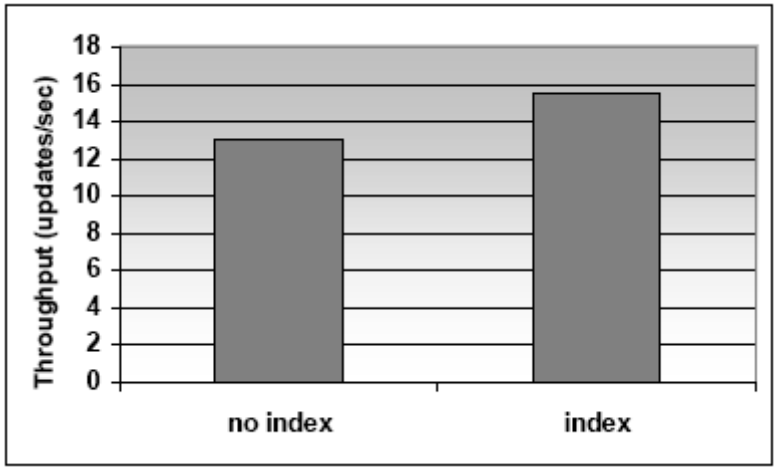


- Covering index performs better than clustering index when first attributes of index are in the where clause and last attributes in the select.
- When attributes are not in order then performance is much worse.

Index on Small Tables

- Tuning manuals suggest to avoid indexes on small tables
  - If all data from a relation fits in one page then an index page adds an I/O
  - If each record fits in a page then an index helps performance

Index on Small Tables



- Small table: 100 records
- Two concurrent processes perform updates (each process works for 10ms before it commits)
- No index: the table is scanned for each update. No concurrent updates.
- A clustered index allow to take advantage of row locking.

Index Tuning Summary

1. Use a hash index for point queries only. Use a B-tree if multipoint queries or range queries are used
2. Use clustering
   - if your queries need all or most of the fields of each records returned
   - if multipoint or range queries are asked
3. Use a dense index to cover critical queries
4. Don’t use an index if the time lost when inserting and updating overwhelms the time saved when querying

Index Tuning Wizard

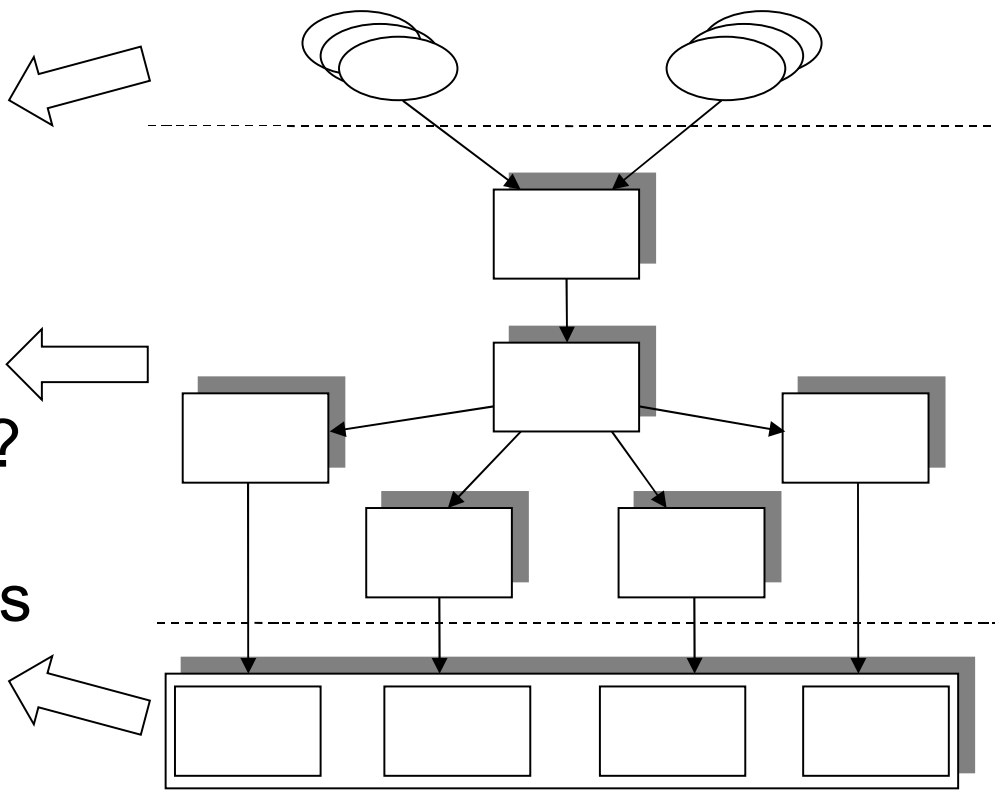
- MS SQL Server 7 and above
- In:
  - A database (schema + data + existing indexes)
  - Trace representative of the workload
- Out:
  - Evaluation of existing indexes
  - Recommendations on index creation and deletion

- The index wizard
  - Enumerates possible indexes on one attribute, then several attributes
  - Traverses this search space using the query optimizer to associate a cost to each index

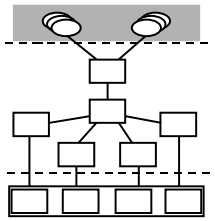
A Systematic Approach to Monitoring

Extract indicators to answer the following questions

- Question 1: Are critical queries being served in the most efficient manner?
- Question 2: Are subsystems making optimal use of resources?
- Question 3: Are there enough primary resources available?



Investigating High Level Consumers

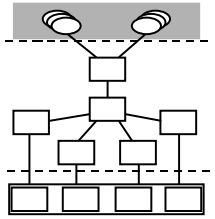


- Answer question 1:

  "Are critical queries being served in the most efficient manner?"

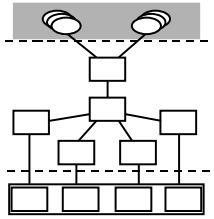
1. Identify the critical queries
2. Analyze their access plans
3. Profile their execution

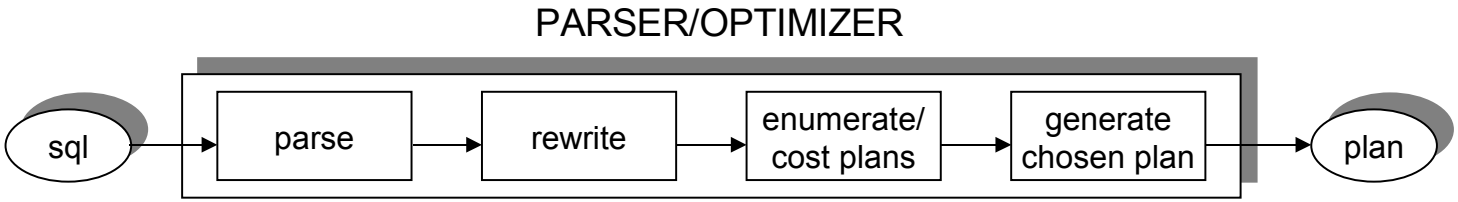
Event Monitors to Identify Critical Queries



- User complains
- Capture usage measurements at end of each query and then sort by usage x time.
- Less overhead than other type of tools because indicators are usually by-product of operations monitored and are accessed in a convenient time
- Typical measures include CPU used, IO used, locks obtained etc.

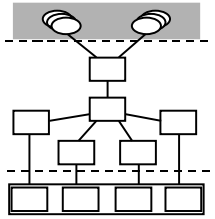
Diagnose Expensive Queries: analyzing access plans





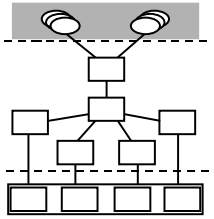
- SQL commands are translated into an internal executable format before they are processed
- After parsing, the optimizer enumerates and estimates the cost of a number of possible ways to execute that query
- The best choice, according to the existing statistics, is chosen

Access Plan Explainers to Analyze Plans

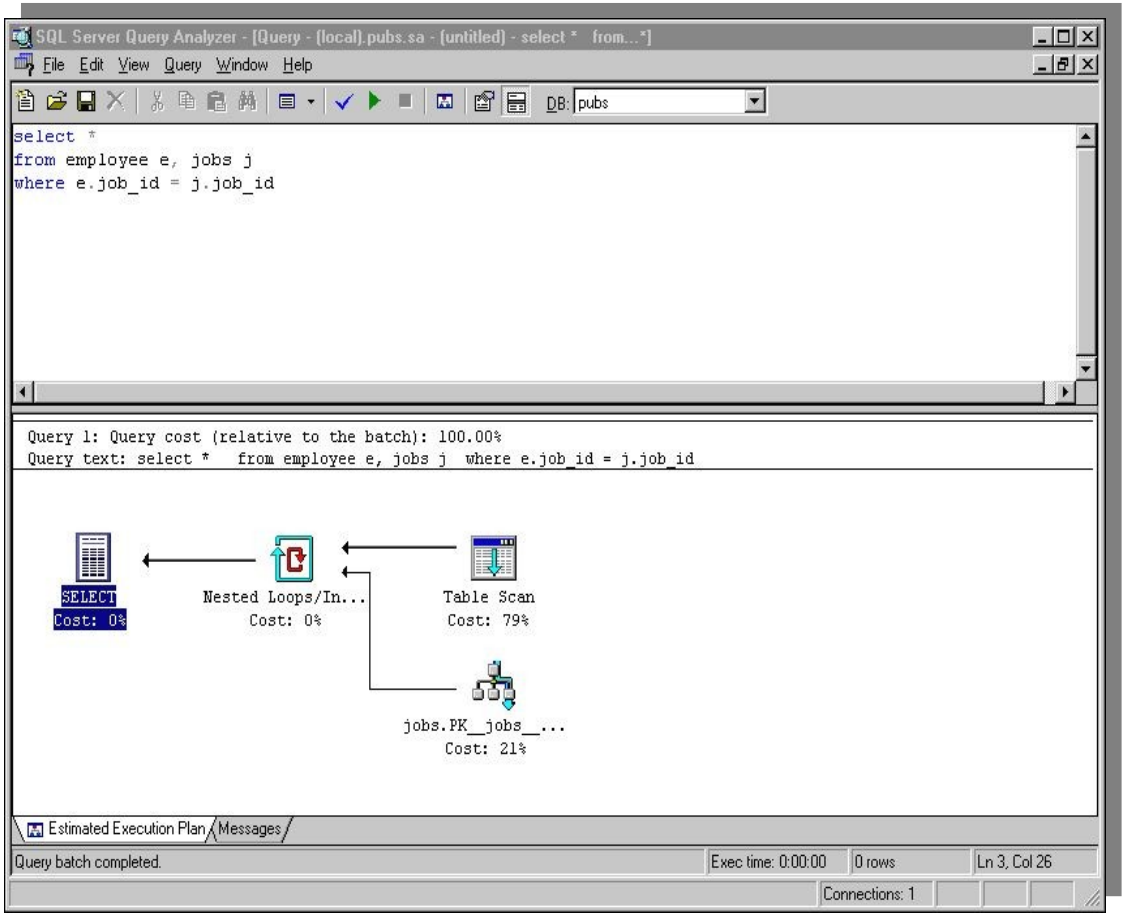


- Explainers usually depict an access plan as a (graphical) single-rooted tree in which sources at the leaf end are tables or indexes, and internal nodes are operators
- These form an assembly line (actually tree) of tuples!
- Most explainers will let you see the estimated costs for CPU consumption, I/O, and cardinality of each operator. If you see something strange, change an index.

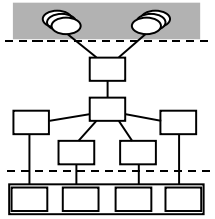
An example Plan Explainer



- Access plan according to SQLServer Query Analyzer
- Similar tools: DB2's Visual Explain and Oracle's SQL Analyze Tool



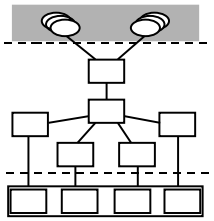
Finding Strangeness in Access Plans



What to pay attention to on a plan

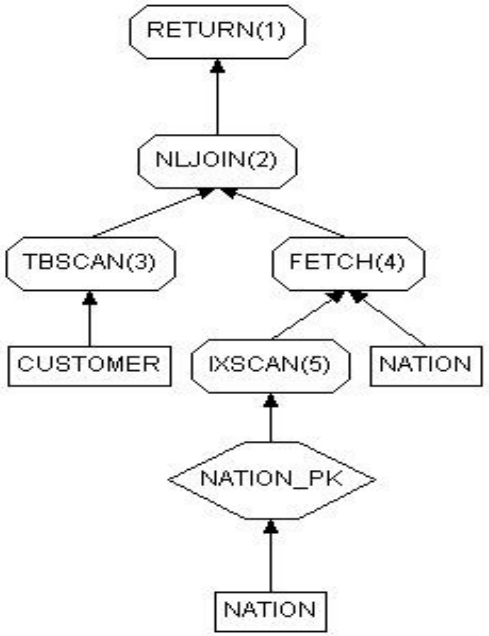
- Access paths for each table
- Sorts or intermediary results
- Order of operations
- Algorithms used in the operators

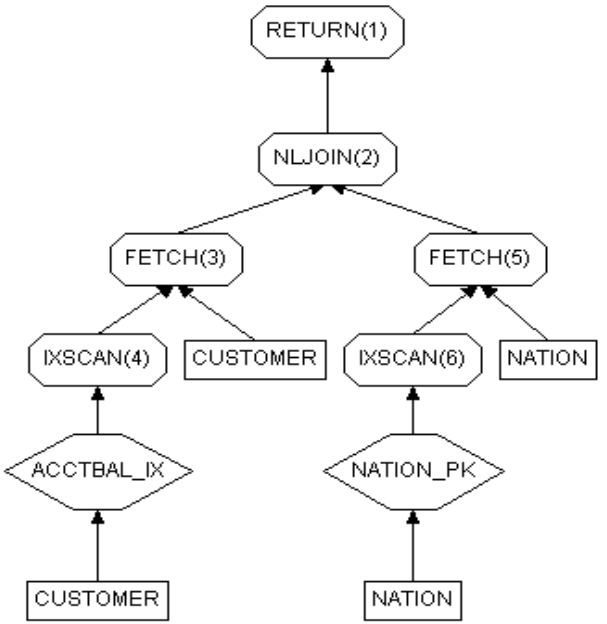
To Index or not to index?



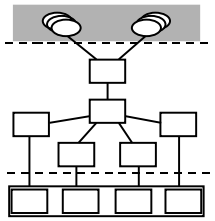
select c_name, n_name from CUSTOMER join NATION on c_nationkey=n_nationkey where c_acctbal > 0

Which plan performs best?
(nation_pk is an non-clustered index over n_nationkey, and similarly for acctbal_ix over c_acctbal)





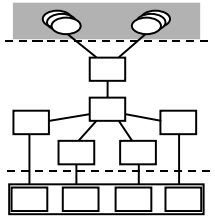
Non-clustering indexes can be trouble



For a low selectivity predicate, each access to the index generates a random access to the table – possibly duplicate! It ends up that the number of pages read from the table is greater than its size, i.e., a table scan is way better

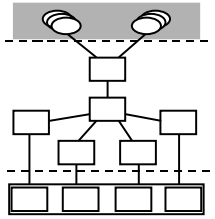
	Table Scan	Index Scan
CPU time	5 sec	76 sec
data logical reads	143,075 pages	272,618 pages
data physical reads	6,777 pages	131,425 pages
index logical reads	136,319 pages	273,173 pages
index physical reads	7 pages	552 pages

Profiling a Query's Execution



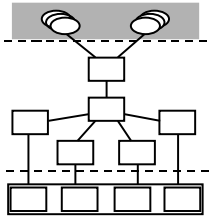
- A query was found critical but its plan looks okay. What's going on? Put it to run. (We just did in the previous example.)
- Profiling it would determine the quantity of the resources used by a query and assessing how efficient this use was
- Resources
  - DBMS subsystems: cache, disk, lock, log
  - OS raw resources: CPU

Performance Monitors to Profiling Queries



- Access or compute performance indicators' values at any time
- Many, many flavors
  - Generic (all indicators) or Specific (indicators of a given subsystem or a given query)
  - Aggregated counters vs. time-base
  - Snapshot, Continuous, or Alarm modes
  - Textual or Graphical

An example Performance Monitor (query level)



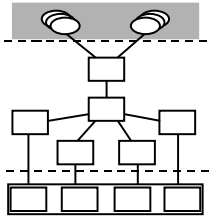
- Details of buffer and CPU consumption on a query's report according to DB2's Benchmark tool
- Similar tools: MSSQL's SET STATISTICS switch and Oracle's SQL Analyze Tool

```
Statement number: 1
select C_NAME, N_NAME
from DBA.CUSTOMER join DBA.NATION on C_NATIONKEY = N_NATIONKEY
where C_ACCTBAL > 0

Number of rows retrieved is:   136308
Number of rows sent to output is:   0
Elapsed Time is:          76.349     seconds

…
Buffer pool data logical reads             = 272618
Buffer pool data physical reads            = 131425
Buffer pool data writes                    = 0
Buffer pool index logical reads            = 273173
Buffer pool index physical reads           = 552
Buffer pool index writes                   = 0
Total buffer pool read time (ms)           = 71352
Total buffer pool write time (ms)          = 0
…
Summary of Results
==================
                 Elapsed          Agent CPU       Rows      Rows
Statement #     Time (s)          Time (s)        Fetched   Printed
1                76.349            6.670          136308    0
```

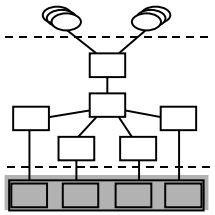
An example Performance Monitor (session level)



- Oracle 10g time based profile

Response Time Component	Duration		# Calls	Dur/Call
SQL*Net message from client	984.0s	49.6%	95,161	0.010340s
SQL*Net more data from client	418.8s	21.1%	3,345	0.125208s
db file sequential read	279.3s	14.1%	45,084	0.006196s
CPU service	248.7s	12.5%	222,760	0.001116s
unaccounted-for	27.9s	1.4%		
latch free	**23.7s**	**1.2%**	**34,695**	**0.000683s**
log file sync	1.1s	0.1%	506	0.002154s
SQL*Net more data to client	0.8s	0.0%	15,982	0.000052s
log file switch completion	0.3s	0.0%	3	0.093333s
enqueue	0.3s	0.0%	106	0.002358s
SQL*Net message to client	0.2s	0.0%	95,161	0.000003s
other	0.2s	0.0%		
Total	1,985.4s	100.0%		

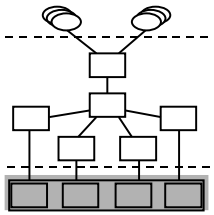
Investigating Primary Resources

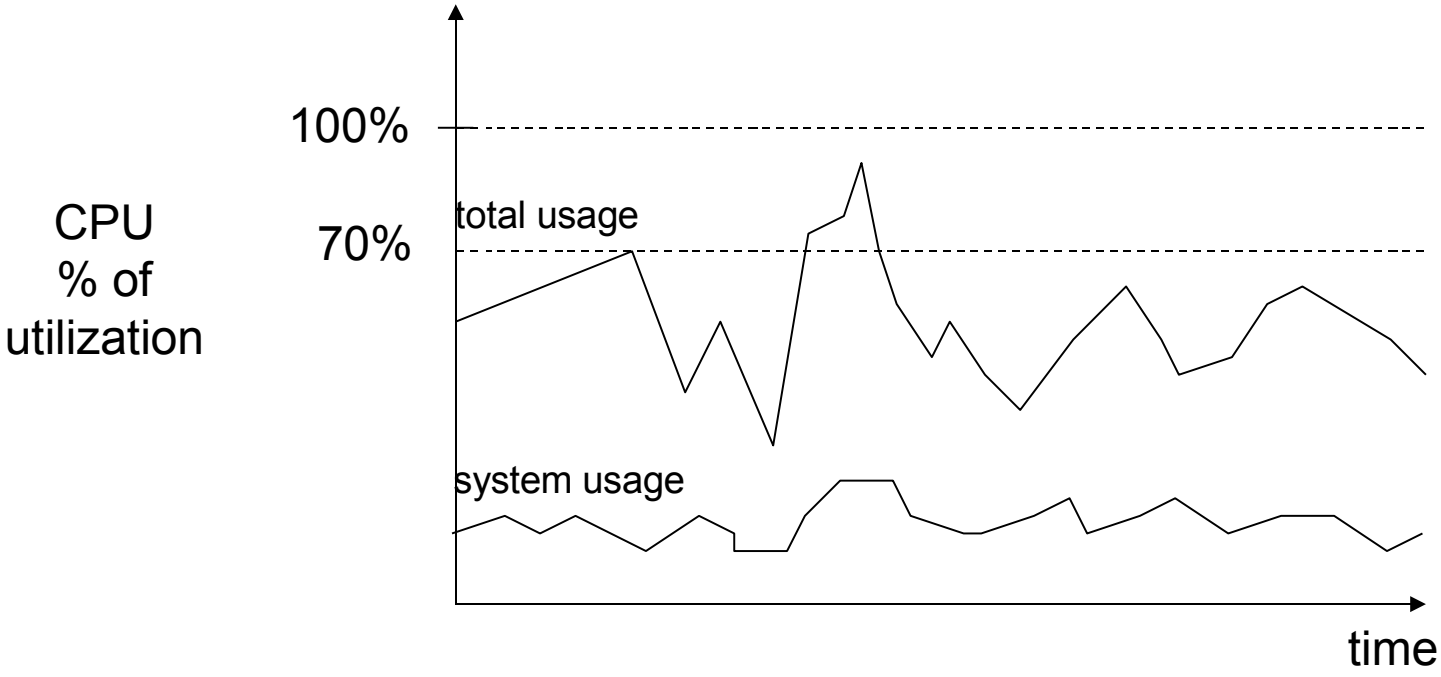


- Answer question 3:
  "Are there enough primary resources available for a DBMS to consume?"

- Primary resources are: CPU, disk/controllers, memory, and network
- Analyze specific OS-level indicators to discover bottlenecks.
- A system-level Performance Monitor is the right tool here

CPU Consumption Indicators at the OS Level



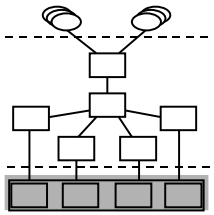


Sustained utilization over 70% should trigger the alert.

System utilization shouldn't be more than 40%.

DBMS (in a non-dedicated machine) should be getting a decent time share.

Disk Performance Indicators at the OS Level



Should be close to zero

Average Queue Size

New requests

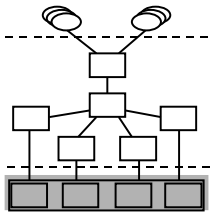
Wait queue

Wait times should also be close to zero

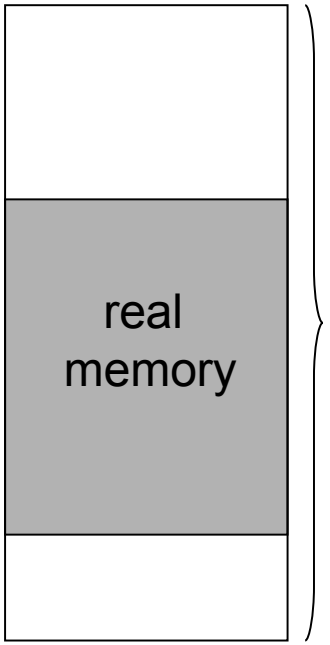
Disk Transfers /second

Idle disk with pending requests? Check controller contention. Also, transfers should be balanced among disks/controllers

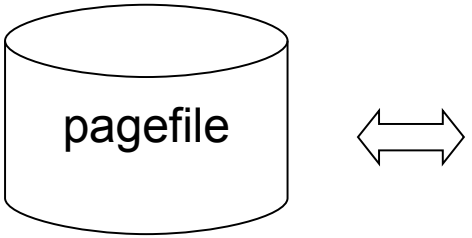
Memory Consumption Indicators at the OS Level



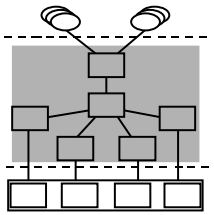
Page faults/time should be close to zero. If paging happens, at least not DB cache pages.



% of pagefile in use (it's used a fixed file/partition) will tell you how much memory is "lacking".



Investigating Intermediate Resources/Consumers

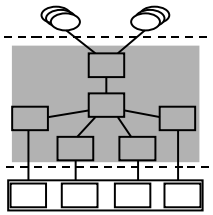


- Answer question 2:

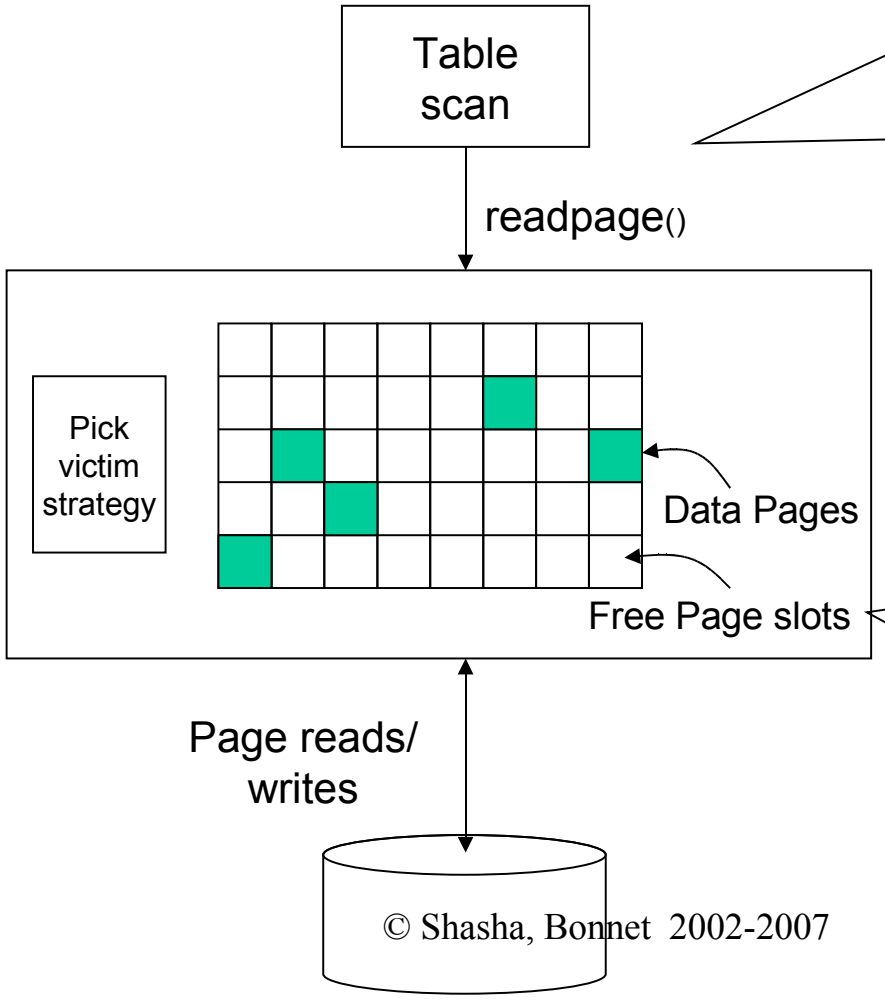
  "Are subsystems making optimal use of resources?"

- Main subsystems: Cache Manager, Disk subsystem, Lock subsystem, and Log/Recovery subsystem
- Similarly to Q3, extract and analyze relevant Pis
- A Performance Monitor is usually useful, but sometimes specific tools apply

Cache Manager Performance Indicators



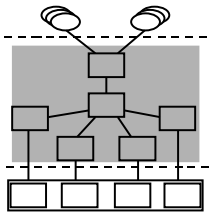
Cache Manager



If page is not in the cache, readpage (logical) generate an actual IO (physical). Ratio of readpages that did not generate physical IO should be 90% or more

Pages are regularly saved to disk to make free space. # of free slots should always be > 0

Disk Manager Performance Indicators



Storage Hierarchy (simplified)

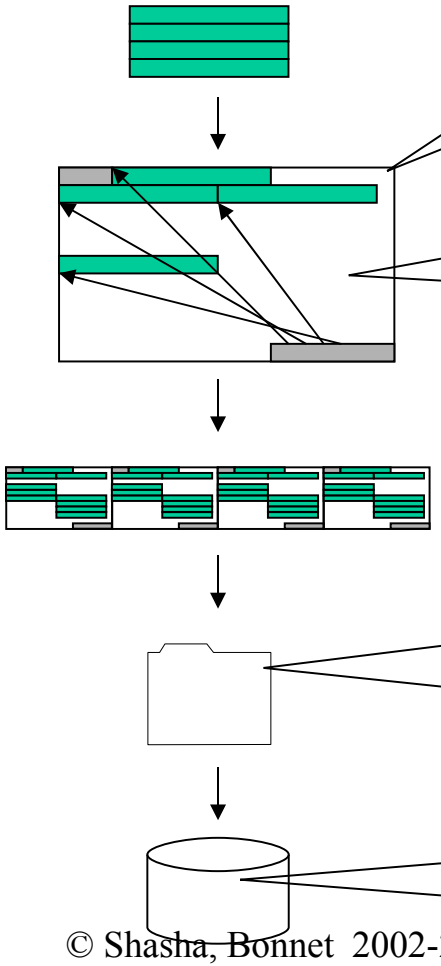
rows

page

extent

file

disk



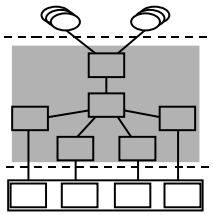
Row displacement: should kept under 5% of rows

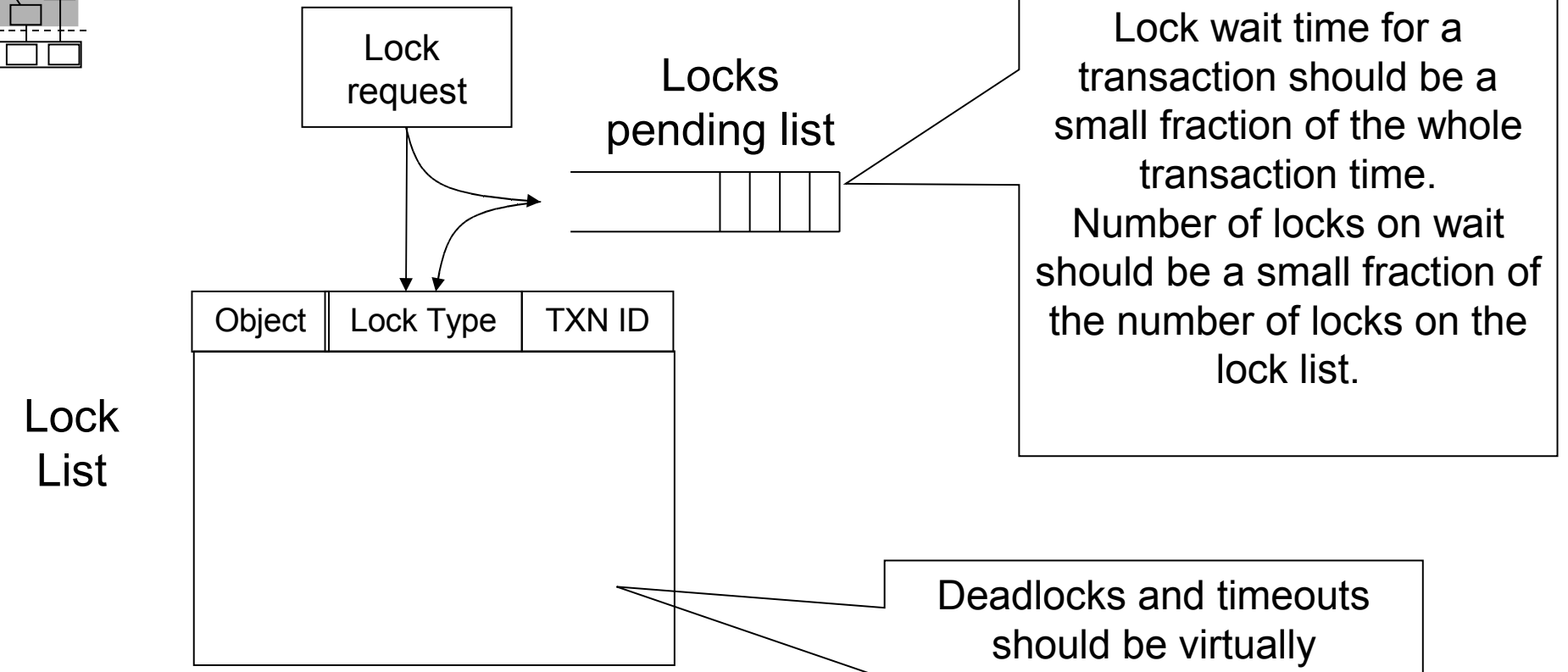
Free space fragmentation: pages with few space should not be in the free list

Data fragmentation: ideally files that store DB objects (table, index) should be in one or few (<5) contiguous extents

File position: should balance workload evenly among all disks

Lock Manager Performance Indicators





Troubleshooting

- Monitoring a DBMS's performance can be done in a systematic way
  - The consumption chain helps distinguishing problems' causes from their symptoms
  - Existing tools help extracting relevant performance indicators
  - The three questions guide the whole monitoring process

Tuning Principles *Leitmotifs*

- Think globally, fix locally (does it matter?)
- Partitioning breaks bottlenecks (temporal and spatial)
- Start-up costs are high; running costs are low (disk transfer, cursors)
- Be prepared for trade-offs (indexes and inserts)

http://www.distlab.dk/dbtune