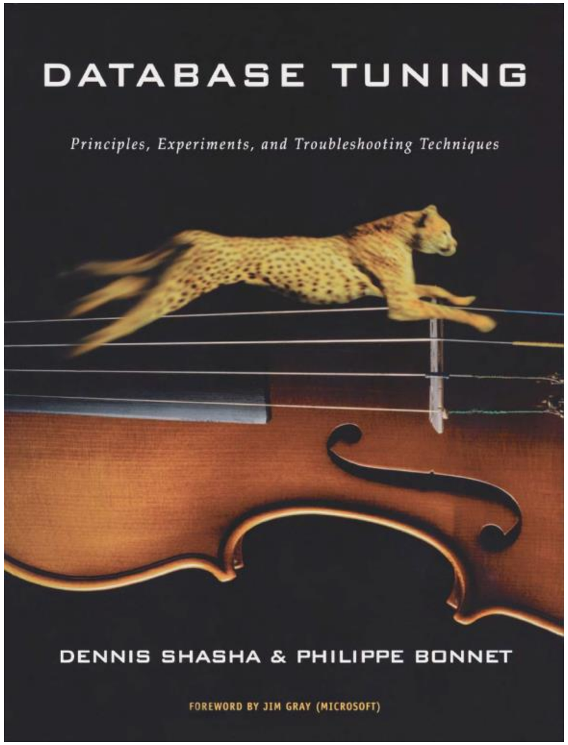
Data Administration in Information Systems

Database tuning (continued)

Database Tuning

- Second part for the course will address different tuning aspects, building on previous knowledge
  - Schema tuning
  - Query tuning
  - Index tuning
  - Lock and log tuning
  - Hardware and OS tuning
  - Database monitoring

Chapter 3



Index Tuning

- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

Types of Queries

1. Point Query

```
SELECT name
FROM Employee
WHERE ssnum = 8478;
```

2. Multipoint Query

```
SELECT name
FROM Employee
WHERE dept = 'Information Systems';
```

Types of Queries (Cont.)

3. Range Query

```
SELECT name
FROM Employee
WHERE salary >= 40000
  AND salary < 60000;
```

4. Prefix Match Query

```
SELECT *
FROM Employee
WHERE name LIKE 'Ke%';
```

Types of Queries (Cont.)

5. Extremal Query

```
SELECT name
FROM Employee
WHERE salary = (SELECT MAX(salary) FROM Employee);
```

6. Ordering Query

```
SELECT *
FROM Employee
ORDER BY salary;
```

Types of Queries (Cont.)

7. Grouping Query

```
SELECT dept, AVG(salary)
FROM Employee
GROUP BY dept;
```

8. Join Query

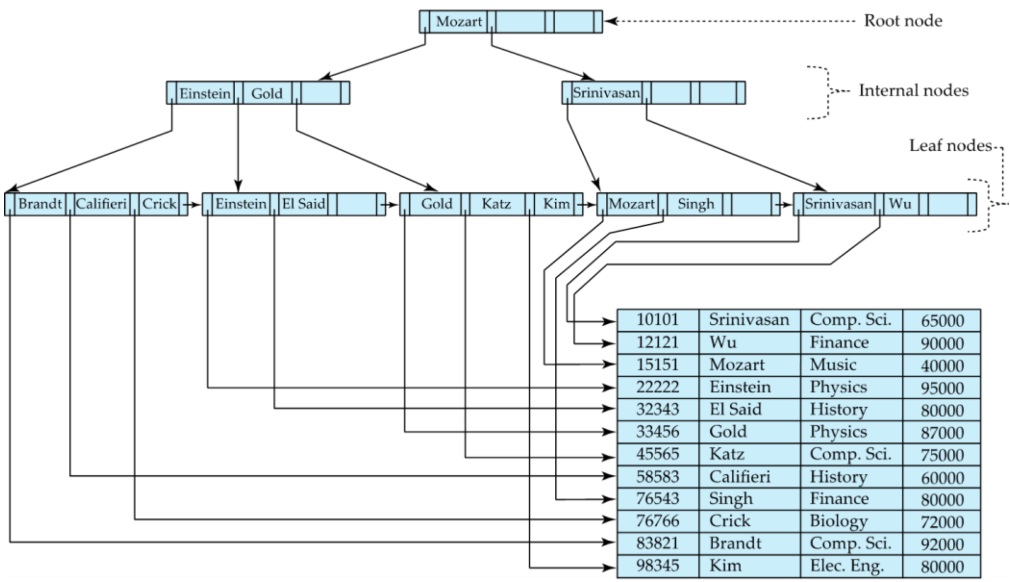
```
SELECT e1.ssnum
FROM Employee e1, Employee e2
WHERE e1.manager = e2.ssnum
  AND e1.salary > e2.salary;
```

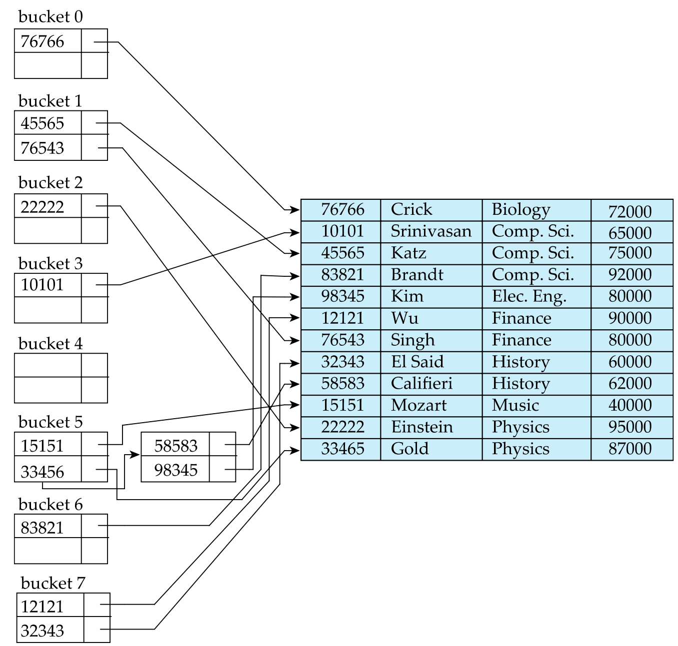
Index Tuning

- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

Index Structure

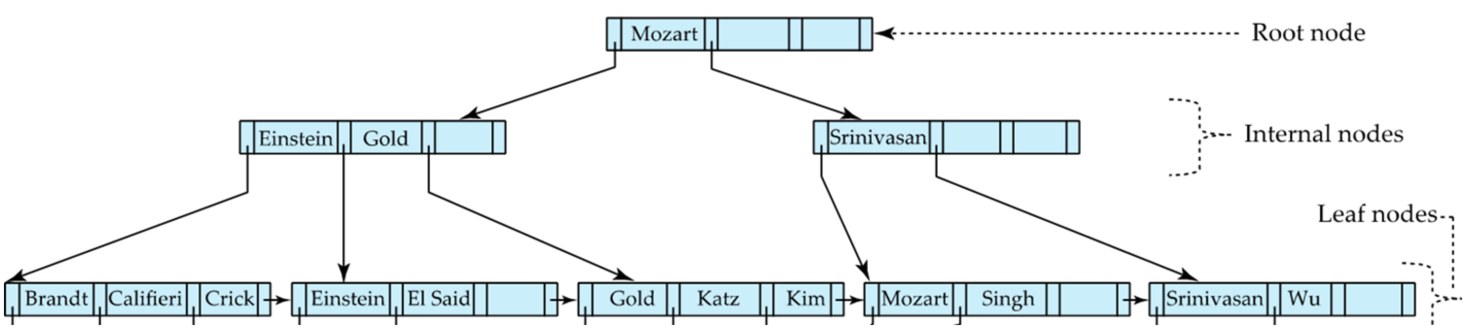
- Most database systems support **B⁺-tree** indexes, and some support **hash** indexes





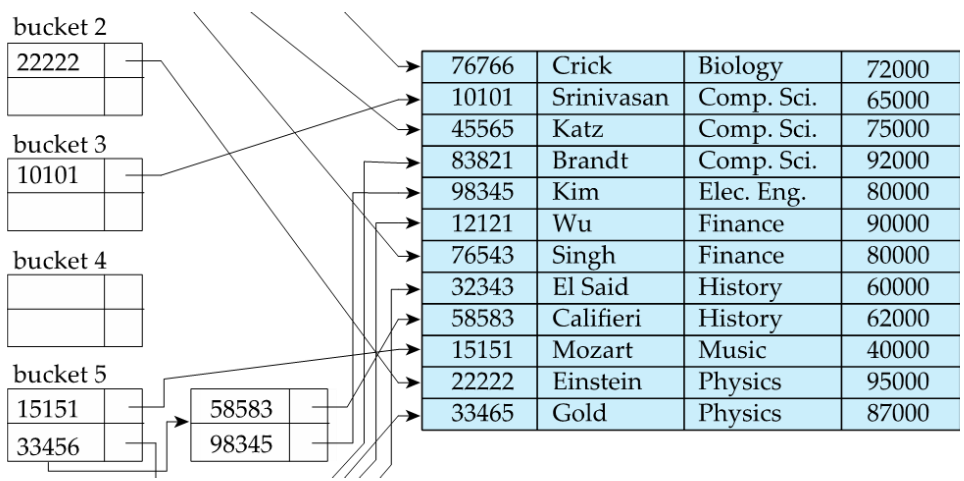
B^+-Tree Index Performance

- Good general-purpose index structure
  - Useful for range queries and extremal queries, where hash indexes are not
- Performance depends on height of the tree
  - Number of nodes or levels from root to leaf of the B^+-Tree
- Minimize height by having more children per node
  - Length of search key influences **fanout** (number of children per node)
  - Choose a small key when creating an index
  - If the key is large, it may be possible to use **key compression**
    - store only part of the key to distinguish between neighbors
    - e.g. store 'Smi', 'Smo', 'Smy' instead of 'Smith', 'Smoot', 'Smythe'



Hash Index Performance

- Faster than B⁺-trees for point queries and multipoint queries
  – provided there are no overflow chains
  – useless for range, prefix or extremal queries; also bad for full scans
- Must be reorganized if there is a significant amount of overflow
  – rebuild entire index or reorganize pages
  – avoiding overflow may require underutilizing the hash space
- Size of hash structure is not related to length of search key
  – hash function is applied on key to locate bucket with pointers

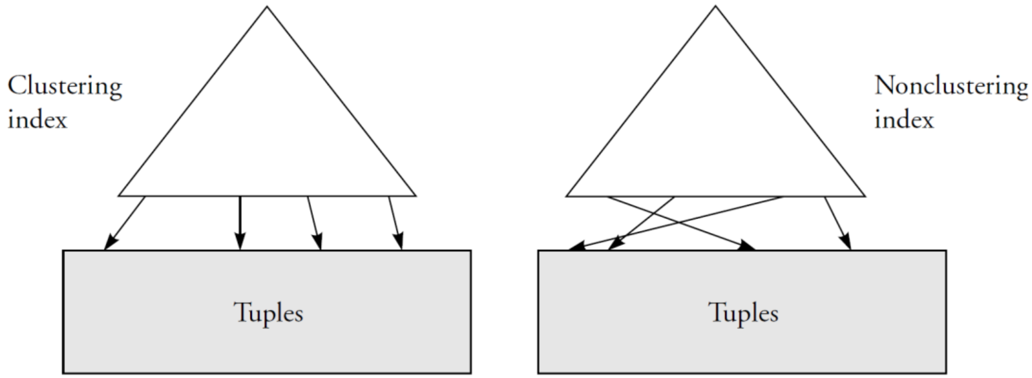


Index Tuning

- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

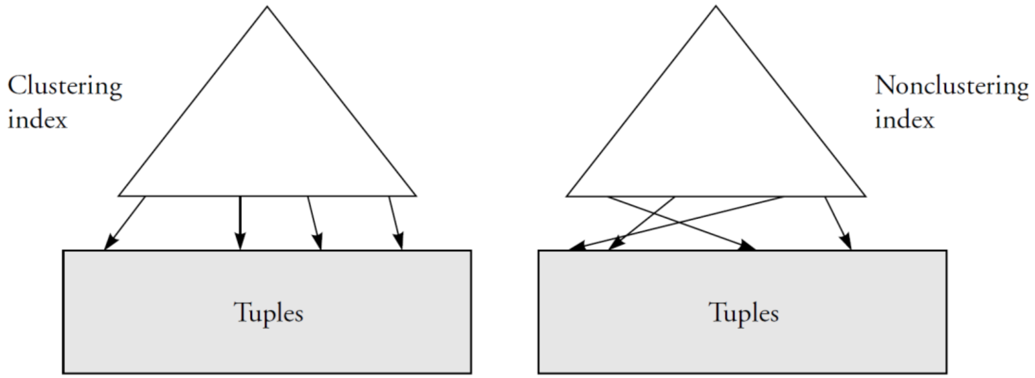
Clustered vs. Non-clustered Indexes

- Clustered index
  - Co-locates records whose values are *near* to one another
    - For B^+-trees, two values are near if they are close in sort order
      - Good for point, multipoint, partial match, general join queries
    - For hash indexes, two values are near only if they are identical
      - Good for point, multipoint, equijoin queries
- Non-clustered index
  - A non-clustered index is independent of the table organization
  - There may be several non-clustered indexes per table



Dense vs. Sparse Indexes

- Clustered indexes can be dense or sparse
- Dense index
  - pointers are associated to records; one index entry per record
- Sparse index
  - pointers are associated to pages; one index entry per page
  - any values $v_1 \leq v < v_2$ can be found on the same page as v_1
- Non-clustered indexes must be dense
  - cannot be sparse, since values $v_1 \leq v < v_2$ can be anywhere

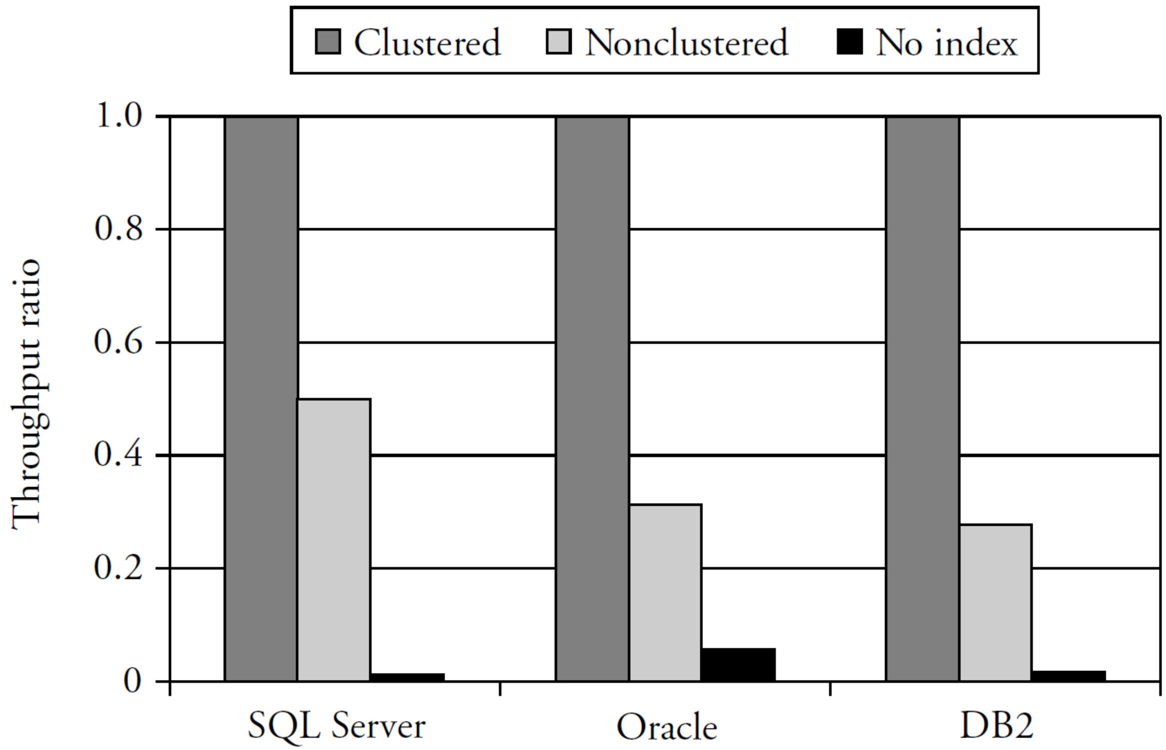


Benefits of a Clustered Index

- A sparse clustered index stores fewer pointers than a dense index
  - This might decrease the height of the B^+-tree
- A clustered index is good for **multipoint queries**
  - A clustered index based on a B^+-tree can support **range**, **prefix**, **extremal** and **ordering queries** well
- A clustered index can improve equality joins
  - If both tables are indexed, a merge-join becomes possible

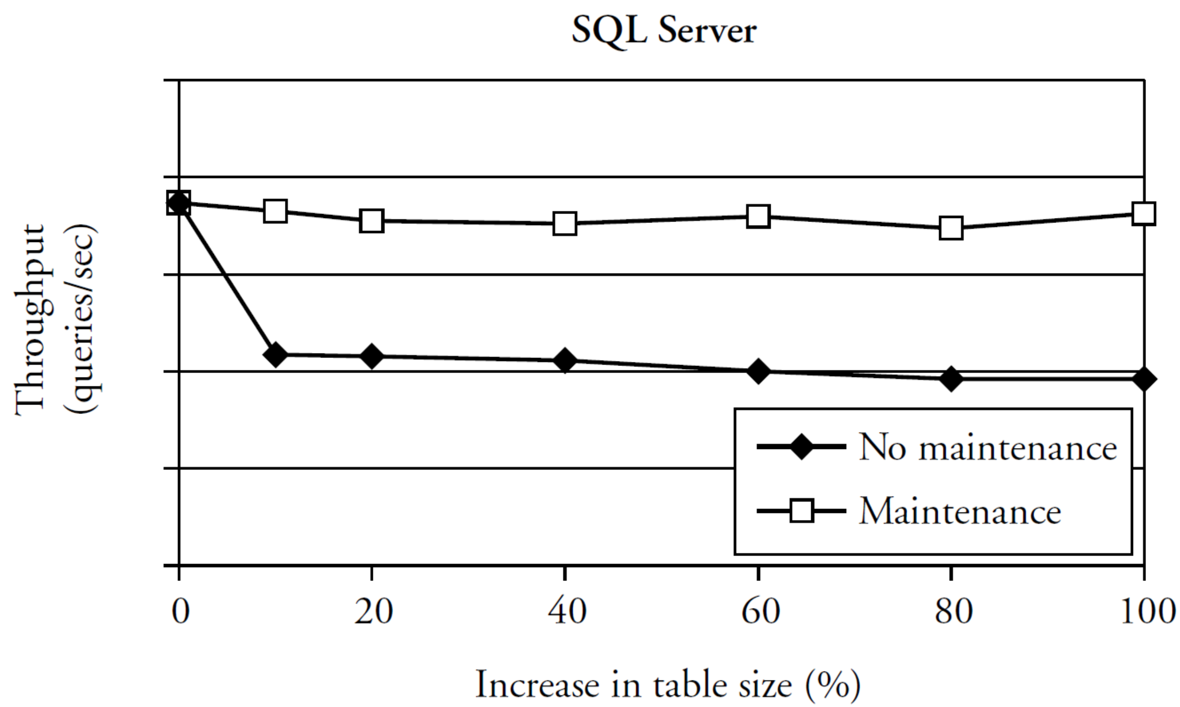
Benefits of a Clustered Index (Cont.)

- Multipoint query that returns 100 records out of 1 000 000
- Clustered index is **twice as fast** as non-clustered index and orders of magnitude faster than a scan



Evaluation of Clustered Indexes with Insertions

- Insertions cause page splits and extra I/O for each query
- Maintenance consists in rebuilding or reorganizing the index
  - With maintenance, performance is constant
  - Without maintenance, performance degrades

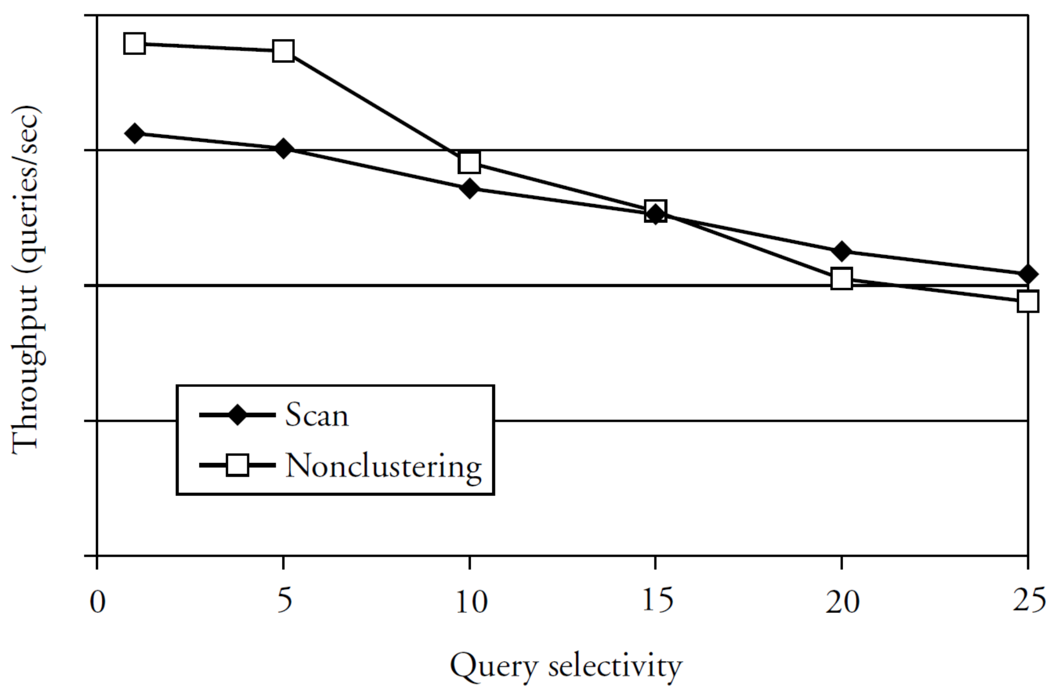


Redundant Tables

- There can be only one clustered index per table
- Question
  - Wouldn't it be nice to have multiple clustered indexes?
- Answer
  - Replicate table to use a clustered index on a different attribute
  - Works well only if low insertion/update rate

Benefits of Non-clustered Indexes

- A non-clustered index is good if the query retrieves few records compared to the number of pages in the table
  - Always useful for point queries
  - Useful for multipoint queries if table contains many distinct values
  - In the experiment below, index is good if query selectivity < 15% of records



Benefits of Non-clustered Indexes (Cont.)

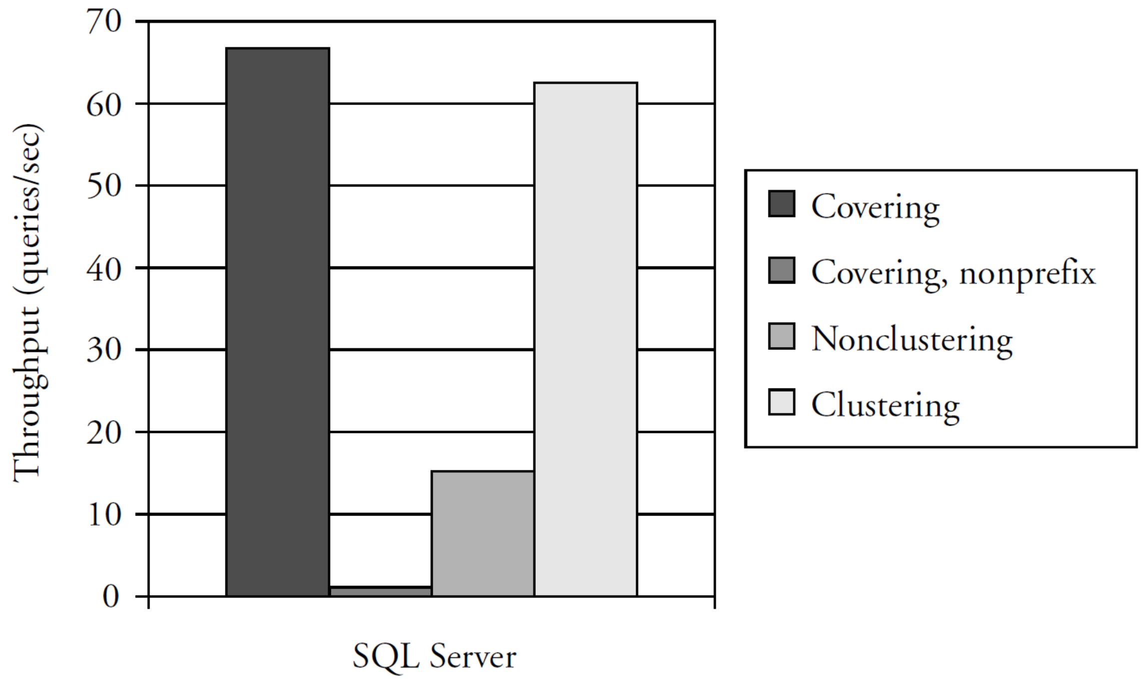
- A non-clustered index can eliminate the need to access the underlying table through **covering**
- Example:
  - with a **composite index** on (*A*, *B*, *C*) or
  - with a **covering index** on *A* that **includes** *B* and *C*
  - the query below can be answered based on the index alone

```
SELECT B, C
FROM R
WHERE A = 5;
```

- It might be worth creating multiple indexes to increase the likelihood that the optimizer finds a covering index

Benefits of Non-clustered Indexes (Cont.)

- Comparing 4 options
  - Covering index
  - Covering index, but with unsuitable order of attributes (unusable)
  - Non-clustered index
  - Clustered index



Index Tuning

- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

Covering/Composite Indexes

- A non-clustered index can eliminate the need to access the underlying table through **covering** or **composite index**
- For the query:

```sql
SELECT name
FROM Employee
WHERE dept = 'Information Systems';
```

  - A good covering index would be on (*dept*, *name*)
  - Index on (*name*, *dept*) would be useless

- Drawbacks
  - Tends to have a large size
  - Update to an attribute causes index to be modified

Composite Search Keys

- To retrieve records with *age*=30 AND *salary*=4000
  - Index on (*age*, *salary*) is better than index on *age* or index on *salary*
  - Index could be clustered or non-clustered
- If condition is 20<*age*<30 AND 3000<*salary*<5000
  - Clustered index on (*age*, *salary*) or (*salary, age*) is best
- If condition is *age*=30 AND 3000<*salary*<5000
  - Clustered index on (*age*, *salary*) is better than (*salary, age*)

Index Tuning

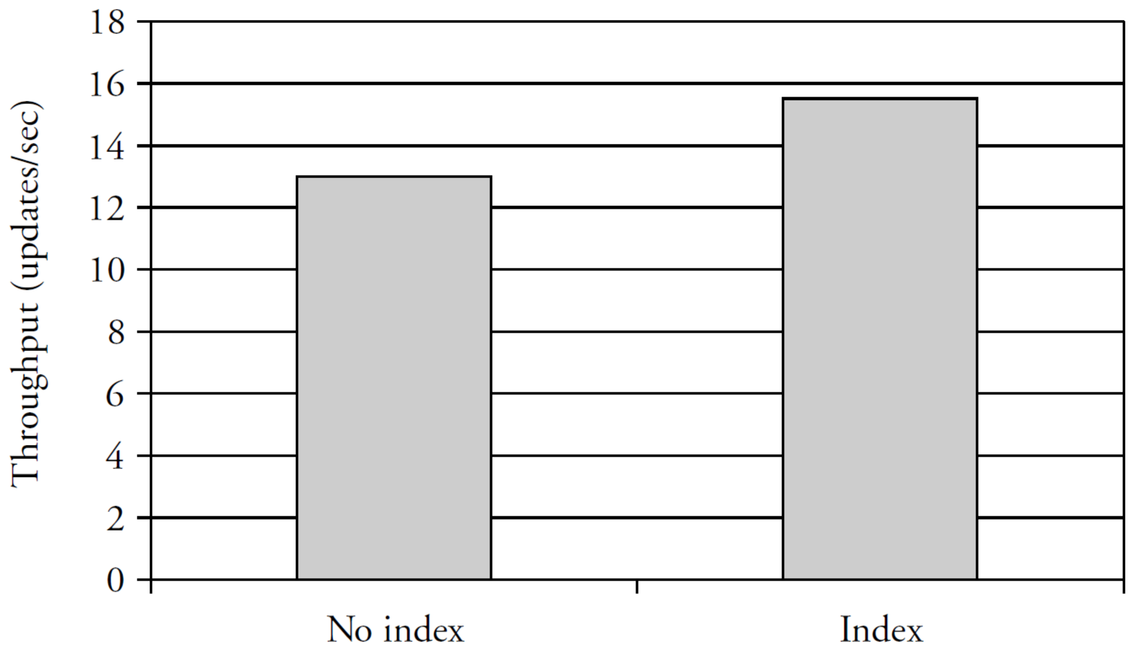
- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

Indexes on Small Tables

- Common practice tells us to avoid indexes on small tables
  - e.g. with fewer than 200 records
  - this number depends on the size of records
- Some examples:
  - If many records fit on a page, and the table has only a few pages, then an index search will require reading one (or more) extra page(s)
  - If each record occupies an entire page, then 100 records require 100 disk accesses, so an index is definitely useful
  - If many inserts execute on a table with a small index, the index itself may become a concurrency control bottleneck
- But having no index can also be harmful
  - Multiple transactions that update a single record will be scanning the entire table and possibly reducing update concurrency

Indexes on Small Tables (Cont.)

- Small table: 100 records
- Two concurrent processes perform updates
- **No index**: the table is scanned for each update
- A **clustered index**: allows to take advantage of less locking



Index Tuning

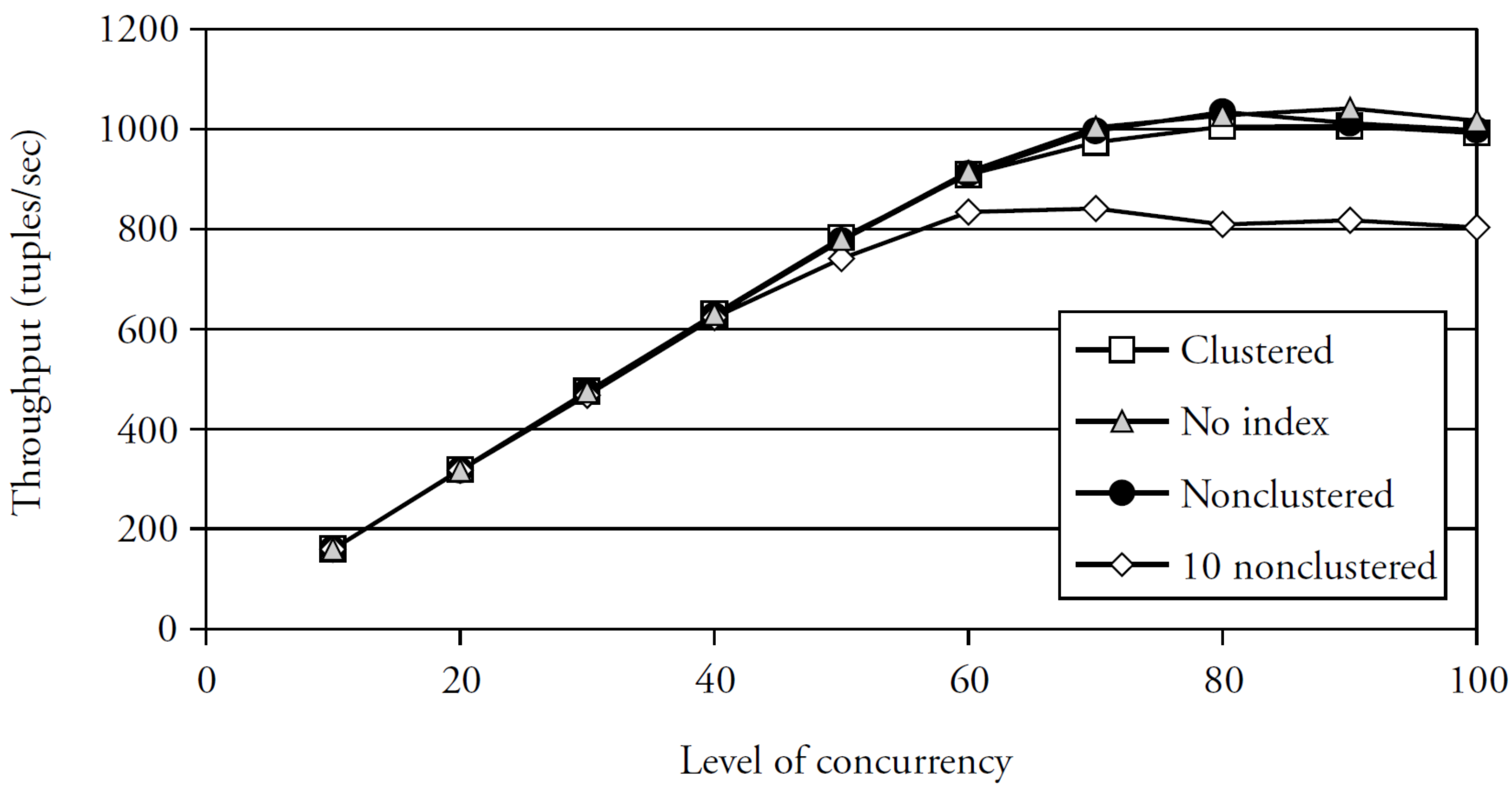
- Topics
  - Types of queries
  - Index structure
  - Clustered vs. non-clustered indexes
  - Covering/composite indexes
  - Indexes on small tables
  - Recommendations

Table Organization and Index Selection

- Use a **hash index** for equality queries
- Use a **B$^+$-tree** if equality and non-equality queries may be used
- Use clustered indexes if:
  - queries need all (or most) fields in records,
  - records are too large for a composite index,
  - and multiple records are returned per query
- If possible, use a covering index for critical queries
- Do not use index if the time lost inserting and updating overwhelms time saved when querying

Table Organization and Index Selection (Cont.)

- Inserting 100 000 records in a benchmark database
- Performance hit is noticeable for many non-clustered indexes at high levels of transaction concurrency



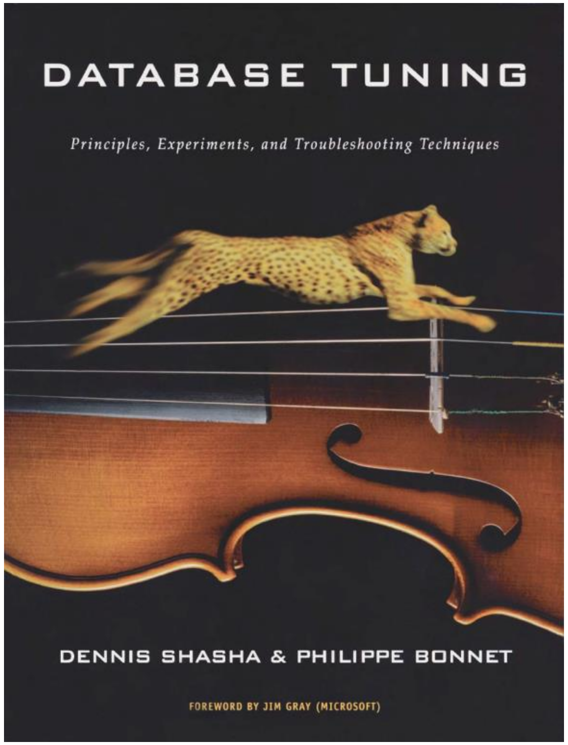
Optimizing Workloads

- Modern database systems: **automated tuning wizards**!
- Provide a workload as input
  - The workload can be a set of queries or a database activity trace
- Two-step approach
  - Find the best index that optimizes each query
  - Find the best subset of indexes that optimizes the entire workload
- Consider both the benefits and the overhead
  - The cost decrease for queries
  - The cost increase for insert/delete statements
  - Final recommendation is the subset of indexes with lowest combined cost

Database Tuning

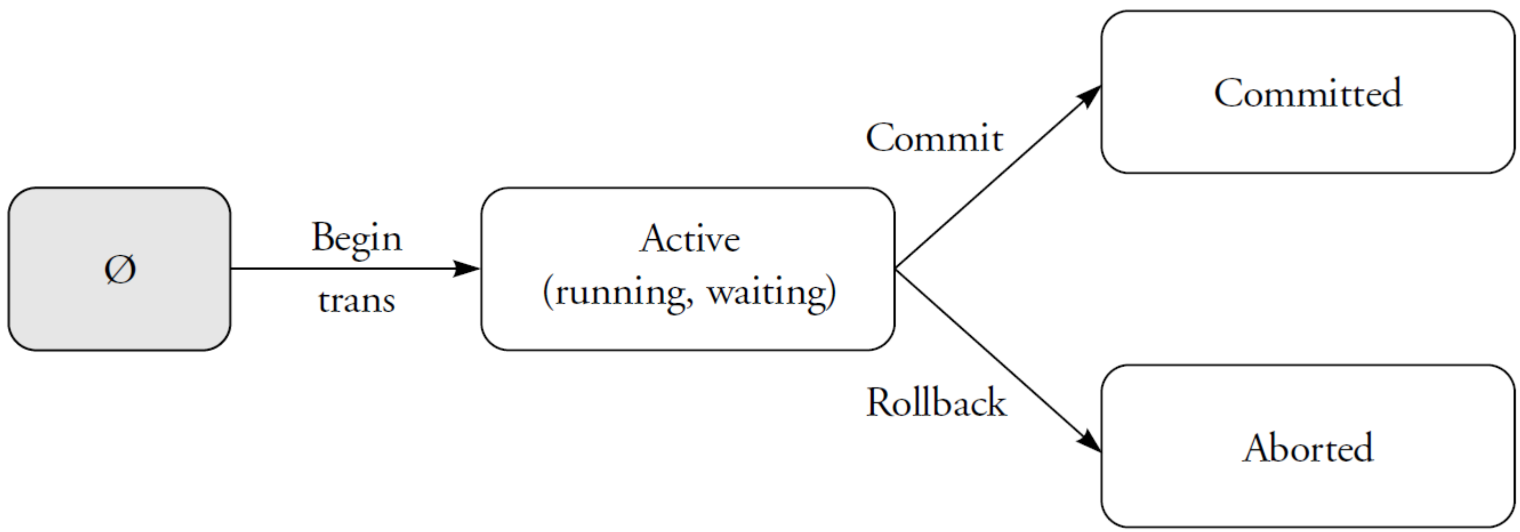
- Second part for the course will address different tuning aspects, building on previous knowledge
  - Schema tuning
  - Query tuning
  - Index tuning
  - Lock and log tuning
  - Hardware and OS tuning
  - Database monitoring

Chapter 2



Atomicity and Durability

- Every transaction either **commits** or **aborts**. It cannot change its mind
- Even in the face of failures:
  - Effects of committed transactions should be permanent;
  - Effects of aborted transactions should leave no trace.

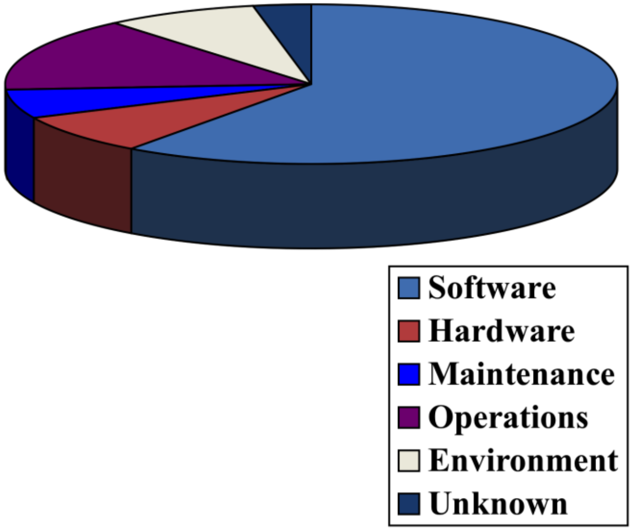


Outages

- Environment
  - Fire in the machine room (Credit Lyonnais, 1996)
- Operations
  - Problem during regular system administration, configuration and operation
- Maintenance
  - Problem during system repair and maintenance
- Hardware
  - Fault in the physical devices: CPU, RAM, disks, network
- Software
  - 99% are Heisenbugs: transient software error related to timing or overload
  - Heisenbugs do not appear when the system is re-started

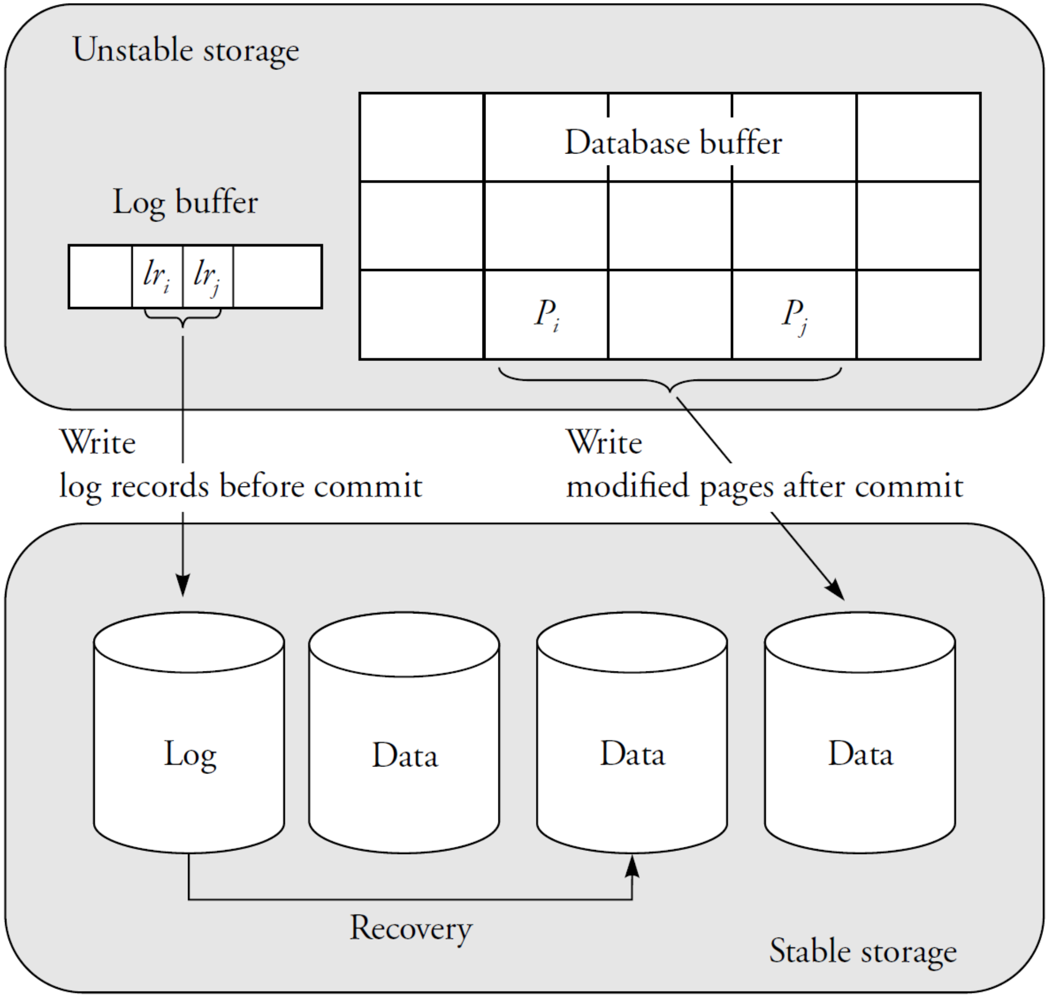
Outages

- A fault tolerant system must provision for all causes of outages
- **Software** is the typical problem
  - Hardware failures cause under 10% of outages
  - Heisenbugs stop the system without damaging the data
- Database systems protect integrity against single hardware failure and some software failures



Stable storage holds data and log

- In case of failure, a consistent state is recovered from log



Tuning the recovery system

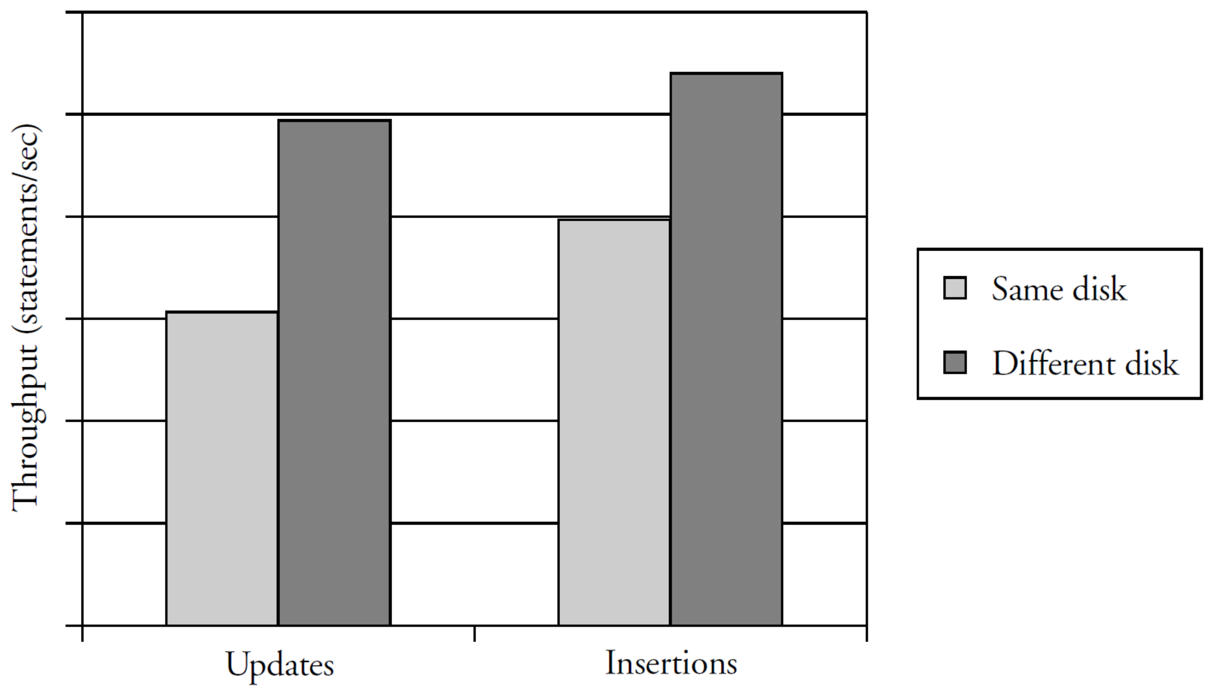
- Topics
  - Put the log on a separate disk
  - Delay output to database disks
  - Database dumps and checkpoints
  - From batch to minibatch transactions

Tuning the recovery system

- Put the log on a separate disk to avoid seeks
  - Writes to log occur sequentially
  - Writes to disk occur (at least) 100 times faster when they occur sequentially than when they occur randomly
- A disk that has the log should have no other data
  - Sequential I/O
  - Log failure independent of database failure

Tuning the recovery system

- 300 000 insert/update statements
- Oracle on Linux without RAID
- Log on separate disk provides 30% performance improvement



Tuning the recovery system

- Topics
  - Put the log on a separate disk
  - Delay output to database disks
  - Database dumps and checkpoints
  - From batch to minibatch transactions

Tuning the recovery system

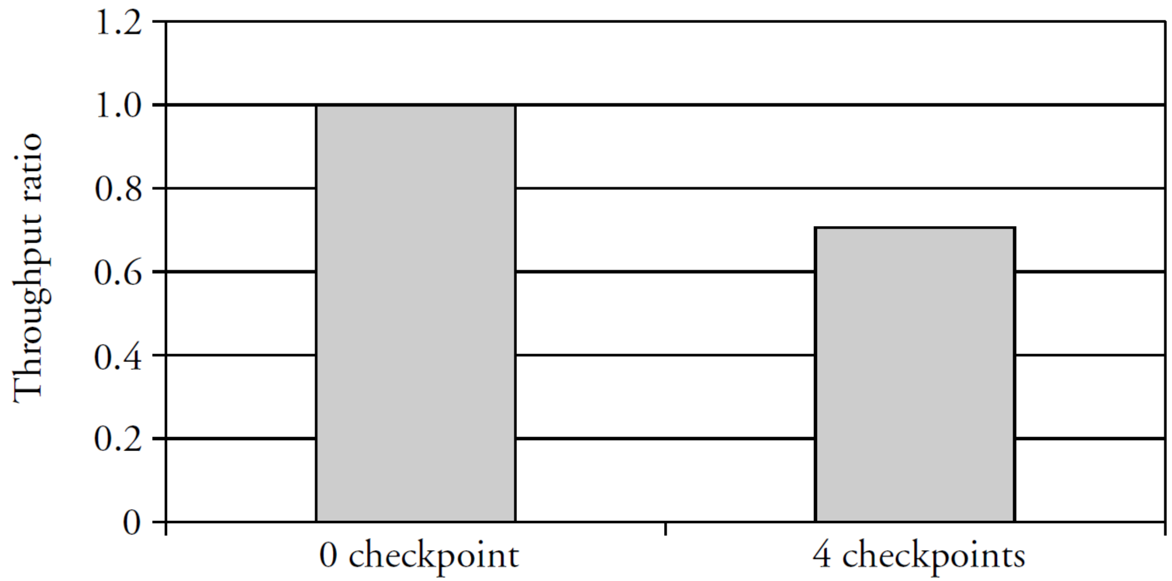
- Tuning database writes
  - delay writing updates to database disks for as long as possible
  - these are random writes; wait for convenient time
- Dirty data is written to disk
  - when the number of dirty pages in buffer is greater than a given parameter (Oracle)
  - when the number of free pages in the buffer crosses a given threshold, e.g. less than 3% of buffer size (SQL Server)
  - DBA can tweak these settings

Tuning the recovery system

- Topics
  - Put the log on a separate disk
  - Delay output to database disks
  - Database dumps and checkpoints
  - From batch to minibatch transactions

Tuning the recovery system

- Setting intervals for database dumps and checkpoints
- A checkpoint is performed
  - at regular intervals
  - or when the log is full (Oracle)
- Checkpoints impact the performance of online processing
  - but they reduce the size of log, and the time to recover from a crash



Tuning the recovery system

- Topics
  - Put the log on a separate disk
  - Delay output to database disks
  - Database dumps and checkpoints
  - From batch to minibatch transactions

Tuning the recovery system

- Reduce the size of large update transactions
- From batch to minibatch
  - A transaction that performs many updates without critical time constraints is called a *batch* transaction
    - transaction is long; recovery from failure can be costly
  - Break transaction into small transactions (**minibatching**)
    - instead of updating all accounts, update them in minibatches
    - commit after each minibatch; keep track of last account updated
    - in case of failure, resume updating from last successful minibatch

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Lock Tuning

- Concurrency Control Goals
- Correctness goals
  - Serializability: each transaction appears to execute in isolation
- 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
- Trade-off between correctness and performance

Lock Tuning

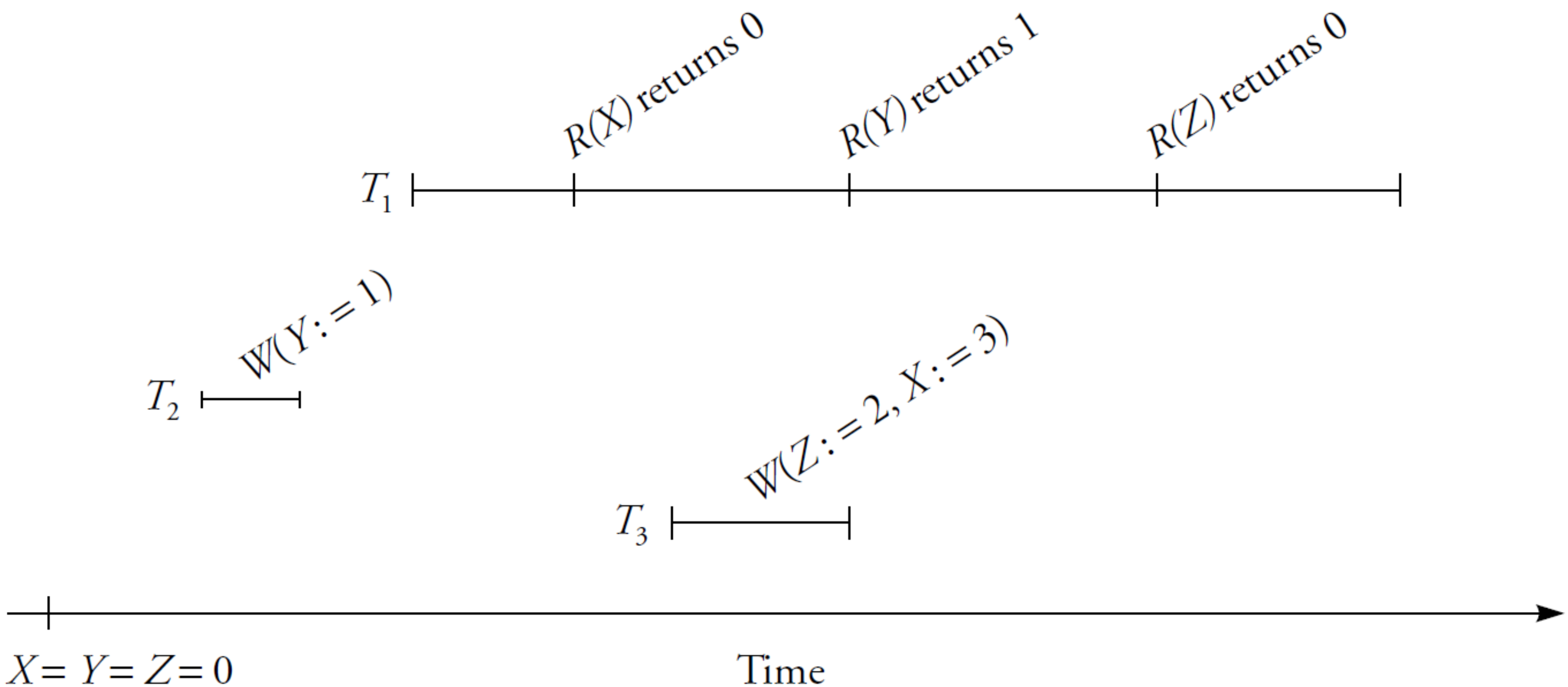
- Ideal Transaction
  - Acquires few locks and favors S-locks over X-locks
    - Reduce the number of conflicts; conflicts are due to X-locks
  - Acquires locks with fine granularity
    - Reduce the scope of each conflict
  - Holds locks for a short time
    - Reduce waiting; but we know the kind of problems this can generate...

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Lock Tuning

- Use special facilities for long reads
  - In some systems, read-only queries hold no locks yet appear to execute in serializable mode
  - Method used (**snapshot isolation**):
    - Each transaction executes against the version of the data that existed when the transaction started



Snapshot Isolation

- Implications of snapshot isolation
  - No locks for read operations; increased concurrency between transactions
  - Time and space overhead in managing multiple versions of data
- Almost serializable; does not guarantee correctness in all cases
  - Consider two transactions
    - T1: x:=y
    - T2: y:=x
  - Initially x=3 and y =17
    - Result of serial execution would be x=y=17 or x=y=3
  - However, with snapshot isolation:
    - Result can be x=17, y=3 (**write skew**)

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Eliminate Unnecessary Locking

- Locking is not necessary when:
  - Only one transaction runs at a time
    - e.g. when loading the database
  - All transactions are read-only
    - e.g. decision support queries on archival database
- Use available options to suppress locking
  - Choose the appropriate isolation level
  - Disable lock escalation
    - when many row or page locks are converted to a table lock
  - Use table hints such as WITH (NOLOCK)

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Isolation Levels

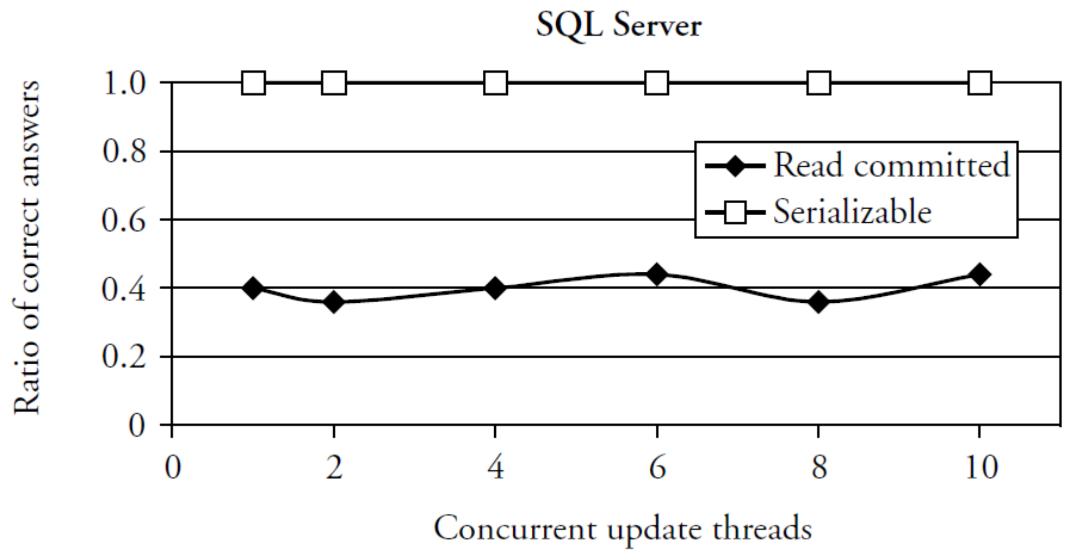
- Use weaker isolation levels when application allows
  - Relaxing correctness to improve performance
- Read Uncommitted
  - Dirty reads; non-repeatable reads; phantom reads
  - Locks on write; no locks on read
- Read Committed
  - Non-repeatable reads; phantom reads
  - Two-phase locking on write; locks on read are released immediately
- Repeatable Read
  - Phantom reads
  - Two-phase locking on write and read
- Serializable
  - Table locking to avoid phantoms

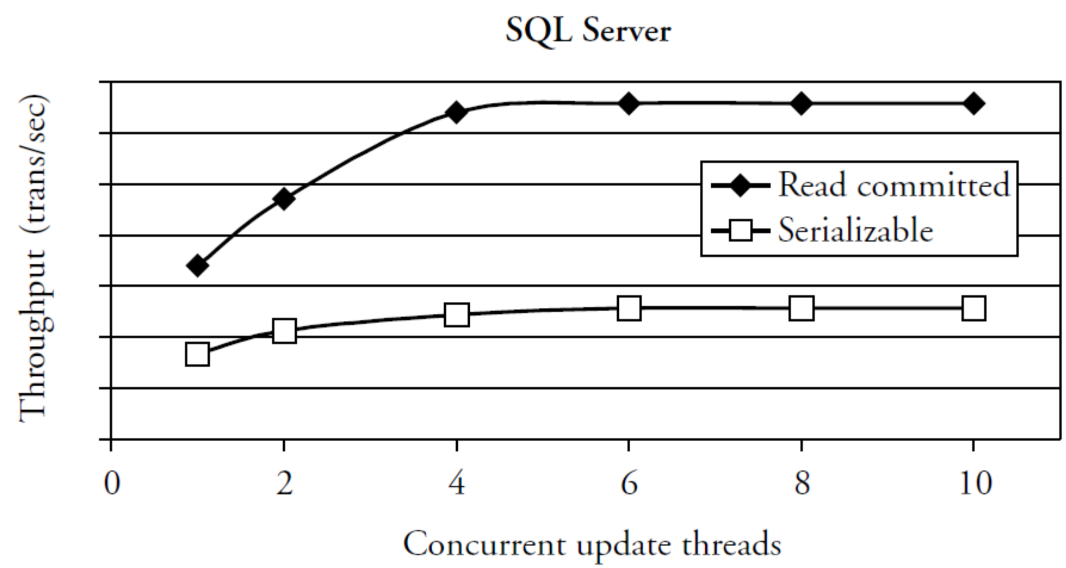
Sacrificing Isolation for Performance

- A transaction that holds locks during a screen interaction is an invitation to bottlenecks
  - Example: Airline Reservation
    - Retrieve list of seats available
    - Talk with customer regarding availability
    - Secure seat
  - Split into two transactions
    - Read the list of seats available
    - Secure the seat that was chosen by the customer
  - Customer may be told that chosen seat could not be secured (correctness is sacrificed)

Experiments

- Correctness vs. performance
  - Tests with summing query running concurrently with update transactions





Recommendation

- Begin with the highest degree of isolation (serializable)
  - If a transaction either suffers extensive deadlock or causes significant blocking, consider lowering the level of isolation
  - Be aware that the answers may not be correct

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Granularity of Locking

- Control the granularity of locking
  - Modern database systems support different granularities of locking
    - e.g. row-level, page-level, table-level, database-level locks
  - Row-level locking is best for online transaction environments where each transaction accesses only a few records spread on different pages
- Then why use table-level locking?
  - Ensure correctness of long transactions (e.g. summing query)
  - Avoid deadlocks between short transactions
  - Reduce locking overhead in scenarios of low concurrency

Recommendation

- Long transactions
  - accessing almost all pages of a table
  - should use table locks mostly to avoid deadlocks
- Short transactions
  - accessing only a few records of a table
  - should use record locks to enhance concurrency

How to Control the Granularity of Locking

- Explicit control of granularity
  - Within a transaction, by requesting (or preventing) a table-level lock
  - Across transactions, by configuring the lock granularity for a table (if possible)
- Configuring the lock escalation point
  - System grants the fine-granularity locks until number of locks exceeds some threshold set by the DBA
  - Threshold should be set high enough so that escalation does not take place in an environment of short transactions

Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Avoid data definition statements

- Data definition statements
  - Interfere with the system catalog or metadata
  - Examples
    - adding or removing tables/indexes
    - changing column width or data type
    - changing the description of attributes
    - etc.
- The system catalog is accessed by every transaction
  - Query parsing, compilation, optimization
- Avoid updates to system catalog during heavy system activity
  - Access to system catalog can become a bottleneck

Lock Tuning

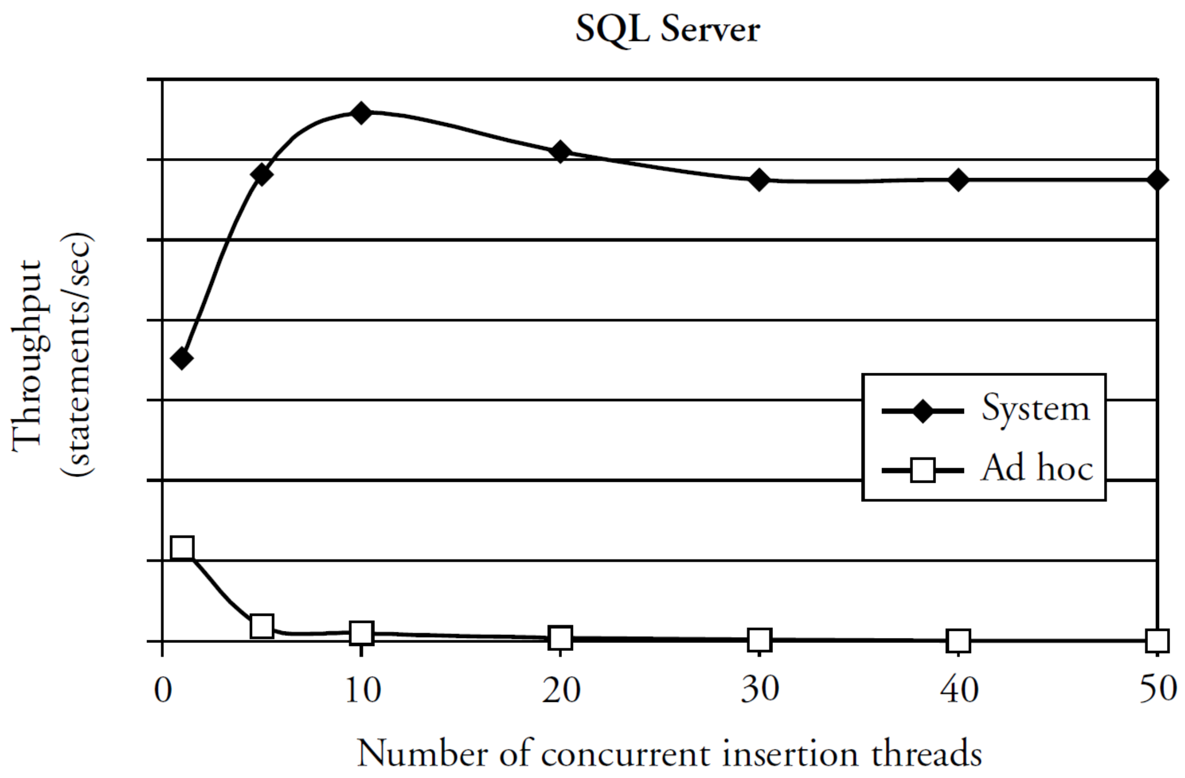
- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Circumvent Hot Spots

- Hot spot
  - Data item accessed by many transactions and updated by some
  - Each update transaction must complete before other transactions can obtain a lock on the item (bottleneck)
- Techniques to circumvent:
  - Access the item as late as possible in the transaction, to minimize the time that the transaction holds the lock on item
  - Use special database management facilities; e.g. use auto increment for sequential key generation, instead of counter

Experiment

- System-generated key vs. ad-hoc counter
  - With an ad-hoc counter, transactions need to access and update the counter themselves
  - The counter value, wherever it is stored, becomes a bottleneck (hot spot)



Lock Tuning

- Topics
  - Correctness vs. performance
  - Use special facilities for long reads
  - Eliminate unnecessary locking
  - Use weaker isolation levels
  - Control the granularity of locking
  - Avoid data definition statements
  - Circumvent hot spots
  - Transaction chopping

Transaction Chopping

- How long should a transaction be?
- Transaction length influences performance:
  - The more locks a transaction *T* requests, the more likely it will have to wait for other transaction to release a lock
  - The longer a transaction *T* executes, the more time other transactions will have to wait for locks being held by *T*
- To mitigate blocking, prefer short transactions to longer ones
  - We may be able to *chop* long transactions in shorter ones without loosing isolation guarantees

Example

- Money transfer between two accounts with balances *X* and *Y*

```
if (X < A) then rollback;
X := X – A
Y := Y + A
```

  – Can be broken into two transactions and still guarantee $X \geq 0$

```
if (X < A) then rollback;
Y := Y + A
X := X – A
```

  – Cannot be broken into two transactions
    - A second transaction may be initiated and result in $X < 0$

Transaction Chopping (Cont.)

- Can a transaction *T* be broken into smaller transactions?
  - This depends on what is running concurrently with *T*
- The are two questions to be answered:
  - If *T* is broken up, will other transactions interfere with *T* ?
    - Making *T* observe or produce an inconsistent state
  - If *T* is broken up, will T interfere with other transactions?
    - Making them observe or produce an inconsistent state

Transaction Chopping (Cont.)

- Rule of thumb
  - Suppose *T* accesses data *X* and *Y*, but other transactions access *X* or *Y* and nothing else
  - Then *T* can be divided into two transactions, one accessing *X* and another accessing *Y*
- Caution
  - Adding a new transaction to a set of existing transactions may invalidate previous choppings