Database Tuning

Introduction, Tuning Principles, Course Organization

Nikolaus Augsten

University of Salzburg
Department of Computer Science
Database Group

Unit 1 – WS 2015/2016

Adapted from "Database Tuning" by Dennis Shasha and Philippe Bonnet.

Outline

Outline

What is Database Tuning?

Activity of making a database application run faster:

- Faster means higher throughput or lower response time
- A 5% improvement is significant

What parameters should be considered for tuning?

- All parameters that help to reach the tuning goal!
- Examples: more or faster disks, more main memory, use indexes effectively, write good queries, avoid unnecessary computations, avoid transaction bottleneck etc.

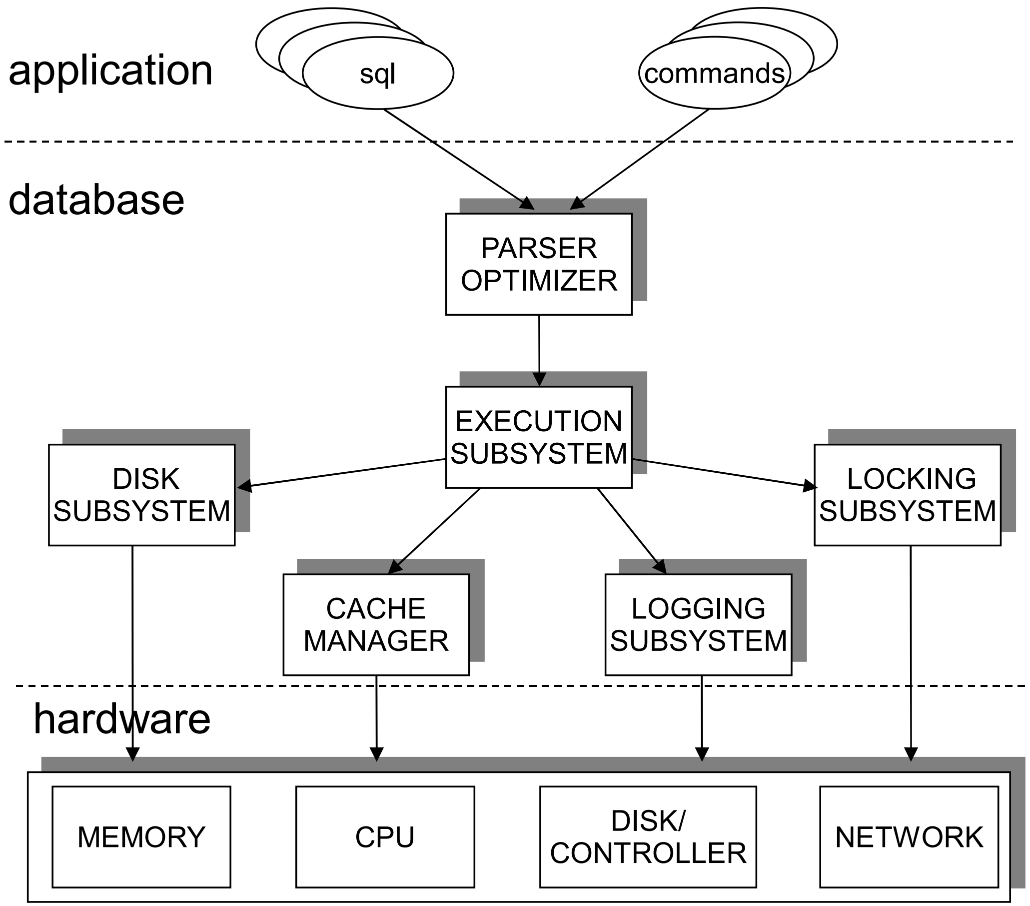
Bad news: There is always a cost/benefit tread-off.
Good news: Sometimes the cost is very low and the benefit very high, e.g., avoiding transaction bottlenecks or queries that run for hours unnecessarily.

Why is Database Tuning hard?

The following query runs too slow:

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

What to do?



Course Objectives

1. Relevant notions concerning the internals of commercial DBMS
    - helps you to understand the manual of your DBMS
    - enables you to take informed tuning decisions
2. Tuning principles, backed by experiments:
    - How do tuning principles impact performances of my system?
3. 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)

Prerequisites

- Programming skills (Java)
- Data structures and algorithms (undergraduate level)
  - lists, trees, arrays, binary search, merge algorithms, etc.
- Databases management systems (undergraduate level)
  - basic SQL knowledge
  - advantageous to know transactions, indexes, buffer management, etc.

How Is This Course (DBT) Different from "Databases II" (DBII)?

- DBT looks at the same topics from a different perspective.
- Algorithmic details vs. black box behavior:
  - DBII: how exactly does a B-tree updated work?
  - DBT: how efficient is a B-tree update and why?
- Theory vs. hands-on:
  - DBII: learn about sort-merge and hash joins on paper
  - DBT: experimentally compare sort-merge and hash join on a real system, interpret the results
- Local vs. Global:
  - DBII: focus on topics in isolation
  - DBT: focus on interaction between system components
- There is a partial overlap, important notions will be revisited!

Outline

Tuning between Theory and Practice

- Practitioner: Apply rules of thumb.
  - Example: “Never use aggregate functions (such as AVG) when transaction response time is critical.”
  - Problem: Blindly applying rules of thumb may not work, e.g., AVG may be OK if only few tuples are accessed via index.
- Theoretician: Mathematically model problem and give guarantees about solution.
  - Example: Runtime behavior of join algorithms with different indexes.
  - Problem: Complex approaches often not applicable in practice since they rest on non-realizable assumptions.

Tuning between Theory and Practice

- Database Tuner: Understand and apply principles!
  - Understanding: The problem is not AVG, but scanning large amounts of data (which AVG often does...).
  - Principle: Do not scan large amounts of data in highly concurrent environments.
  - Understanding the principles is necessary to decide, whether they apply in a particular situation.

Five Basic Tuning Principles

- Five general and basic principles in tuning:
  1. think globally; fix locally
  2. partitioning breaks bottlenecks
  3. start-up costs are high; running costs are low
  4. render on the server what is due on the server
  5. be prepared for trade-offs

Think Globally; Fix Locally (I/II)

- Tuner should be like a good physician:
  - think globally: identify the problem (vs. treating symptoms)
  - fix locally: minimalist intervention (reduce side effects)
- Example: Disk activity is very high. What to do?
- Solution 1: Buy more disks (local thinking).
  - Disk activity is a symptom.
  - Global thinking: Where is the disc activity generated?
    - missing index on frequent query (add index)
    - database buffer is too small (increase buffer)
    - log and frequently accessed data share disk (move log to other disk)
  - Solving the problem is cheaper and more effective than fighting the symptom.

Think Globally; Fix Locally (II/II)

- Solution 2: Speed up query with the longest runtime.
  - Slowest query might be infrequent and take only 1% of overall runtime.
  - Speedup by factor 2 will increase system performance only by 0.5%!
  - Speed up important queries!
- Solution 3: Speed up query with largest share in runtime.
  - The query that slows down the system might be unnecessary.
  - Talk to application programmers. Is the query necessary? Can you achieve the same thing in a simpler way?
- Lesson learned: Look at the whole system when you identify the problem (think globally). Fix the problem where it occurs (fix locally).

Partitioning Breaks Bottlenecks

- What is a bottleneck?
  - rarely *all* parts of a system are saturated.
  - often on part limits the overall performance of the system.
  - bottleneck: the limiting part of the system.
- Example: Highway traffic jam:
  - e.g. due to narrow street segment or merging streets
  - bottleneck: road segment with greatest portion of cars per lane
- Solutions for traffic jam:
  1. make drivers drive faster through narrow street segment
  2. create more lanes
  3. encourage drivers to avoid rush hours
- Solution 1 is a local fix (e.g., add index)
- Solutions 2 and 3 are called partitioning.

Partitioning Breaks Bottlenecks – Strategies

- Partitioning in mathematics:
  - divide a set into mutually disjoint (non-intersecting) parts
  - Example: $A = \{a, b, c, d, e\}$ is a set, $\{\{a, c\}, \{d\}, \{b, e\}\}$ is a partitioning of A
  - database tuning: query load is partitioned
- The two basic partitioning strategies are:
  - divide load over more resources (add lanes)
  - spread load over time (avoid rush hours)

Partitioning Breaks Bottlenecks – Example

- Example 1: Bank accounts
  - A bank has N branches.
  - Most clients access accounts from their home branch.
  - Centralized system is overloaded.
- Solution: Partition in space
  - put account data of clients with home branch i into subsystem i
  - partitioning of physical resources in space

Partitioning Breaks Bottlenecks – Example

- Example 2: Lock contention on free list.
  - free list: list of unused database buffer pages
  - a thread that needs a free page locks the free list
  - during the lock no other thread can get a free page
- Solution: Logical partitioning
  - create several free lists
  - each free list contains pointers to a portion of free pages
  - a thread that needs a free page randomly selects a list
  - with n free lists the load per list is reduced by factor $1/n$
  - logical partitioning of lockable resources

Partitioning Breaks Bottlenecks – Example

- Example 3: Lock and resource contention in system with long and short "online" transactions that access the same data.
- Lock and resource contention:
  - lock contention: many threads lock the same resource (e.g., DB table)
  - resource contention: many threads access the same resource (e.g., disk)
- Long and online transactions:
  - long transactions (e.g., data warehouse loads) hold many locks (e.g., on multiple tables)
  - online transactions are short and need fast response time

Partitioning Breaks Bottlenecks

- Problems:
  - deadlocks may force long transactions to abort
  - online transactions slow because
    - they have to wait for long transactions to finish and release the locks
    - long transactions use up resources (e.g., memory buffer)
- Solution: Partition in time or space
  - partition in time: run long transactions when there is little online transaction activity
  - partition in space: run long transactions (if read only) on out-of-date data on separate hardware
  - serialize long transactions so that they don't interfere with one another

Partitioning Breaks Bottle Necks – Summary

- Types of partitioning:
  - partitioning in space (bank branches)
  - logical partitioning (free lists)
  - partitioning in time (long and short transactions)
- Partition with care: performance not always improved!
  - bank branches: additional communication cost
  - free lists: if one list is empty, need to go to next list
  - transactions: additional offline system
- Lesson learned: When you find a bottleneck,
  1. try to speed up that component (fix locally)
  2. if that does not work, then partition

Start-Up Costs Are High; Running Costs Are Low

- In man-made objects start-up time is often long:
  - cars: ignition system
  - light bulbs: lifetime depends on the number of times they are turned on
  - database systems :-)

Start-Up Costs Are High; Running Costs Are Low

- Reads from disk:
  - expensive to begin read operation
  - once read has started, data can be delivered at high speed
  - Example: reading 64 KB (128 sectors) from a single disk track is less than 2 times slower than reading 512 bytes (1 sector)
- Conclusions:
  - frequently scanned tables should be laid out sequentially on disk
  - frequent query that projects few columns from table with hundreds of columns: vertically partition table
- Note: Holds also for RAM!
  - scanning sequential data from RAM much faster than accessing the same data in different positions
  - RAM (random access memory) is not really random...

Start-Up Costs Are High; Running Costs Are Low

- Network latency:
  - overhead of sending a message is very high
  - additional cost of sending large message over small message is small
  - Example: sending 1 byte packet (message) is almost as expensive as sending 1 KB packet (message)
- Conclusion:
  - sending few large data chunks is better than sending many small ones

Start-Up Costs Are High; Running Costs Are Low

- Query overhead:
  - before a query is executed by the database
    - it is parsed
    - it is optimized
    - and access paths to the data are selected
  - even for small queries: approx. 10000 instructions
- Compiled queries:
  - cache the results of parsing, optimizing, and access path selection
  - next execution of the cached query saves this overhead
  - cached query can be called with different parameters
  - example: queries generated by a form that asks for customers; only the customer data changes, the structure of the query remains unchanged
- Conclusion:
  - compile often executed queries

Start-Up Costs Are High; Running Costs Are Low

- Connection overhead from programming languages:
  - applications written in C++, Java, etc. make calls to databases
  - opening connection: significant overhead
    - establish network connection
    - user authentication
    - negotiate connection parameters
- Connection caching and pooling:
  - open a pool of connections and keep them open
  - new request for a connection uses a free connection from the pool
- Conclusion:
  - do *one* SELECT and loop over results (rather than doing SELECTs in a loop)
  - cache and pool connections

Start-Up Costs Are High; Running Costs Are Low

- Different meanings of start-up cost:
  - obtaining first byte of a read
  - sending first byte of a message
  - preparing a query for execution
  - opening a connection to the database
- Lessen learned: Obtain the effect you want with the fewest possible start-ups.

Render on the Server What Is Due on the Server

- Where to allocate the work?
  - database system (server)
  - application program (client)
- Decision depends on three main factors:
  - relative computing resources of client and server
  - where the relevant information is located
  - whether the database task interacts with the screen

Render on the Server What Is Due on the Server

- Relative computing resources of client and server.
  - if server is overloaded, off-load tasks to clients
  - good candidates: computing (CPU) intensive tasks
- Do computation where the relevant information is located.
  - Example: application responds (e.g., screen message) to database change (e.g., insertions to a table)
  - Client solution: polling
    - periodically query the table for changes
    - inefficient (many queries)
  - Server solution: trigger
    - fires only when change happens
  - Since relevant info is on server, server solution is more efficient

Render on the Server What Is Due on the Server

- Does the database task interact with screen?
  - screen interaction should not be done in a transaction (i.e., not server side)
  - reason: screen transactions take a long time (at least seconds)
  - solution: split transaction as follows
    1. first transaction retrieves data from server
    2. interactive session at the client side (outside any transaction)
    3. second transaction installs changes on server

Be Prepared for Trade-Offs

- Increasing speed has a cost:
  - adding main memory
  - adding disk storage
  - adding CPUs
  - adding new computer systems (e.g., offline system for loads)
  - maintain additional systems
- Making one query faster may slow down an other query!
- Example: index makes critical queries fast, but
  - additional disk space is required
  - index slows down inserts and updates that don't use index
- Lesson learned: You want speed? How much are you willing to pay?

Outline

All Info Regarding Lecture and Lab:

http://dbresearch.uni-salzburg.at/teaching/2015ws/dbt