Systems Infrastructure for Data Science

Web Science Group
Uni Freiburg
WS 2012/13

Lecture VI: Performance Tuning and Benchmarking in Databases

Performance Tuning

- Performance tuning involves adjusting various parameters and design choices to improve a system’s performance **for a specific application**.

- Tuning is best done by
  1. identifying **bottlenecks**, and
  2. eliminating them.

Performance Tuning

- A database system can be tuned at 3 levels:
  - **Hardware:** Examples: adding disks to speed up I/O, adding memory to increase buffer hits, moving to a faster processor.
  - **Database system parameters:** Examples: setting buffer size to avoid paging of buffer, setting checkpointing intervals to limit log size. (System may have automatic tuning.)
  - **Higher level database design:** Examples: tuning the schema, indices, and transactions.

Bottlenecks

- Performance of most systems (at least before they are tuned) is usually limited by the performance of one or a few components: these are called **“bottlenecks”**.
  - Example: 80% of the code may take up 20% of the time, while 20% of the code taking up 80% of the time.
    - It is worth spending most time on 20% of the code that take 80% of the time.
- Bottlenecks may be in hardware (e.g., disks are very busy, CPU is idle), or in software.
- Removing one bottleneck often exposes another.
- “De-bottlenecking” consists of repeatedly finding bottlenecks and removing them.

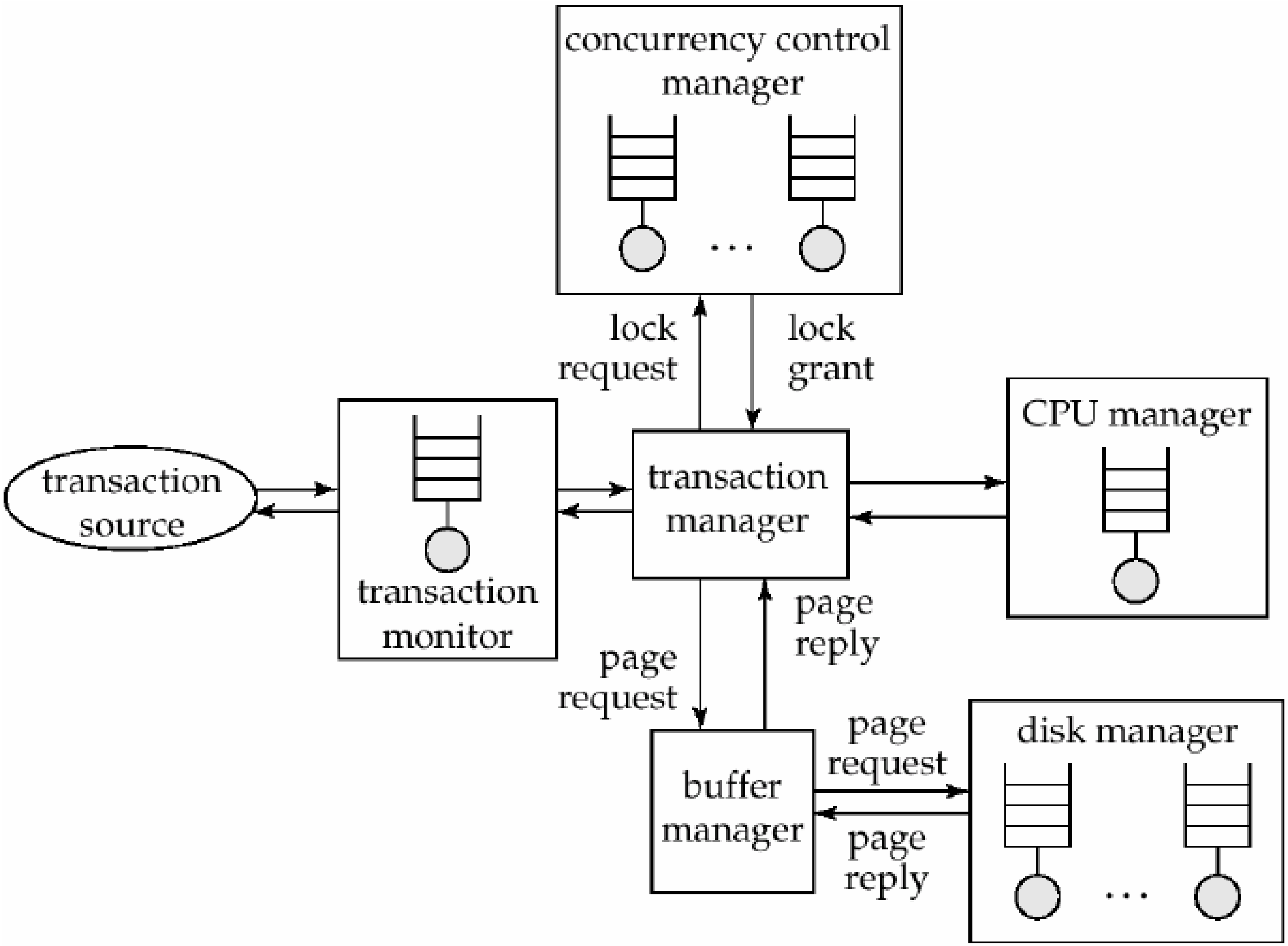
Identifying Bottlenecks

- Transactions request a sequence of services from a database system.
  - Examples: CPU cycles, Disk I/O, locks for concurrency control.

- With concurrent transactions, transactions may have to wait for a requested service while other transactions are being served.

- We can model a database system as a **queueing system** with a queue for each service.
  - Transactions repeatedly do the following:
    - Request a service; Wait in queue for the service; Get serviced.

Queues in a Database System



Identifying Bottlenecks (Cont'd)

- Bottlenecks in a database system typically show up as very **high utilizations** (and correspondingly, very long queues) of a particular service.
  - Example: Disk vs. CPU utilization.

- 100% utilization leads to very long waiting time.
  - **Rule of thumb:** Design the system for about 70% utilization at peak load.
  - Utilizations over 90% should be avoided.

Tunable Parameters

- Database administrators can tune a system at three levels:
  - Hardware level (lowest level)
  - Database system parameters level (system-dependent)
    - Provided in manuals or via automatic tools
  - Database design level (system-independent) (highest level)
    - Tuning of schema
    - Tuning of indices
    - Tuning of materialized views
    - Tuning of transactions
- There is interaction across the levels, and tuning at a higher level may change the bottleneck and affect tuning at the lower levels.

Tuning of Hardware

- Even well-tuned transactions typically require a few I/O operations.
  - Example: Consider a disk that supports about 100 random I/O operations per second of 4KB each.
  - Suppose each transaction requires just 2 random I/O operations. Then to support *n* transactions per second, we need to stripe data across *n*/50 disks. (*n*=50 => 1 disk)
- Number of I/O operations per transaction can be reduced by keeping more data in memory.
  - If all data is in memory, I/O is needed only for writes.
  - Keeping frequently used data in memory reduces disk accesses, reducing number of disks required, but has a memory cost.
  - Memory is much more expensive than disk.

Hardware Tuning: Five-Minute Rule

- Question: Which data to keep in memory?
  - If a page is accessed *n* times per second, keeping it in memory saves:

$$\boxed{n * \frac{\text{price-per-disk-drive}}{\text{accesses-per-second-per-disk}}}$$

  - Cost of keeping page in memory:

$$\boxed{\frac{\text{price-per-MB-of-memory}}{\text{pages-per-MB-of-memory}}}$$

  - Break-even point: value of *n* for which above costs are equal.
    - If accesses are more, then saving is greater than cost.
  - Solving above equation with current disk and memory prices leads to:
    - ➢ **5-Minute Rule:** If a page that is **randomly accessed** is used more frequently than once in 5 minutes, it should be kept in memory (by buying sufficient memory!).

Hardware Tuning: One-Minute Rule

- For **sequentially accessed** data, more pages can be read per second. Assuming sequential reads of 1MB of data at a time:
  - **1-Minute Rule:** Sequentially accessed data that is accessed once or more in a minute should be kept in memory.

- Prices of disk and memory have changed greatly over the years, but the ratios have not changed much.
  - So, the rules still remain as 5-Minute and 1-Minute rules, not 1-Hour or 1-Second rules!

Hardware Tuning: References

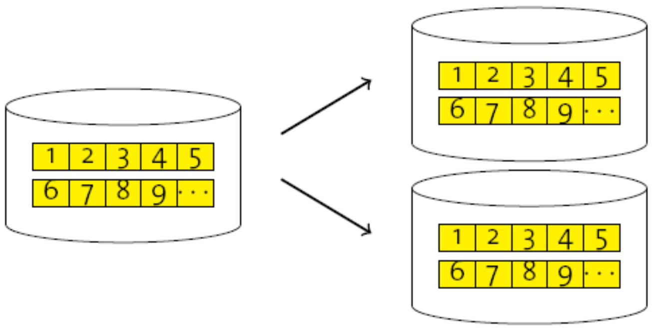
- J. Gray, G. F. Putzolu, “The Five-Minute Rule for Trading Memory for Disk Accesses, and the 10 Byte Rule for Trading Memory for CPU Time”, ACM SIGMOD Conference, June 1987.

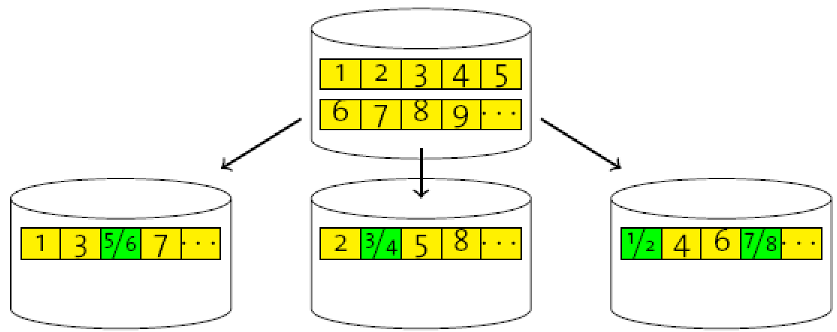
- J. Gray, G. Graefe, “The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb”, ACM SIGMOD Record 26:4, December 1997.

- G. Graefe, “The Five-Minute Rule 20 Years Later, and How Flash Memory Changes the Rules”, ACM Queue 6:4, July/August 2008.

Hardware Tuning: Choice of RAID Level

- To use RAID 1 (disk mirroring) or RAID 5 (disk striping with parity)?





- Depends on ratio of reads and writes.
  - RAID 5 requires 2 block reads and 2 block writes to write out 1 data block (*Note that this is required for parity handling: read old data block + read old parity block + write new data block + write new parity block. Old blocks are needed to compare with the new write request for determining the change in the parity block.*).

Hardware Tuning: Choice of RAID Level

- If an application requires *r* reads and *w* writes per second:
  - RAID 1 requires: $r + 2w$ I/O operations per second.
  - RAID 5 requires: $r + 4w$ I/O operations per second.

- For reasonably large *r* and *w*, this requires lots of disks to handle workload
  - RAID 5 may require more disks than RAID 1 to handle load!
  - Apparent saving of number of disks by RAID 5 (by using parity, as opposed to the mirroring done by RAID 1) may be illusory!

Hardware Tuning: Choice of RAID Level

- **Rule of Thumb:** RAID 5 is fine when writes are rare and data is very large, but RAID 1 is preferable otherwise.
- If you need more disks to handle I/O load, just mirror them, since disk capacities these days are enormous!

Tuning the Database Design: Schema Tuning

- Schema Tuning
  1. **Vertically partition** relations to isolate the data that is accessed more often (i.e., only fetch needed information).
     - Example: *account(<u>account-number</u>, branch-name, balance)*
     - Split *account* into two relations:
       - *account-branch*(<u>*account-number*</u>, *branch-name*)
       - *account-balance*(<u>*account-number*</u>, *balance*)
       - *branch-name* need not be fetched unless required.
     - Normal forms are kept.

Tuning the Database Design: Schema Tuning

- Schema Tuning
  2. Improve performance by storing a **denormalized relation**.
     - Example: Store join of *account* and *depositor*.
       - *account(<u>account-number</u>, branch-name, balance)*
       - *depositor(<u>customer-name, account-number</u>)*
       - *depositor-account(<u>customer-name, account-number</u>, branch-name, balance)*
     - *branch-name* and *balance* information is repeated for each holder of an account, but join need not be computed repeatedly.
     - Price paid: More space and more work for programmer to keep relation consistent on updates.
     - Better to use "materialized views", where the database would maintain the consistency automatically.

Tuning the Database Design: Schema Tuning

- Schema Tuning
  3. **Cluster** together on the same disk page records that would match in a frequently required join ("multi-table clustering file organization").
     - Compute join very efficiently when required.
     - This would be an alternative to (2).

Tuning the Database Design: Index Tuning

- Index Tuning
  - Create appropriate indices to speed up slow queries/updates.
  - Speed up slow updates by removing excess indices (tradeoff between queries and updates).
  - Choose type of index (B-tree/hash) appropriate for most frequent types of queries.
  - Choose which index to make clustered (only one per relation).
  - Index tuning wizards look at past history of queries and updates (the workload) and recommend which indices would be best for the workload.

Tuning the Database Design: Materialized Views

- Materialized Views
  - **Views** are virtual relations. A database normally stores only the query defining the view.
  - A **materialized view** is one whose contents are computed and stored in the database.
  - Materialized views constitute redundant data, but it is useful when we can directly access their contents without recomputing them.

Tuning the Database Design: Materialized Views

- Materialized Views
  - Materialized views can help speed up certain queries (aggregate queries in particular).
  - Overheads
    - Space + Time (for view maintenance):
      - Immediate view maintenance (done as part of update transaction)
      - Deferred view maintenance (done only when required)
        - » until updated, the view may be out-of-date.
  - Preferable to denormalized schema, since view maintenance is system’s responsibility, not programmer’s.
    - Avoids inconsistencies caused by errors in update programs.

Tuning the Database Design: Materialized Views

- Materialized Views
  - How to choose the set of materialized views?
    - Helping one transaction type by introducing a materialized view may hurt others.
    - Choice of materialized views depends on costs.
      - Users often have no idea of actual cost of operations.
    - Overall, manual selection of materialized views is tedious.
  - Some database systems provide tools to help DBA choose views to materialize.
    - “Materialized view selection wizards”

Tuning of Transactions

- Two basic approaches for improving transaction performance:
  1. Improve set orientation
  2. Reduce lock contention

- Improving set orientation:
  - In client-server systems, communication overhead and query handling overheads are significant parts of cost of each call.
  - Combine multiple embedded SQL/ODBC/JDBC queries into a single set-oriented query (which leads to fewer calls to the database).
  - Example: Given a relation *expenses(date, employee, department, amount)*, find total expenses of a given department. Repeat this for a given list of departments.
    - ➢ Instead of repeating the same query for each department one by one, use a single GROUP-BY query (single scan).

Tuning of Transactions

- Reducing lock contention
  - Long transactions (typically read-only) that examine large parts of a relation result in lock contention with update transactions.
  - Example: Large query to compute bank statistics and regular bank transactions.
  - To reduce contention:
    - Use multi-version concurrency control.
      - Example: Oracle “snapshots” which support multi-version 2PL.
    - Use degree-two consistency (read-committed/cursor-stability) for long transactions.
      - Drawback: result may be approximate.

Tuning of Transactions

- Long update transactions cause several problems:
  - Exhaust lock space
  - Exhaust log space
  - Increase recovery time after a crash

- Use “mini-batch transactions” to limit number of updates that a single transaction can carry out (e.g., if a single large transaction updates every record of a very large relation, log may grow too big).
  - Split large transaction into batch of mini-transactions, each performing part of the updates.
  - Hold locks across transactions in a mini-batch to ensure serializability.
  - If lock table size is a problem can release locks, but at the cost of serializability.
  - In case of failure during a mini-batch, must complete its remaining portion on recovery, to ensure atomicity.

Performance Simulation

- Performance simulation using a queueing model is useful to predict bottlenecks as well as the effects of tuning changes, even without access to a real system.
- The queuing model that we saw earlier models the activities that go on in parallel.
- Simulation model is quite detailed, but usually omits some low level of details.
  - Model service time, but disregard details of service, e.g., approximate disk read time by using an average disk read time.

Performance Simulation

- Experiments can be run on the model, and provide an estimate of measures such as average throughput/ response time.
- Service times can be varied to see how sensitive the performance is to each of them.
- Parameters can be tuned in model and then replicated in real system (e.g., number of disks, memory, algorithms, etc.).

Performance Benchmarks

- Benchmarks are suites of standardized tasks used to characterize and quantify the performance of software systems.
- They are useful to get a rough idea of the hardware and software requirements of an application, even before the application is built.
- They are important in comparing database systems, especially as systems become more standards compliant.

Performance Benchmarks

- Commonly used performance measures:
  - **Throughput** (transactions per second, or tps)
  - **Response time** (delay from submission of transaction to return of result)
  - **Availability** or mean time to failure

Performance Benchmarks

- Beware when computing average throughput of different transaction types.
  - Example: Suppose a system runs transaction type A at 99 tps and transaction type B at 1 tps.
  - Given an **equal mixture** of types A and B, throughput is **not** (99+1)/2 = 50 tps.
  - Running one transaction of each type takes time 1+.01 seconds, giving a throughput of 1.98 tps.
  - To compute average throughput, use **"harmonic mean"** of *n* throughputs $t_1, .., t_n$ as follows:

$$\boxed{\frac{n}{1/t_1 + 1/t_2 + \ldots + 1/t_n}}$$

  - Use the above only if the transactions do not interfere with each other (due to lock contention).

Database Application Classes

- **OnLine Transaction Processing (OLTP) applications**
  - require high concurrency and clever techniques to speed up commit processing, to support a high rate of update transactions.
- **Decision support applications**
  - including OnLine Analytical Processing (OLAP) applications.
  - require good query evaluation algorithms and query optimization.
- The architecture of some database systems has been tuned to one of the two classes.
  - Example: Teradata has been tuned to decision support.
- Others try to balance the two requirements.
  - Example: Oracle, with snapshot support for long read-only transactions.

TPC Benchmark Suites

- The Transaction Processing Performance Council (TPC) benchmark suites are widely used.
  - **TPC-A and TPC-B:** Simple OLTP application modeling a bank teller application with and without communication.
    - Not used anymore.
  - **TPC-C:** Complex OLTP application modeling an inventory system.
    - Current standard for OLTP benchmarking.

TPC Benchmark Suites

- TPC benchmarks
  - **TPC-D:** Complex decision support application.
    - Superseded by TPC-H and TPC-R.
  - **TPC-E:** Newer benchmark simulating the OLTP workload of a brokerage firm.
    - Models a central database that executes transactions related to the firm's customer accounts.
    - More read intensive compared to TPC-C.

TPC Benchmark Suites

- TPC benchmarks
  - **TPC-H:** (H for ad hoc) Based on TPC-D with some extra queries.
    - Models ad hoc queries which are not known beforehand (a total of 22 queries with emphasis on aggregation).
    - Prohibits materialized views.
    - Permits indices only on primary and foreign keys.

TPC Benchmark Suites

- TPC benchmarks
  - **TPC-R:** (R for reporting) Same as TPC-H, but without any restrictions on materialized views and indices.
    - Not used any more.
  - **TPC-W:** (W for web) End-to-end web service benchmark modeling a web bookstore, with combination of static and dynamically generated pages.
    - Not used any more.

TPC Performance Measures

- TPC performance measures
  - **transactions-per-second** with specified constraints on response time (TPC-W: web interactions per second (WIPS)).
  - **transactions-per-second-per-dollar** accounts for cost of owning a system (TPC-W: price per WIPS).
- TPC benchmark requires database sizes to be scaled up with increasing transactions-per-second.
  - Reflects real world applications where more customers means more database size and more transactions-per-second.
- External audit of TPC performance numbers mandatory.
  - TPC performance claims can be trusted.

TPC Performance Measures

- Two types of tests for TPC-H and TPC-R
  - **Power test:** Runs queries and updates sequentially, then takes mean to find queries per hour.
  - **Throughput test:** Runs queries and updates concurrently.
    - Multiple streams running in parallel each generates queries, with one parallel update stream.
  - **Composite query per hour metric:** Square root of product of power and throughput metrics.
  - **Composite price/performance metric**

Other Benchmarks

- Object-oriented databases (OODB) transactions require a different set of benchmarks.
  - OO7 benchmark has several different operations, and provides a separate benchmark number for each kind of operation.
  - Reason: Hard to define what is a typical OODB application.

- Other benchmarks under discussion for:
  - XML databases
  - Stream data management
  - Cloud data management

Summary

- Performance tuning
  - Identify bottlenecks and remove them.
  - At 3 levels: Hardware (5-minute rule), Database system parameters (system-dependent, automatic tools), Higher-level design (schema, indices, transactions)
  - Performance simulation

- Performance benchmarking
  - OLTP vs. OLAP workloads
  - TPC benchmark suites