“Capítol 24: Advanced Application Development”

Fitxers i bases de dades

- Performance Tuning
- Performance Benchmarks
- Standardization
- E-Commerce
- Legacy Systems

Performance Tuning

- Adjusting various parameters and design choices to improve system performance for a specific application.
- Tuning is best done by
  1. identifying bottlenecks, and
  2. eliminating them.
- Can tune a database system at 3 levels:
  - **Hardware** - e.g., add disks to speed up I/O, add memory to increase buffer hits, move to a faster processor.
  - **Database system parameters** - e.g., set buffer size to avoid paging of buffer, set checkpointing intervals to limit log size. System may have automatic tuning.
  - **Higher level database design**, such as the schema, indices and transactions (more later)

Bottlenecks

- Performance of most systems (at least before they are tuned) usually limited by performance of one or a few components: these are called bottlenecks
  - E.g., 80% of the code may take up 20% of time and 20% of code takes up 80% of time

    Worth spending most time on 20% of code that take 80% of 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
  - This is a heuristic

Identifying Bottlenecks

- Transactions request a sequence of services
  - E.g., CPU, Disk I/O, locks
- With concurrent transactions, transactions may have to wait for a requested service while other transactions are being served
- Can model database as a queueing system with a queue for each service
  - Transactions repeatedly do the following
    - request a service, wait in queue for the service, and get serviced
- Bottlenecks in a database system typically show up as very high utilizations (and correspondingly, very long queues) of a particular service
  - E.g., disk vs. CPU utilization
  - 100% utilization leads to very long waiting time:
    - Rule of thumb: design system for about 70% utilization at peak load
    - utilization over 90% should be avoided

Queues In A Database System

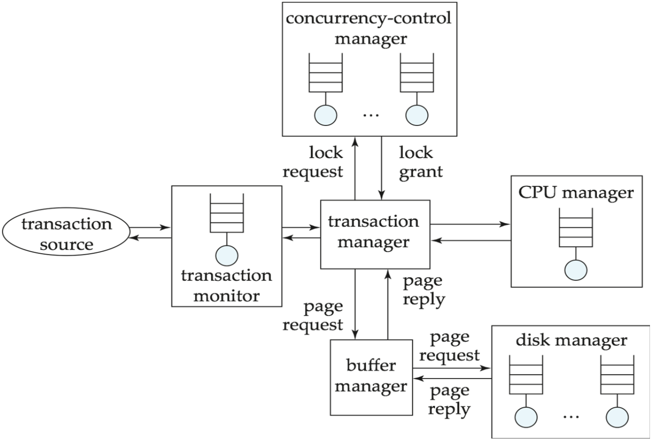

Figure: Example.

Tunable Parameters

- Tuning of hardware
- Tuning of schema
- Tuning of indices
- Tuning of materialized views
- Tuning of transactions

Tuning of Hardware

- Even well-tuned transactions typically require a few I/O operations
  - Typical disk supports about 100 random I/O operations per second
  - 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 (ignoring skew)
- Number of I/O operations per transaction can be reduced by keeping more data in memory
  - If all data is in memory, I/O needed only for writes
  - Keeping frequently used data in memory reduces disk accesses, reducing number of disks required, but has a memory cost

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
    - $n * \frac{\textit{price-per-disk-drive}}{\textit{accesses-per-second-per-disk}}$
  - Cost of keeping page in memory
    - $\frac{\textit{price-per-MB-of-memory}}{\textit{ages-per-MB-of-memory}}$
  - 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 rules remain as 5 minute and 1 minute rules, not 1 hour or 1 second rules!

Hardware Tuning: Choice of RAID Level

- To use RAID 1 or RAID 5?
  - Depends on ratio of reads and writes
    - RAID 5 requires 2 block reads and 2 block writes to write out one data block
- 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!
- Thumb rule: 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
  - Vertically partition relations to isolate the data that is accessed most often – only fetch needed information.
    - E.g., split account into two, (account-number, branch-name) and (account-number, balance).
      - Branch-name need not be fetched unless required
  - Improve performance by storing a denormalized relation
    - E.g., store join of account and depositor; 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 (more on this later..)
  - Cluster together on the same disk page records that would match in a frequently required join
    - Compute join very efficiently when required.

Tuning the Database Design (Cont.)

- 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
- 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 (Cont.)

Materialized Views

- Materialized views can help speed up certain queries
  - Particularly aggregate queries
- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update txn
      - time overhead paid by update transaction
    - Deferred view maintenance: done only when required
    - Update transaction is not affected, but system time is spent on view maintenance
      - Until updated, the view may be out-of-date
- Preferable to denormalized schema since view maintenance is systems responsibility, not programmers
  - Avoids inconsistencies caused by errors in update programs

Tuning the Database Design (Cont.)

- How to choose 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

- Basic approaches to tuning of transactions
  - Improve set orientation
  - Reduce lock contention
- Rewriting of queries to improve performance was important in the past, but smart optimizers have made this less importan
- Communication overhead and query handling overheads significant part of cost of each call
  - Combine multiple embedded SQL/ODBC/JDBC queries into a single set-oriented query
    - Set orientation $\longrightarrow$ fewer calls to database
    - E.g., tune program that computes total salary for each department using a separate SQL query by instead using a single query that computes total salaries for all department at once (using **group by**)
  - Use stored procedures: avoids re-parsing and re-optimization of query

Tuning of Transactions (Cont.)

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

Tuning of Transactions (Cont.)

- Long update transactions cause several problem
  - Exhaust lock space
  - Exhaust log space
    - and also greatly increase recovery time after a crash, and may even exhaust log space during recovery if recovery algorithm is badly designed!
- 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 queuing model useful to predict bottlenecks as well as the effects of tuning changes, even without access to real system
- Queuing model as we saw earlier
  - Models activities that go on in parallel
- Simulation model is quite detailed, but usually omits some low level details
  - Model service time, but disregard details of service
  - E.g., approximate disk read time by using an average disk read time
- Experiments can be run on model, and provide an estimate of measures such as average throughput/response time
- Parameters can be tuned in model and then replicated in real system
  - E.g., number of disks, memory, algorithms, etc.

Performance Benchmarks

- Suites of tasks used to quantify the performance of software systems
- Important in comparing database systems, especially as systems become more standards compliant.
- 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 (Cont.)

- Suites of tasks used to characterize performance
  - single task not enough for complex systems
- Beware when computing average throughput of different transaction types
  - E.g., 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:

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

  - Interference (e.g., lock contention) makes even this incorrect if different transaction types run concurrently

Database Application Classes

- Online transaction processing (OLTP)
  - requires 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, or OLAP applications
  - require good query evaluation algorithms and query optimization.
- Architecture of some database systems tuned to one of the two classes
  - E.g., Teradata is tuned to decision support
- Others try to balance the two requirements
  - E.g., Oracle, with snapshot support for long read-only transaction

Benchmarks Suites

- The Transaction Processing 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

Benchmarks Suites (Cont.)

- TPC benchmarks (cont.)
  - **TPC-D**: complex decision support application
    - Superceded by TPC-H and TPC-R
  - **TPC-H:** (H for ad hoc) based on TPC-D with some extra queries
    - Models ad hoc queries which are not known beforehand
      - Total of 22 queries with emphasis on aggregation
    - prohibits materialized views
    - permits indices only on primary and foreign keys
  - **TPC-R**: (R for reporting) same as TPC-H, but without any restrictions on materialized views and indices
  - **TPC-W**: (W for Web) End-to-end Web service benchmark modeling a Web bookstore, with combination of static and dynamically generated pages

TPC Performance Measures

- TPC performance measures
  - transactions-per-second with specified constraints on response time
  - transactions-per-second-per-dollar accounts for cost of owning system
- 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

- 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
- Benchmarks for XML being discussed

Standardization

- The complexity of contemporary database systems and the need for their interoperation require a variety of standards.
  - syntax and semantics of programming languages
  - functions in application program interfaces
  - data models (e.g., object oriented/object relational databases)
- Formal standards are standards developed by a standards organization (ANSI, ISO), or by industry groups, through a public process
- De facto standards are generally accepted as standards without any formal process of recognition
  - Standards defined by dominant vendors (IBM, Microsoft) often become de facto standards
  - De facto standards often go through a formal process of recognition and become formal standards

Standardization (Cont.)

- Anticipatory standards lead the market place, defining features that vendors then implement .
  - Ensure compatibility of future products
  - But at times become very large and unwieldy since standards bodies may not pay enough attention to ease of implementation (e.g., SQL-92 or SQL:1999)
- Reactionary standards attempt to standardize features that vendors have already implemented, possibly in different ways.
  - Can be hard to convince vendors to change already implemented features. E.g., OODB systems

SQL Standards History

- SQL developed by IBM in late 70s/early 80s
- SQL-86 first formal standard
- IBM SAA standard for SQL in 1987
- SQL-89 added features to SQL-86 that were already implemented in many systems
  - Was a reactionary standard
- SQL-92 added many new features to SQL-89 (anticipatory standard)
  - Defines levels of compliance (entry, intermediate and full)
  - Even now few database vendors have full SQL-92 implementation

SQL Standards History (Cont.)

- SQL:1999
  - Adds variety of new features - extended data types, object orientation, procedures, triggers, etc.
  - Broken into several parts
    - SQL/Framework (Part 1): overview
    - SQL/Foundation (Part 2): types, schemas, tables, query/update statements, security, etc.
    - SQL/CLI (Call Level Interface) (Part 3): API interface
    - SQL/PSM (Persistent Stored Modules) (Part 4): procedural extensions
    - SQL/Bindings (Part 5): embedded SQL for different embedding languages

SQL Standards History (Cont.)

- More parts undergoing standardization process
  - Part 7: SQL/Temporal: temporal data
  - Part 9: SQL/MED (Management of External Data)
    - Interfacing of database to external data sources
      - Allows other databases, even files, can be viewed as part of the database
  - Part 10 SQL/OLB (Object Language Bindings): embedding SQL in Java
  - Missing part numbers 6 and 8 cover features that are not near standardization yet

Database Connectivity Standards.)

- **Open DataBase Connectivity (ODBC)**: standard for database interconnectivity
  - based on Call Level Interface (CLI) developed by X/Open consortium
  - defines application programming interface, and SQL features that must be supported at different levels of compliance
- **JDBC** standard used for Java
- **X/Open XA** standards define transaction management standards for supporting distributed 2-phase commit
- **OLE-DB**: API like ODBC, but intended to support non-database sources of data such as flat files
  - OLE-DB program can negotiate with data source to find what features are supported
  - Interface language may be a subset of SQL
- **ADO (Active Data Objects)**: easy-to-use interface to OLE-DB functionality

Object Oriented Databases Standards.

- **Object Database Management Group (ODMG)** standard for object-oriented databases
  - version 1 in 1993 and version 2 in 1997, version 3 in 2000
  - provides language independent Object Definition Language (ODL) as well as several language specific bindings
- **Object Management Group (OMG)** standard for distributed software based on objects
  - **Object Request Broker (ORB)** provides transparent message dispatch to distributed objects
  - **Interface Definition Language (IDL)** for defining language-independent data types
  - **Common Object Request Broker Architecture (CORBA)** defines specifications of ORB and IDL

XML-Based Standards.

- Several XML based Standards for E-commerce
  - E.g., RosettaNet (supply chain), BizTalk
  - Define catalogs, service descriptions, invoices, purchase orders, etc.
  - XML wrappers are used to export information from relational databases to XML
- Simple Object Access Protocol (SOAP): XML based remote procedure call standard
  - Uses XML to encode data, HTTP as transport protocol
  - Standards based on SOAP for specific applications
    - E.g., OLAP and Data Mining standards from Microsoft

E-Commerce.

- E-commerce is the process of carrying out various activities related to commerce through electronic means
- Activities include:
  - Presale activities: catalogs, advertisements, etc.
  - Sale process: negotiations on price/quality of service
  - Marketplace: e.g., stock exchange, auctions, reverse auctions
  - Payment for sale
  - Delivery related activities: electronic shipping, or electronic tracking of order processing/shipping
  - Customer support and post-sale service

E-Catalogs.

- Product catalogs must provide searching and browsing facilities
  - Organize products into intuitive hierarchy
  - Keyword search
  - Help customer with comparison of products
- Customization of catalog:
  - Negotiated pricing for specific organizations
  - Special discounts for customers based on past history
    - E.g., loyalty discount
  - Legal restrictions on sales
    - Certain items not exposed to under-age customers
- Customization requires extensive customer-specific information

Marketplaces.

- Marketplaces help in negotiating the price of a product when there are multiple sellers and buyers
- Several types of marketplaces
  - Reverse auction
  - Auction
  - Exchange
- Real world marketplaces can be quite complicated due to product differentiation
- Database issues:
  - Authenticate bidders
  - Record buy/sell bids securely
  - Communicate bids quickly to participants
    - Delays can lead to financial loss to some participants
  - Need to handle very large volumes of trade at times
    - E.g., at the end of an auction

Types of Marketplace.

- Reverse auction system: single buyer, multiple sellers.
  - Buyer states requirements, sellers bid for supplying items. Lowest bidder wins. (also known as tender system)
  - Open bidding vs. closed bidding
- Auction: Multiple buyers, single seller
  - Simplest case: only one instance of each item is being sold
  - Highest bidder for an item wins
  - More complicated with multiple copies, and buyers bid for specific number of copies
- Exchange: multiple buyers, multiple sellers
  - E.g., stock exchange
  - Buyers specify maximum price, sellers specify minimum price
  - exchange matches buy and sell bids, deciding on price for the trade
    - e.g., average of buy/sell bids

Order Settlement.

- Order settlement: payment for goods and delivery
- Insecure means for electronic payment: send credit card number
  - Buyers may present some one else's credit card numbers
  - Seller has to be trusted to bill only for agreed-on item
  - Seller has to be trusted not to pass on the credit card number to unauthorized people
- Need secure payment systems
  - Avoid above-mentioned problems
  - Provide greater degree of privacy
    - E.g., not reveal buyers identity to seller
  - Ensure that anyone monitoring the electronic transmissions cannot access critical information

Secure Payment Systems.

- All information must be encrypted to prevent eavesdropping
  - Public/private key encryption widely used
- Must prevent person-in-the-middle attacks
  - E.g., someone impersonates seller or bank/credit card company and fools buyer into revealing information
    - Encrypting messages alone doesn't solve this problem
    - More on this in next slide
- Three-way communication between seller, buyer and credit-card company to make payment
  - Credit card company credits amount to seller
  - Credit card company consolidates all payments from a buyer and collects them together
    - E.g., via buyer's bank through physical/electronic check payment

Secure Payment Systems (Cont).

- Digital certificates are used to prevent impersonation/man-in-the middle attack
  - Certification agency creates digital certificate by encrypting, e.g., seller's public key using its own private key
    - Verifies sellers identity by external means first!
  - Seller sends certificate to buyer
  - Customer uses public key of certification agency to decrypt certificate and find sellers public key
    - Man-in-the-middle cannot send fake public key
  - Sellers public key used for setting up secure communication
- Several secure payment protocols
  - E.g., Secure Electronic Transaction (SET)

Digital Cash.

- Credit-card payment does not provide anonymity
  - The SET protocol hides buyers identity from seller
  - But even with SET, buyer can be traced with help of credit card company
- Digital cash systems provide anonymity similar to that provided by physical cash
  - E.g., Dig Cash
  - Based on encryption techniques that make it impossible to find out who purchased digital cash from the bank
  - Digital cash can be spent by purchaser in parts
    - much like writing a check on an account whose owner is anonymous

Legacy Systems.

- Legacy systems are older-generation systems that are incompatible with current generation standards and systems but still in production use
  - E.g., applications written in Cobol that run on mainframes
    - Today's hot new system is tomorrows legacy system!
- Porting legacy system applications to a more modern environment is problematic
  - Very expensive, since legacy system may involve millions of lines of code, written over decades
    - Original programmers usually no longer available
  - Switching over from old system to new system is a problem
    - more on this later
- One approach: build a wrapper layer on top of legacy application to allow interoperation between newer systems and legacy application
  - E.g., use ODBC or OLE-DB as wrapper

Legacy Systems (Cont.)

- Rewriting legacy application requires a first phase of understanding what it does
  - Often legacy code has no documentation or outdated documentation
  - reverse engineering: process of going over legacy code to
    - Come up with schema designs in ER or OO model
    - Find out what procedures and processes are implemented, to get a high level view of system
- Re-engineering: reverse engineering followed by design of new system
  - Improvements are made on existing system design in this process

Legacy Systems (Cont.)

- Switching over from old to new system is a major problem
  - Production systems are in every day, generating new data
  - Stopping the system may bring all of a company's activities to a halt, causing enormous losses
- Big-bang approach:
  1. Implement complete new system
  2. Populate it with data from old system
     1. No transactions while this step is executed
     2. scripts are created to do this quickly
  3. Shut down old system and start using new system

  - Danger with this approach: what if new code has bugs or performance problems, or missing features
    - Company may be brought to a halt

Legacy Systems (Cont.)

- Chicken-little approach:
  - Replace legacy system one piece at a time
  - Use wrappers to interoperate between legacy and new code
    - E.g., replace front end first, with wrappers on legacy backend
      - Old front end can continue working in this phase in case of problems with new front end
    - Replace back end, one functional unit at a time
      - All parts that share a database may have to be replaced together, or wrapper is needed on database also
  - Drawback: significant extra development effort to build wrappers and ensure smooth interoperation
    - Still worth it if company's life depends on system

FINAL DEL CAPÍTOL 24