Performance Tuning

Tuning the Database Design

- **Schema tuning**
  - Vertically partition relations to isolate the data that is accessed most often -- only fetch needed information.
- Given relation of NID database as:

Person(NID, name, f-name, m-name, DOB, H-no, street, city, thana, district, division, blood-group, height, weight, BMI, income, profession, qualification, spouse-NID)

Person-BIS (…)

Person-EHR(…..)

Applications are

a. Bank account verification system
b. National electronic health record system

Question 36-1: Perform schema tuning for the above 2 applications

Tuning the Database Design

- **Schema tuning**
  - Improve performance by storing a **denormalized relation**
  - Given relations are:

student (id, name, CGPA, DOB)

Takes (ID, course_id, semester, year, grade)

Most of the query needs access to id, name, course_id, grade.

Question 36-2:

a. Perform schema tuning for the above query by denormalization.
b. Explain the drawbacks.
c. How these drawbacks can be removed?

Price paid: more space and more work for programmer to keep relation consistent on updates
Better to use materialized views

Tuning the Database Design

- **Schema tuning**

- Cluster together on the same disk page records that would match in a frequently required join
  - Compute join very efficiently when required.

Question 36-3: Explain how the join will be computed very efficiently in the above case?

Tuning the Database Design (Cont.)

- **Index tuning**
  a. Create appropriate indices to speed up slow queries/updates **(Explain!!)**

  b. Speed up slow updates by removing excess indices (tradeoff between queries and updates) **(Explain!!)**

- Given relation of NID database as:

Person(NID, name, f-name, m-name, DOB, H-no, street, city, thana, district, division, blood-group, height, weight, BMI, income, profession, qualification, spouse-NID)

Queries on {NID, name}, {name, street, city}, {blood-group}, {income}, {weight}, {profession}

Tuning the Database Design (Cont.)

- **Index tuning**
  - Choose type of index (B-tree/hash) appropriate for most frequent types of queries.
  - Choose which index to make clustered

Tuning the Database Design (Cont.)

- **Index tuning**

- 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

- Given relation of NID database as:

Person(<u>NID</u>, name, f-name, m-name, DOB, H-no, street, city, thana, district, division, blood-group, height, weight, BMI, income, profession, qualification, spouse-NID)

Queries on {thana, district, division, avg-income}, {district, division, avg-income}, {division, avg-income}

Option 1: Can be processed from person

Option 2: Can be processed using materialized view. **Explain how?**

Tuning the Database Design (Cont.)

Materialized Views

- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update
      - 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

Tuning the Database Design (Cont.)

Materialized Views

- 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”

Improving Set Orientation

- When SQL queries are executed from an application program, it is often the case that a query is executed frequently, but with different values for a parameter.

- Each call has an overhead of communication with the server, in addition to processing overheads at the server

- For example, consider a program that steps through each department, invoking an embedded SQL query to find the total salary of all instructors in the department:

select sum(*salary*)
from *instructor*
where *dept name*= ?

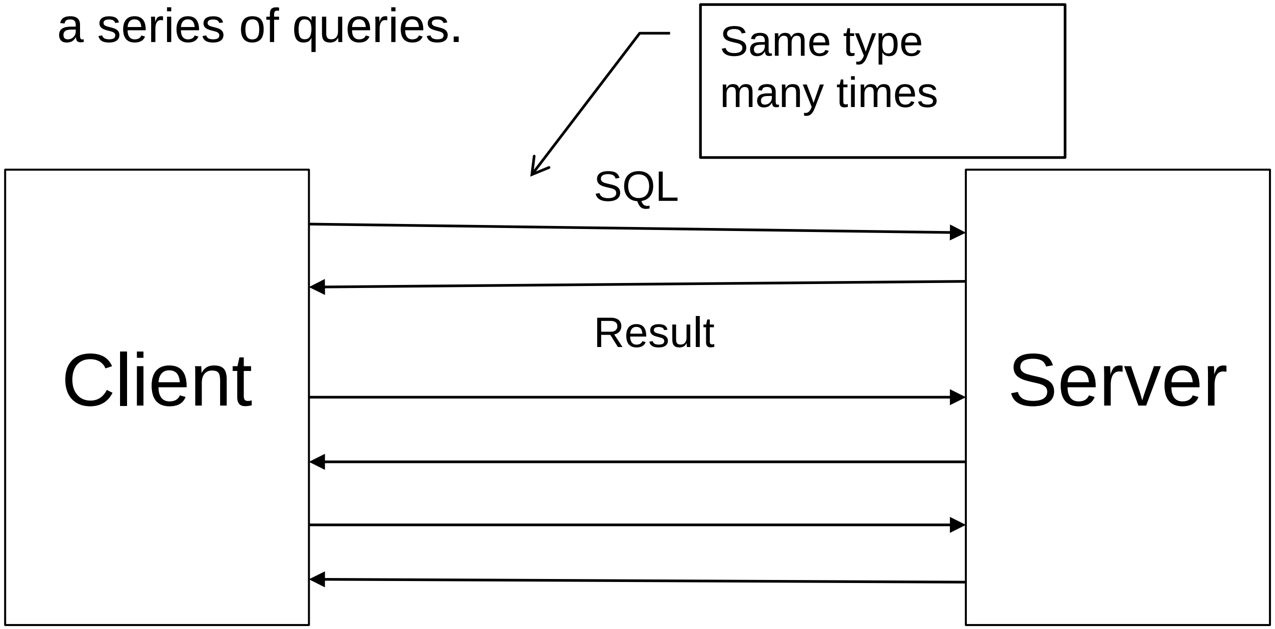
Client side queries are: {CSE}, {ME}, {CE}, …

Question 37-1: How can you improve the above queries?

Improving Set Orientation

- Another technique used widely in client-server systems to reduce the cost of communication and SQL compilation is to use stored procedures, where queries are stored at the server in the form of procedures, which may be precompiled.

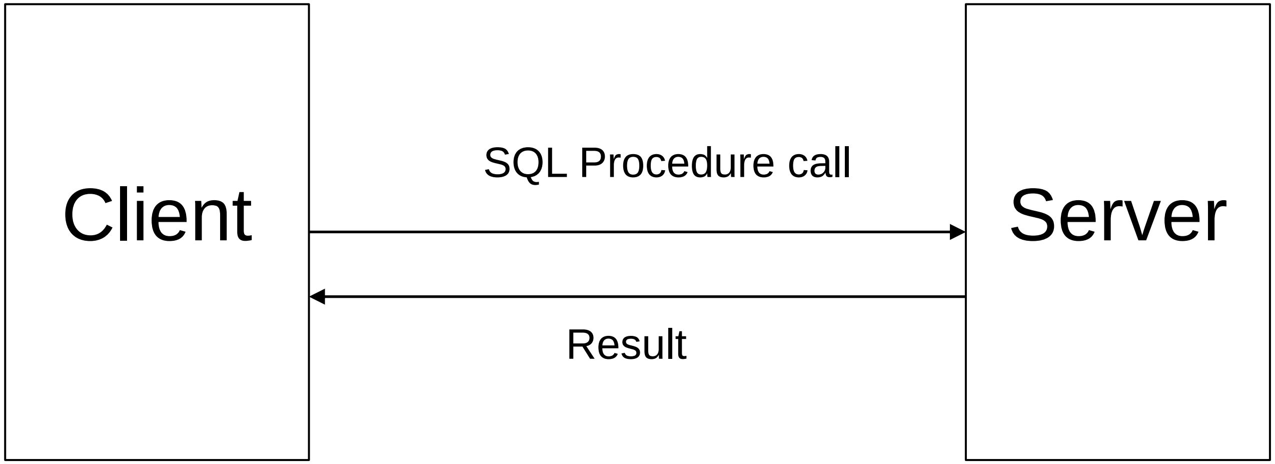
- Clients can invoke these stored procedures rather than communicate a series of queries.



Improving Set Orientation

- Another technique used widely in client-server systems to reduce the cost of communication and SQL compilation is to use stored procedures, where queries are stored at the server in the form of procedures, which may be precompiled.

- Clients can invoke these stored procedures rather than communicate a series of queries.



Tuning of Bulk Loads and Updates

- When loading a large volume of data into a database (called a bulk load operation), performance is usually very poor if the inserts are carried out as separate SQL insert statements **(Why ?)**

- performing integrity constraint checks and index updates separately for each inserted tuple results in a large number of random I/O operations

- To support bulk load operations, most database systems provide a bulk import utility and a corresponding bulk export utility.

- The bulk-import utility reads data from a file and performs integrity constraint checking as well as index maintenance in a very efficient manner

Tuning of Transactions (Cont.)

- Concurrent execution of different types of transactions can sometimes lead to poor performance because of contention on locks

- **read-write contention**

- As an example of read-write contention, consider the following situation on a banking database. During the day, numerous small update transactions are executed almost continuously.

- Suppose that a large query **(select type sum(balance) from account group by type)** that computes statistics on branches is run at the same time. If the query performs a scan on a relation, it may block out all updates on the relation while it runs, and that can have a disastrous effect on the performance of the system.

- Solution - **snapshot isolation**: queries are executed on a snapshot of the data, and updates can go on concurrently

Tuning of Transactions (Cont.)

- Concurrent execution of different types of transactions can sometimes lead to poor performance because of contention on locks

- **Write-write contention**
- Log record <TID, Data item(emp1), old value, new value>
- Lock-x(emp1, TID)
- Long update transactions (**update employee set salary = salary * 1.2**) cause several problems
  - Exhaust lock space (one lock for every update)
  - Exhaust log space (one log record for every update)
    - and also greatly increase recovery time after a crash, and may even exhaust log space during recovery if recovery algorithm is badly designed!

Question 38-1:

a. Lock space is exhausted in Long update transactions. **Explain.**
b. Log space is exhausted in Long update transactions. **Explain.**
c. Recovery time is increased in Long update transactions. **Explain.**

Tuning of Transactions (Cont.)

- 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 Benchmarks

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.)

- Since most software systems, such as databases, are complex, there is a good deal of variation in their implementation by different vendors. As a result, there is a significant amount of variation in their performance on different tasks.

- One system may be the most efficient on a particular task; another may be the most efficient on a different task. Hence, a single task is usually insufficient to quantify the performance of the system.

- Instead, the performance of a system is measured by suites of standardized tasks, called *performance benchmarks*

Performance Benchmarks (Cont.)

- Combining the performance numbers from multiple tasks must be done with care.
- Suppose that we have two tasks, *T*1 and *T*2, and that we measure the throughput of a system as the number of transactions of each type that run in a given amount of time— say, 1 second.
- Suppose that system A runs *T*1 at 99 transactions per second and *T*2 at 1 transaction per second.
- Similarly, let system B run both *T*1 and *T*2 at 50 transactions per second. Suppose also that a workload has an equal mixture of the two types of transactions.
- **Question 39-1:** “The average throughput of system A = (99+1)/2 = 50 tps and for system B = (50+50)/2 = 50 tps”, justify it for the workload with an equal mixture of the two types T1 and T2 of transactions.

Performance Benchmarks (Cont.)

- 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 = (2/1.01) = 1.98 tps.
- To compute average throughput, use **harmonic mean**:

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

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
  - Home page
  - http://www.tpc.org/tpcc/
  - Download
  - http://tpc.org/tpc_documents_current_versions/current_specifications5.asp

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

Question 39-2:

a. Why TPC-H queries emphasises on aggregation?

b. Why TPC-H prohibits materialized view?

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

- TPC-H and TPC-R Performance measure
  - **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:** defined by dividing the system price by the composite metric.

Other Benchmarks

- There are several other TPC benchmarks, such as a data integration benchmark (TPC-DI),

- benchmarks for big data systems based on Hadoop/Spark (TPCx-HS), and

- for back-end processing of internet-of-things data (TPCx-IoT).