Physical Tuning

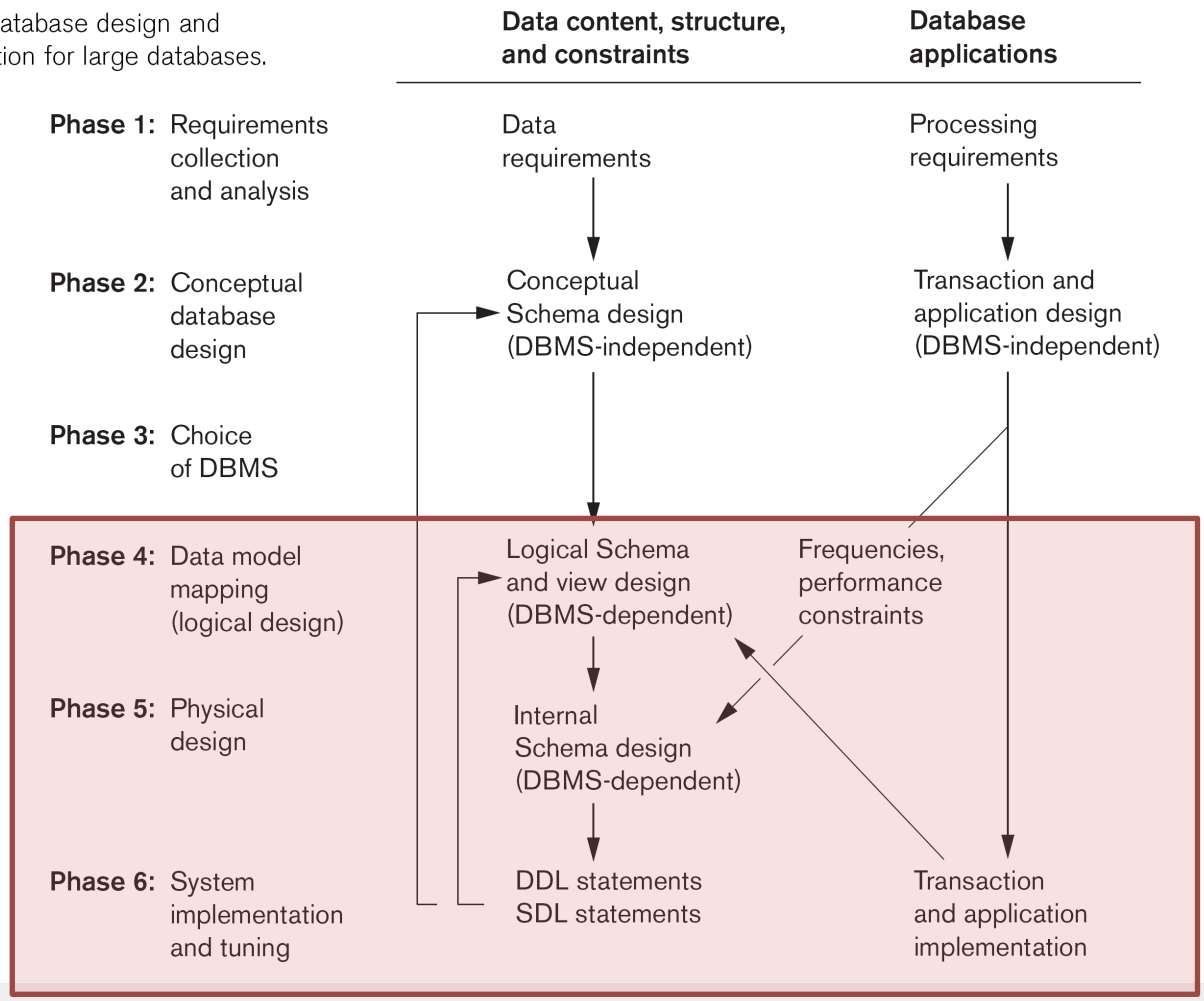
Lecture 12

Outline

- Context
- Influential Factors
- Knobs
  - Denormalization
  - Database Design
  - Query Design

Database Design and Implementation Process

Figure 10.1
Phases of database design and implementation for large databases.



Factors that Influence Physical Tuning

- Attributes w.r.t. Queries/Transactions
  - Queried = potentially good for indexes
  - Updated = bad for indexes
  - Unique = could be indexed
- Frequency of Queries/Transactions
  - 80/20 rule
  - Updates
- Performance Constraints w.r.t. Queries/Transactions
  - e.g. must complete within X seconds
- Profiling
  - Storage allocation
  - I/O performance
  - Query execution time

Tools at Your Disposal

- Indexes
  - Covered in last lecture
  - Note: there may be DBMS-specific configuration settings that can improve performance
- Denormalization
  - Materialized views
- Database design
- Query design

Denormalization

- The goal of normalization is to yield a database schema that is free from redundancies
- Depending upon performance constraints and the job mix, sometimes it is appropriate to *introduce* redundancies (i.e. *denormalize* to 1/2NF) in the name of performance improvement (e.g. to avoid joins)
- **Note**: a schema should always be fully normalized first, and denormalization considered during physical tuning upon analysis of constraints/performance
  - This technique should be deliberate and is not an excuse for sloppy database design

Example: Employee Assignment Roster

ASSIGN(Emp_id, Proj_id, Emp_name, Emp_job_title, Percent_assigned, Proj_name, Proj_mgr_id, Proj_mgr_name)

Proj_id → Proj_name, Proj_mgr_id
Proj_mgr_id → Proj_mgr_name
Emp_id → Emp_name, Emp_job_title

EMP(Emp_id, Emp_name, Emp_job_title)
PROJ(Proj_id, Proj_name, Proj_mgr_id)
EMP_PROJ(Emp_id, Proj_id, Percent_assigned)

Main Approaches to Denormalizing

- Use a **materialized view**
  - Create a new relation on disk, DBMS responsible for automatically updating w.r.t. base relations

- Denormalize the logical data design
  - Implement constraints via DBMS (e.g. triggers) or application logic

Common Denormalization Uses

- Storing derived attributes
  - *Every iPhone has a list of prior owners, each with a name and e-mail. The price of the device depends upon how many prior owners there have been.*

- Adding attributes to a relation from another relation with which it will be joined
  - *Profiling has shown us that every query on employee project assignments has needed the project name.*

- Storing results of calculations on one or more fields within the same relation
  - *We need to store chemicals in base units (e.g. mL), but our most frequent query depends upon larger units (e.g. L)*

Database Design Tuning

Denormalization is one method by which to alter database design to achieve performance goals

Others common approaches…

- Vertical partitioning
- Horizontal partitioning

Vertical Partitioning

Given a normalized relation [typically with many attributes], break into two or more relations, each duplicating the PK, but separating attribute groups

Example:

- Given **R(K,A,B,C,G,H,…)**
  - Knowing that **(A,B,C)** typically together, distinct from **(G, H,…)**
- Yield **R1(K,A,B,C)** and **R2(K,G,H,…)**

Horizontal Partitioning

Given a normalized relation [typically with many rows], break into two or more relations, each with the same columns, but a different subset of rows

Example:

- Given **`ORDER(ID,REGION_ID,…)`**
  - Knowing that typical queries are specific to a region
- Yield **`ORDER_R1(ID,…)`**, **`ORDER_R2(ID,…)`**, …
  - Will require multiple queries/UNION if all orders are to be considered at once

Query Design Tuning

- Indications
  - Profiling indicates too much I/O and/or time
  - The query plan (via **`EXPLAIN`**) shows that relevant indexes are not being used

- The following slides offer common situations in which query tuning might be applicable. For any particular DBMS, see vendor documentation and trade literature.

- Generally speaking, do not attempt to pre-optimize for these situations – let the DBMS/profiling tell you when there is a problem (i.e. avoid premature optimization).

Query Issues (1)

Many query optimizers do not use indexes in the presence of…

- Arithmetic expressions
  - `Salary/2000 > 10.50`
- Numerical comparisons of attributes of different sizes and precision
  - `Aqty = Bqty`, where `Aqty` is `INTEGER` and `Bqty` is `SMALLINTEGER`
- `NULL` comparisons
  - `ReportsTo IS NULL`
- Substring comparisons
  - `Lname LIKE '%mann'`

Some of this (e.g. arithmetic expressions) can be ameliorated with denormalization

Query Issues (2)

Indexes are often not used for nested queries using IN:

```
SELECT Ssn FROM EMPLOYEE
WHERE Dno IN ( SELECT Dnumber FROM DEPARTMENT
WHERE Mgr_ssn = '333445555' );
```

The DBMS may not use the index on **Dno** in **EMPLOYEE**, whereas using **Dno=Dnumber** in the **WHERE**-clause with a single block query may cause the index to be used.

Introducing additional calls to your application may alleviate this type of issue, assuming communication I/O is not prohibitively expensive.

Query Issues (3)

Some **`DISTINCT`**s may be redundant and can be avoided without changing the result. A **`DISTINCT`** often causes a sort operation and must be avoided as much as possible

Query Issues (4)

Avoid correlated queries where possible. Consider the following query, which retrieves the highest paid employee in each department:

```
SELECT Ssn
FROM EMPLOYEE E
WHERE Salary = (SELECT MAX(Salary)
FROM EMPLOYEE M WHERE M.Dno=E.Dno);
```

This has the potential danger of searching all of the inner **EMPLOYEE** table M for each tuple from the outer **EMPLOYEE** table **E**

To make the execution more efficient, the process can be re-written such that one query computes the maximum salary in each department and then is joined

Query Issues (5)

If multiple options for a join condition are possible, choose one that avoids string comparisons

For example, assuming that the Name attribute is a candidate key in **EMPLOYEE** and **STUDENT**, it is better to use **EMPLOYEE.Ssn = STUDENT.Ssn** as a join condition rather than **EMPLOYEE.Name = STUDENT.Name**

Query Issues (6)

One idiosyncrasy with some query optimizers is that the order of tables in the **`FROM`**-clause may affect the join processing. If that is the case, one may have to switch this order so that the smaller of the two relations is scanned and the larger relation is used with an appropriate index.

Some DBMSs have commands by which to influence query optimization (e.g. **`HINT`**)

Query Issues (7)

A query with multiple selection conditions that are connected via **OR** may not be prompting the query optimizer to use any index. Such a query may be split up and expressed as a union of queries, each with a condition on an attribute that causes an index to be used. For example,

```
SELECT Fname, Lname, Salary, Age FROM EMPLOYEE
WHERE Age > 45 OR Salary < 50000;
```

may be executed using table scan giving poor performance. Splitting it up as

```
SELECT Fname, Lname, Salary, Age FROM EMPLOYEE
WHERE Age > 45
UNION
SELECT Fname, Lname, Salary, Age FROM EMPLOYEE
WHERE Salary<50000;
```

may utilize indexes on **Age** as well as on **Salary**