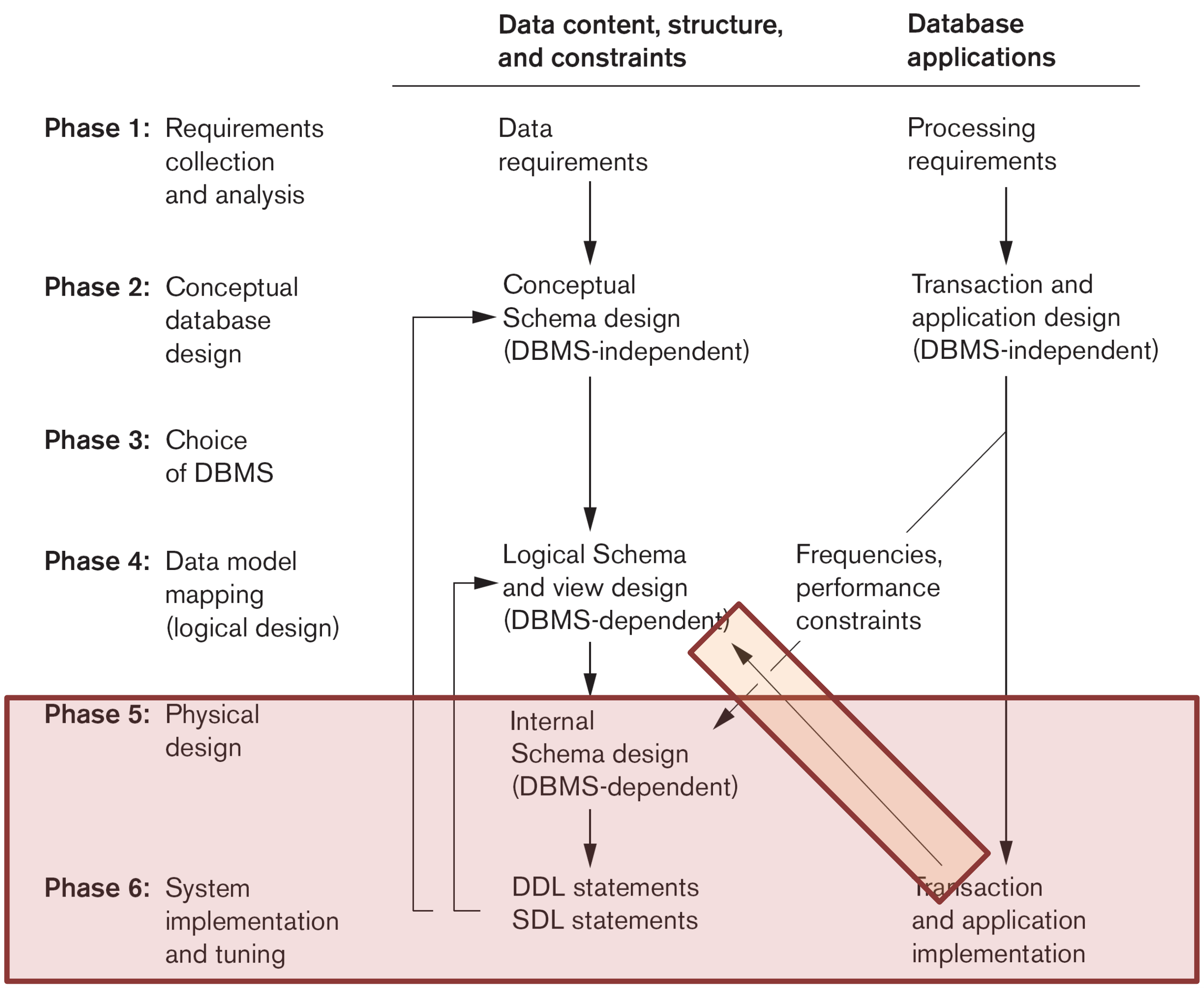
Physical Design & Tuning

Lecture 12

Outline

- Context
- Influential Factors
- Knobs at Your Disposal
  - Indexing
    - Functionality
    - Utility tradeoffs
      - Selectivity
    - Index types
  - Database Design
    - Denormalization
  - Query Design

Database Design and Implementation Process


Data content, structure, and constraints
Database applications
Phase 1: Requirements collection and analysis
Data requirements
Processing requirements
Phase 2: Conceptual database design
Conceptual Schema design (DBMS-independent)
Transaction and application design (DBMS-independent)
Phase 3: Choice of DBMS
Phase 4: Data model mapping (logical design)
Logical Schema and view design (DBMS-dependent)
Frequencies, performance constraints
Phase 5: Physical design
Internal Schema design (DBMS-dependent)
Phase 6: System implementation and tuning
DDL statements
SDL statements
Transaction and application implementation

Factors that Influence Performance

- Attributes in Queries
  - Queried = potentially good for indexes
  - Updated = bad for indexes
  - Unique = could be indexed
- Relative 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

As a general rule with RDBMS's: design for correctness first, profile/gather data, then optimize for performance requirements

What is an Index?

- Persistent data structure, stored in the database
- Primary mechanism to get improved query performance
- Many interesting issues (see Ch. 16-17); we will focus on usage, tradeoffs

Creating an Index

```
CREATE [UNIQUE] INDEX index_name
ON table_name (c_name1, …)
[OPTIONS];
```

Notes

- Ordering of columns is VERY important
- Options often refer to the type of index being used and other important flags

Functionality

An index answers *certain kinds* of questions very efficiently (<u>depends upon type of index</u>)

- **Equality**: fieldname=value
- **Range/ordering**: fieldname>value
  - Only index that maintains ordering (e.g., tree-based)

Can be used for **`WHERE`** clause, as well as **`JOIN`** and **`ORDER BY`**

Comparison (1)

```
SELECT * FROM T
WHERE …
```

- No indexes (indices)
  Anything = full **table scan**
- Index on (A)
  `A = 'panda'` (fast)
  `A > 'dog'` (fast, if ordered index)
  `ORDER BY A` (fast, if ordered)
- Index on (B)
  `B = 1` (fast)
  `B <= 5` (fast, if ordered)
  `ORDER BY B` (fast, if ordered)
- Index on (A, B)
  `A = 'cat'` (fast)
  `A = 'cat' AND B >= 3` (fast, if ordered)
  `A <= 'panda' ORDER BY B` (fast, if ordered)
  Anything not starting with A = full table scan
- Index on (C,A), (C,B), … (i.e. start with C)
  Anything not starting with C = full table scan

T	A	B	C
1	cat	1	…
2	dog	3	…
3	panda	7	…
4	cat	4	…
5	cat	5	…
6	panda	9	…
7	moose	10	…
8	dog	8	…
9	dog	10	…

Comparison (2)

T1	A	B	C
1	cat	1	…
2	dog	3	…
3	panda	7	…
4	cat	4	…

T2	X	Y	Z
i	felidae	1	…
ii	canidae	3	…
iii	bear	7	…
iv	felidae	4	…

`T1 JOIN T2 ON T1.B=T2.Y`

- No indexes: scan T1, scan T2 (n^2)
- Index on T1(B): scan T2, fast search in T1
- Index on T2(Y): scan T1, fast search in T2
- Index on T1(B), T2(Y): merge sort (if ordered)

Utility

Pro

- Can make the difference between full table scan and log/constant lookup

Con

- Extra space
  - Linear with # rows
- Extra time
  - Creation (moderate)
  - Maintenance (can offset savings)

Choosing the Index(es) to Create

- Table size
  - Many rows = larger cost to table scan
- Data distribution (selectivity)
  - Fewer distinct values = higher likelihood needing to touch many rows, independent of index usage
    - Index can lead to lots of IO/cache misses vs. sequential scan via clustered index
- Query vs. update load
  - Many updates = higher relative index maintenance cost
  - Analysis of frequent queries leads to choosing key attributes that get you the most bang for your buck

Selectivity

- **Cardinality**: # distinct values in a column

```
SELECT COUNT(DISTINCT col_name)
FROM table_name;
```

- **Selectivity**: 100% * cardinality / # rows
  - Compare for 10K rows…
    - Yes/No (~ even split)
    - Country (195)
    - Birthday (366: Jan. 1 -> Dec. 31)

General Advice

- Use narrow indexes (i.e., few columns); these are more efficient than compound indices
- Avoid a large number of indices on a table
- Avoid “overlapping” indices that contain shared columns (often a single index can service multiple queries)
- For indices that contain more than one column: given no other constraints, place the most selective column first
- Unless you have very good reason, always define a PK (in most RDBMSs, results in a *clustered* index, more shortly)

Index Types

- Clustered vs. Non-clustered
- Covering (w.r.t. a query)
- Balanced Trees (B+-Trees)
- Hash Tables
- Other

Clustered vs. Non-clustered

- Clustered: affects physical order on disk
  - At most one per table (for some RDBMSs, PK)
  - Fast when data accessed in order/reverse
- Non-clustered: induces logical ordering
  - Arbitrary number per table
  - Depending on the query/data, can lead to significant slowdown due to cache misses and frequent disk access

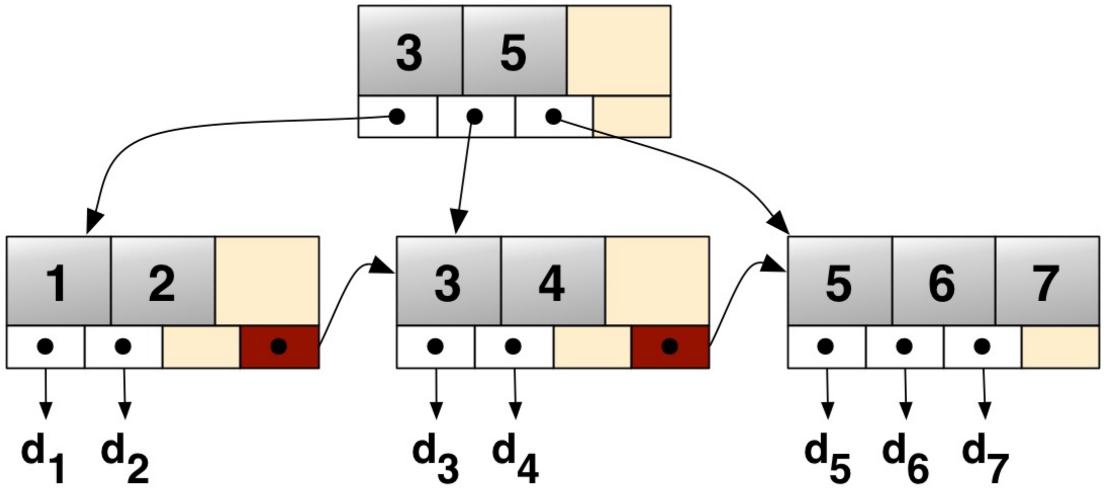
Covering

- Typically indexes help the DBMS *find* the row of interest
  - ID -> Name
  - Name->ID

ID	Name
1	Alice
2	Bob
3	Carol
4	Dan

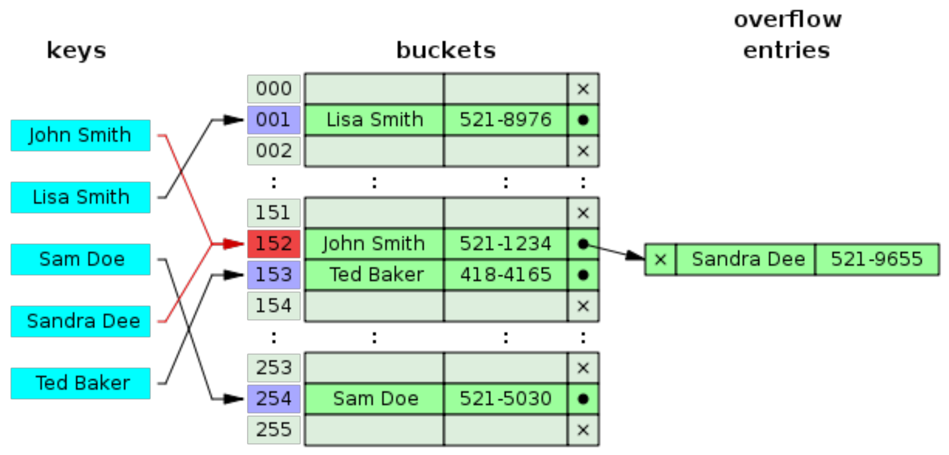
- A covering index contains all the necessary data within the index itself (w.r.t. to query or queries)
  - More storage vs. IO savings
  - (ID, Name) or (Name, ID)

B+-Trees



- Balanced, constant out-degree (within range)
- Values (i.e. row pointer) only at leaves
  - Distinguishes from a B-tree
  - Linked list at leaves, in order
- Logarithmic traversal, constant at leaf
  - Top k levels usually kept in memory (e.g., 2-3)
- Typical default index for DBMS; also used in file systems, etc.

Hash Table



- "Constant" access time (under certain assumptions, amortized)
- No range queries

Other

- Bitmap
  - Useful for low-update systems (e.g., read-only) with low cardinality attributes
- Trie
  - Useful for sequence queries (e.g., bioinformatics)
- Spatial (e.g., R-tree)
  - Useful for queries about space (e.g., what stores are close to me? what planes are within 1 mile of each other?)
- Inverted
  - Useful for full-text search (e.g., search engines)

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

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 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.*
  - *We need to store chemicals in base units (e.g., mL), but our most frequent query depends upon larger units (e.g., L)*
- 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.*

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(`$\underline{\texttt{K}}$`,A,B,C,G,H,…)`**
  - Knowing that **`(A,B,C)`** typically together, distinct from **`(G,H,…)`**
- Yield **`R1(`$\underline{\texttt{K}}$`,A,B,C)`** and **`R2(`$\underline{\texttt{K}}$`,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`**

Summary

- There are many factors that affect **physical design**, which is design related to performance
- In general, design for correctness first, profile/gather data, then optimize for performance requirements
- Common tools at your disposal include indexes, schema design, and query tuning
  - Each has their own issues/tradeoffs