Lec1:

ANSI definition of data:

1 A representation of facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or by automatic means.

2 Any representation such as characters or analog quantities to which meaning is or might be assigned. Generally, we perform operations on data or data items to supply some information about an entity.

Volatile vs persistent data

Our concern is primarily with persistent data

Definition (Database):

A large and persistent collection of factual data and metadata organized in a way that facilitates efficient retrieval and revision.

Definition (Data Model)

A data model determines the nature of the metadata and how retrieval and revision is expressed.

Database Management System (DBMS):

Definition: A program (or set of programs) that implements a data model.

Idea: Abstract common functions and create a uniform well defined interface for applications that require a database.

1. Supports an underlying data model (all data stored and manipulated in a well-defined way)
2. Access control (various data can be accessed or revised only by authorized people)
3. Concurrency control (multiple concurrent applications can access data)
4. Database recovery (reliability; nothing gets accidentally lost)
5. Database maintenance (e.g., revising metadata)

Definition (Schema)

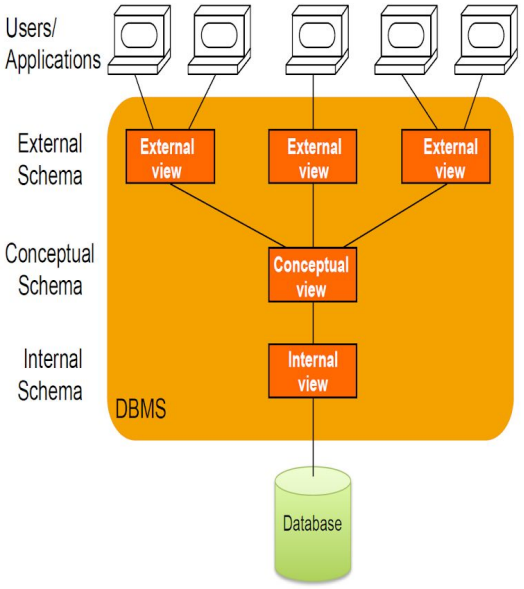
A database schema is a collection of metadata conforming to an underlying data model.

Definition (Instance)

A database instance is a collection of factual data as defined by a given database schema.

Three Level Schema Architecture

1. External schema (view): what the application programs and user see. May differ for different users of the same database.
2. Conceptual schema: description of the logical structure of *all* data in the database.
3. Physical schema: description of physical aspects (selection of files, devices, storage algorithms, etc.)



Data Independence

Idea

Applications do not access data directly but, rather through an abstract data model *provided by the DBMS.*

Two kinds of data independence:

- Physical: applications immune to changes in storage structures
- Logical: modularity:

  The `WAREHOUSE` table is not accessed by the payroll app; the `EMPLOYEE` table is not accessed by the inventory control app.

Transactions (cont'd)

Definition (Transaction)

An application-specified atomic and durable unit of work.

Properties of transactions ensured by the DBMS:

Atomic:	a transaction occurs entirely, or not at all
Consistency:	each transaction preserves the consistency of the database
Isolated:	concurrent transactions do not interfere with each other
Durable:	once completed, a transaction's changes are permanent

Types of Database Users

End user:

- Accesses the database indirectly through forms or other query-generating applications, or
- Generates ad-hoc queries using the DML.

Application developer:

- Designs and implements applications that access the database.

Database administrator (DBA):

- Manages conceptual schema.
- Assists with application view integration.
- Monitors and tunes DBMS performance.
- Defines internal schema.
- Loads and reformats database.
- Is responsible for security and reliability.

Lec2:

Interfacing to the DBMS

Data Definition Language (DDL): for specifying schemas

- may have different DDLs for external schema, conceptual schema, physical schema

Data Manipulation Language (DML): for specifying retrieval and revision requests

- **navigational** (procedural)
- **non-navigational** (declarative)

Using a DBMS to manage data helps:
to remove common code from applications
to provide uniform access to data
to guarantee data integrity
to manage concurrent access
to protect against system failure
to set access policies for data

Set comprehension syntax for questions:

$$\{(x_1, \ldots, x_k) \mid \langle condition \rangle\}$$

Answers:

All *k-tuples* of *values* that satisfy $\langle condition \rangle$.

The Relational Model
Idea
All information is organized in (a finite number of) relations.

Relational Databases

Components:

Universe
- a set of *values* **D** with equality ($=$)

Relation
- predicate name R, and arity k of R (the number of columns)
- instance: a relation $\mathbf{R} \subseteq \mathbf{D}^k$.

Database
- signature: finite set ρ of predicates
- instance: a relation $\mathbf{R_i}$ for each R_i

Notation

Signature: $\rho = (R_1, \ldots, R_n)$ *Instance:* $\mathbf{DB} = (\mathbf{D}, =, \mathbf{R_1}, \ldots, \mathbf{R_n})$

The integers, with addition and multiplication:

$\rho = (\mathtt{PLUS}, \mathtt{TIMES})$ $\mathbf{DB} = (\mathbf{Z}, =, \mathbf{PLUS}, \mathbf{TIMES})$

Predicates (also called table headers):
AUTHOR(aid, name)
WROTE(author, publication)
Relations (also called tables):
AUTHOR = { (1, John), (2, Sue) }
WROTE = { (1, 1), (1, 4), (2, 3) }

Conditions in the Relational Calculus

Idea

Conditions can be formulated using the language of first-order logic.

Definition (Syntax of Conditions)

Given a database schema $\rho = (R_1, \ldots, R_k)$ and a set of variable names $\{x_1, x_2, \ldots\}$, conditions are *formulas* defined by

$$\varphi ::= \underbrace{\underbrace{\underbrace{R_i(x_{i_1}, \ldots, x_{i_k}) \mid x_i = x_j \mid \varphi \wedge \varphi \mid \exists x_i.\varphi}_{\text{conjunctive formulas}} \mid \varphi \vee \varphi}_{\text{positive formulas}} \mid \neg\varphi}_{\text{first-order formulas}}$$

First-order Variables and Valuations

How do we *interpret* variables?

Definition (Valuation)

A valuation is a function θ that maps *variable names* to values in the universe:

$$\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}.$$

To denote a modification to θ in which variable x is instead mapped to value v, one writes:

$$\theta[x \mapsto v].$$

Idea

Answers to queries $\Leftrightarrow$ valuations to free variables that make the formula true with respect to a database.

Complete Semantics for Conditions

Definition

The *truth* of a formula φ is defined with respect to

1. a **database instance DB** $= (\mathbf{D}, =, \mathbf{R}_1, \mathbf{R}_2, \ldots)$, and
2. a **valuation** $\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}$

as follows:

$\mathbf{DB}, \theta \models R(x_{i_1}, \ldots, x_{i_k})$	if $R \in \rho, (\theta(x_{i_1}), \ldots, \theta(x_{i_k})) \in \mathbf{R}$
$\mathbf{DB}, \theta \models x_i = x_j$	if $\theta(x_i) = \theta(x_j)$
$\mathbf{DB}, \theta \models \varphi \wedge \psi$	if $\mathbf{DB}, \theta \models \varphi$ and $\mathbf{DB}, \theta \models \psi$
$\mathbf{DB}, \theta \models \neg\varphi$	if not $\mathbf{DB}, \theta \models \varphi$
$\mathbf{DB}, \theta \models \exists x_i.\varphi$	if $\mathbf{DB}, \theta[x_i \mapsto v] \models \varphi$ for some $v \in \mathbf{D}$

Definition (Queries)

A *query* in the relational calculus is a set comprehension of the form

$$\{(x_1, \ldots, x_k) \mid \varphi\}.$$

Definition (Query Answers)

An *answer* to a *query* $\{(x_1, \ldots, x_k) \mid \varphi\}$ over **DB** is the **relation**

$$\{(\theta(x_1), \ldots, \theta(x_k)) \mid \mathbf{DB}, \theta \models \varphi\},$$

where $\{x_1, \ldots, x_k\} = FV(\varphi)^\dagger$.

$^\dagger FV$ denotes the *free variables* of φ.

Definition (Free Variables)

The *free variables* of a formula φ, written $FV(\varphi)$, are defined as follows:

$$\begin{array}{lcl} FV(R(x_{i_1}, \ldots, x_{i_k})) & \equiv & \{x_{i_1}, \ldots, x_{i_k}\} \\ FV(x_i = x_j) & \equiv & \{x_i, x_j\} \\ FV(\varphi \wedge \psi) & \equiv & FV(\varphi) \cup FV(\psi) \\ FV(\neg\varphi) & \equiv & FV(\varphi) \\ FV(\exists x_i.\varphi) & \equiv & FV(\varphi) - \{x_i\} \end{array}$$

A formula that has no free variables expresses is called a *sentence*.

Integrity Constraints

A relational signature captures only the structure of relations.

Idea:

Valid database instances satisfy additional integrity constraints.

Definition (Relational Database Schema)

A *relational database schema* is a signature ρ and a (finite) set of integrity constraints Σ over ρ.

Definition

A relational database instance **DB** (over a schema ρ) *conforms to a schema* Σ (written $\mathbf{DB} \models \Sigma$) if and only if $\mathbf{DB}, \theta \models \varphi$ for any integrity constraint $\varphi \in \Sigma$ and any valuation θ.

Unsafe Queries

- $\{(y) \mid \neg\exists x.\texttt{author}(x, y)\}$
- $\{(x, y, z) \mid \texttt{book}(x, y, z) \vee \texttt{proceedings}(x, y)\}$
- $\{(x, y) \mid x = y\}$

$\Rightarrow$ we want only queries with finite answers (over finite databases).

Definition (Domain-independent Query)

A query $\{(x_1, \ldots, x_k) \mid \varphi\}$ is *domain-independent* if

$$\mathbf{DB}_1, \theta \models \varphi \iff \mathbf{DB}_2, \theta \models \varphi$$

for any pair of database instances $\mathbf{DB}_1 = (\mathbf{D_1}, =, \mathbf{R_1}, \ldots, \mathbf{R_k})$ and $\mathbf{DB}_2 = (\mathbf{D_2}, =, \mathbf{R_1}, \ldots, \mathbf{R_k})$ and all θ.

Theorem

Answers to domain-independent queries contain only values that exist in $\mathbf{R_1}, \ldots, \mathbf{R_k}$ *(the active domain).*

Domain-independent + finite database $\Rightarrow$ "safe"

Theorem:
Satisfiability1 of first-order formulas is undecidable;
Theorem:
Domain-independence of first-order queries is undecidable.

Proof.

φ is satisfiable iff $\{(x, y) \mid (x = y) \wedge \varphi\}$ is not domain-independent. $\square$

Definition (Range restricted formulas)

A formula φ is *range restricted* when, for φ_i that are also range restricted, φ has the form

$$
\begin{array}{ll}
R(x_{i_1}, \ldots, x_{i_k}), & \\
\varphi_1 \wedge \varphi_2\,, & \\
\varphi_1 \wedge (x_i = x_j) & (\{x_i, x_j\} \cap FV(\varphi_1) \neq \emptyset), \\
\exists x_i.\varphi_1 & (x_i \in FV(\varphi_1)), \\
\varphi_1 \vee \varphi_2 & (FV(\varphi_1) = FV(\varphi_2)), \text{ or} \\
\varphi_1 \wedge \neg\varphi_2 & (FV(\varphi_2) \subseteq FV(\varphi_1)).
\end{array}
$$

Theorem

Range-restricted $\Rightarrow$ *Domain-independent.*

DISTINCT(φ). - range restricted

Theorem
Every domain-independent query can be written equivalently as a range restricted query.
Proof:

1. restrict every variable in φ to active domain,
2. express the active domain using a unary query over the database instance.

Definition (Query Subsumption)

A query $\{(x_1, \ldots, x_k) \mid \varphi\}$ *subsumes* a query $\{(x_1, \ldots, x_k) \mid \psi\}$ with respect to a database schema Σ if

$$\{(\theta(x_1), \ldots, \theta(x_k)) \mid \mathbf{DB}, \theta \models \psi\} \subseteq \{(\theta(x_1), \ldots, \theta(x_k)) \mid \mathbf{DB}, \theta \models \varphi\}$$

for every database **DB** such that $\mathbf{DB} \models \Sigma$.

What queries cannot be expressed in RC?
RC is not Turing-complete
- there must be computable queries that cannot be written in RC.

Lec3:

SQL (cont.)

Three major parts of the language:

1. DML (Data Manipulation Language)
   - ⇒ Query language
   - ⇒ Update language

   Also: Embedded SQL (SQL/J) and ODBC (JDBC)
   ⇒ necessary for application development

2. DDL (Data Definition Language)
   - ⇒ defines *schema* for relations
   - ⇒ creates (modifies/destroys) database objects.

3. DCL (Data Control Language)
   - ⇒ access control

<query> must be unary means there is only one column in the query

---

Lec4:

Duplicates and Queries

How do we define what an answer to a query is now?

Ideas

1. A finite valuation can appear k times ($k > 0$) as a query answer.
2. The number of duplicates is a function of the numbers of duplicates in formulas.
3. $\mathbf{DB}, \theta, k \models \varphi$ reads "finite valuation θ appears k times in φ's answer".

Definition (Multiset Semantics for the Relational Calculus)

$\mathbf{DB}, \theta, 0 \models \varphi$	if	$\mathbf{DB}, \theta \not\models \varphi$
$\mathbf{DB}, \theta, k \models R(x_1, \ldots, x_k)$	if	$(\theta(x_1), \ldots, \theta(x_k)) \in \mathbf{R}$ k times
$\mathbf{DB}, \theta, m \cdot n \models \varphi \wedge \psi$	if	$\mathbf{DB}, \theta, m \models \varphi$ and $\mathbf{DB}, \theta, n \models \psi$
$\mathbf{DB}, \theta, m \models \varphi \wedge (x_i = x_j)$	if	$\mathbf{DB}, \theta, m \models \varphi$ and $\theta(x_i) = \theta(x_j)$
$\mathbf{DB}, \theta, \sum_{v \in D} n_v \models \exists x.\varphi$	if	$\mathbf{DB}, \theta[x := v], n_v \models \varphi$
$\mathbf{DB}, \theta, m + n \models \varphi \vee \psi$	if	$\mathbf{DB}, \theta, m \models \varphi$ and $\mathbf{DB}, \theta, n \models \psi$
$\mathbf{DB}, \theta, m - n \models \varphi \wedge \neg\psi$	if	$\mathbf{DB}, \theta, m \models \varphi$, $\mathbf{DB}, \theta, n \models \psi$ and $m > n$

Outer Join

Idea

Allow "NULL-padded" answers that "fail to satisfy" a conjunct in a conjunction

- Extension of syntax for the `FROM` clause
  ⇒ `FROM` *R* `<j-type>` `JOIN` *S* `ON` *C*
  ⇒ the `<j-type>` is one of `FULL`, `LEFT`, `RIGHT`, or `INNER`
- Semantics (for $R(x, y)$, $S(y, z)$, and $C = (r.y = s.y)$).
  1. $\{(x, y, z) : R(x, y) \wedge S(y, z)\}$
  2. $\{(x, y, \text{NULL}) : R(x, y) \wedge \neg(\exists z.S(y, z))\}$ for `LEFT` and `FULL`
  3. $\{(\text{NULL}, y, z) : S(y, z) \wedge \neg(\exists x.R(x, y))\}$ for `RIGHT` and `FULL`

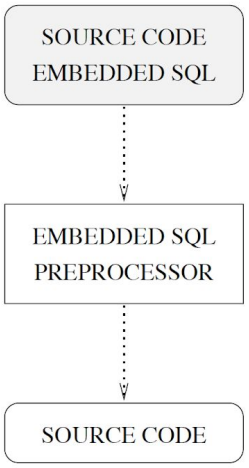
  ⇒ syntactic sugar for `UNION ALL`

Lec5:

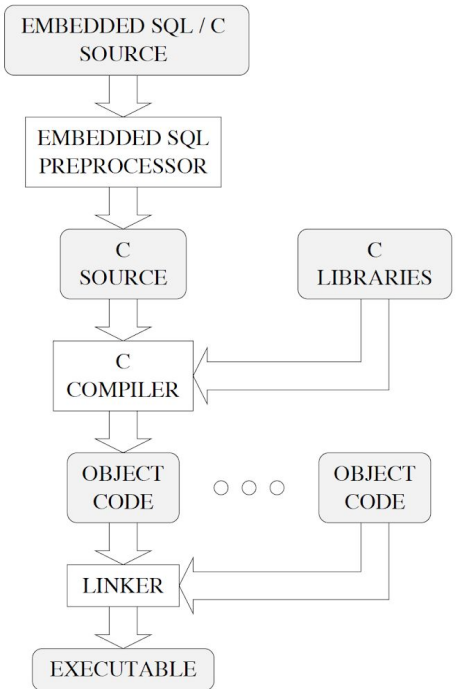
- SQL isn't sufficient to write general applications.
  ⇒ connect it with a general-purpose PL!
- Language considerations:
  ⇒ Library calls (CLI/ODBC)
  ⇒ Embedded SQL
  ⇒ Advanced *persistent* PL (usually OO)
- Client-server:
  ⇒ SQL runs on the server
  ⇒ Application runs on the client

- SQL Statements are *embedded* into a *host language* (C, C++, FORTRAN, . . .)
- The application is *preprocessed*
  pure host language program + library calls
  - Advantages:
    * Preprocessing of (static) parts of queries
    * MUCH easier to use
  - Disadvantages:
    * Needs precompiler
    * Needs to be *bound* to a database

Development Process for Embedded SQL Applications



General structure



- Host Variables are used to pass values between a SQL and the rest of the program
- parameters in SQL statements: communicate single values between SQL a statement and host language variables
- must be declared within SQL declare section:
  EXEC SQL BEGIN DECLARE SECTION;
  …
  EXEC SQL END DECLARE SECTION;
- can be used in the EXEC SQL statements:
  to distinguish them from SQL identifiers they are preceded by ‘:’ (colon)

NULLs and Indicator Variables

- what if a host variable is assigned a NULL?

  ⇒ not a valid value in the datatype
  ⇒ ESQL uses an extra *Indicator* variable, e.g.:

```
smallint ind;
SELECT firstname INTO :firstname
                 INDICATOR :ind
FROM   ...
```

  then if `ind` $<$ 0 then `firstname` is NULL

- if the indicator variable is not provided and the result is a null we get an run-time error
- the same rules apply for host variables in updates.

Don't forget to check errors!
EXEC SQL WHENEVER SQLERROR GO TO error;

Stored Procedures:
A stored procedure executes application logic directly inside the DBMS process.

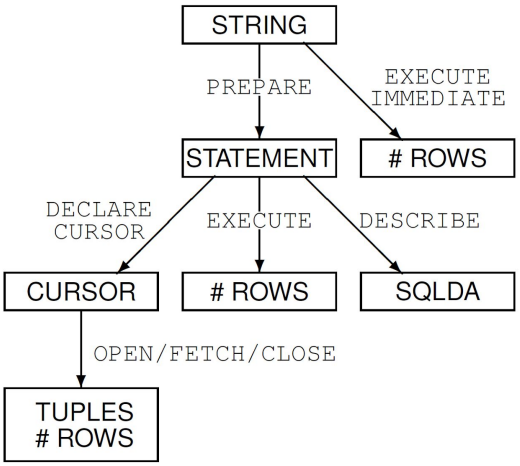
- Possible implementations
  - invoke externally-compiled application
  - SQL/PSM (or vendor-specific language)
- Possible advantages of stored procedures:
  1. minimize data transfer costs
  2. centralize application code
  3. logical independence

Lec6:

Dynamic SQL: execute a string as a SQL statement

we develop an "adhoc" application that accepts an SQL statement as an argument and executes it (and prints out answers, if any).

Dynamic SQL: a Roadmap



EXEC SQL EXECUTE IMMEDIATE :string;
:string is compiled every time we pass through.

PREPARE

We better **compile** a `:string` into a `stmt`...

```
EXEC SQL PREPARE stmt FROM :string;
```

`stmt` can now used for repeatedly executed statements

$\Rightarrow$ avoids recompilation each time we want to execute then

- `:string` may be a query (and return answers).
- `:string` may contain parameters.
- `stmt` is **not** a host variable but an identifier of the statement used by the preprocessor (careful: can't be used in recursion!)

Parametric Statements

How do we pass parameters into SQL statements?

- Static embedded SQL

  ⇒ host variables as parameters

- Dynamic SQL (strings) and **parameters**?

  ⇒ we can change the string (recompilation)
  ⇒ use **parameter marker**: a `"?"` in the string

Idea

Values for `"?"`s are substituted when the statement is to be executed

```
EXEC SQL EXECUTE stmt
         USING   :var1 [,...,:vark];
```

SQLDA: a description of tuple structure

The `sqlda` data structure is a SQL **description area** that defines how a single tuple looks like, where are the data, etc. . .
this is how the DBMS communicates with the application.

DESCRIBE

A prepared statement can be **described**; the description is stored in the SQLDA structure.

EXEC SQL DESCRIBE stmt INTO sqlda

Summary

- given a string:

  ⇒ unknown: `DESCRIBE`
  ⇒ simple statement used once: `EXECUTE IMMEDIATE`
  ⇒ otherwise: `PREPARE`

- given a statement handle (using `PREPARE`):

  ⇒ simple statement: `EXECUTE`
  ⇒ query: `DECLARE CURSOR`
  and then process as a ordinary cursor

Lec7:

Call Level Interface/ODBC

An interface built on a library calls:

- Applications are developed without access to the DB (and without additional tools: no precompilation)
- incorporates ODBC (MS) and X/Open standards
- but it is harder to use and doesn't allow preprocessing (e.g., no checking of your SQL code and data types)

Three fundamental objects in an ODBC program:

- Environments
- Connections
- Statements

- CLI/ODBC can do everything Embedded SQL can.

- However, all statements are *dynamic*

  $\Rightarrow$ no precompilation
  $\Rightarrow$ explicit binding of parameters (user has to make types match!)

- An almost standard (ODBC, X/Open)

  $\Rightarrow$ independence on DBMS
  $\Rightarrow$ but: the standard has 100's of functions

LEC8:

E-R Model: Used for (and designed for) database (conceptual schema) design

World/enterprise described in terms of

- entities
- attributes
- relationships

Visualization: **E-R diagram**

Entity: a distinguishable object
Entity set: set of entities of same type

Attributes: describe properties of entities
Examples (for Student-entities): StudentNum, StudentName, Major
Domain: set of permitted values for an attribute

Relationship: representation of the fact that certain entities are related to each other
Relationship set: set of relationships of a given type
- In order for a relationship to exist, the participating entities must exist.

Role: the function of an entity set in a relationship set
Role name: an explicit indication of a role
- Role labels are needed whenever an entity set has multiple functions in a relationship set.

Relationships may also have attributes

Primary Keys
Each entity must be distinguishable from any other entity in an entity set by its attributes
Primary key: selection of attributes chosen by designer values of which determines the particular entity.

Existence Dependencies
Sometimes the existence of an entity depends on the existence of another entity
If x is existence dependent on y, then
y is a **dominant** entity
x is a **subordinate** entity

Weak entity set: an entity set containing **subordinate** entities
Strong entity set: an entity set containing **no subordinate** entities
Attributes of weak entity sets only form key relative to a given dominant entity
- A weak entity set **must have a many-to-one** relationship to a distinct entity set

Discriminator of a weak entity set: set of attributes that distinguish subordinate entities of the set, for a particular dominant entity
Primary key for a weak entity set: discriminator + primary key of entity set for dominating entities

General cardinality constraints determine lower and upper bounds on the number of relationships of a given relationship set in which a component entity may participate

Structured Attributes
Composite attributes: composed of fixed number of other attributes

Multi-valued attributes: attributes that are set-valued

Aggregation
Relationships can be viewed as higher-level entities

Specialization
A specialized kind of entity set may be derived from a given entity set

Generalization
Several entity sets can be abstracted by a more general entity set

Disjointness
Specialized entity sets are usually disjoint but can be declared to have entities in common

---

Lec9:

Main ideas:

- Each entity set maps to a new table
- Each attribute maps to a new table column
- Each relationship set maps to either new table columns or to a new table

Weak entity set *E* translates to table *E*

Columns of table *E* should include

- Attributes of the weak entity set
- Attributes of the identifying relationship set
- Primary key attributes of entity set for dominating entities

Representing Relationship Sets

- If the relationship set is an identifying relationship set for a weak entity set then no action needed
- If we can deduce the general cardinality constraint (1,1) for a component entity set E then add following columns to table E
  - Attributes of the relationship set
  - Primary key attributes of remaining component entity sets
- Otherwise: relationship set $R \rightarrow$ table R
- Columns of table R should include
  - Attributes of the relationship set
  - Primary key attributes of each component entity set
- Primary key of table R determined as follows
  - If we can deduce the general cardinality constraint (0,1) for a component entity set E, then take the primary key attributes for E
  - Otherwise, choose primary key attributes of each component entity

Create a table for each lower-level entity set only

Columns of new tables should include

- Attributes of lower level entity set
- Attributes of the superset

Definition (View)
A view is a relation whose instance is determined by the instances of other relations.

Types of Views:
Virtual: Views are used only for querying; they are not stored in the database
Materialized: The query that makes up the view is executed, the view constructed and stored in the database.

Updating Views:
- Modifications to a view's instance must be propagated back to instances of

relations in conceptual schema.
- Some views cannot be updated unambiguously.

Materialized Views
Problem:
When a base table changes, the materialized view may also change.

- Solution?
  - Periodically reconstruct the materialized view.
  - Incrementally update the materialized view.

Post Midterm:

Data Control Language

assigns *access rights* to database objects

- Syntax:

```
GRANT  <what> ON <object> TO <user(s)>
REVOKE <what> ON <object> FROM <user(s)>
```

Lecture 10:

How to Find and Fix Anomalies?

Detection: How do we know an *anomaly* exists?

(certain families) of Integrity Constraints postulate
regularities in schema instances that lead to anomalies.

Repair How can we *fix* it?

Certain Schema Decompositions avoid the anomalies
while retaining *all information* in the instances.

Dependencies between attributes in a single relation lead to **improvements** in schema design.

Functional Dependencies (FDs)

Idea: to express the fact that in a relation **schema**

(values of) a set of attributes uniquely **determine**
(values of) another set of attributes.

Definition: Let R be a relation schema, and $X, Y \subseteq R$ sets of attributes. The **functional dependency** $X \to Y$ is the formula

$$\forall v_1, \ldots, v_k, w_1, \ldots, w_k . R(v_1, \ldots, v_k) \wedge R(w_1, \ldots, w_k) \wedge \left(\bigwedge_{j \in X} v_j = w_j\right) \to \left(\bigwedge_{i \in Y} v_i = w_i\right)$$

We say that (the set of attributes) X **functionally determines** Y (in R).

A set F *logically implies* a FD $X \to Y$ if $X \to Y$ holds in *all instances* of R that satisfy F.

The **closure of** F^+ of F is the set of all functional dependencies that are *logically implied by F*

Logical implications can be derived by using inference rules called **Armstrong's axioms**

- (reflexivity) $Y \subseteq X \Rightarrow X \to Y$
- (augmentation) $X \to Y \Rightarrow XZ \to YZ$
- (transitivity) $X \to Y, Y \to Z \Rightarrow X \to Z$

The axioms are

- sound (anything derived from F is in F^+)
- complete (anything in F^+ can be derived)

Additional rules can be derived

- (union) $X \to Y, X \to Z \Rightarrow X \to YZ$
- (decomposition) $X \to YZ \Rightarrow X \to Y$

Keys: formal definition

Definition:

- $K \subseteq R$ is a **superkey** for relation schema R if dependency $K \rightarrow R$ holds on R.

- $K \subseteq R$ is a **candidate key** for relation schema R if K is a superkey and no subset of K is a superkey.

Primary Key = a candidate key choosen by the DBA.

function $ComputeX^+(X, F)$
begin
    $X^+ := X$;
    while true do
        if there exists $(Y \rightarrow Z) \in F$ such that
            (1) $Y \subseteq X^+$, and
            (2) $Z \not\subseteq X^+$
        then $X^+ := X^+ \cup Z$
        else exit;
    return X^+;
end

Theorem: X is a superkey of R if and only if

$$ComputeX^+(X, F) = R$$

Theorem: $X \rightarrow Y \in F^+$ if and only if

$$Y \subseteq ComputeX^+(X, F)$$

Computing a Decomposition

Decomposition

Let R be a relation schema (= set of attributes). The collection $\{R_1, \ldots, R_n\}$ of relation schemas is a **decomposition** of R if

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

A good decomposition does not

- lose information
- complicate checking of constraints
- contain anomalies (or at least contains fewer anomalies)

Lossless-Join Decompositions (cont.)

A decomposition $\{R_1, R_2\}$ of R is lossless if and only if the common attributes of R_1 and R_2 form a superkey for either schema, that is

$$R_1 \cap R_2 \to R_1 \quad \text{or} \quad R_1 \cap R_2 \to R_2$$

A decomposition $D = \{R_1, \ldots, R_n\}$ of R is **dependency preserving** if there is an equivalent set F' of functional dependencies, none of which is interrelational in D.

Avoiding Anomalies

What is a "good" relational database schema?

Rule of thumb: Independent facts in separate tables:

"Each relation schema should consist of a primary key and a set of mutually independent attributes"

$\Rightarrow$ achieved by transformation of a schema to a normal form

Boyce-Codd Normal Form (BCNF)

Schema R is in **BCNF** (w.r.t. F) if and only if
whenever $(X \to Y) \in F^+$ and $XY \subseteq R$, then either

- $(X \to Y)$ is trivial (i.e., $Y \subseteq X$), or
- X is a superkey of R

A database schema $\{R_1, \ldots, R_n\}$ is in BCNF if each relation schema R_i is in BCNF.

Formalization of the goal that **independent relationships** are stored in **separate tables**.

Lossless-Join BCNF Decomposition

function *ComputeBCNF*(R, F)
begin
  Result := $\{R\}$;
  while *some* $R_i \in$ *Result and* $(X \to Y) \in F^+$
      violate the BCNF condition **do begin**
    Replace R_i *by* $R_i - (Y - X)$;
    Add $\{X, Y\}$ *to Result;*
  end;
  return *Result;*
end

It is possible that no lossless join dependency preserving BCNF decomposition exists:

Consider $R = \{A, B, C\}$ *and* $F = \{AB \to C, C \to B\}$.

Third Normal Form (3NF)

Schema R is in **3NF** (w.r.t. F) if and only if whenever $(X \to Y) \in F^+$ and $XY \subseteq R$, then either

- $(X \to Y)$ is trivial, or
- X is a superkey of R, or
- each attribute of Y contained in a candidate key of R

A schema $\{R_1, \ldots, R_n\}$ is in 3NF if each relation schema R_i is in 3NF.

- 3NF is looser than BCNF
  $\Rightarrow$ allows more redundancy
  $\Rightarrow$ $R = \{A, B, C\}$ and $F = \{AB \to C, C \to B\}$.
- lossless-join, dependency-preserving decomposition into 3NF relation schemas always exists.

Minimal Cover

Definition: Two sets of dependencies F and G are **equivalent** iff $F^+ = G^+$.

There are different sets of functional dependencies that have the same logical implications. Simple sets are desirable.

Definition: A set of dependencies G is **minimal** if

1. every right-hand side of an dependency in F is a single attribute.
2. for no $X \to A$ is the set $F - \{X \to A\}$ equivalent to F.
3. for no $X \to A$ and Z a proper subset of X is the set $F - \{X \to A\} \cup \{Z \to A\}$ equivalent to F.

Theorem: For every set of dependencies F there is an equivalent minimal set of dependencies (**minimal cover**).

Finding Minimal Covers

A minimal cover for F can be computed in four steps. Note that each step must be repeated until it no longer succeeds in updating F.

Step 1.
Replace $X \to YZ$ with the pair $X \to Y$ and $X \to Z$.

Step 2.
Remove $X \to A$ from F if $A \in \mathit{ComputeX}^+(X, F - \{X \to A\})$.

Step 3.
Remove A from the left-hand-side of $X \to B$ in F if
B is in $\mathit{ComputeX}^+(X - \{A\}, F)$.

Step 4.
Replace $X \to Y$ and $X \to Z$ in F by $X \to YZ$.

Computing a 3NF Decomposition

A lossless-join 3NF decomposition that is dependency preserving can be efficiently computed

```
function Compute3NF(R, F)
begin
    Result := ∅;
    F' := a minimal cover for F;
    for each (X → Y) ∈ F' do
        Result := Result ∪ {XY};
    if there is no R_i ∈ Result such that
        R_i contains a candidate key for R then begin
          compute a candidate key K for R;
          Result := Result ∪ {K};
    end;
    return Result;
end
```

- Goals: to decompose relational schamas in such a way that the decomposition is

  (1) lossless-join
  (2) dependency preserving
  (3) BCNF (and if we fail here, at least 3NF)

Beyond Functional Dependencies

There exist anomalies/redundancies in relational schemas that cannot be captured by FDs.

Example: consider the following table:

Course	**T**eacher	**B**ook
Math	Smith	Algebra
Math	Smith	Calculus
Math	Jones	Algebra
Math	Jones	Calculus
Advanced Math	Smith	Calculus
Physics	Black	Mechanics
Physics	Black	Optics

There are no (non-trivial) FDs that hold on this scheme; therefore the scheme (Course, Set-of-teachers, Set-of-books) is in BCNF.

How do we Execute Queries?

1. Parsing, typechecking, etc.
2. Relational Calculus (SQL) translated

   to *Relational Algebra*
3. Optimization:

   $\Rightarrow$ generates an efficient *query plan*

   $\Rightarrow$ uses statistics collected about the stored data
4. Plan execution:

   $\Rightarrow$ *access methods* to access stored relations

   $\Rightarrow$ *physical relational operators* to combine relations

Relational Algebra

Idea

Define a *set of operations* on the universe of finite relations...
...called a *RELATIONAL ALGEBRA*.

$$(\mathcal{U}; R_0, \ldots, R_k, \times, \sigma_\varphi, \pi_V, \cup, -)$$

Constants:

- R_i: one for each relational scheme

Unary operators:

- σ_φ: selection (keeps only tuples satisfying φ)
- π_V: projection (keeps only attributes in V)

Binary operators:

- $\times$: Cartesian product
- $\cup$: union
- $-$: set difference

Relational Calculus/SQL to Algebra

How do we know that these operators are sufficient to evacuate *all* Relational Calculus queries?

Theorem (Codd)

For every domain independent Relational Calculus query there is an equivalent Relational Algebra expression.

$$
\begin{aligned}
RCtoRA(R_i(x_1, \ldots, x_k)) &= R_i \\
RCtoRA(Q \wedge x_i = x_j) &= \sigma_{\#i=\#j}(RCtoRA(Q)) \\
RCtoRA(\exists x_i.Q) &= \pi_{FV(Q)-\{\#i\}}(RCtoRA(Q)) \\
RCtoRA(Q_1 \wedge Q_2)) &= RCtoRA(Q_1) \times RCtoRA(Q_2) \\
RCtoRA(Q_1 \vee Q_2)) &= RCtoRA(Q_1) \cup RCtoRA(Q_2) \\
RCtoRA(Q_1 \wedge \neg Q_2)) &= RCtoRA(Q_1) - RCtoRA(Q_2)
\end{aligned}
$$

. . . queries in $\wedge$ must have disjoint sets of free variables

. . . we must *invent* consistent way of referring to attributes

Iterator Model for RA

How do we avoid (mostly) storing *intermediate* results?

Idea

We use the *cursor OPEN/FETCH/CLOSE interface*.

Every *implementation* of an Relational Algebra operator:

1. implements the cursor interface to produce answers
2. uses the *same* interface to get answers from its children

. . . we make (at least) one *physical implementation* per operator.

Physical Operators (cont.)

The rest of the lot:

product:
simple nested loops algorithm

projection:
eliminate *unwanted attributes* from each tuple

union:
simple concatenation

set difference:
nested loops algorithm that checks for tuples on r.h.s.

WARNING!

This doesn't quite work: projection and union may produce *duplicates*
. . . need to be followed by a *duplicate elimination operator*

How to make the query faster?
1. use (disk-based) data structures for efficient searching INDEXING (used, e.g., in selections)
2. use better algorithms to implement the operators commonly based on SORTING or HASHING
3. rewrite the RA expression to an equivalent, but more efficient one remove unnecessary operations (e.g., duplicate elimination) enable the use of better algorithms/data structures

Atomic Relations:
We use the **Access Methods** (defined in last lecture) to gain access to the stored data:
if an index Rindex (x) (where x is the search attribute) is available we replace a subquery of the form

- σx=c(R)

with accessing Rindex (x) directly,
Otherwise: check all file blocks holding tuples for R.

Even if an index is available, scanning the entire relation may be faster in certain circumstances:

- the relation is very small
- the relation is large, but we expect most of the tuples in the relation to satisfy the selection criteria

Joins:
The MOST studied operation of relational algebra; There are many other ways to perform a join.

1. The Nested Loop Join
   ```
   for t in R do
       for u in S do
           if C(t,u) then
               output (tu)
   ```
   => with the optional use of indices on S
2. The Sort-Merge Join
   sort the tuples of R and of S on the common values, then merge the sorted relations.
3. The Hash Join
   hash each tuple of R and of S to “buckets” by applying a hash function to columns involved in the join condition. Within each bucket, look for tuples with the matching values.

the cost of the join depends on the chosen method

Duplicates and Aggregates:
How do we eliminate duplicates in results of operations?
How do we group tuples for aggregation?

Similar solution:
- sort the result and then eliminate duplicates/aggregate
- hash the result and do the same

=> often an index (e.g. a B+tree) can be used to avoid the sorting/hashing phase

The rest of the lot:
- we assume a natural implementation for selection, duplicate-preserving projection, and duplicate preserving union.
- set difference can be evaluated similarly to a join

additional operations:
- sorts. Use and **external** sort algo(essentially a merge-sort adopted for disk)
- temporary store(to avoid recomputation of subqueries; can be inserted anywhere in the query plan)

Query Optimization:
- Many possible **query plans** for a single query:
  1. equivalences in Relational Algebra
  2. choice of Operator Implementation

=> performance differs greatly

How do we choose the best plan?
1. “always good” transformations
2. cost-based model

=> find an **optimal plan** is computationally

General Approach:

1. generate all physical plans equivalent to the query
2. pick the one with the lowest cost

Cannot be done in general:

1. It is **undecidable** if a query satisfiable or unsatisfiable
   equivalent to an empty plan
2. Very **expensive** even for **conjunctive** queries
   => the Join-ordering problem
3. In practice:
   => only plans of certain form are considered(restrictions on the search space)
   => the goal is to eliminate the really bad ones

How do we determine which plan is the best one?
- We estimate the cost based on stats collected by the DBMS for all relations

A **Simple Cost Model** for disk I/O; Assumptions:
Uniformity: all possible values of an attribute are equally likely to appear in a relation

Independence: the likelihood that an attribute has a particular value(in a tuple) does not depend on the values of other attributes.

For a stored relation R with an attribute A we keep:
|R|: the cardinality of R(the number of tuples in R)
b(R): the **blocking factor** for R
min(R,A): the minimum value for A in R
max(R,A): the maximum value for A in R
distinct(R,A): the number of distinct values of A
- Based on these values we try to estimate the **cost of physical plans**.

EXAMPLE:
Cost of Retrieval

```
Mark(Studnum, Course, Assignnum, Mark)
SELECT Studnum, Mark
FROM Mark
WHERE Course = 'PHYS'
AND Studnum = 100
AND Mark > 90
```

Indices:
1. clustering index CourseInd on Course
2. non-clustering index StudnumInd on Studnum

Assume:

|Mark| = 10000
b(Mark) = 50
500 different students
100 different courses
100 different marks

Strategy 1: Use CourseInd
- Assuming uniform distribution of tuples over the courses, there will be about |Mark|/100 = 100 tuples with Course = PHYS.
- Searching the CourseInd index has a cost of 2. Retrieval of the 100 matching tuples adds a cost of **100/b(Mark)** data blocks. The total cost of 4.

Selection of N tuples from relation R using a clustered index has a cost of **2 + N/b(R)**.

Strategy 2: Use StudnumInd
- Assuming uniform distribution of tuples over student numbers, there will be about |Mark|/500 = 20 tuples for each student.
- Searching the StudnumInd has a cost of 2. Since this is not a clustered index, we will make the pessimistic assumption that each matching record is on a separate data block, i.e., 20 blocks will need to be read. The total cost is 22.

Selection of N tuples from relation R using a non-clustered index has a cost of **2 + N**.

Strategy 3: Scan the relation
The relation occupies 10,000/50 = 200 blocks, so 200 block I/O operations will be required.

Selection of N tuples from relation R by scanning the entire relation has a cost of |R|/b(R).

Cost of other Relational Operations

Costs of **physical** operations (in I/O's):

- Selection: $\text{cost}(\sigma_c(E)) = (1 + \epsilon_c)\,\text{cost}(E)$.
- Nested-Loop Join (R is the **outer** relation):
$$\text{cost}(R \bowtie S) = \text{cost}(R) + (|R|/b)\,\text{cost}(S)$$
- Index Join (R is the outer relation, and S is the inner relation: B-tree with depth d_S):
$$\text{cost}(R \bowtie S) = \text{cost}(R) + d_S|R|$$
- Sort-Merge Join:
$$\text{cost}(R \bowtie S) = \text{cost}(\text{sort}(R)) + \text{cost}(\text{sort}(S))$$
where $\text{cost}(\text{sort}(E)) = \text{cost}(E) + (|E|/b)\log(|E|/b)$.
- ...

Size Estimation

In the cost estimation we need to know sizes of results of operations: we use the **selectivity**, defined, for a condition $\sigma_{\text{condition}}(R)$, as:

$$\text{sel}(\sigma_{\text{condition}}(R)) = \frac{|\sigma_{\text{condition}}(R)|}{|R|}$$

Again, the optimizer will *estimate* selectivity using simple rules based on its statistics:

$$\text{sel}(\sigma_{A=c}(R)) \approx \frac{1}{\text{distinct}(R, A)}$$

$$\text{sel}(\sigma_{A\leq c}(R)) \approx \frac{c - \min(R, A)}{\max(R, A) - \min(R, A)}$$

$$\text{sel}(\sigma_{A\geq c}(R)) \approx \frac{\max(R, A) - c}{\max(R, A) - \min(R, A)}$$

Size Estimation (cont.)

For Joins:

- General Join (on attribute A):

$$|R \bowtie S| \approx |R| \frac{|S|}{\text{distinct}(S, A)}$$

  or as

$$|R \bowtie S| \approx |S| \frac{|R|}{\text{distinct}(R, A)}$$

- Foreign key Join (Student and Enrolled joined on Sid):

$$|R \bowtie S| = |S| \frac{|R|}{|S|} = |R|$$

  May joins are foreign key joins, like this one.

Plan Generation:

1. apply “always good” transformations
   => **heuristics** that work in the majority of cases
2. cost-based join-order selection
   =>applied on **conjunctive subqueries** (the “select blocks”)
   => still computationally not tractable

- Push selections:

$$\sigma_{\varphi}(E_1 \bowtie_{\theta} E_2) = \sigma_{\varphi}(E_1) \bowtie_{\theta} E_2$$

  for φ involving columns of E_1 only (and vice versa).
- Push projections:

$$\pi_V(R \bowtie_{\theta} S) = \pi_V(\pi_{V_1}(R) \bowtie_{\theta} \pi_{V_2}(S))$$

  where V_1 is the set of all attributes of R involved in θ and V (similarly for V_2).
- Replace products by joins:

$$\sigma_{\varphi}(R \times S) = R \bowtie_{\varphi} S$$

$\Rightarrow$ also reduces the space of plans we need to search

Join Order Selection

- Joins are associative $R \bowtie S \bowtie T \bowtie U$ can be equivalently expressed as
  1. $((R \bowtie S) \bowtie T) \bowtie U$
  2. $(R \bowtie S) \bowtie (T \bowtie U)$
  3. $R \bowtie (S \bowtie (T \bowtie U))$

  $\Rightarrow$ try to minimize the intermediate result(s).
- Moreover, we need to decide which of the subexpressions is evaluated first

  $\Rightarrow$ e.g., Nested Loop join's cost is **not** symmetric!

Pipelined Plans:
- all operators (except sorting) operate without storing intermediate results
=> iterator protocols in constant storage
=> no recomputation for **left-deep** plans

Temporary Store:
- General pipelined plans lead to recomputation
- We introduce an additional store operator
=> allows us to store intermediate results in a relation
=> we can also built a (hash) index on top of the result
Semantically, the operator represents the **identity**

- The costs of plans:
  1. cumulative cost—to compute the value of the expression and store then in a relation (once):
  $$\text{cost}_c(\text{store}(E)) = \text{cost}_c(E) + \text{cost}_s(E) + |E|/b$$
  2. scanning cost—to "read" all the tuples in the stored result of the expression:
  $$\text{cost}_s(\text{store}(E)) = |E|/b$$

Parallelism in Query Execution:
Another approach to improving performance: **take advantage of parallelism in hardware**
- mass storage usually reads/writes data in blocks
- multiple mass storage units can be accessed in parallel
- relational operators amenable to parallel execution

Summary

Relational Algebra is the basis for efficient implementation of SQL

- provides a connection between conceptual and physical level
- breaks query execution to (easily) manageable pieces
- allows the use of efficient algorithms/data structures
- provides mechanism for *query optimization* based on logical transformations (including simplifications based on integrity constraints, etc.)

Performance of database operations depends on the way queries (and updates) are executed against a particular *physical schema/design*.

. . . understanding *basics* of query processing is necessary to making *physical design decisions*

. . . performance also depends on *transaction management* (later)

Chapter 13:

Physical Database Design and Tuning:
Physical Design: The process of selecting a physical schema(collection of data structures) to implement the conceptual schema

Tuning: Periodically adjusting the physical and/or conceptual schema of a working system to adapt to changing requirement and/or performance characteristics

Good design and tuning requires understanding the database **workload**

Workload Modeling:
Definition(Workload Description):
A workload description contains

- the most important queries and their frequency
- the most important updates and their frequence
- the desired performance goal for each query or update

- For each query:
  - Which relations are accessed?
  - Which attributes are retrieved?
  - Which attributes occur in selection/join conditions? How *selective* is each condition?
- For each update:
  - Type of update and relations/attributes affected.
  - Which attributes occur in selection/join conditions? How *selective* is each condition?

Database Tuning:
Goal: Make a set of applications execute “as fast as possible”

How can we affect performance (as DBAs)?

- make queries run faster (data structures, clustering, replication)
- make updates run faster (locality of data items)
- minimize congestion due to concurrency

The Physical Schema:
A storage strategy is chosen for each relation

- Possible storage options: Unsorted(Heap) file, Sorted file, Hash file

Indexes are then added

- Speed up queries
- Extra update overhead
- Possible index types: B-trees, R trees, Hash tables, ISAM,VSAM

Create Indexes:

```
CREATE INDEX LastnameIndex
ON Employee(Lastname) [CLUSTER]

DROP INDEX LastnameIndex
```

Primary effects of `LastnameIndex`:

- Substantially reduce execution time for selections that specify conditions involving `Lastname`
- Increase execution time for insertions
- Increase or decrease execution time for updates or deletions of tuples from `Employee`
- Increase the amount of space required to represent `Employee`

Clustering vs. Non-Clustering Indexes:
- An index on attribute A of a relation is a clustering index if tuples in the relation with similar values for A are stored together in the same block.
- Other indices are non-clustering (or secondary) indices

Note: A relation may have at most one clustering index, and any number of non-clustering indices.

Co-Clustering:
Definition: Two relations are co-clustered if their tuples are interleaved within the same file

Co-clustering is useful for storing hierarchical data (1:N relationships)

Effects on performance:
- Can speed up joins, particularly foreign-key joins
- Sequential scans of either relation become slower

Range Queries:

- B-trees can also help for **range queries**:

```
SELECT *
FROM R
WHERE A ≥ c
```

- If a B-tree is defined on A, we can use it to find the tuples for which $A = c$. Using the forward pointers in the leaf blocks, we can then find tuples for which $A > c$.

Multi-Attribute Indices:

It is possible to create an index on several attributes of the same relation. For example:

```
CREATE INDEX NameIndex
ON Employee(Lastname,Firstname)
```

The order in which the attributes appear is important. In this index, tuples (or tuple pointers) are organized first by `Lastname`. Tuples with a common surname are then organized by `Firstname`.

Physical Design Guidelines

1. Don't index unless the performance increase outweighs the update overhead
2. Attributes mentioned in WHERE clauses are candidates for index search keys
3. Multi-attribute search keys should be considered when
   - a WHERE clause contains several conditions; or
   - it enables index-only plans
4. Choose indexes that benefit as many queries as possible
5. Each relation can have at most one clustering scheme; therefore choose it wisely
   - Target important queries that would benefit the most
     - Range queries benefit the most from clustering
     - Join queries benefit the most from co-clustering
   - A multi-attribute index that enables an index-only plan does not benefit from being clustered

Index Selection and Tools

Idea

Convert *physical design* into another *optimization problem*

- generate the space of all possible physical designs
- pick the best one *based on a given WORKLOAD*

Workload

An abstraction of applications executed against a database:

- list of queries
- list of updates
- frequencies/probabilities of the above
- sequencing constraints
- ...

Schema Tuning and Normal Forms

So far we only *added data structures* to improve performance.
what to do if this isn't enough?

Changes to the *conceptual design*

Goals:

- avoid expensive operations in query execution (joins)
- retrieve *related data* in fewer operations

Techniques:

- alternative normalization/weaker normal form
- co-clustering of relations (if available)/denormalization
- vertical/horizontal partitioning of data (and views)
- avoiding concurrency hot-spots

Tuning the Conceptual Schema:
Adjustments can be made to the conceptual schema:

- Re-normalization
- Denormalization
- Partitioning

Normalization is the process of decomposing schemas to reduce redundancy
Denormalization is the process of merging schemas to intentionally increase redundancy

In general, redundancy increases update overhead (due to change anomalies) but decreases query overhead.
The appropriate choice of normal form depends **heavily** upon the workload.

Partitioning:

- Very large tables can be a source of performance bottlenecks
- *Partitioning* a table means splitting it into multiple tables for the purpose of reducing I/O cost or lock contention
  1. Horizontal Partitioning
     - Each partition has all the original columns and a subset of the original rows
     - Tuples are assigned to a partition based upon a (usually natural) criteria
     - Often used to separate operational from archival data
  2. Vertical Partitioning
     - Each partition has a subset of the original columns and all the original rows
     - Typically used to separate frequently-used columns from each other (concurrency *hot-spots*) or from infrequently-used columns

Tuning Queries:

- Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.
- Sometimes desirable to target performance of specific queries or applications
- Guidelines for tuning queries:
  1. Sorting is expensive. Avoid unnecessary uses of ORDER BY, DISTINCT, or GROUP BY.
  2. Whenever possible, replace subqueries with joins
  3. Whenever possible, replace correlated subqueries with uncorrelated subqueries
  4. Use vendor-supplied tools to examine generated plan. Update and/or create statistics if poor plan is due to poor cost estimation.