Introduction to Database

Database Lecture Notes

1. Introduction

- **Definition:** Collection of interrelated data with programs to access it.
- **Purpose:** Efficiently store and retrieve data for enterprises, ensuring safety and preventing anomalies.

1.1 Database-System Applications

- **Evolution:** From simple commercial data management to complex, global enterprises.
- **Common Elements:** Data centrality, crucial for modern corporations' value and functionality.

Abstraction and Complexity Management

- **Abstraction:** Simplifies complex systems for user interaction.
- **Database Systems:** Provide a unified view of data, managing complexity effectively.

Representative Applications

- **Enterprise Information:** Sales, Accounting, Human Resources, Manufacturing.
- **Banking and Finance:** Banking, Credit Card Transactions, Finance.
- **Other Sectors:** Universities, Airlines, Telecommunication, Web-based services, Document databases, Navigation systems.

User Interaction with Databases

- **Evolution:** From back-office systems to direct customer interactions via web and mobile applications.
- **Usage:** Nearly every interaction with modern technology involves accessing databases.

Modes of Database Usage

1. **Online Transaction Processing:** Many users, small data retrieval and updates.
2. **Data Analytics:** Processing data for drawing conclusions, driving business decisions.

Data Analytics in Practice

- **Examples:** Loan approvals, Advertisement targeting, Manufacturing and Retail decisions.
- **Techniques:** Data mining combines knowledge discovery and statistical analysis on large databases.

1.2 Purpose of Database Systems

In a university organization, managing information about instructors, students, departments, and courses is crucial. Traditional file-based systems store this data with application programs to manipulate it. However, this approach faces several challenges:

1. **Data Redundancy and Inconsistency:** Different file structures and programming languages lead to duplicated data and inconsistencies.

2. **Difficulty in Accessing Data:** Retrieving specific data requires custom application programs, leading to inefficiency.
3. **Data Isolation:** Scattered data in various formats make developing new applications challenging.
4. **Integrity Problems:** Enforcing consistency constraints across different files is complex.
5. **Atomicity Problems:** Ensuring all or nothing transactions in case of system failures is difficult.
6. **Concurrent-Access Anomalies:** Simultaneous data updates may result in inconsistent states.
7. **Security Problems:** Enforcing access control is cumbersome due to ad hoc application additions.

These challenges drove the development of database systems in the 1960s and 1970s. Database systems address these issues through concepts and algorithms, providing efficient and secure data management solutions.

1.3 View of Data

1.3.1 Data Models

- A **database system** consists of interrelated data and programs for accessing and modifying them.
- **Data model:** Conceptual tools for describing data, relationships, semantics, and consistency constraints.
- Four categories of data models:
  - **Relational Model:** Uses tables to represent data and relationships, widely used in current systems.
  - **Entity-Relationship Model:** Utilizes entities and relationships among them.
  - **Semi-structured Data Model:** Allows data with varying attributes, like JSON and XML.
  - **Object-Based Data Model:** Integrates object-oriented concepts into databases, often used with Java, C++, or C#.

1.3.2 Relational Data Model

- Data represented in tables with multiple columns.
- Each row represents one piece of information.
- Illustration with sample university instructors and department details.

1.3.3 Data Abstraction

- **Efficient data retrieval** requires complex structures, abstracted for user simplicity.
- Levels of abstraction:
  - **Physical level:** Describes actual data storage.
  - **Logical level:** Describes stored data and relationships, offering physical data independence.
  - **View level:** Presents only relevant portions of the database to users, providing simplified interaction.

1.3.4 Instances and Schemas

- **Instances:** Collection of stored information at a specific moment.
- **Schemas:** Overall database design.
- Partitioned into physical, logical, and view schemas.
- **Logical schema** crucial for application development, offering physical data independence.
- Need for well-designed schemas to avoid issues like redundant information.

Note: Emphasizing key concepts and categorizing data models and abstraction levels.

1.4 Database Languages

A **database system** includes a **data-definition language (DDL)** for schema specification and a **data-manipulation language (DML)** for queries and updates. These are often parts of a unified language like **SQL**. Integrity constraints, such as **domain constraints** and **referential integrity**, ensure data consistency. **Authorization** controls user access levels. DDL statements generate output stored in a **data dictionary**, containing metadata.

The SQL Data-Definition Language

SQL's rich DDL defines tables and integrity constraints. For instance, a SQL statement defines a **department table** with specific columns and types. SQL supports primary keys and referential integrity, ensuring data consistency.

Data-Manipulation Language

DMLs enable data retrieval, insertion, deletion, and modification. Procedural DMLs specify data retrieval methods, while declarative DMLs focus on what data is needed without specifying retrieval methods. Queries, a subset of DML, request data retrieval. SQL, a nonprocedural language, is widely used for querying.

Database Access from Application Programs

SQL, not as powerful as general-purpose languages, requires embedded queries in host languages like C/C++, Java, or Python. Application programs interact with databases for various tasks like student registration, GPA calculation, etc. DML and DDL statements are sent to the database using APIs like **ODBC** or **JDBC**.

1.5 Database Design

Database systems manage large bodies of information within enterprises. The design process mainly focuses on **database schema**. Understanding user requirements is crucial, followed by choosing a **data model** and translating requirements into a **conceptual schema**. This schema defines data relationships and ensures satisfaction of functional requirements.

In relational model context, decisions involve selecting attributes and grouping them into tables. The process can employ either the **entity-relationship model** or **normalization algorithms**.

Moving from abstract to implementation involves logical design, mapping the conceptual schema to the system-specific data model, and then the physical design, specifying storage structures.

1.6 Database Engine

The database system comprises modules: storage manager, query processor, and transaction management.

- **Storage Manager:** Interfaces between low-level data and applications, includes components like authorization, transaction, file, and buffer managers.
- **Query Processor:** Includes DDL interpreter, DML compiler, and query evaluation engine. Translates DML statements into low-level instructions and executes them.
- **Transaction Management:** Ensures atomicity, consistency, isolation, and durability (ACID properties) of database transactions. Manages concurrent access to data and system failures.

Modern database engines emphasize **parallel processing** for handling large data efficiently and often utilize **solid-state disks (SSDs)** for faster storage.

For detailed reading:

- Database Design: Chapters 6 and 7
- Database Engine: Chapters 12, 13, 14, 15, 16
- Transaction Management: Chapters 17, 18, 19

1.7 Database and Application Architecture

The architecture of a centralized database system is depicted in Figure 1.3, illustrating user interaction and database engine components. This centralized structure suits shared-memory server setups, but for larger data volumes and higher speeds, parallel and distributed databases are utilized. Chapters 20 through 23 explore system structures, query processing, transaction processing, and maintenance in these contexts.

Modern database applications exhibit either two-tier or three-tier architecture, as shown in Figure 1.4. Traditional two-tier setups involve client-side applications interacting directly with server-side databases. Conversely, contemporary three-tier configurations separate the client, application server, and database, enhancing security and performance.

1.8 Database Users and Administrators

Database system users fall into four categories: Naive users, Application programmers, Sophisticated users, and Database Administrators (DBAs). Each interacts with the system differently, utilizing distinct user interfaces.

DBAs play a crucial role in database management, overseeing schema definition, storage structure, user authorization, and routine maintenance tasks.

1.9 History of Database Systems

- **1950s-1970s:** Evolution from punched cards to magnetic tapes, then hard disks. Relational model introduced by Edgar Codd in 1970.
- **Late 1970s-1980s:** Commercialization of relational databases, including IBM's System R and Oracle.
- **1990s:** Emergence of decision support applications, parallel databases, and object-relational features. Growth of the World Wide Web impacts database usage.
- **2000s:** Evolution of data storage formats (XML, JSON), growth of open-source databases, and development of graph databases.
- **2010s:** Rise of NoSQL systems, outsourcing of data management to cloud services, and increased focus on data security and privacy regulations.

2. Introduction to the Relational Model

- **Importance:** Relational model is primary for commercial data-processing due to its simplicity, evolving over 50 years with object-relational features, XML support, and tools for semi-structured data.
- **Chapter Focus:** Fundamentals, database theory, relational schema design, query processing efficiency, and formal relational languages.

2.1 Structure of Relational Databases

- **Table Collection:** Relational databases comprise tables, each uniquely named.
- **Example:** Tables like 'instructor' and 'course' contain rows with corresponding attributes.

- **Row Representation:** Each row represents a relationship among values, akin to tuples in mathematical relations.
- **Terminology:** Relation refers to a table, tuple to a row, and attribute to a column.
- **Instance:** Specific set of rows in a relation.
- **Attribute Domain:** Permitted values for each attribute, ensuring atomicity.

Atomicity and Null Values

- **Atomic Domains:** All attribute domains must be atomic, treating elements as indivisible.
- **Null Value:** Signifies unknown or nonexistent value, posing challenges in database operations.
- **Elimination:** Preferably avoid null values due to operational difficulties.

Practical Advantages and Limitations

- **Advantages:** Strict relational structure aids in data storage and processing efficiencies.
- **Limitations:** Less suitable for dynamic applications where data and structure change over time.

Note: Detailed discussions on relational theory, schema design, and database operations are provided in subsequent chapters.

2.2 Database Schema

When discussing a database, we distinguish between the **database schema** (logical design) and the **database instance** (snapshot of data at a specific time). A **relation** corresponds to a variable, while a **relation schema** is akin to a type definition in programming. It consists of attributes and domains.

- **Relation Instance:** Comparable to the value of a variable, it may change over time.
- **Relation Schema:** Attributes and domains, usually static.

For simplicity, the same name (e.g., "instructor") is often used for both schema and instance, though explicit differentiation is possible. Common attributes in schemas facilitate relating tuples across different relations.

Consider the "department" relation schema: department (dept name, building, budget). Shared attributes like "dept_name" facilitate data retrieval across related relations.

A university database might include various relations:

- **section:** Describing class offerings (course id, sec id, semester, year, building, room number, time slot id).
- **teaches:** Associating instructors with class sections they teach (ID, course id, sec id, semester, year).

Additional relations in a university database include:

- **student:** Records student information (ID, name, dept_name, total_credit).
- **advisor:** Associates students with their advisors (s id, i id).
- **takes:** Tracks student enrollment in courses (ID, course id, sec id, semester, year, grade).
- **classroom:** Details about classrooms (building, room number, capacity).
- **time slot:** Information about time slots for classes (time slot id, day, start_time, end_time).

Note: SQL queries for creating these tables would depend on the specific database management system being used.

2.3 Keys

- **Distinguishing Tuples:** Attributes must uniquely identify tuples within a relation.
- **Superkey:** Set of attributes allowing unique tuple identification.
- **Candidate Key:** Minimal superkey; no proper subset is a superkey.
- **Primary Key:** Chosen key for tuple identification; a candidate key designated by the database designer.
- **Primary Key Constraint:** Prohibits identical key attribute values in different tuples.
- **Underlining:** Indicates primary key attributes in a relation schema.
- **Foreign-Key Constraint:** Ensures values in one relation's attribute(s) correspond to the primary key of another relation.
- **Referential-Integrity Constraint:** Requires values in specified attributes of one relation to appear in specified attributes of another relation.
- **Foreign Key:** Attribute(s) in one relation referencing the primary key of another relation.
- **Referencing Relation:** Contains the foreign key(s).
- **Referenced Relation:** Contains the primary key referenced by the foreign key.
- **Generalization:** Referential-integrity constraints relax the requirement that referenced attributes must form the primary key.
- **Support:** Database systems typically support foreign-key constraints but not referential integrity constraints where the referenced attribute is not a primary key.

```sql
-- SQL to create a table with primary key
CREATE TABLE classroom (
    building VARCHAR(50),
    room_number INT,
    capacity INT,
    PRIMARY KEY (building, room_number)
);

-- SQL to create a table with foreign key constraint
CREATE TABLE instructor (
    ID INT PRIMARY KEY,
    name VARCHAR(50),
    dept_name VARCHAR(50),
    FOREIGN KEY (dept_name) REFERENCES department(dept_name)
);
```

2.4 Schema Diagrams

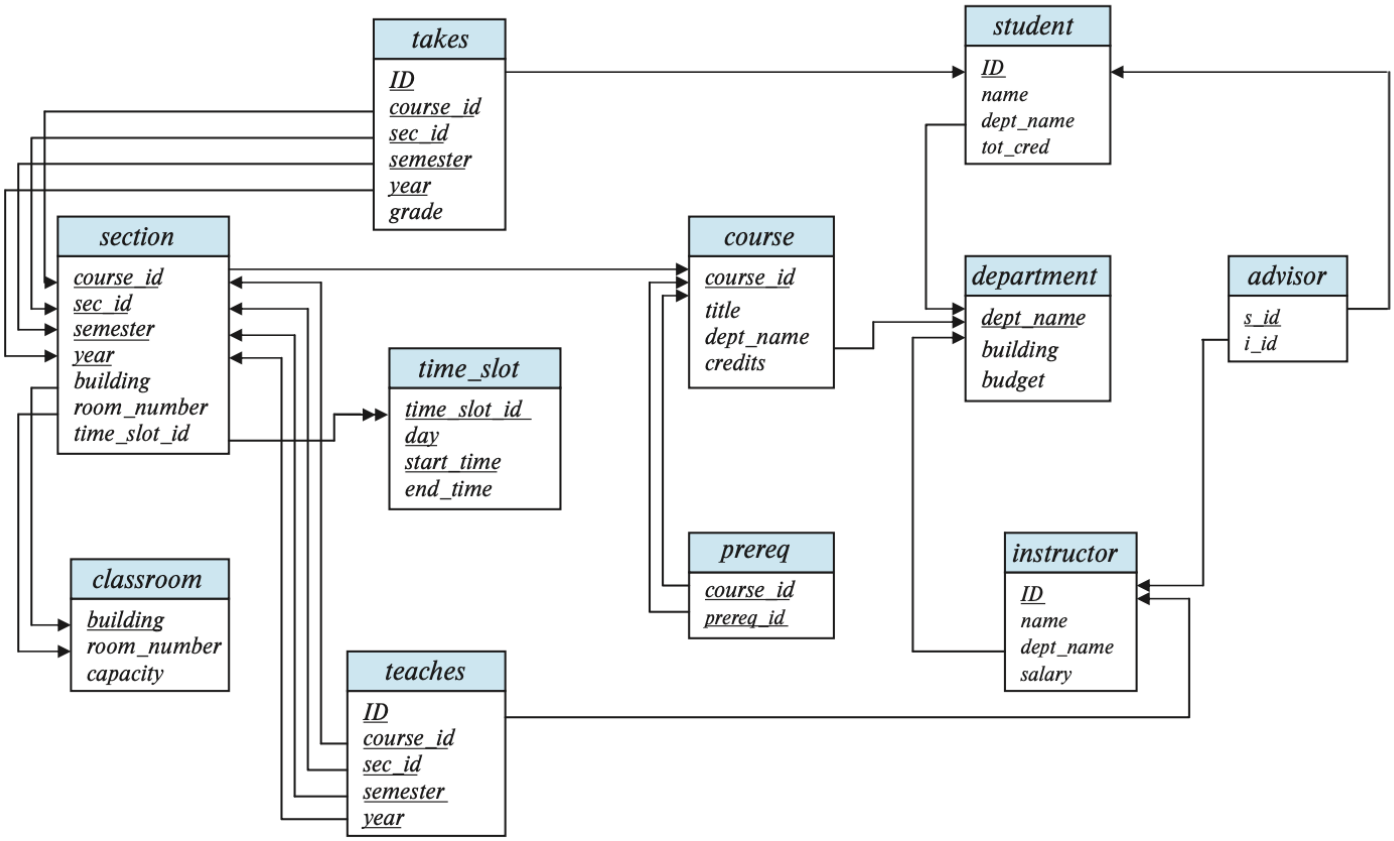


Figure 2.9 Schema diagram for the university database.

A **database schema** is represented with schema diagrams, incorporating **primary key** and **foreign key** constraints. Each relation is depicted as a box with the relation name highlighted in blue at the top, and attributes listed inside. Primary key attributes are underlined, while foreign key constraints are indicated with arrows from referencing attributes to the primary key of referenced relations. Two-headed arrows denote referential integrity constraints distinct from foreign key constraints.

Figure 2.9 illustrates this with a two-headed arrow from 'time slot id' in the 'section' relation to 'time slot id' in the 'time slot' relation, symbolizing the referential integrity constraint between them. While many database systems offer graphical tools for creating such diagrams, another representation called the **entity-relationship diagram** will be discussed later in Chapter 6.

2.5 Relational Query Languages

A **query language** facilitates user requests from the database and typically operates at a higher level than standard programming languages. These languages fall into categories: imperative, functional, or declarative.

- **Imperative:** Instructs the system with a sequence of operations, often involving state variables.
- **Functional:** Expresses computation as the evaluation of functions without side effects.
- **Declarative:** Describes desired information without specifying steps, often using mathematical logic.

"Pure" query languages include:

- **Relational algebra:** Functional in nature, forming the theoretical basis of SQL.
- **Tuple relational calculus** and **domain relational calculus:** Declarative approaches described in Chapter 27.

While these languages lack the "syntactic sugar" of commercial languages, they illustrate fundamental data extraction techniques. Practical query languages, like SQL, blend elements from imperative, functional, and

declarative paradigms. SQL, extensively used, will be covered from Chapter 3 to Chapter 5.

```
-- Sample SQL query for retrieving data
SELECT column1, column2
FROM table_name
WHERE condition;
```

2.6 The Relational Algebra

The **relational algebra** encompasses operations on one or two relations, yielding a new relation as the result. Unary operations (e.g., select, project, rename) work on one relation, while binary operations (e.g., union, Cartesian product, set difference) operate on pairs of relations.

Though relational algebra forms the basis for SQL, database systems typically don't allow direct relational algebra queries. However, implementations exist for practicing relational algebra queries.

Relations contain unique tuples per mathematical definition, but duplicates are often permitted in database tables unless constrained. In formal relational algebra, duplicates are eliminated as per set theory.

**2.6.1 The Select Operation

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 2.10 Result of $\sigma_{dept\_name = \text{“Physics”}}(instructor)$. **

Select operation chooses tuples satisfying a predicate denoted by the lowercase Greek letter sigma (σ). The predicate appears as a subscript, and the argument relation is in parentheses. For instance, to select Physics department instructors:

```
σ_{dept_name = “Physics”}(instructor)
```

Selection predicates can combine multiple conditions using logical connectives like and (∧), or (∨), and not (¬). Comparison operators (=, ≠, <, ≤, >, and ≥) are also valid.

2.6.2 The Project Operation

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Figure 2.11 Result of $\Pi_{ID,\ name,\ salary}(instructor)$.

Project operation (Π) returns a relation with specific attributes, eliminating duplicates. It's denoted by the uppercase Greek letter pi (Π), with desired attributes listed as a subscript. For example:

```
Π_{ID,name,salary}(instructor)
```

A generalized version allows expressions involving attributes in the list.

2.6.3 Composition of Relational Operations

Relational operations can be composed into expressions. Resulting expressions maintain the relational type, allowing for seamless composition.

```
Π_{name}(σ_{dept_name = “Physics”}(instructor))
```

2.6.4 The Cartesian-Product Operation

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 2.12 Result of the Cartesian product *instructor* × *teaches*.

Cartesian product (×) combines information from two relations. Unlike mathematical Cartesian products, tuples are concatenated. Naming schema conventions are crucial for distinguishing attributes. For example, the result schema for `instructor × teaches`:

```
(instructor.ID, instructor.name, instructor.dept_name, instructor.salary, teaches.ID,
course_id, sec_id, semester, year)
```

The resulting relation contains tuples generated from all possible pairs of tuples from the input relations, leading to a potentially large result set.

2.6.5 Join Operation

instructor.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Figure 2.13 Result of $\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$.

To obtain information about instructors along with the course IDs of all courses they have taught, we utilize the join operation. While the Cartesian product of instructor and teaches combines data from both relations, it associates every instructor with every course, regardless of whether they taught it. By selecting tuples where instructor.ID = teaches.ID, we filter out irrelevant pairs. This results in instructors and their respective taught courses, eliminating those who didn't teach any course. Duplication of instructor IDs is resolved through projection to remove the teaches.ID column.

```sql
SELECT *
FROM instructor
JOIN teaches ON instructor.ID = teaches.ID;
```

2.6.6 Set Operations

Set operations like union (∪), intersection (∩), and set-difference (−) are employed to manipulate relations.

For instance, to find courses taught in Fall 2017 or Spring 2018, we perform selections on section relation based on semester and year. Then, the union of these sets gives the desired result, ensuring compatibility in terms of arity and attribute types.

```sql
(SELECT course_id FROM section WHERE semester = 'Fall' AND year = 2017)
UNION
(SELECT course_id FROM section WHERE semester = 'Spring' AND year = 2018);
```

To find courses taught in both Fall 2017 and Spring 2018, we perform set intersection on respective selections. Alternatively, set-difference identifies courses taught in Fall 2017 but not in Spring 2018.

```sql
(SELECT course_id FROM section WHERE semester = 'Fall' AND year = 2017)
INTERSECT
(SELECT course_id FROM section WHERE semester = 'Spring' AND year = 2018);

(SELECT course_id FROM section WHERE semester = 'Fall' AND year = 2017)
EXCEPT
(SELECT course_id FROM section WHERE semester = 'Spring' AND year = 2018);
```

Relational Algebra Operations

2.6.7 The Assignment Operation

- **Functionality:** Assign parts of a relational-algebra expression to temporary relation variables.
- **Symbol:** Denoted by ←.
- **Example:**

```
courses_fall_2017 ← Π_{course_id} (σ_{semester = “Fall” ∧ year=2017}(section))
courses_spring_2018 ← Π_{course_id} (σ_{semester = “Spring” ∧ year=2018}(section))
courses_fall_2017 ∩ courses_spring_2018
```

- **Purpose:** Enables expressing complex queries as sequential programs.

2.6.8 The Rename Operation

- **Purpose:** Provides names to relational-algebra expressions.
- **Symbol:** Denoted by ρ (rho).
- **Usage:**
  - Assigns a name to a result: ρ (E) x.
  - Renames attributes: ρ_{x(A1, A2, ..., An)}(E).
- **Advantages:** Facilitates referencing the same relation multiple times in a query.
- **Example:**

```
Π ((σ (ρ (instructor) × σ (ρ (instructor)))))) i.ID,i.name i.salary > w.salary i
w.id=12121 w
```

- **Note:** Alternative positional notation exists but is less convenient for humans.

2.6.9 Equivalent Queries

- **Observation:** Multiple ways to express a query in relational algebra.
- **Example:**
  - Query 1: σ*{dept_name = “Physics”}(instructor* ⋈{instructor.ID = teaches.ID} teaches)
  - Query 2: (σ*{dept_name = “Physics”}(instructor))* ⋈{instructor.ID = teaches.ID} teaches
- **Equivalent Queries:** May have different sequences but yield the same result.
- **Optimization:** Database optimizers find efficient ways to compute results, not strictly following query sequence.

```sql
-- Example SQL query for equivalent queries
SELECT *
FROM instructor
JOIN teaches ON instructor.ID = teaches.ID
WHERE dept_name = 'Physics';
```

Note: Relational algebra offers flexibility in query expression, allowing for optimization.

3. Introduction to SQL

3.1 Overview

- **SQL:** Widely used database query language with functionalities beyond querying, including data structure definition, modification, and security constraints.
- **Purpose:** Present fundamental constructs and concepts of SQL; actual implementations may vary.

SQL Language Evolution

- **Origin:** Developed by IBM in the 1970s as Sequel, evolved into SQL (Structured Query Language).
- **Standardization:** ANSI and ISO published SQL standards: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, SQL:2016.
- **Parts:** Data Definition Language (DDL), Data Manipulation Language (DML), Integrity, View Definition, Transaction Control, Embedded SQL, Dynamic SQL, Authorization.

Chapter Contents

- **Basic Features:** Survey of basic DML and DDL features since SQL-92.
- **Chapter 4:** Detailed coverage of SQL query language, including joins, views, transactions, integrity constraints, type system, and authorization.
- **Chapter 5:** Advanced features, such as accessing SQL from programming languages, functions, procedures, triggers, recursive queries, aggregation, and data analysis.

Implementation Variations

- **Differences:** SQL implementations may support nonstandard features and omit advanced or recent features.
- **Consultation:** Refer to user manuals for specific database systems to verify supported features

3.2 SQL Data Definition

3.2.1 Basic Types

- **SQL DDL:** Defines relations in a database, including schema, attribute types, integrity constraints, indices, and storage structure.
- **Types:** char(n), varchar(n), int, smallint, numeric(p,d), real, double precision, float(n), nvarchar.
- **Null Value:** Represents an unknown or non-existent value; can be prohibited in certain cases.

3.2.2 Basic Schema Definition

- **Create Table Command:** Defines relation schema.
- **Attributes:** Name, type, optional constraints.
- **Primary Key:** Specifies unique non-null attributes.
- **Foreign Key:** References primary key attributes of another relation.
- **Not Null Constraint:** Prohibits null values for an attribute.

Integrity Constraints

- **Primary Key Constraint:** Ensures uniqueness and non-nullity.
- **Foreign Key Constraint:** Enforces referential integrity.
- **Not Null Constraint:** Excludes null values for an attribute.

Commands

- **Create Table:** Defines relation schema.
- **Drop Table:** Deletes relation and its schema.
- **Alter Table:** Adds or drops attributes from an existing relation.

```sql
-- Example: Create a department relation
CREATE TABLE department (
    dept_name VARCHAR(20),
    building VARCHAR(15),
    budget NUMERIC(12,2),
    PRIMARY KEY (dept_name)
);

-- Example: Add an attribute to an existing relation
ALTER TABLE department ADD location VARCHAR(50);

-- Example: Drop an attribute from an existing relation
ALTER TABLE department DROP building;
```

3.3 Basic Structure of SQL Queries

The basic structure of an SQL query comprises three clauses: **select**, **from**, and **where**. Queries operate on relations listed in the **from** clause, manipulate them as specified in the **where** and **select** clauses, and yield a relation as the output.

3.3.1 Queries on a Single Relation

- **Example 1:** Find the names of all instructors.

```sql
select name from instructor;
```

- **Example 2:** Find the department names of all instructors.

```sql
select dept_name from instructor;
```

- **Duplicate Handling:** SQL allows duplicates, but **distinct** keyword eliminates them.

```sql
select distinct dept_name from instructor;
```

Arithmetic Expressions

- The **select** clause can include arithmetic expressions.

```sql
select ID, name, dept_name, salary * 1.1 from instructor;
```

The Where Clause

- Filters rows based on specified predicates.
- Example: Find instructors in the Computer Science department with salary > $70,000.

```
select name from instructor where dept_name = 'Comp. Sci.' and salary > 70000;
```

3.3.2 Queries on Multiple Relations

- Combines information from multiple relations.
- Example: Retrieve instructor names, along with their department names and building names.

```
select name, instructor.dept_name, building from instructor, department where
instructor.dept_name=department.dept_name;
```

General SQL Query Form

```
select A1, A2,...,An from r1, r2,...,rm where P;
```

- **A1, A2,...,An:** Attributes desired in the output.
- **r1, r2,...,rm:** Relations accessed.
- **P:** Predicate involving attributes in the **from** clause.

Cartesian Product

- Formed by relations listed in the **from** clause.
- Each tuple in one relation combines with every tuple in another.

Understanding SQL Queries

1. Generate Cartesian product of relations.
2. Apply predicates specified in the **where** clause.
3. Output attributes specified in the **select** clause.

Note: Real SQL implementations optimize evaluation differently.

Remember: Be cautious with predicates to avoid generating large Cartesian products.

3.4 SQL Additional Basic Operations

3.4.1 Rename Operation

- SQL supports renaming attributes in result relations using the `AS` clause.
- It allows changing attribute names and avoiding duplication.
- Syntax: `old-name AS new-name`.

Examples:

1. Renaming attributes in the result:

```
SELECT name AS instructor_name, course_id
FROM instructor, teaches
WHERE instructor.ID = teaches.ID;
```

2. Renaming relations:

```
SELECT T.name, S.course_id
FROM instructor AS T, teaches AS S
WHERE T.ID = S.ID;
```

3.4.2 String Operations

- SQL uses single quotes for string values.
- String functions include concatenation, substring extraction, case conversion, trimming, etc.
- Pattern matching is done using the `LIKE` operator with `%` and `_` wildcards.

Example:

- Finding departments with building names containing 'Watson':

```
SELECT dept_name
FROM department
WHERE building LIKE '%Watson%';
```

3.4.3 Attribute Specification in Select Clause

- `*` symbol selects all attributes.
- `table_name.*` selects all attributes from a specific table.

Example:

- Selecting all attributes from the `instructor` table:

```
SELECT instructor.*
FROM instructor, teaches
WHERE instructor.ID = teaches.ID;
```

3.4.4. Ordering the Display of Tuples

- `ORDER BY` clause sorts tuples in the result.
- Default order is ascending, but can be specified as descending.
- Supports sorting by multiple attributes.

Example:

- Listing instructors in Physics department alphabetically:

```
SELECT name
FROM instructor
WHERE dept = 'Physics'
ORDER BY name;
```

3.4.5 Where-Clause Predicates

- `BETWEEN` operator simplifies range queries.
- Allows specifying a range of values.

Example:

- Finding instructors with salary between $90,000 and $100,000:

```
SELECT name
FROM instructor
WHERE salary BETWEEN 90000 AND 100000;
```

- Using row constructors for tuple comparison:

```
SELECT name, course_id
FROM instructor, teaches
WHERE (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

3.5 Set Operations in SQL

3.5.1 Union Operation

- Query to find courses taught either in Fall 2017 or Spring 2018:

```
(select course_id from section where semester = 'Fall' and year = 2017)
union
(select course_id from section where semester = 'Spring' and year = 2018);
```

- Automatically eliminates duplicates from the result.

3.5.2 Intersect Operation

- Query to find courses taught in both Fall 2017 and Spring 2018:

```
(select course_id from section where semester = 'Fall' and year = 2017)
intersect
(select course_id from section where semester = 'Spring' and year = 2018);
```

- Automatically eliminates duplicates from the result.

3.5.3 Except Operation

- Query to find courses taught in Fall 2017 but not in Spring 2018:

```
(select course_id from section where semester = 'Fall' and year = 2017)
except
(select course_id from section where semester = 'Spring' and year = 2018);
```

- Outputs tuples from the first input that do not occur in the second input.

Note: Parentheses around each select-from-where statement are optional but useful for readability.

Result Interpretation

- Union: Automatically eliminates duplicates.
- Intersect: Automatically eliminates duplicates.
- Except: Outputs tuples not occurring in the second input.

The number of duplicate tuples in the result depends on the number of duplicates in the input relations.

3.6 Null Values

Null values pose challenges in relational operations, including arithmetic, comparison, and set operations.

- **Arithmetic Operations:** Result is null if any input value is null. For instance, "r.A + 5" yields null if "r.A" is null.
- **Comparison Operations:** Comparisons with nulls are problematic. SQL considers comparison with null as unknown, introducing a third logical value alongside true and false.
- **Boolean Operations:** SQL extends boolean operations to handle the unknown value:
  - "and": true and unknown is unknown, false and unknown is false, unknown and unknown is unknown.
  - "or": true or unknown is true, false or unknown is unknown, unknown or unknown is unknown.
  - "not": not unknown is unknown.
- **Where Clause Predicate:** If the predicate evaluates to false or unknown, the tuple is excluded from the result.
- **SQL Syntax:**
  - To find null values in "salary" column: `SELECT name FROM instructor WHERE salary IS NULL;`
  - To test if a comparison result is unknown: `SELECT name FROM instructor WHERE salary > 10000 IS UNKNOWN;`
- **Distinct Clause:** When comparing tuples for distinctness, SQL treats null values as identical. Thus, {('A',null), ('A',null)} are considered identical.
- **Set Operations:** Union, intersection, and except treat tuples as identical if they have the same values for all attributes, even if some values are null.

Note: Null values introduce complexity in SQL operations and require careful handling to ensure accurate results.

3.7 Aggregate Functions in SQL

Aggregate functions in SQL take a collection of values and return a single value. SQL offers five standard built-in aggregate functions:

- **Average:** `avg`
- **Minimum:** `min`
- **Maximum:** `max`
- **Total:** `sum`
- **Count:** `count`

The input to `sum` and `avg` must be a collection of numbers, while the other operators can work with non-numeric data types as well.

Basic Aggregation

For example, to find the average salary of instructors in the Computer Science department:

```
SELECT AVG(salary) AS avg_salary
FROM instructor
WHERE dept_name = 'Comp. Sci.';
```

Retaining duplicates is crucial for accurate aggregation. However, in cases where duplicates must be eliminated, the `distinct` keyword is used.

Aggregation with Grouping

The `group by` clause is utilized to apply aggregate functions to groups of tuples. For instance, to find the average salary in each department:

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

The `having` clause is used to specify conditions on groups rather than individual tuples. For example, to find departments where the average salary exceeds $42,000:

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

Aggregation with Null and Boolean Values

Null values are handled according to specific rules. Aggregate functions treat nulls by ignoring them in their input collection. The `count (*)` function returns 0 for an empty collection, and other aggregate operations return null. Additionally, Boolean data types introduced in SQL:1999 allow for aggregation with `some` and `every` functions.

3.8 Nested Subqueries

SQL provides a mechanism for **nesting subqueries**, which are select-from-where expressions within another query. Subqueries are commonly used to perform tests for **set membership**, make set comparisons, and determine set cardinality. We explore various uses of nested subqueries in the where clause (Sections 3.8.1 to 3.8.4) and in the from clause (Section 3.8.5). Additionally, a class of subqueries called **scalar subqueries** is discussed in Section 3.8.7.

3.8.1 Set Membership

- **in and not in operators**: Test for set membership or absence.
- Example: Finding courses taught in both Fall 2017 and Spring 2018 semesters using nested subqueries.

```
select distinct course_id
from section
where semester = 'Fall' and year = 2017 and
course_id in (select course_id from section
where semester = 'Spring' and year = 2018);
```

- **not in** construct: Used similarly to **in**, example: Finding courses taught in Fall 2017 but not in Spring 2018.

```
select distinct course_id
from section
where semester = 'Fall' and year = 2017 and course_id not in (select course_id
from section
where semester = 'Spring' and year = 2018);
```

- **Enumerated sets**: The operators can be used on enumerated sets as well.

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

3.8.2 Set Comparison

- **> some and > all comparisons**: Allow comparisons of sets.
- Example: Finding names of instructors whose salary is greater than at least one in the Biology department.

```
select name
from instructor
where salary > some (select salary
from instructor
where dept_name = 'Biology');
```

- **> all comparison**: Finds names of instructors with salary greater than all in the Biology department.

```
select name
from instructor
where salary > all (select salary
from instructor
where dept_name = 'Biology');
```

- **Other comparisons**: SQL also supports < some, <= some, >= some, = some, <> some, and similarly for **all**.

3.8.3 Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The `exists` construct returns true if the argument subquery is nonempty. We can use it to find courses taught in both Fall 2017 and Spring 2018 semesters:

```sql
SELECT course_id
FROM section AS S
WHERE semester = 'Fall' AND year = 2017 AND
      EXISTS (SELECT *
              FROM section AS T
              WHERE semester = 'Spring' AND year = 2018 AND
                    S.course_id = T.course_id);
```

A subquery that uses a correlation name from an outer query is called a correlated subquery. Scoping rules apply for correlation names, allowing usage only within the subquery itself or its containing query.

We can test for the nonexistence of tuples in a subquery using the `not exists` construct. It can simulate set containment operations. For instance, to find students who have taken all Biology department courses:

```sql
SELECT S.ID, S.name
FROM student AS S
WHERE NOT EXISTS ((SELECT course_id
                   FROM course
                   WHERE dept_name = 'Biology')
                  EXCEPT
                  (SELECT T.course_id
                   FROM takes AS T
                   WHERE S.ID = T.ID));
```

3.8.4 Test for the Absence of Duplicate Tuples

SQL includes the `unique` construct for testing duplicate tuples in a subquery. It returns true if the subquery contains no duplicates. To find courses offered at most once in 2017:

```sql
SELECT T.course_id
FROM course AS T
WHERE UNIQUE (SELECT R.course_id
              FROM section AS R
              WHERE T.course_id = R.course_id AND R.year = 2017);
```

`not unique` can be used to test for the existence of duplicate tuples. For instance, to find courses offered at least twice in 2017:

```sql
SELECT T.course_id
FROM course AS T
WHERE NOT UNIQUE (SELECT R.course_id
                  FROM section AS R
                  WHERE T.course_id = R.course_id AND R.year = 2017);
```

3.8.5 Subqueries in the From Clause

SQL allows subquery expressions in the `from` clause. A select-from-where expression returns a relation, insertable into another select-from-where. For example, to find average instructors' salaries of departments with average salary > $42,000:

```sql
SELECT dept_name
FROM (SELECT dept_name, AVG(salary) AS avg_salary
      FROM instructor
      GROUP BY dept_name) AS dept_avg
WHERE avg_salary > 42000;
```

Nested subqueries in the `from` clause are supported by most SQL implementations. The `lateral` keyword allows accessing attributes of preceding tables or subqueries in the same `from` clause. For example:

```sql
SELECT name, salary, avg_salary
FROM instructor AS I1, LATERAL (SELECT AVG(salary) AS avg_salary
                                FROM instructor AS I2
                                WHERE I2.dept_name = I1.dept_name);
```

3.8.6 The With Clause

The **WITH** clause in SQL allows the definition of temporary relations, accessible only within the query it's used in. It enhances query readability and logic, particularly useful for complex queries. Introduced in SQL:1999, it's supported by many, but not all, database systems.

Example:

```sql
WITH max_budget (value) AS (
    SELECT MAX(budget) FROM department
)
SELECT budget
FROM department, max_budget
WHERE department.budget = max_budget.value;
```

3.8.7 Scalar Subqueries

Scalar subqueries in SQL return a single value and can be used wherever an expression returning a value is permitted. They're commonly used in select, where, and having clauses. These subqueries may involve aggregates or not, and if they return more than one tuple at runtime, it results in an error.

Example:

```sql
SELECT dept_name, (SELECT COUNT(*) FROM instructor WHERE department.dept_name =
instructor.dept_name) AS num_instructors
FROM department;
```

3.8.8 Scalar Without a From Clause

Certain queries necessitate calculation without reference to any relation or have subqueries without a from clause. In such cases, a special dummy relation like "dual" can be used to facilitate these queries.

Example:

```sql
SELECT (SELECT COUNT(*) FROM teaches) / (SELECT COUNT(*) FROM instructor) FROM dual;
```

Oracle provides a predefined relation called dual, containing a single tuple, for such purposes. To ensure precision, subquery results can be converted to floating point numbers before division.

3.9 Modification of the Database

3.9.1 Deletion

- **Delete Statement:** Removes tuples from a relation.
- Syntax: `delete from r where P;`
  - `r`: Relation name
  - `P`: Predicate
- Examples:
  - Delete instructors from the Finance department: `delete from instructor where dept_name = 'Finance';`
  - Delete instructors with salary between $13,000 and $15,000: `delete from instructor where salary between 13000 and 15000;`
  - Delete tuples associated with departments in the Watson building:

```
delete from instructor
where dept_name in (select dept_name from department where building =
'Watson');
```

3.9.2 Insertion

- **Insert Statement:** Adds tuples to a relation.
- Syntax:
  - Single tuple: `insert into r values (val1, val2, ...);`
  - With attribute names: `insert into r(attr1, attr2, ...) values (val1, val2, ...);`
- Examples:
  - Insert a course: `insert into course values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);`
  - Insert based on query result:

```
insert into instructor
select ID, name, dept_name, 18000
from student where dept_name='Music' and tot_cred>144;
```

3.9.3 Updates

- **Update Statement:** Modifies tuples in a relation.
- Syntax: `update r set attr = val where P;`
- Examples:
  - Increase all instructor salaries by 5%: `update instructor set salary= salary * 1.05;`
  - Increase salaries for instructors below $70,000: `update instructor set salary = salary * 1.05 where salary < 70000;`
  - Conditional updates:

```
update instructor
set salary = case
             when salary <= 100000 then salary * 1.05
             else salary * 1.03
             end;
```

SQL provides flexibility in data manipulation, allowing precise modifications to database content.

4. Intermediate SQL

4.1 Join Expressions Overview

- **Cartesian Product**: Previously used in combining multiple relations, except when using set operations.
- **Join Operations**: Introduced to express queries more naturally and handle complex queries that Cartesian products alone cannot efficiently manage.
- **Example Context**: Utilizes the 'student' and 'takes' relations, where attributes like 'grade' may have null values indicating unassigned grades.

4.1.1 The Natural Join

- **Basic SQL Query**:

```
select name, course_id
from student, takes
where student.ID = takes.ID;
```

- **Natural Join Usage**:
  - Automates the equating of attributes with the same name across relations.
  - More efficient and cleaner than using Cartesian products with additional conditions.
- **SQL Example with Natural Join**:

```
select name, course_id
from student natural join takes;
```

- **Attributes Management**:
  - Common attributes appear once.
  - Order: common attributes, unique to first relation, unique to second relation.

4.1.2 Join Variations and Conditions

- **Extended Usage in SQL**:
  - Combines multiple relations using natural join within a single 'from' clause.
  - Can handle complex queries involving additional relations and conditions.
- **Complex SQL Query Example**:

```
select name, title
from student natural join takes, course
```

```
where takes.course_id = course.course_id;
```

- **Using Clause**:
  - Allows specification of exactly which columns should match in the join, enhancing control over join conditions.
- **SQL Example with Using Clause**:

```
select name, title
from (student natural join takes) join course using (course_id);
```

The 'On' Condition

- **General Predicate for Joins**:
  - Similar to 'where' clause but used specifically within join contexts.
  - Allows specifying complex join conditions directly in the join expression.
- **SQL Example with On Condition**:

```
select *
from student join takes on student.ID = takes.ID;
```

- **Comparative Analysis**:
  - 'On' condition can specify any SQL predicate, allowing for more expressive join conditions.
  - Often interchangeable with a corresponding 'where' clause, but can enhance readability and specificity in queries.
- **Significance in SQL**:
  - Particularly useful in outer joins where behavior differs from where conditions.
  - Improves readability by segregating join conditions from other query predicates.

4.1.3 Outer Joins

Concept of Outer Joins

- **Problem with Natural Joins**: Fails to include students who have not taken any courses because no matching tuples exist in the 'takes' relation.
- **Outer Joins**: Ensures inclusion of all relevant tuples by adding tuples with null values for non-matching attributes.

Types of Outer Joins

- **Left Outer Join**: Preserves all tuples from the left relation. If no matching tuples exist in the right relation, attributes from the right relation are set to null.
- **Right Outer Join**: Preserves all tuples from the right relation. Functions like a left outer join, but in reverse.
- **Full Outer Join**: Combines the effects of both left and right outer joins. Preserves tuples from both relations, filling non-matching attributes with nulls.

SQL Queries for Outer Joins

- **Left Outer Join**:

```
SELECT *
FROM student NATURAL LEFT OUTER JOIN takes;
```

  Includes students who haven't taken any courses by filling missing 'takes' attributes with nulls.
- **Query to Find Students Without Courses**:

```
SELECT ID
FROM student NATURAL LEFT OUTER JOIN takes
WHERE course_id IS NULL;
```

- **Right Outer Join** (Example):

```
SELECT *
FROM takes NATURAL RIGHT OUTER JOIN student;
```

  Similar to left outer join but swaps relation order.
- **Full Outer Join Example**:

```
SELECT *
FROM (SELECT *
      FROM student
      WHERE dept_name='Comp. Sci.')
NATURAL FULL OUTER JOIN (SELECT *
                         FROM takes
                         WHERE semester='Spring' AND year=2017);
```

  Displays all students in the Comp. Sci. department and all courses from Spring 2017, regardless of enrollment.

Handling Join Conditions

- **ON Clause**: Specifies conditions for joins, affecting how tuples are matched.
- **Example with ON Clause**:

```
SELECT *
FROM student LEFT OUTER JOIN takes ON student.ID = takes.ID;
```

  This query distinguishes non-matched tuples using the ON condition rather than WHERE, ensuring inclusion of unmatched tuples in the results.

Distinction Between Join Types

- **Inner Joins** (default join type):

```
SELECT *
FROM student JOIN takes USING (ID);
```

```
SELECT *
FROM student INNER JOIN takes USING (ID);
```

- **Equivalent to Natural Inner Join**:

```
SELECT *
FROM student NATURAL JOIN takes;
```

- **Join Types and Conditions**: The SQL syntax allows for specifying different types of joins (inner, left outer, right outer, full outer) combined with different join conditions (natural, using, or on).

4.2 Views

- **Purpose of Views**: Not all users need to see all data. Views help restrict access to specific parts of the database for security and data organization tailored to user needs.

4.2.1 View Definition

- **SQL Command for Creating Views**:

```
create view v as <query expression>;
```

  - **Example for Restricting Data** (hiding salary data from clerks):

```
create view faculty as select ID, name, dept from instructor;
```

  - **Example for Course Information** (listing Physics courses in Fall 2017):

```
create view physics_fall_2017 as
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall'
and section.year = 2017;
```

4.2.2 Using Views in SQL Queries

- **Usage Example**:
  - Finding Physics courses in a specific building:

```
select course_id
from physics_fall_2017
where building = 'Watson';
```

  - Views behave like regular tables in queries but are computed on demand.

4.2.3 Materialized Views

- **Definition**: Stored views that are updated when underlying data changes, enhancing query performance but requiring maintenance.
- **Maintenance**:
  - Immediate update or lazy update upon access.
  - Systems may allow periodic updates; contents may be stale if not updated frequently.

4.2.4 Update of a View

- **Challenges in Modifying Views**:
  - Not all views are updatable due to the complexity of reflecting changes in the underlying tables.
  - Example problematic update:

```
insert into faculty values ('30765', 'Green', 'Music');
```

    - Requires a default or null salary which may not be acceptable.
- **Conditions for Updatable Views**:
  - Single table in the `FROM` clause.
  - Only attribute names in the `SELECT` clause, no aggregates or expressions.
  - Attributes not listed can be null.
- **Example of Restricted Update**:
  - View where only history department instructors are shown:

```
create view history_instructors as select *
from instructor
where dept_name = 'History';
```

  - Problems with inserting tuples that do not match the view's filter condition.
- **SQL Standards for View Updates**:
  - SQL:1999 specifies more complex rules for view updates.
  - Use of `WITH CHECK OPTION` to ensure inserted or updated rows satisfy the view's filter.

4.3 Transactions

- A transaction consists of a sequence of query and/or update statements and is a unit of work
- The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed
- The transaction must end with one of the following statements
  - Commit: the updates performed by the transaction become permanent in the database
  - Rollback: all the updates performed by the SQL statements in the transaction are undone
- Thus, the atomicity of a transaction is guaranteed
  - Either fully executed or rolled back as if it never occurred

4.4 Integrity Constraints

- **Purpose**: Ensure database consistency by guarding against accidental data corruption, contrasting with security constraints which prevent unauthorized access.
- **Examples**:
  - Instructor names cannot be null.

    - Unique instructor IDs.
    - Department names in courses must match those in the department relation.
    - Department budgets must exceed $0.00.
- **Specification**: Typically specified during schema design and included in SQL `CREATE TABLE` commands or added later via `ALTER TABLE`.

4.4.1 Constraints on a Single Relation

- **Primary SQL Commands**: `CREATE TABLE` which may include constraints like `NOT NULL`, `UNIQUE`, and `CHECK(<predicate>)`.

4.4.2 Not Null Constraint

- **Purpose**: Prevent null values in critical fields, ensuring data completeness.
- **SQL Example**:

```
CREATE TABLE student (
    name VARCHAR(20) NOT NULL,
    budget NUMERIC(12, 2) NOT NULL
);
```

- **Use Case**: Null values are prohibited in primary keys to maintain entity integrity.

4.4.3 Unique Constraint

- **Definition**: Ensures no two tuples have the same values in specified attributes.
- **SQL Syntax**:

```
UNIQUE (attribute1, attribute2, ...)
```

- **Details**: Attributes under `UNIQUE` constraint can still be null unless explicitly declared as `NOT NULL`.

4.4.4 The Check Clause

- **Purpose**: Enforces that attributes satisfy specified conditions, acting as a dynamic assertion within the database.
- **SQL Example**:

```
CREATE TABLE department (
    budget NUMERIC(12, 2),
    CHECK (budget > 0)
);
CREATE TABLE section (
    semester VARCHAR(6),
    CHECK (semester IN ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

- **Special Cases**: Null values do not violate `CHECK` constraints unless explicitly stated by a `NOT NULL` constraint.
- **Limitations**: Current SQL standards do not support subqueries within `CHECK` predicates.

4.4.5 Referential Integrity

- **Referential Integrity Constraints**: Ensure a value in one relation (referencing relation) also appears in another (referenced relation), typically implemented using foreign keys.
  - **Foreign Key Example**: In a university database, the `course` table references the `department` table to ensure all listed departments exist.
- **SQL Foreign Key Declaration**: Specified within the `CREATE TABLE` statement using the `FOREIGN KEY` clause.

```
CREATE TABLE course (
  dept_name VARCHAR(20),
  FOREIGN KEY (dept_name) REFERENCES department(dept_name)
);
```

- **Referential Actions**: SQL allows specifying actions like `ON DELETE CASCADE` and `ON UPDATE CASCADE` to handle changes that might violate integrity constraints.

```
CREATE TABLE course (
  ...
  dept_name VARCHAR(20),
  FOREIGN KEY (dept_name) REFERENCES department ON DELETE CASCADE ON UPDATE
CASCADE,
  ...
);
```

- **Handling Null Values**: SQL allows foreign keys to contain null values unless specified as `NOT NULL`. Null foreign keys automatically satisfy integrity constraints.

4.4.6 Assigning Names to Constraints

- **Naming Constraints**: Enhances manageability by allowing specific constraints to be named and later dropped if necessary.

```
ALTER TABLE instructor DROP CONSTRAINT minsalary;
```

4.4.7 Integrity Constraint Violation During a Transaction

- **Deferred Constraint Checking**: SQL supports deferring integrity checks to the end of a transaction, allowing intermediate states that might otherwise violate constraints.
  - **Example Scenario**: Inserting related data across tables (e.g., spouses in a `person` table) that temporarily violates foreign key constraints until all inserts are complete.

4.4.8Complex Check Conditions and Assertions

- **Advanced Integrity Constraints**: SQL standards allow specifying complex conditions using subqueries and assertions, though not widely supported.
  - **Assertion Example**: Ensuring total credits in the `student` table match sum of completed course credits.

```
CREATE ASSERTION credits_earned CHECK (
  NOT EXISTS (
    SELECT ID FROM student
    WHERE tot_cred <> (
      SELECT COALESCE(SUM(credits), 0)
      FROM takes NATURAL JOIN course
      WHERE student.ID = takes.ID AND grade IS NOT NULL AND grade <> 'F'
    )
  )
);
```

- **Use of Assertions**: Provides a powerful but costly method to enforce data integrity. Care is advised due to potential performance impact. Assertions are not typically supported by common database systems, but similar functionality can be achieved using triggers.

4.5 SQL Data Types and Schemas

This chapter extends the discussion from Chapter 3 on SQL data types and introduces user-defined types alongside built-in types.

4.5.1 Date and Time Types in SQL

- **Built-in Types**:
  - **date**: Stores a calendar date (year, month, day).
  - **time**: Stores time of day; `time(p)` variant includes fractional seconds and optional time-zone.
  - **timestamp**: Combines date and time; `timestamp(p)` variant allows fractional seconds and optional time-zone.
- **SQL Functions**:
  - **date 'YYYY-MM-DD'**: Sets a date.
  - **time 'HH:MM:SS'**: Sets a time.
  - **timestamp 'YYYY-MM-DD HH:MM:SS.FF'**: Sets a timestamp.
  - **extract(field from d)**: Extracts parts of date/time (e.g., year, month, day).
  - **current_date**, **current_time**, **localtime**, **current_timestamp**, **localtimestamp**: Functions to get current date and/or time.

4.5.2 Type Conversion and Formatting Functions

- **Type Conversion**:
  - **cast(e as t)**: Converts expression `e` to type `t`.
  - SQL example: `SELECT cast(ID as numeric(5)) as inst_id FROM instructor ORDER BY inst_id;` to convert and sort ID correctly.
- **Formatting Functions**:
  - **format**: Available in MySQL to format numbers.
  - **to_char**, **to_number**, **to_date**: Available in Oracle and PostgreSQL for data formatting.
  - **convert**: Used in SQL Server for data conversion.
- **Handling Null Values**:
  - **coalesce**: Returns the first non-null argument.
  - Example: `SELECT ID, coalesce(salary, 0) as salary FROM instructor;` shows salaries, substituting null with 0.

- **decode**: Oracle-specific, allows replacing values based on a condition, e.g., replacing null with 'N/A'.
- SQL example: `SELECT ID, decode(salary, null, 'N/A', salary) as salary FROM instructor;`

4.5.3 Default Values

- **Default Attributes in SQL Tables**:
  - SQL example: `CREATE TABLE student (ID varchar(5), name varchar(20) NOT NULL, dept_name varchar(20), tot_cred numeric(3,0) DEFAULT 0, PRIMARY KEY (ID));`
  - Illustrates setting a default value (0) for the `tot_cred` attribute. An `INSERT` statement can omit `tot_cred`, and it will default to 0.

4.5.4 Large-Object Types

- **Data Types for Large Objects**:
  - **clob**: Character Large Object, for storing large text data.
  - **blob**: Binary Large Object, for storing large binary data (e.g., images, videos).
  - SQL example: Declaring large object types, such as `book review clob(10KB)`, `image blob(10MB)`, `movie blob(2GB)`.
- **Handling Large Objects in SQL**:
  - Large objects are typically manipulated using locators, which are references to the data, allowing for efficient handling of very large data by fetching parts as needed, similar to file handling in operating systems.

4.5.5 User-Defined Types

- **SQL User-Defined Types**: SQL supports user-defined types, which include distinct types and structured data types.
  - **Distinct Types**: Used for defining simple types with unique domains. For example, monetary values in different currencies can be defined as distinct types to avoid programming errors during assignments or comparisons.

```
CREATE TYPE Dollars AS NUMERIC(12,2) FINAL;
CREATE TYPE Pounds AS NUMERIC(12,2) FINAL;
```

  - **Example Usage**:

```
CREATE TABLE department (
  dept_name VARCHAR(20),
  building VARCHAR(15),
  budget Dollars
);
```

  - **Type Conversion**: Requires explicit casting to convert between types and perform operations.

```
CAST(department.budget AS NUMERIC(12,2))
```

    - **Structured Data Types**: Not covered in this section; discussed in Section 8.2, includes complex structures like nested records and arrays.
- **Domains vs. Types**:
    - **Domains**: Can have constraints (e.g., NOT NULL) and default values. Less strictly typed than user-defined types.

```
CREATE DOMAIN DDollars AS NUMERIC(12,2) NOT NULL;
```

    - **Constraints**: Domains allow for specific constraints that must be met for the values.

```
CREATE DOMAIN YearlySalary AS NUMERIC(8,2) CONSTRAINT salary_value_test CHECK
(value >= 29000.00);
```

4.5.6 Generating Unique Key Values

- **Automatic Key Generation**: SQL supports auto-generating unique values for primary keys.
    - **Oracle/DB2 Syntax**:

```
ID NUMBER(5) GENERATED ALWAYS AS IDENTITY;
```

    - **Inserting Data without Specifying ID**:

```
INSERT INTO instructor (name, dept_name, salary) VALUES ('Newprof', 'Comp.
Sci.', 100000);
```

4.5.7 Create Table Extensions

- **Create Table Like**:

```
CREATE TABLE temp_instructor LIKE instructor;
```

- **Create Table As**:

```
CREATE TABLE t1 AS (SELECT * FROM instructor WHERE dept_name = 'Music') WITH DATA;
```

4.5.8 Schemas, Catalogs, and Environments

- **Naming and Organization**:
    - **Schemas and Catalogs**: Provide a hierarchical structure for organizing database objects, preventing name clashes.
    - **Connection and Environment Setup**: Involves specifying a default catalog and schema that uniquely identifies database relations.
    - **SQL Environment**: Includes user identifiers and operates within the context of a schema.
- **Schema and Catalog Management**:
    - **Creating/Dropping Schemas**:

```
CREATE SCHEMA univ_schema;
DROP SCHEMA univ_schema;
```

- **Catalogs**: Creation and management of catalogs vary across different SQL implementations and are not standardized.

4.6 Index Definition in SQL

- **Purpose of Indexes**: Enhance the efficiency of queries that access only a small fraction of records, avoiding the need to scan all records.
- **Index Usage**: Allows quick retrieval of tuples with specific attribute values, such as department names or IDs in a relation.
- **Physical vs. Logical Schema**: Indices are part of the physical schema and are used to improve transaction processing and integrity constraint enforcement, though they are not essential for correctness.
- **Control Over Indices**:
  - SQL provides control over index creation and removal through data-definition-language commands.
  - Indices can be on a single attribute or a list of attributes.
- **SQL Commands for Indices**:
  - Create Index: `CREATE INDEX <index-name> ON <relation-name> (<attribute-list>);`
  - Unique Index: `CREATE UNIQUE INDEX <index-name> ON <relation-name> (<attribute-list>);` marks the search key as a candidate key.
  - Drop Index: `DROP INDEX <index-name>;`
  - Indices can be B-tree, hash, or clustered, details of which are studied further in databases.
- **Creating an Index**:

  ```
  CREATE INDEX <index-name> ON <relation-name> (<attribute-list>);
  ```

  Example:

  ```
  CREATE INDEX deptindex ON instructor (dept_name);
  ```

- **Unique Index Creation**:

  ```
  CREATE UNIQUE INDEX <index-name> ON <relation-name> (<attribute-list>);
  ```

  Example:

  ```
  CREATE UNIQUE INDEX deptindex ON instructor (dept_name);
  ```

- **Dropping an Index**:

  ```
  DROP INDEX <index-name>;
  ```

4.7 Authorization

- **Types of Privileges**:
  - **Read**: Authorization to read data.
  - **Insert**: Authorization to insert new data.
  - **Update**: Authorization to update data.
  - **Delete**: Authorization to delete data.
  - Privileges can be granted on specific parts of the database like relations or views.
- **Authorization Process**:
  - SQL checks if a user's query or update is authorized based on granted privileges.
  - Unauthorized queries or updates are rejected.
- **Additional Authorizations**:
  - Users may have schema-level authorizations (e.g., create, modify, or drop relations).
  - Privileges can be passed on (granted) or withdrawn (revoked) by users who have them.
- **Database Administrator**:
  - Holds the highest level of authority, similar to a superuser in an OS, capable of authorizing users and restructuring the database.

4.7.1 Granting and Revoking of Privileges

- **SQL Privileges**:
  - Includes `select`, `insert`, `update`, `delete`, and `all privileges`.
  - `grant` statement assigns privileges; `revoke` removes them.

```
grant <privilege list> on <relation name or view name> to <user/role list>;
revoke <privilege list> on <relation name or view name> from <user/role list>;
```

- **Example of Granting Privileges**:
  - `grant select on department to Amit, Satoshi;`
  - Grants select privilege on the department relation to users Amit and Satoshi.
- **Revoking Privileges**:
  - `revoke select on department from Amit, Satoshi;`
  - Removes select privilege from users Amit and Satoshi.

4.7.2 Roles

- **Role-Based Authorization**:
  - Roles group authorizations for ease of management (e.g., instructor, dean).
  - Users are granted roles which simplify assigning and managing permissions.

```
create role instructor;
grant select on takes to instructor;
```

- **Role Inheritance**:
  - Roles can inherit privileges from other roles.
  - E.g., a dean may inherit privileges from the instructor and teaching assistant roles.

4.7.3 Authorization on Views

- **Views for Restricted Access**:
  - Users can be granted access to specific data through views without giving them access to entire relations.

```
create view geo_instructor as (select * from instructor where dept_name = 'Geology');
```

- **Authorization Checks on Views**:
  - When a query involves a view, the system checks if the user has the necessary privileges on the underlying data.

4.7.4 Authorizations on Schema

- **Schema Modifications**:
  - Only the schema owner can modify it unless specific privileges like `references` are granted.

```
grant references (dept_name) on department to Mariano;
```

4.7.5 Transfer of Privileges

- **With Grant Option**:
  - Allows users to pass on the privileges they have been granted to others.

```
grant select on department to Amit with grant option;
```

- **Revocation and its Impact**:
  - Revoking a privilege can have a cascading effect, impacting users who were indirectly granted the privilege.

4.7.6 Revoking of Privileges

- **Cascading Revocation**:
  - Default behavior in SQL; can be controlled with `restrict` or `cascade` options in the `revoke` statement.

4.7.7 Row-Level Authorization

- **Fine-Grained Access Control**:
  - Some systems support authorization at the row level, allowing for precise control over who can see or manipulate individual rows.

Overall, SQL provides a robust framework for managing database authorizations at various levels, including the ability to assign and revoke privileges and the use of roles to simplify and secure access management.

5. Advanced SQL

5.1 Accessing SQL from a Programming Language

SQL, while powerful and declarative for querying, lacks the full capabilities of general-purpose languages, necessitating integration with such languages for comprehensive application development. Here are the key reasons and methods for combining SQL with general-purpose languages:

- **Reasons for Integration**:
  1. **Expressiveness Limitations**: Some complex queries require the advanced capabilities of languages like C, Java, or Python, which are not available in SQL.
  2. **Nondeclarative Actions**: Tasks such as outputting reports, user interaction, or integrating query results into user interfaces must be handled outside SQL.
- **Methods of Accessing SQL**:
  1. **Dynamic SQL**:
     - Enables runtime construction and execution of SQL queries within general-purpose programs.
     - Supported by procedural and object-oriented languages through functions or methods that allow queries to be built as character strings, executed, and their results retrieved into program variables.
     - **Standards and APIs**:
       - **JDBC**: Java API that facilitates database connection and operations.
       - **ODBC**: Initially for C, now supports C++, C#, Ruby, Go, PHP, and Visual Basic.
       - **Python Database API**: Connects Python programs to databases.
       - **ADO.NET**: For .NET languages (e.g., C# and Visual Basic), allows data access similar to JDBC but also supports non-relational data sources.
  2. **Embedded SQL**:
     - Incorporates SQL directly within the code, identified during the compilation.
     - Uses a preprocessor to convert SQL into function calls that interact with the database at runtime through an API providing Dynamic SQL capabilities.
     - Specific to the database being used and recognized at compile time, contrasting with the runtime flexibility of Dynamic SQL.
- **Challenges**:
  - **Data Manipulation Mismatch**: SQL operates on relational data and returns sets (relations), while general-purpose languages work on individual variables. Bridging this gap requires mechanisms to adapt set-oriented SQL results into the variable-oriented structure of programming languages.
- **Alternative Approaches**:
  - **Embedded Databases**: Discussed as another option for integrating databases within applications, offering localized data management without server connectivity.

Code Example: SQL Queries

```
-- Example of a Dynamic SQL query in a programming language (Pseudocode)
query = "SELECT * FROM users WHERE age > 21";
result = execute_query(query);
while (has_more_results(result)) {
    user = fetch_next(result);
    print(user.name);
}
```

This lecture highlights the necessity and methods for integrating SQL with general-purpose programming languages to create robust, multifunctional applications capable of handling both complex data operations and non-database functionalities.

Subsection Details:

- **JDBC (Java Database Connectivity)**:
  - **Connection Establishment**: Utilizes `getConnection()` from `java.sql.DriverManager` to establish connections using database URLs, user IDs, and passwords.
  - **Executing SQL Statements**: SQL commands are sent to the database for execution via the `Statement` class. Methods like `executeQuery()` and `executeUpdate()` handle SQL queries and updates respectively.
  - **Handling Results**: Results are managed via `ResultSet` objects, with data fetched using methods like `getString()` and `getFloat()`.
  - **Prepared Statements**: Enhances security and efficiency by using placeholders for values in SQL commands, preventing SQL injection risks.
  - **Exception Handling and Resource Management**: Utilizes try-with-resources for safe management of database connections and statements to avoid resource leaks.
- **Metadata Handling**:
  - **ResultSetMetaData**: Provides metadata about the result set of a query, such as column names and types.
  - **DatabaseMetaData**: Offers detailed information about the database system, including supported features and schema details.
- **Advanced JDBC Features**:
  - **Callable Statements**: Supports SQL stored procedures and functions.
  - **Transaction Management**: Controls transaction commitment and rollback with `setAutoCommit()`, `commit()`, and `rollback()`.
  - **Large Object Handling**: Methods like `getBlob()` and `setBlob()` manage large data objects efficiently, using streams.
- **Security Considerations**:
  - **Prepared Statements**: Crucial for preventing SQL injection, ensuring that user inputs are safely incorporated into SQL commands.
  - **SQL Injection Risks**: Demonstrates the dangers of constructing SQL queries directly from user inputs and the importance of using prepared statements.

SQL Code Examples:

- **Connecting to a Database**:

```
getConnection("jdbc:oracle:thin:@db.yale.edu:2000:univdb", "user", "password");
```

- **Executing an Update**:

```
stmt.executeUpdate("INSERT INTO instructor VALUES (...)");
```

- **Fetching Query Results**:

```
ResultSet rset = stmt.executeQuery("SELECT * FROM instructor");
```

- **Prepared Statement for Insert**:

```
PreparedStatement prepStmt = conn.prepareStatement("INSERT INTO instructor VALUES
(?, ?, ?, ?)");
prepStmt.setString(1, "88878");
prepStmt.setString(2, "Perry");
prepStmt.setString(3, "Finance");
prepStmt.setFloat(4, 125000);
prepStmt.executeUpdate();
```

5.1.2 Database Access from Python

- **Database Operations in Python**:
  - **Insert Queries**: Use placeholders ("%s") to parameterize SQL queries, similar to JDBC prepared statements.
  - **Transaction Management**: Updates must be manually committed using the `commit()` method.
  - **Error Handling**: Utilizes `try`, `except` blocks to catch and print exceptions.
  - **Result Iteration**: A `for` loop is used to traverse and access query results.
  - **Database Drivers**: Examples include `psycopg2` for PostgreSQL, `MySQLdb` for MySQL, and `cx_Oracle` for Oracle. The `pyodbc` driver supports ODBC connections.
  - **API Differences**: Minor variations exist in the Python Database API across different drivers, particularly in connection parameters.

5.1.3 ODBC (Open Database Connectivity)

- **ODBC API**: Allows applications to connect to any ODBC-supported database server for querying and updates.
- **ODBC Usage**:
  - **Connection Setup**: Involves allocating SQL environment and database connection handles, using `SQLConnect`.
  - **Query Execution**: Uses `SQLExecDirect` to send SQL commands.
  - **Result Binding**: `SQLBindCol` binds C variables to SQL query results.
  - **Data Fetching**: `SQLFetch` retrieves query results into bound C variables.
  - **Transaction Control**: `SQLSetConnectOption` to disable auto-commit, requiring explicit commit or rollback.
  - **ODBC Conformance Levels**: Vary from core features to more advanced features involving arrays and detailed catalog information.

5.1.4 Embedded SQL

- **Embedded SQL in Host Languages**: SQL can be embedded into programming languages like C, Java, Cobol, etc.
- **Preprocessing and Compilation**:
  - **Preprocessor Role**: Translates embedded SQL into host language code before compilation.
  - **Runtime Execution**: Unlike JDBC, embedded SQL is preprocessed, which can catch SQL-related errors early.
- **Cursor Usage**: Cursors manage row-by-row processing of query results.
- **Embedded SQL Syntax**: Uses `EXEC SQL` to denote embedded SQL statements; host language variables within SQL statements are prefixed with a colon (":").

- **Disadvantages of Embedded SQL**:
  - **Debugging Complexity**: Generated host language code complicates debugging.
  - **Syntax Clashes**: Potential conflicts with updates in host language syntax.
- **Alternatives to Embedded SQL**: Most systems now use dynamic SQL due to its flexibility; Microsoft's LINQ is an exception, integrating query capabilities into the host language.

```
-- Example of an SQL statement with parameters
INSERT INTO department VALUES(?, ?, ?);
```

```
-- Turning off auto-commit on a connection
SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0);

-- Explicitly committing a transaction
SQLTransact(conn, SQL_COMMIT);
```

5.2 Functions and Procedures

- **Purpose**: Allows developers to write and store their own functions and procedures in the database for invocation from SQL statements. Useful for specialized data types (e.g., images, geometric objects).
- **Business Logic Storage**: Enables encapsulating business rules as database-stored procedures, enhancing maintainability and accessibility.
- **Implementation Variability**: Although based on the SQL standard, most databases implement proprietary versions (e.g., PL/SQL for Oracle, TransactSQL for Microsoft SQL Server, PL/pgSQL for PostgreSQL).

5.2.1 Declaring and Invoking SQL Functions and Procedures

- **Function Example**: Function to count instructors in a department, usable in queries to fetch department names and budgets with more than 12 instructors.

```
select dept_name, budget
from department
where dept_count(dept_name) > 12
```

- **Performance Considerations**: Potential performance issues when invoking complex user-defined functions on large datasets.
- **Table Functions**: Functions that return tables, acting like parameterized views.

```
select *
from table(instructor_of('Finance'));
```

- **Procedures Example**: Procedures can handle input (`in`) and output (`out`) parameters, invoked using SQL or embedded SQL.

```
create procedure dept_count_proc(in dept_name varchar(20), out d_count integer)
begin
  select count(*) into d_count from instructor
  where instructor.dept_name = dept_count_proc.dept_name
end
```

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

5.2.2 Language Constructs for Procedures and Functions

- **SQL as a Programming Language**: Supports variables, assignments, compound statements, loops, and conditional statements to enable complex procedural logic within SQL.
- **Transaction Control**: The `begin atomic ... end` statement ensures all contained statements are executed as a single transaction.
- **Exception Handling**: SQL can signal and handle exceptions, supporting robust error management within procedural code.

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
  -- sequence of statements
end
```

5.2.3 External Language Routines

- **Integration with External Languages**: SQL functions/procedures can be defined in languages like Java, C#, C, or C++ for enhanced efficiency and capability.
- **Security and Performance Considerations**: While external routines can be more performant, they may introduce security risks or system instability. Safe execution environments like sandboxes can mitigate some risks.

```
create procedure dept_count_proc(in dept_name varchar(20), out count integer)
language C
external name '/usr/avi/bin/dept_count_proc';
```

- **Support Across Databases**: Different databases support various external languages, impacting the portability and maintenance of database applications.

5.3 Triggers

- **Definition**: A statement executed automatically by the system due to a database modification.
- **Configuration**:
  - **Event**: Specifies when the trigger is checked.
  - **Condition**: Must be met for the trigger execution to proceed.
  - **Action**: Defined operations that execute when the trigger fires.

5.3.1 Need for Triggers

- **Purpose**:
  - Enforce integrity constraints not possible through standard SQL constraints.
  - Automate tasks and alerts based on specific database changes.
- **Examples**:

- **Course Credits**: Automatically update a student's total credits when they enroll in a new course.
- **Inventory Management**: Auto-generate reorder entries when inventory falls below a minimum level.

5.3.2 Triggers in SQL

- **Purpose and Usage**: Triggers in SQL are used to ensure data integrity and automate data management tasks by responding to specific changes or events within a database.
- **Standard vs. Non-standard Syntax**: While the SQL standard defines a specific syntax for triggers, most databases implement variations of this syntax, making it crucial to understand the concepts broadly applicable across different systems.
- **Referential Integrity Example**:
  - **Insert Trigger**: Ensures that `time slot id` in the `section` relation is valid upon insertions. The trigger checks each new row and rolls back the transaction if it violates referential integrity.
  - **SQL Query**:

```
-- This example code checks the validity of time slot IDs during insertion
create trigger check_time_slot_id after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id is not valid)
begin atomic
  rollback transaction;
end;
```

  - **Delete Trigger**: Ensures that a `time slot id` being deleted is not referenced in the `section` relation, maintaining referential integrity.
- **Handling Updates and Deletes**:
  - Triggers must also manage updates and deletes to both the `section` and `time slot` tables to ensure ongoing referential integrity.
- **Update Specific Attributes**:
  - Triggers can be configured to respond only to changes in specific attributes, which prevents unnecessary executions and enhances performance.
  - **SQL Query**:

```
create trigger update_credits after update of takes on grade
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null and (orow.grade = 'F' or
orow.grade is null)
begin atomic
  update student set tot_cred = tot_cred + (select credits from course where
course.course_id = nrow.course_id where student.id = nrow.id);
end;
```

- **Trigger Options**:
  - **Before and After**: Triggers can be set to activate before or after an event, allowing them to either prevent or correct errors.

    - **Row vs. Statement Level**: Triggers can operate on individual rows or on all rows affected by a SQL statement, with different implications for performance and complexity.
    - **Enabling and Disabling**: Triggers can be dynamically enabled or disabled, allowing for flexible database management.
- **Advanced Uses and Customization**:
    - **Set Value Modification**: Triggers can modify data as it is being inserted or updated, such as changing blank values to null.
    - **SQL Query**:

```
create trigger set_null before update of takes on grade
referencing new row as nrow
for each row
when (nrow.grade = ' ')
begin atomic
  set nrow.grade = null;
end;
```

5.3.3 When Not to Use Triggers

Triggers are powerful tools in database management, but there are instances where their use is not advisable or necessary. Alternative solutions often offer more efficient or clearer implementations.

- **Alternative to Triggers**:
    - **Foreign-Key Constraints**: Using triggers to mimic the 'on delete cascade' feature is more complex than using the built-in cascade feature of foreign-key constraints, making the database harder to understand.
    - **Materialized Views**: While triggers can maintain materialized views, such as updating a count of students per course section, modern databases support automatic maintenance of materialized views, negating the need for manual trigger management.
        - **SQL Example**:

```
CREATE TABLE section_registration (
    course_id VARCHAR(20),
    sec_id VARCHAR(10),
    semester VARCHAR(10),
    year INT,
    total_students INT
);

SELECT course_id, sec_id, semester, year, COUNT(ID) AS total_students
FROM takes
GROUP BY course_id, sec_id, semester, year;
```

- **Database Replication**: Triggers were historically used to maintain replicas of databases by recording changes in delta relations. Modern systems, however, offer built-in replication facilities that are more efficient and robust than trigger-based solutions.
- **Triggers in Backup and Replication Scenarios**:
    - **Unintended Execution**: Triggers may execute unintended actions during data loads from backups or when updates at a primary site are replicated at a backup site. To manage this, triggers can be

disabled or set as 'not for replication' to prevent execution in replication scenarios.
    - **System Variables**: Some systems use a variable to indicate a database is a replica, allowing trigger bodies to exit without executing if this condition is true.
- **Care in Trigger Use**:
    - **Runtime Errors**: Errors within triggers can cause the transaction that activated the trigger to fail.
    - **Chain Reactions**: Improperly designed triggers can initiate a chain of triggers, potentially leading to infinite loops. Database systems may limit the number of permissible trigger cascades to prevent such issues.
- **Alternatives to Triggers**:
    - **Stored Procedures**: Many tasks assigned to triggers can be more appropriately handled by stored procedures, which offer more direct control and can be easier to manage.
    -

5.4 Recursive Queries

- **Context**: Discussion on how to find both direct and indirect prerequisites for a course (e.g., CS-347) at a university.
- **Scenario**: Courses like CS-319, CS-315, and CS-101 indirectly act as prerequisites for CS-347 through a series of prerequisite chains.
- **Transitive Closure**: Describes the relationship where a course is a direct or indirect prerequisite of another course. This concept applies to multiple hierarchies such as organizational structures or mechanical parts.

5.4.1 Transitive Closure Using Iteration

- **Method**: Utilizes iterative queries to find all courses that are prerequisites using three temporary tables (`c_prereq`, `new_c_prereq`, `temp`) to track and update the list of prerequisites.
- **Process**:
    - Insert direct prerequisites into `new_c_prereq`.
    - Move these courses to `c_prereq`.
    - Find new prerequisites, store in `temp`, and update `new_c_prereq`.
    - Continue until no new courses are added, indicating all indirect prerequisites are found.
- **SQL Features**: Use of `create temporary table` and handling potential cycles with the `except` clause to ensure accurate results despite unusual data structures like cycles.

5.4.2 Recursion in SQL

- **Recursion in SQL**: Utilized to specify transitive closure more conveniently than iteration, allowing for recursive view definitions.
- **Example Use Case**: Defining courses that are prerequisites (both directly and indirectly) for a specific course such as CS-347.
- **SQL Standard on Recursion**:
    - Supports a limited form of recursion using the `with recursive` clause.
    - Recursive views are temporary and defined in terms of themselves.
- **Recursive View Definition**:
    - Must be the union of a **base query** (nonrecursive) and a **recursive query**.
    - The process continues iteratively until no new tuples are added to the view, reaching a **fixed point** where the view contains the tuples in the fixed-point instance.

- **Query Example**:
  - To find all prerequisites of CS-347, modify the query with a `where` clause filtering by the course ID.
- **SQL Query Syntax for Recursive Operation**:

```
with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id from prereq
    union
    select rec_prereq.course_id, prereq.prereq_id
    from rec_prereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
)
select * from rec_prereq;
```

- This recursive SQL query efficiently finds all direct and indirect prerequisites for a course.
- **Restrictions on Recursive Queries**:
  - Must be **monotonic**, meaning the result must grow as more tuples are added to the view relation.
  - Cannot use operations like aggregation on the recursive view, `not exists` with a subquery involving the recursive view, or set difference (`except`) where the right-hand side uses the recursive view.
- **Other SQL Implementations**:
  - Some databases like Oracle use different syntax such as `start with / connect by prior` for hierarchical queries.

5.5 Advanced Aggregation Features in SQL

5.5.1 Ranking

- **Purpose**: Ranking helps determine the position of a value within a set, such as ranking students by GPA.
- **SQL Constructs for Ranking**:
  - Standard SQL ranking is challenging and may combine SQL with other programming languages for efficiency.
  - Example ranking query:

```
select ID, rank() over (order by GPA desc) as s_rank
from student_grades;
```

  - An additional `order by` is required to sort by rank:

```
select ID, rank() over (order by GPA desc) as s_rank
from student_grades
order by s_rank;
```

- **Handling Ties**:
  - The `rank` function assigns the same rank to tied values, creating gaps in subsequent rankings.
  - The `dense_rank` function does not create gaps after ties, providing a sequential ranking.
- **Null Values**:
  - Nulls can be treated as the highest values or controlled with `nulls first` or `nulls last` options:

```
select ID, rank() over (order by GPA desc nulls last) as s_rank
from student_grades;
```

- Traditional SQL approach to simulate rank:

```
select ID, (1 + (select count(*) from student_grades B where B.GPA > A.GPA)) as
s_rank
from student_grades A
order by s_rank;
```

- **Partitioned Ranking**:
  - Ranking within categories (e.g., by department) uses the `partition by` clause:

```
select ID, rank() over (partition by dept_name order by GPA desc) as dept_rank
from dept_grades
order by dept_name, dept_rank;
```

- **Efficient Ranking**:
  - Nested queries for top-n results:

```
select *
from (select ID, rank() over (order by GPA desc) as s_rank from student_grades)
where s_rank <= 5;
```

  - This may return more than `n` results due to ties.
- **Alternative Ranking Functions**:
  - **Percent Rank**: Calculates rank as a fraction of the total count.
  - **Cume Dist**: Cumulative distribution function.
  - **Row Number**: Assigns a unique number to each row based on order.
  - **Ntile**: Divides data into nearly equal buckets, useful for histograms:

```
select ID, ntile(4) over (order by GPA desc) as quartile
from student_grades;
```

  - These functions offer various ways to interpret and analyze data rankings differently depending on the specific requirements of the analysis.

This comprehensive exploration of SQL ranking functions underscores their versatility and power in data analysis, allowing for detailed and nuanced insights into data sets.

5.5.2 Windowing

- **Window Queries**: Compute aggregate functions over ranges of tuples, useful for analyzing trends over time. Windows may overlap, allowing tuples to contribute to multiple windows, unlike partitions where a tuple contributes to only one.
- **Use Cases**:
  - **Trend Analysis**: Analyzing sales data to understand trends affected by external factors like weather. Similarly, stock market trends can be analyzed using various moving averages.

- **SQL Windowing Feature**:
  - Simplifies queries that compute aggregates over sliding windows of data.
  - Example: Calculating moving averages over fixed periods without writing cumbersome SQL for each window.
- **SQL Queries for Windowing**:
  - **Fixed Window Size Example**:

```
SELECT year, AVG(num_credits) OVER (ORDER BY year ROWS 3 PRECEDING) AS
avg_total_credits
FROM tot_credits;
```

    This query calculates the average number of credits over the three preceding years.
  - **Unbounded Preceding Window Example**:

```
SELECT year, AVG(num_credits) OVER (ORDER BY year ROWS UNBOUNDED PRECEDING) AS
avg_total_credits
FROM tot_credits;
```

    This computes the average over all prior years to a given year.
  - **Variable Size Window Example**:

```
SELECT year, AVG(num_credits) OVER (ORDER BY year ROWS BETWEEN 3 PRECEDING AND
2 FOLLOWING) AS avg_total_credits
FROM tot_credits;
```

    Averages credits over a window that starts three years before and ends two years after the current year.
  - **Department-Specific Window Example**:

```
SELECT dept_name, year, AVG(num_credits) OVER (PARTITION BY dept_name ORDER BY
year ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) AS avg_total_credits
FROM tot_credits_dept;
```

    This query partitions data by department and computes the average credits for the department over a defined window.
- **Range vs. Rows**:
  - `ROWS`: Specifies the window in terms of physical rows.
  - `RANGE`: Covers all tuples with a particular value rather than a specific number of tuples.
  - Note: The `RANGE` keyword might not be fully implemented in all SQL systems.

5.5.3 Pivoting

- **Purpose**: To transform rows into columns, typically used in data analysis to view data from different perspectives.
- **SQL Query for Pivot**:

```
SELECT * FROM sales PIVOT (SUM(quantity) FOR color IN ('dark', 'pastel', 'white'));
```

- **Application**: Useful for summarizing sales data by different attributes like item name, color, and size in a compact form.

5.5.4 Rollup and Cube

- **Purpose**: To perform multiple `GROUP BY` queries in a single SQL query, useful for generating comprehensive summaries across various dimensions.
- **Rollup Example**:

```
SELECT item_name, color, SUM(quantity) FROM sales GROUP BY ROLLUP(item_name,
color);
```

- **Cube Example**:

```
SELECT item_name, color, clothes_size, SUM(quantity) FROM sales GROUP BY
CUBE(item_name, color, clothes_size);
```

- **Grouping Sets**: Provides the ability to specify exact groupings.

```
SELECT item_name, color, clothes_size, SUM(quantity) FROM sales GROUP BY GROUPING
SETS((color, clothes_size), (clothes_size, item_name));
```

- **Handling Nulls with Grouping Functions**: Distinguishes between nulls generated by rollup/cube operations and actual data nulls.

```
SELECT (CASE WHEN grouping(item_name) = 1 THEN 'all' ELSE item_name END) AS
item_name, (CASE WHEN grouping(color) = 1 THEN 'all' ELSE color END) AS color,
SUM(quantity) AS quantity FROM sales GROUP BY ROLLUP(item_name, color);
```

These features enhance SQL's capability to handle complex data aggregation and analysis tasks efficiently and effectively.

6. Database Design Using the E-R Model

6.1 Overview of the Design Process

- **Complexity**: Involves designing database schema, access/update programs, and a security scheme.
- **User-Centric**: Designing process primarily driven by user needs.

6.1.1 Design Phases

- **Initial Phase**:
  - **User Interaction**: Essential for characterizing data needs through extensive interaction with domain experts and users.
  - **Outcome**: Specification of user requirements, typically captured textually.
- **Conceptual Design**:
  - **Data Model Selection**: Choosing a data model to translate requirements into a conceptual schema.

    - **Entity-Relationship Model**: Commonly used to represent the conceptual schema through entities, attributes, relationships, and constraints.
    - **Entity-Relationship Diagram**: Provides a graphical representation of the schema.
    - **Schema Review**: Ensures all data requirements are met and checks for redundancies.
    - **Functional Requirements**: Schema should support specified operations like modifying, searching, and deleting data.
- **Logical Design**:
    - **Schema Mapping**: Translates high-level conceptual schema into a relational schema suitable for implementation.
    - **Implementation Data Model**: Typically the relational model.
- **Physical Design**:
    - **Database Implementation**: Specifies physical features like file organization and indexing.
    - **Schema Flexibility**: Physical schema is more adaptable post-implementation compared to logical schema.

6.1.2 Design Alternatives

- **Entity Representation**:
    - **Entities**: Distinct items such as people, places, and products.
    - **Examples**: In a university database, entities might include instructors, students, departments, courses, and sections of courses.
- **Avoiding Design Pitfalls**:
    - **Redundancy**: Avoid repeating information across the database to prevent inconsistencies (e.g., course titles stored with each offering).

```
-- Example to avoid redundancy by normalizing data
CREATE TABLE Courses (
  CourseID int PRIMARY KEY,
  Title varchar(100),
  DeptName varchar(100),
  Credits int
);

CREATE TABLE CourseOfferings (
  OfferingID int PRIMARY KEY,
  CourseID int,
  Semester varchar(10),
  Year int,
  FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

    - **Incompleteness**: Ensure all necessary aspects of the enterprise can be modeled, avoiding schemas that cannot represent all required information.
- **Modeling Relationships**:
    - **Consideration**: Whether a relationship (like a sale) is an interaction between entities or an entity itself.

6.2 The Entity-Relationship Model

- **Purpose**: Facilitates database design through an enterprise schema representing the logical structure of a database.
- **Utility**: Maps real-world interactions onto a conceptual schema, widely used in database design tools.
- **Key Concepts**:
  - **Entity Sets**: Groups of entities (real-world objects) sharing properties. Examples include people in a university categorized as 'instructors' or 'students'.
  - **Relationship Sets**: Associations among entities, such as the advisor relationship between instructors and students.
  - **Attributes**: Properties of entities, such as a person's ID or name.
- **E-R Diagrams**: Visual representation of databases using entity sets (rectangles) and relationship sets (diamonds).

6.2.1 Entity Sets

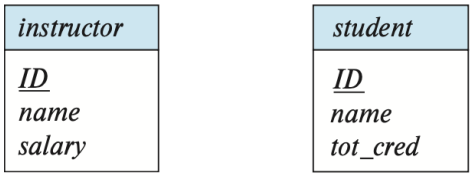


Figure 6.1 E-R diagram showing entity sets *instructor* and *student*.

- **Definition**: An entity is an object distinct from other objects, identified by a unique set of properties (e.g., a person identified by a person ID).
- **Types of Entities**: Can be concrete (like a book) or abstract (like a course offering).
- **Attributes**: Descriptive properties unique to each entity within a set. Common attributes include ID, name, and department name. Key attributes are underlined in diagrams.

6.2.2 Relationship Sets

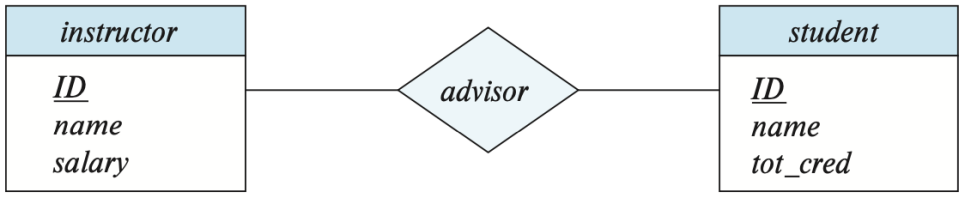


Figure 6.3 E-R diagram showing relationship set *advisor*.

- **Definition**: Associations among multiple entities, such as an instructor advising a student.
- **Complex Relationships**: In addition to binary relationships (involving two entity sets), relationships can be ternary (involving three entity sets) or even more complex.
- **Attributes of Relationships**: Relationships can have descriptive attributes (like grades in a student-section relationship).

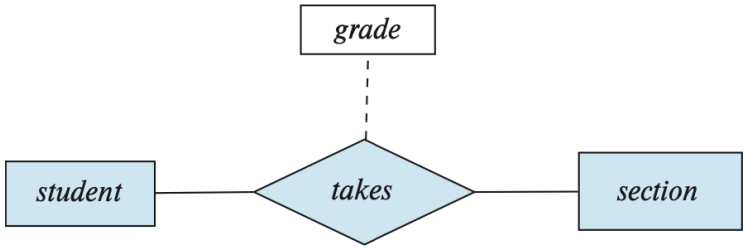


Figure 6.5 E-R diagram with an attribute attached to a relationship set.

- **E-R Diagram Representation**:
  - Entities are shown as rectangles divided into two sections for the entity name and its attributes.
  - Relationships are depicted with diamonds, linked to entities with lines. Attributes of relationships are shown with dashed lines connecting to the diamonds.

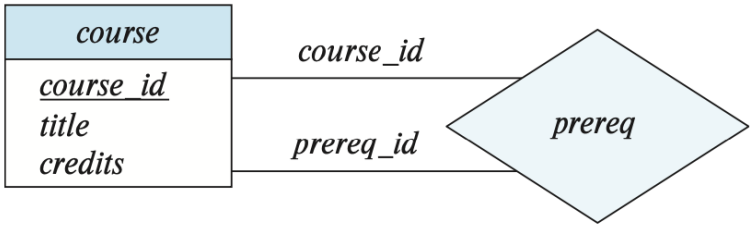


Figure 6.4 E-R diagram with role indicators.

- **Role of Entities in Relationships**: Entities may participate in multiple roles within relationships. For example, a course might be related to another as a prerequisite.

SQL Representation (Hypothetical Queries)

```sql
-- Assuming the existence of tables for entity sets and relationships:
-- Query to find the advisor for a specific student:
SELECT instructor.name
FROM advisor
JOIN instructor ON advisor.instructor_id = instructor.id
JOIN student ON advisor.student_id = student.id
WHERE student.name = 'Shankar';

-- Query to list all courses and their prerequisites:
SELECT course.title AS course, prereq.title AS prerequisite
FROM course
JOIN prereq ON course.prereq_id = prereq.course_id;
```

6.3 Complex Attributes

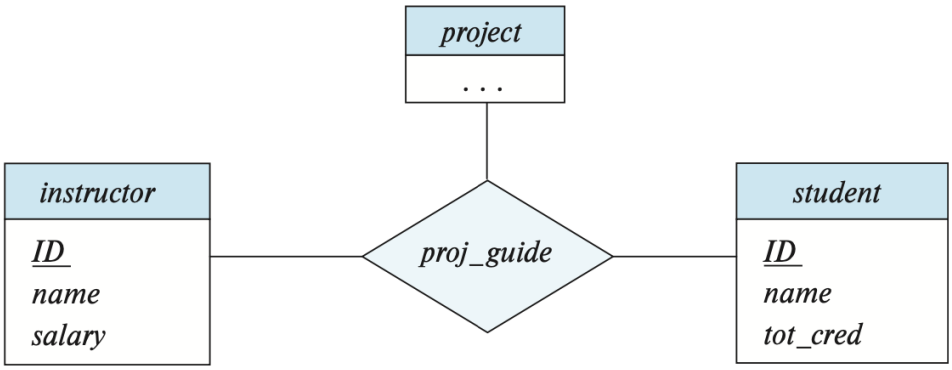


Figure 6.6 E-R diagram with a ternary relationship *proj_guide*.

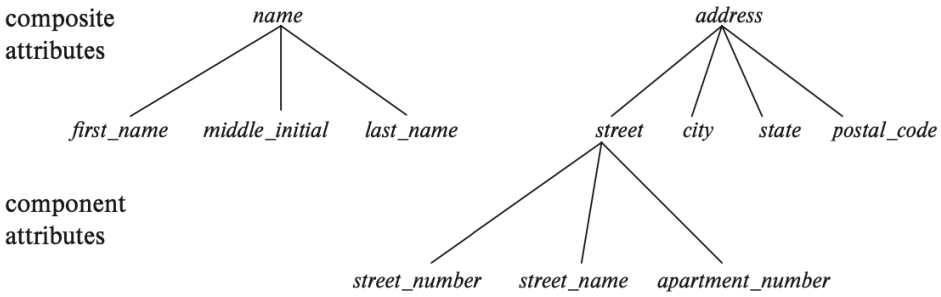


Figure 6.7 Composite attributes instructor *name* and *address*.

- **Domain**: Each attribute has a set of allowed values known as its domain. For instance, the domain for `course_id` might include all text strings of a specified length, whereas the domain for `semester` might be the strings {Fall, Winter, Spring, Summer}.
- **Attribute Types in the E-R Model**:
  - **Simple and Composite Attributes**:
    - **Simple Attributes**: These are not divided into smaller parts.
    - **Composite Attributes**: These can be broken down into smaller subparts or attributes. For example, an `address` attribute might be a composite of street, city, state, and postal code, allowing for a more structured data approach.
    - **Hierarchical Composite Attributes**: A composite attribute like `street` could further be divided into street number, street_name, and apartment number, forming a hierarchical structure.
  - **Single-valued and Multivalued Attributes**:
    - **Single-valued Attributes**: Attributes that have only one value for each entity, like a `student ID`.
    - **Multivalued Attributes**: Attributes that can have multiple values for a single entity. For instance, an instructor might have multiple phone numbers or several dependents, represented as `{phone number}` or dependent names respectively.
  - **Derived Attributes**: Attributes whose values are calculated from other attributes. An example is `students_advised`, where the number of students an instructor advises can be counted from associated student entities.
- **Null Values in Attributes**:
  - **Indications of Null Values**:
    - **Not Applicable**: The attribute value does not exist for the entity.
    - **Unknown**: The attribute value is missing (exists but not known) or it is not known whether the value exists.

- **Examples**: A null value for `middle initial` might indicate a non-applicable situation (e.g., the person has no middle name). A null in `apartment number` could imply the absence of an apartment in the address or unknown information about the apartment.

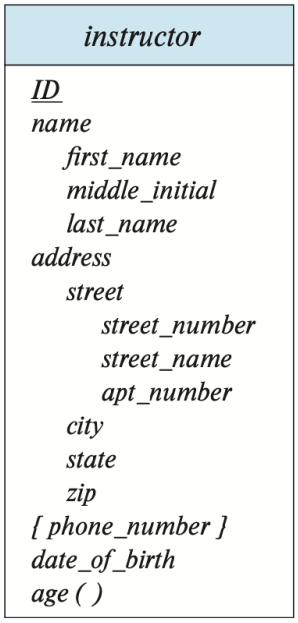


Figure 6.8 E-R diagram with composite, multivalued, and derived attributes.

Representation in E-R Notation:

- **Composite Attributes**: Illustrated as components within a larger attribute, such as an `address` consisting of street number, street name, and apartment number.
- **Multivalued Attributes**: Denoted by curly braces around the attribute, e.g., `{phone number}`.
- **Derived Attributes**: Represented with parentheses, e.g., `age ()`.

SQL Representation:

- **Handling Multivalued Attributes**: In SQL, these might be handled by creating separate tables to store the multiple values due to the relational model's constraints.
- **Derived Attributes**: Can be managed through SQL queries that calculate values based on other data in the database.

```sql
-- Example SQL query to derive 'students_advised' attribute
SELECT instructor_id, COUNT(student_id) AS students_advised
FROM advising_relationships
GROUP BY instructor_id;
```

6.4 Mapping Cardinalities

- **Mapping Cardinalities or Cardinality Ratios**: Express the number of entities in one entity set that can be associated with entities in another entity set via a relationship set. Primarily used in binary relationships but applicable to more complex sets.

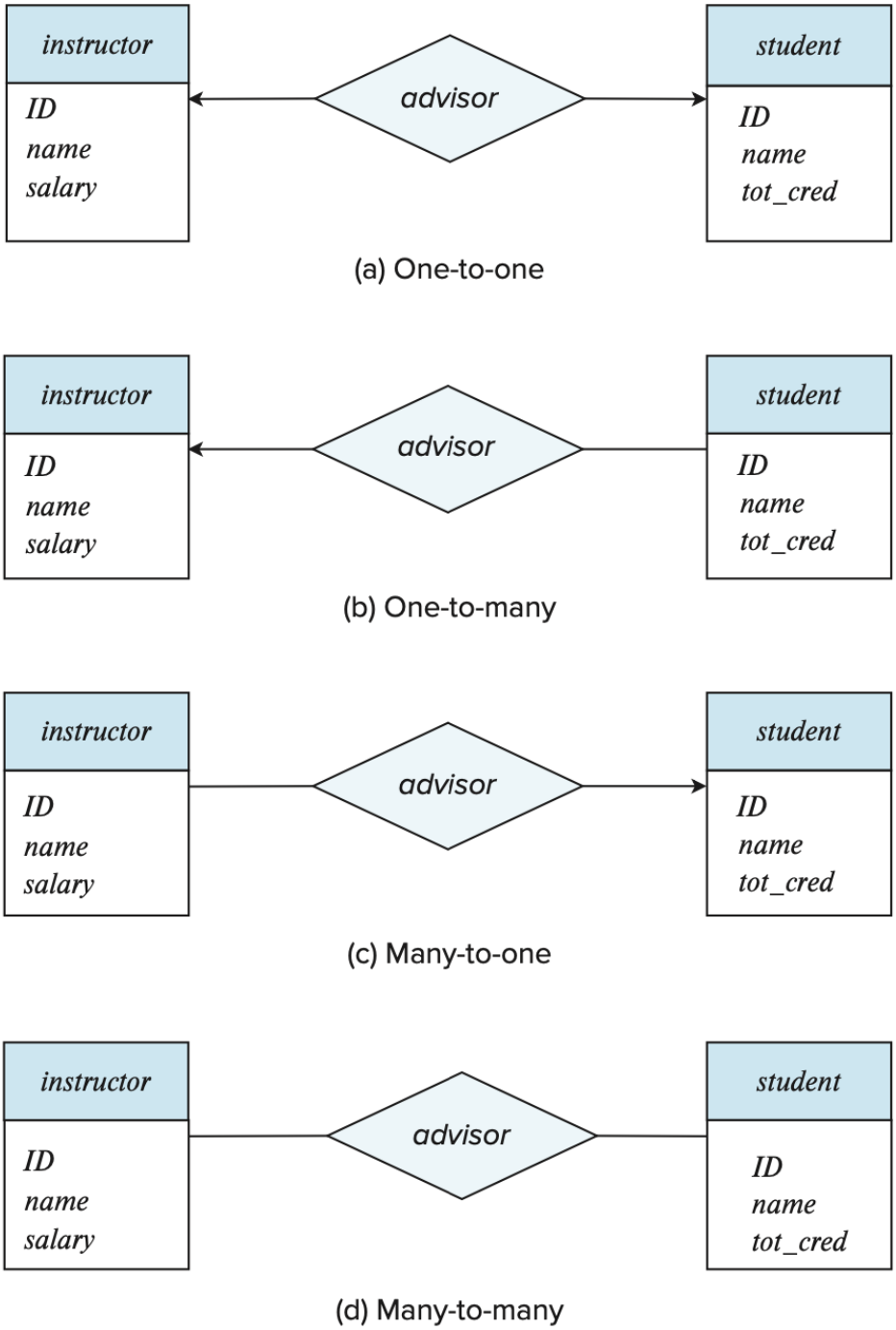


Figure 6.11 Relationship cardinalities.

- **Types of Cardinalities in Binary Relationship Sets**:
  - **One-to-one**: Each entity in entity set A is associated with at most one entity in entity set B and vice versa.
  - **One-to-many**: One entity in A can be associated with multiple entities in B, but each entity in B can only be associated with one entity in A.
  - **Many-to-one**: One entity in B can be associated with multiple entities in A, but each entity in A can only be associated with one entity in B.
  - **Many-to-many**: Entities in A can be associated with any number of entities in B and vice versa.
- **Real-world Implications**:
  - Used to model real-world relationships such as advising relationships at universities.
  - Depending on university policies, an advising relationship could be one-to-many (one instructor advises many students) or many-to-many (students advised by multiple instructors).
- **Representation in E-R Diagrams**:

    - **Cardinality Representation**: Indicated with directed lines (arrows) for one-to-one or one-to-many relationships, and undirected lines for many-to-one or many-to-many relationships.
    - **Participation**: Total participation is shown with double lines, indicating that every entity in the set participates in at least one relationship.
    - **Cardinality Constraints**: Often noted on the lines connecting entities, with notation like 1..1 (exactly one), 0..* (zero or more).
- **Examples**:
    - **Advisor Relationship Set**:
        - Total participation of students (every student must have an advisor).
        - Partial participation of instructors (not all instructors must advise).

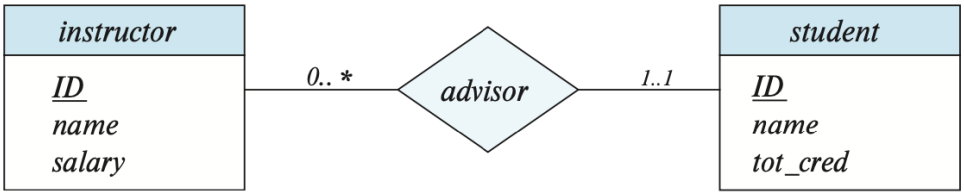


Figure 6.13 Cardinality limits on relationship sets.

- **Complex Constraints**:
    - Allows specifying minimum and maximum numbers of participations of entities in relationships.
    - E-R diagrams can alternatively show these constraints with lines and arrows instead of cardinality limits.
- **SQL Representation**:
    - Specific cardinality constraints can influence SQL schema design, affecting keys and foreign key constraints, but detailed SQL code generation from cardinality specifications isn't directly applicable without the broader context of the database schema design.

6.5 Primary Key

- **Purpose**: Distinct identification of entities within an entity set and relationships within a relationship set based on their attribute values.

6.5.1 Entity Sets

- **Uniqueness**: Each entity must have attribute values that uniquely identify it, ensuring no two entities in an entity set have the same values for all attributes.
- **Key Types**:
    - **Superkey**: A set of attributes that can uniquely identify entities.
    - **Candidate Key**: A minimal superkey; minimal set of attributes needed to uniquely identify entities.
    - **Primary Key**: A candidate key selected to uniquely identify entities in a database table.

6.5.2 Relationship Sets

- **Identification of Relationships**: Using the primary keys of the participating entity sets to form a unique primary key for the relationship set.
- **Composition of Primary Key**:
    - If **no attributes** are associated with a relationship set (R), then the primary key is the union of the primary keys from all participating entity sets (\text{primary-key}(E1) \cup \text{primary-key}(E2)

\cup \ldots \cup \text{primary-key}(En)).
    - If **attributes are associated** (a1, a2, \ldots, am) with the relationship set (R), the primary key includes these attributes alongside the primary keys from the entity sets.
- **Handling Non-Unique Attribute Names**: Attributes are renamed to include the entity set name to maintain uniqueness.
- **Special Cases**:
    - **Many-to-Many Relationships**: The primary key is the union of the primary keys of the involved entities.
    - **Many-to-One and One-to-Many Relationships**: The primary key of the "many" side is used.
    - **One-to-One Relationships**: The primary key of either entity can be used.
    - **Nonbinary Relationships**: The primary key depends on cardinality constraints; if constraints are complex, a combination of primary keys from multiple entities may form the candidate key.

6.5.3 Weak Entity Sets

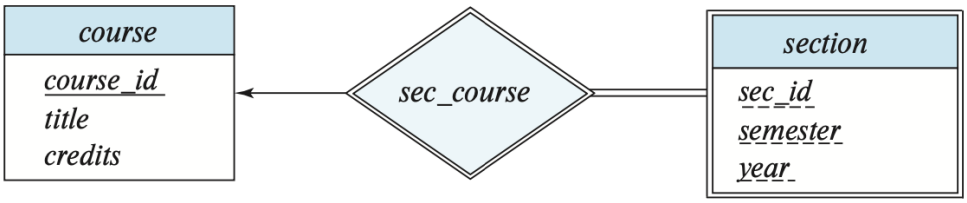


Figure 6.14 E-R diagram with a weak entity set.

- **Definition**: Entities that do not have enough attributes to form a primary key on their own and are dependent on another entity set, termed as an identifying entity set.
- **Identification**: Weak entities use the primary key of the identifying entity set combined with discriminator attributes to form a unique identifier.
- **Total Participation**: Every weak entity must be associated with an identifying entity, indicating a many-to-one relationship from the weak entity to the identifying entity.
- **ER Diagram Representation**: Weak entity sets are depicted with a double rectangle and connected to their identifying entity set with a double diamond.

SQL Representation of Primary Key Constraints:
For example, defining primary keys within an SQL database for an entity set and a relationship set could be represented as:

```
-- Creating an entity table with a primary key
CREATE TABLE Entity (
    EntityID INT NOT NULL,
    Attribute1 VARCHAR(255),
    Attribute2 VARCHAR(255),
    PRIMARY KEY (EntityID)
);

-- Creating a relationship table with a composite primary key
CREATE TABLE Relationship (
    Entity1ID INT NOT NULL,
    Entity2ID INT NOT NULL,
    PRIMARY KEY (Entity1ID, Entity2ID),
    FOREIGN KEY (Entity1ID) REFERENCES Entity(EntityID),
```

```
    FOREIGN KEY (Entity2ID) REFERENCES Entity(EntityID)
);
```

This SQL code defines primary keys for a hypothetical entity and its relationship set, ensuring uniqueness and referential integrity in a relational database system.

6.6 Removing Redundant Attributes in Entity Sets

When constructing a database using the E-R model, identifying relevant entity sets and attributes is crucial. In the context of a university organization, key entity sets such as **student** and **instructor** were determined. Attributes for each entity set need careful selection to avoid redundancy and accurately reflect the organization's structure.

Entity and Attribute Selection

- **Instructor Entity Set**: Attributes include ID, name, salary, and dept_name.
- **Department Entity Set**: Contains dept_name, building, and budget. Dept_name is the primary key.

Identifying Redundancies
Redundant attributes across entity sets can complicate the database schema and should be minimized:

- **Dept_name**: Redundant in the **instructor** entity set since it's already defined as a primary key in the **department** entity set. Such redundancy is avoided by establishing relationships rather than duplicating attributes.
- **Time Slot ID and Building/Room Number**: Similar redundancies were identified in the **section** entity set, which contains attributes that are primary keys in the **time slot** and **classroom** entity sets, respectively.

Relationship Sets
Properly defining relationship sets helps eliminate unnecessary attributes within entity sets by using relationships:

- **inst_dept**: Connects instructors with their departments.
- **stud_dept**: Associates students with departments.
- **teaches**, **takes**, **sec_class**, **sec_time_slot**, **advisor**, **prereq**: Additional relationships defining interactions between courses, sections, students, and instructors.

E-R Design Validation
The final E-R design was validated to ensure no entity set contains attributes made redundant by relationship sets. The design replaces redundant attributes with relationship sets, capturing all necessary information while maintaining simplicity and integrity.

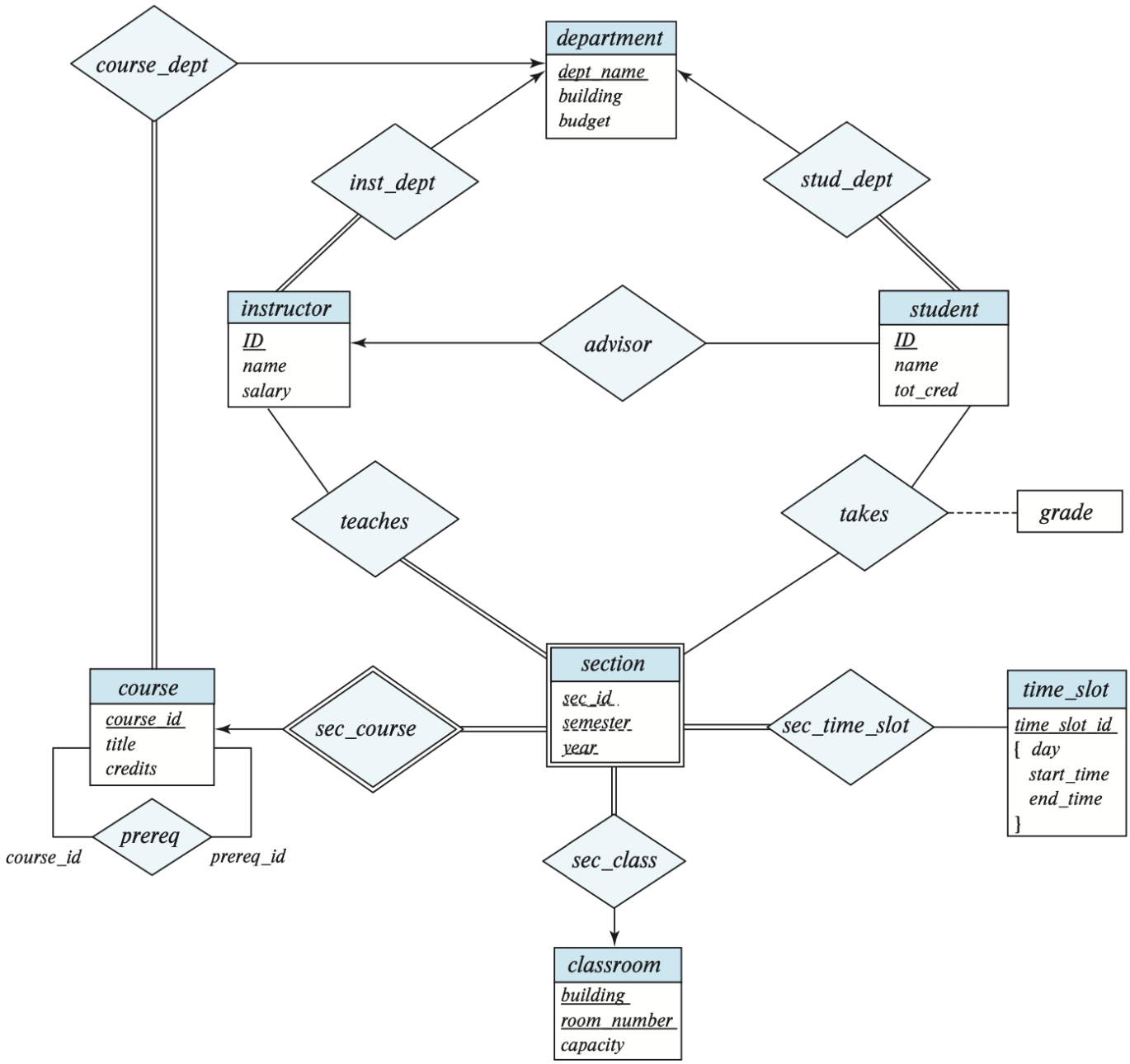


Figure 6.15 E-R diagram for a university enterprise.

Diagram and Constraints

- **E-R Diagram**: Reflects the university structure with constraints like total participation of instructors in departments, indicated by a double line between the **instructor** and **inst_dept**.
- **Unique Constraints**: Arrows from relationship sets to departments indicate that each entity (instructor, course, student) is associated with exactly one department.

SQL Considerations

In relational databases derived from E-R diagrams, SQL is used to manage and query data. For instance, removing an instructor's department could be represented in SQL as follows:

```
ALTER TABLE Instructor
DROP COLUMN dept_name;
```

Reduction to Relational Schemas

Reduction to Relational Schemas

- Entity sets and relationship sets can be expressed uniformly as relational schemas that represent the contents of the database
- A database which conforms to an ER diagram can be represented by a collection of schemas
- For each entity set and relationship set, there is a unique schema that is assigned the name of the corresponding entity set or relationship set
- Each schema has a number of columns (generally corresponding to attributes), which have unique names

Representing Entity Sets

- A strong entity set reduces to a schema with the same attributes student (ID, name, tot_cred)
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set
  section (course_id, sec_id, sem, year)

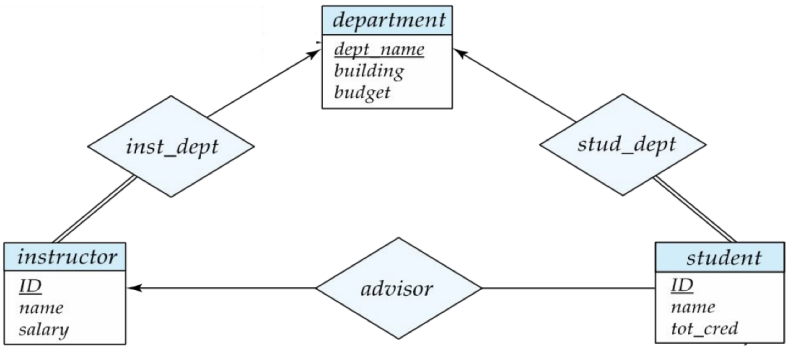
Entity Sets with Composite Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - e.g., given entity set instructor with a composite attribute name with component attributes first_name, middle_initial, and last_name, the schema corresponding to the entity set has three attributes name_first_name, name_middle_initial, and name_last_name
  - The prefix can be omitted if there is no ambiguity
    - name_first_name could be first_name
- Ignoring multivalued attributes, extended instructor schema is instructor(ID, first_name, middle_initial, last_name, street_number, street_name, apt_number, city, state, zip_code, date_of_birth)

Entity Sets with Multivalued Attributes

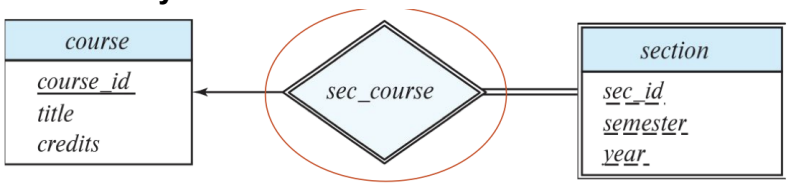
- A multivalued attribute M of an entity E is represented by a separate schema EM
- The schema EM has attributes corresponding to the primary key of E and an attribute corresponding to the multivalued attribute M
  - e.g., a multivalued attribute phone_number of instructor is represented by a schema: inst_phone = (ID, phone_number)
- Each value of the multivalued attribute maps to a separate tuple of the relation on the schema EM
  - For example, an instructor entity with the primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples: (22222, 456-7890) and (22222, 123-4567)

Representing Relationship Sets



- A many-to-many relationship set is represented as a schema with attributes for the primary keys of the two participating entity sets and any descriptive attributes of the relationship set
  - e.g., schema for a relationship set advisor advisor = (s_id, i_id)
- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
  - e.g., instead of creating a schema for a relationship set inst_dept, add an attribute dept_name to the schema arising from the entity set instructor
- For one-to-one relationship sets, either side can be chosen to act as the “many” side
  - An extra attribute can be added to either of the tables corresponding to the two entity sets If participation is partial on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values
  - e.g., suppose, in the previous slide, we added an attribute instructor_ID to the schema for the entity set student instead of creating a schema for a relationship set advisor; if a student does not have an advisor, its instructor_ID will become null

Redundancy of Schemas



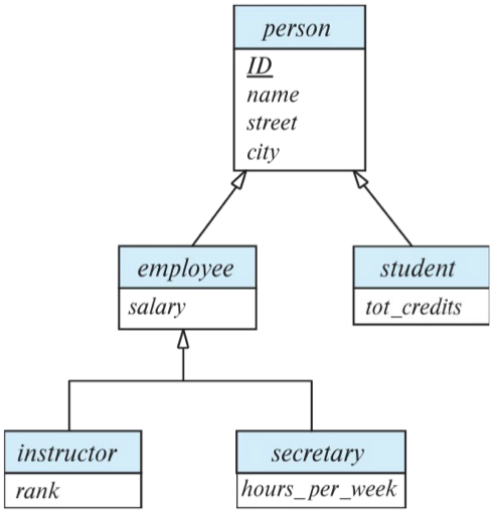
- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant
  - e.g., the section schema already contains the attributes that would appear in the sec_course schema

Extended ER Features

Specialization

- Top-down design process: we designate sub-groupings within an entity set that are distinctive from other entities in the set
- These sub-groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set
- It is depicted by a triangle component labeled ISA
  - e.g., an instructor “is a” person
- Attribute inheritance: a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked

Specialization Example



- Overlapping: employee and student
- Disjoint: instructor and secretary
- Total and partial

Representing Specialization via Schemas
Method 1:

schema	attributes
person	ID, name, street, city
student	ID, tot_credits
employee	ID, salary
![[Pasted image 20240422142117.png	250]]

- Forming a schema for the higher-level entity
- Forming a schema for each lower-level entity set and including the primary key of the higher-level entity set and local attributes
- Drawback: getting information about an employee requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema
  Method 2:

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_credits
employee	ID, name, street, city, salary
![[Pasted image 20240422142139.png	250]]

- Forming a schema for each entity set with all local and inherited attributes
- Drawback: name, street, and city may be stored redundantly for people who are both students and employees

Generalization

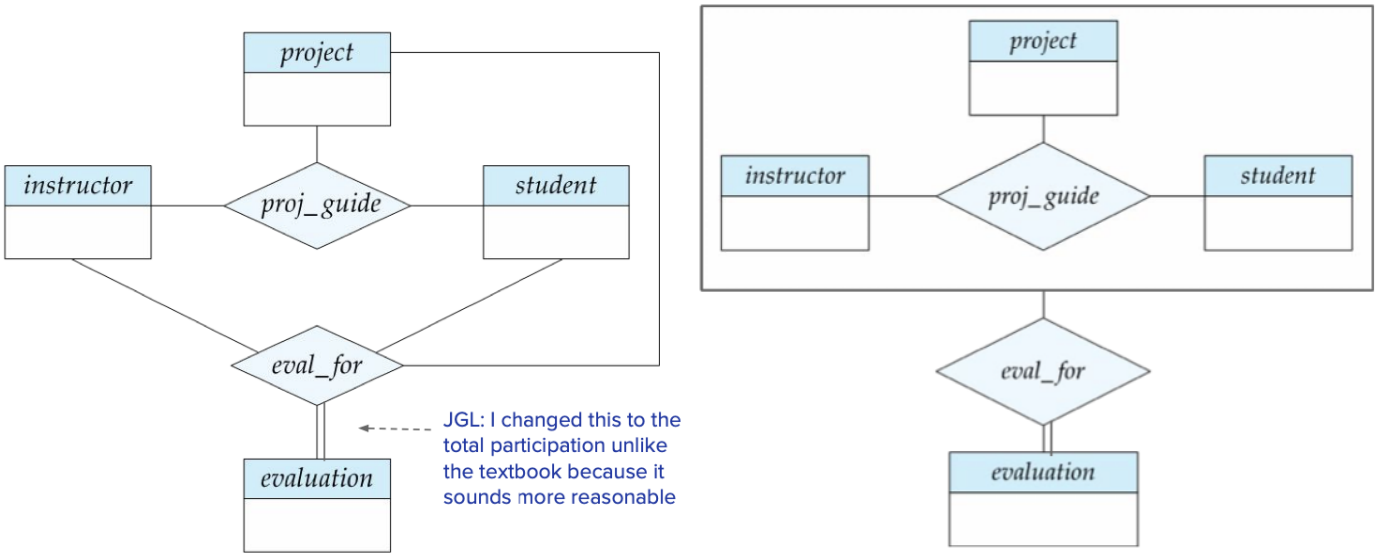
- Bottom-up design process: we combine a number of entity sets that share the same features into a higher-level entity set

- Specialization and generalization are simple inversions of each other; they are represented in an ER diagram in the same way
- The terms specialization and generalization are used interchangeably

Completeness Constraint

- Completeness constraint: specifying whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization
  - Total: an entity must belong to one of the lower-level entity sets
  - Partial: an entity need not belong to one of the lower-level entity sets
- Partial generalization is the default
- We can specify total generalization in an ER diagram by adding the keyword “total” in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies or to the set of hollow arrow-heads to which it applies
- e.g., the student generalization is total: All student entities must be either graduate or undergraduate

Aggregation



- Consider the ternary relationship proj_guide, which we saw earlier
- Suppose we want to record evaluations of a student by a guide on a project
- Relationship sets eval_for and proj_guide represent overlapping information
  - Every eval_for relationship corresponds to a proj_guide relationship
  - However, some proj_guide relationships may not correspond to any eval_for relationships
    - So we cannot discard the proj_guide relationship
- Eliminate this redundancy via aggregation
  - Treating a relationship as an abstract entity
  - Allowing a relationship between relationships
  - Abstraction of a relationship into a new entity
- Eliminate this redundancy via aggregation without introducing redundancy, the following diagram represents:
  - A student is guided by a particular instructor on a particular project
  - A student, instructor, project combination may have an associated evaluation

Reduction to Relational Schemas

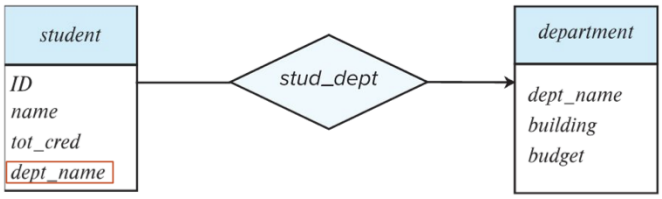
- To represent aggregation, create a schema containing

- The primary key of the aggregated relationship
- The primary key of the associated entity set
- Any descriptive attributes, if exists

- In our example:
  - The schema eval_for is:
    - eval_for (s_ID, project_id, i_ID, evaluation_id)
  - The schema proj_guide is redundant, provided that we are willing to store null values for the score attribute in the relation on evaluation (added by JGL)
    - evaluation (id, score): the score attribute could be null if the corresponding instructor, student, project combination was not evaluated

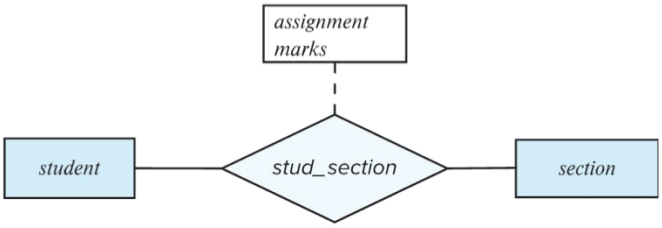
Design Issues

Common Mistakes in E-R Diagrams



Use of a primary key of an entity set as an attribute of another entity set

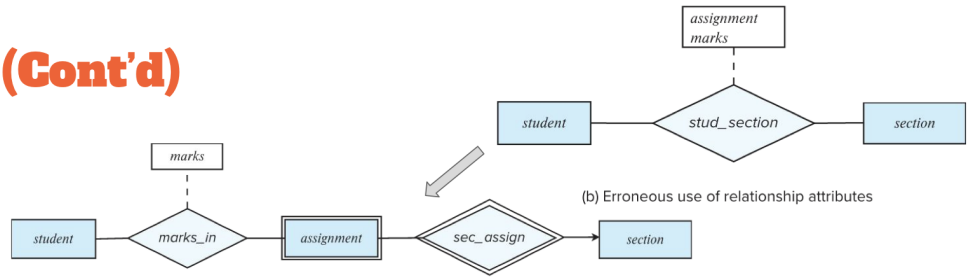
(a) Incorrect use of attribute



Use a relationship with a single-valued attribute in a situation that requires a multi-valued attribute

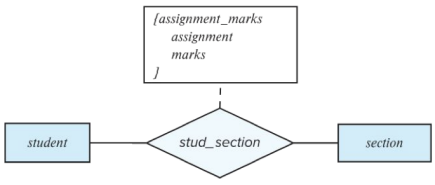
(b) Erroneous use of relationship attributes

(Cont'd)



(b) Erroneous use of relationship attributes

(c) Correct alternative to erroneous E-R diagram (b)



(d) Correct alternative to erroneous E-R diagram (b)

Attributes vs. Entity Sets

- Use of attributes vs. entity sets
  - e.g., exactly one phone number vs. multiple phone numbers
    - Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

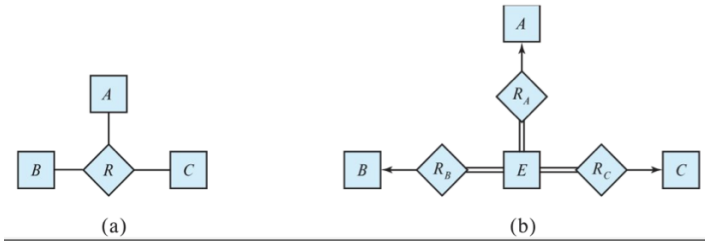
Relationship Sets vs. Entity Sets

- Use of relationship sets vs. entity sets
  - e.g., both designs accurately represent the university's information, but the use of takes is more compact and probably preferable
  - A possible guideline is to designate a relationship set to describe an action that occurs between entities
- Placement of relationship attributes
  - e.g., the attribute date as an attribute of advisor or as an attribute of student

Binary vs. Non-Binary Relationships

- Although it is possible to replace any non-binary (n-ary, for n > 2) relationship set by a number of distinct binary relationship sets, a n-ary relationship set shows more clearly that several entities participate in a single relationship
- Some relationships that appear to be non-binary may be better represented using binary relationships
  - For example, a ternary relationship parents, relating a child to his/her father and mother, is best replaced by two binary relationships, father and mother
    - Using two binary relationships allows partial information (e.g., only mother being known)
  - But there are some relationships that are naturally non-binary
    - e.g., proj_guide

Converting Non-Binary Relationships



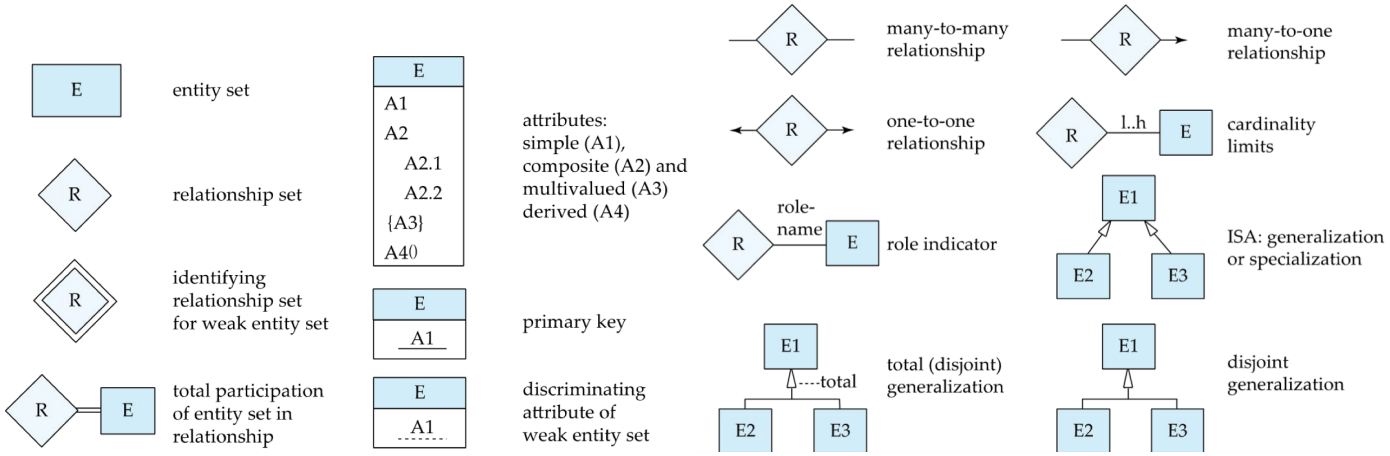
- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set
  - Replace R between entity sets A, B, and C by an entity set E and three relationship sets:
    - 1. RA, relating E and A 2. RB, relating E and B 3. RC, relating E and C
  - Create an identifying attribute for E and add any attributes of R to E
  - For each relationship (ai, bi, ci) in R,
    - 1. create a new entity ei in the entity set E 2. add (ei , ai) to RA 3. add (ei, bi) to RB 4. add (ei , ci) to RC
- There may not be a way to translate constraints on the ternary relationship into those on the binary relationships
  - e.g., consider a constraint saying that R is many-to-one from A, B to C; that is, each pair of entities from A and B is associated with at most one C entity
    - This constraint cannot be expressed by using cardinality constraints on RA, RB, and RC

ER Design Decisions

- The use of an attribute or entity set to represent an object
- Whether a real-world concept is best expressed by an entity set or a relationship set
- The use of a ternary relationship versus a pair of binary relationships
- The use of a strong or weak entity set

- The use of specialization/generalization ← contributes to modularity in the design
- The use of aggregation ← can treat the aggregate entity set as a single unit without concern for the details of its internal structure

Symbols Used in ER Notation



7. Normalization EXAM! - EASY TO MAKE PROBLEMS

Overview of Normalization

Features of Good Relational Designs

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- Suppose we combine instructor and department into in_dep, which represents the natural join on the relations instructor and department
- There is repetition of information - the red box(Physics, Watson, 70000)
- Null values are needed for a new department with no instructor(When initiating a new department)

Combined Schema Without Repetition

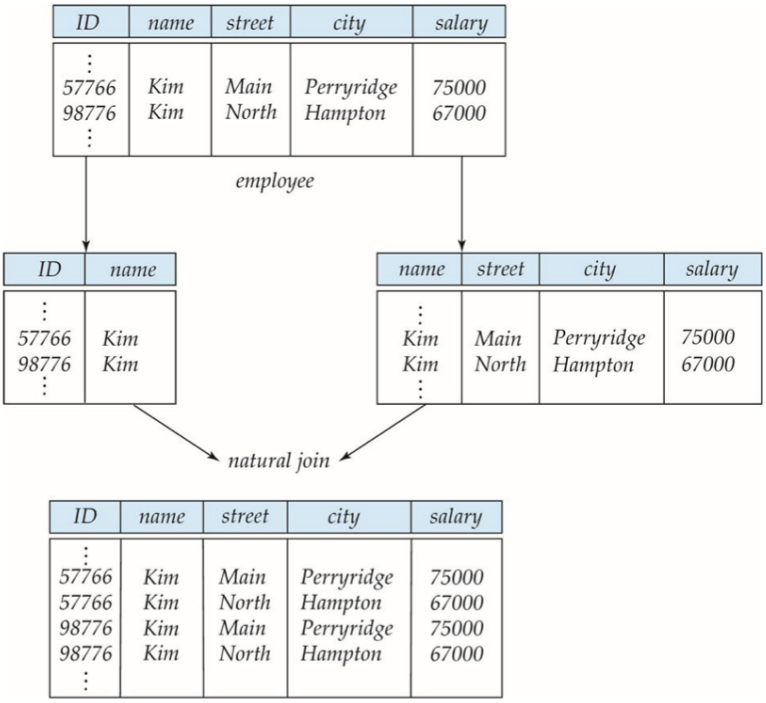
- Not all combined schemas result in repetition of information
- e.g., consider combining relations
  - `sec_class(sec_id, building, room_number)` and `section(course_id, sec_id, semester, year)` ⇒ `section(course_id, sec_id, semester, year, building, room_number)`
  - No repetition in this case

Decomposition

- The only way to avoid the repetition-of-information problem in the in_dep schema is to decompose it into two schemas: instructor and department schemas

- Not all decompositions are good. Suppose we decompose
  - `employee (ID, name, street, city, salary)` ⇒ `employee1 (ID, name), employee2 (name, street, city, salary)`
    - A problem arises when we have two employees with the same name
  - The next slide shows how we lose information
    - We cannot reconstruct the original employee relation; so, this is a lossy decomposition

A Lossy Decomposition



Lossless Decomposition

Decomposition of $R = (A, B, C)$: $R_1(A, B)$ $R_2(B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A (r) \bowtie \Pi_B (r)$

A	B	C
α	1	A
β	2	B

- Let R be a relation schema and let R1 and R2 form a decomposition of R, i.e., R = R1 U R2
- We say that the decomposition is a lossless decomposition if there is no loss of information by replacing R with the two relation schemas R1 U R2
- Formally, πR1 (r) ⋈ πR2 (r) = r
- And, conversely a decomposition is lossy if r ⊂ πR1 (r) ⋈ πR2 (r)

Normalization Theory

- Decide whether a particular relation R is in “good” form
- In the case that a relation R is not in “good” form, decompose it into a set of relations {R1, R2, ..., Rn} such that
  - Each relation is in good form

    - The decomposition is a lossless decomposition
- Our theory is based on:
    - Functional dependencies
    - Multivalued dependencies

Functional Dependencies

- There are usually a variety of constraints (rules) on the data in the real world
- For example, some of the constraints that are expected to hold in a university database are:
    - Students and instructors are uniquely identified by their ID
    - Each student and instructor has only one name
    - Each instructor and student is (primarily) associated with only one department
    - Each department has only one value for its budget and only one associated building
- An instance of a relation that satisfies all such real-world constraints is called a legal instance of the relation
- A legal instance of a database is one where all the relation instances are legal instances
- A functional dependency is a generalization of the notion of a key

Functional Dependencies Definition

- Let R be a relation schema $\alpha \subseteq R$ and $\beta \subseteq R$
- The functional dependency $\alpha \rightarrow \beta$ holds on R if and only if for any legal relations r(R), whenever any two tuples t1 and t2 of r agree on the attributes α, they also agree on the attributes β, i.e., $t1[\alpha] = t2[\alpha] \Rightarrow t1[\beta] = t2[\beta]$ (α, β may be multiple attributes)
- e.g., Consider r(A, B) with the following instance of r

1	4
1	5
3	7

    - On this instance, $B \rightarrow A$ hold; $A \rightarrow B$ does not hold

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F
    - If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by F is the closure of F
- We denote the closure of F by F+

Keys and Functional Dependencies

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if $K \rightarrow R$, and for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys (superkeys right side is fixed with R, but functional dependencies can be anything)
- Example
    - in_dep (ID, name, salary, dept_name, building, budget).
    - We expect these functional dependencies to hold:
        - dept_name$\rightarrow$budget
        - ID$\rightarrow$building

- but would not expect the following to hold:
  - dept_name⟶salary

Use of Functional Dependencies

- We use functional dependencies:
  - To test relations to see if they are legal under a given set of functional dependencies
    - If a relation r is legal under a set F of functional dependencies, we say that r satisfies F
  - To specify constraints on the set of legal relations
    - We say that F holds on R if all legal relations on R satisfy the set of functional dependencies F
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances
  - e.g., a specific instance of instructor may, by chance, satisfy name⟶ID

Trivial Functional Dependencies

- A functional dependency is trivial if it is satisfied by all instances of a relation e.g., ID, name⟶ID, name⟶ name
- In general, α⟶β is trivial if β ⊆ α

Lossless Decomposition

- We can use functional dependencies to show when certain decompositions are lossless
- For the case of R = (R1, R2), we require that for all possible relations r on schemaR,r=πR1 (r)⋈πR2 (r)
- A decomposition of R into R1 and R2 is lossless decomposition if at least one of the following dependencies is in F+:
  - R1∩R2⟶R1
  - R1∩R2⟶R2
- The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies
- Example
  - R = (A, B, C) , F = {A⟶B, B⟶C}
  - R1 = (A, B), R2 = (B, C)
    - Lossless decomposition: R1 ∩ R2 = {B} and B⟶BC
  - R1 = (A, B), R2 = (A, C)
    - Lossless decomposition: R1 ∩ R2 = {A} and A⟶AB
  - Note: B⟶BC is a shorthand notation for B⟶{B, C}

Dependency Preservation

- Testing functional dependency constraints each time the database is updated can be costly
- It is useful to design the database in a way that constraints can be tested efficiently
- If testing a functional dependency can be done by considering just one relation, then the cost of testing this constraint is low
- When decomposing a relation, it may be no longer possible to do the testing without having to perform a Cartesian product or join
- A decomposition that makes it computationally hard to enforce functional dependency is said to be not dependency preserving
- Example

- Consider a schema: dept_advisor(s_ID, i_ID, dept_name)
- With function dependencies:
  - i_ID ⟶ dept_name
  - s_ID, dept_name ⟶ i_ID
- In the above design, we are forced to repeat the department name once for each time an instructor participates in a dept_advisor relationship
- To fix this, we need to decompose dept_advisor
- Any decomposition will not include all the attributes in
  - s_ID, dept_name ⟶ i_ID
- Thus, the decomposition will not be dependency preserving

Normal Forms

Boyce-Codd Normal Form

- A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F+ of the form α⟶β where α ⊆ R and β ⊆ R, at least one of the following holds:
  - α⟶β is trivial (i.e., β ⊆ α)
  - α is a superkey for R
- An example schema that is not in BCNF:
  - in_dep (**ID**, name, salary, **dept_name**, building, budget) because
  - dept_name ⟶ building, budget holds on in_dep but
  - dept_name is not a superkey
- When decomposing in_dept into instructor and department
  - instructor is in BCNF
  - department is in BCNF

Decomposing a Schema into BCNF

- Let R be a schema R that is not in BCNF; let α⟶β be the FD that causes a violation of BCNF
- We decompose R into:
  - (α∪β)
  - (R – (β – α))
- In our example of in_dep,
  - α = dept_name
  - β = building, budget
- and in_dep is replaced by
  - (α ∪ β) = (dept_name, building, budget)
  - (R – (β – α)) = (ID, name, dept_name, salary)
- Examples
  - R = (A, B, C) F = {A⟶B, B⟶C}
  - R1 = (A, B), R2 = (B, C)
    - Lossless-join decomposition: R1 ∩ R2 = {B} and B⟶BC
      - Always given using this method
    - Dependency preserving
      - Not always given in this method
  - R1 = (A, B), R2 = (A, C)

- Lossless-join decomposition: R1 ∩ R2 = {A} and A⟶AB
- Not dependency preserving (cannot check B⟶C without computing R1 ⋈ R2)

BCNF and Dependency Preservation

- It is not always possible to achieve both BCNF and dependency preservation
- Consider a schema: dept_advisor(s_ID, i_ID, dept_name)
- With function dependencies:
  - i_ID ⟶ dept_name
  - s_ID, dept_name ⟶ i_ID
- dept_advisor is not in BCNF
  - i_ID is not a superkey
- Following our rule for BCNF decomposition, we get:
  - (s_ID, i_ID)
  - (i_ID, dept_name)
- Any decomposition of dept_advisor will not include all the attributes in
  - s_ID, dept_name ⟶ i_ID
- Thus, the decomposition is not dependency preserving

Third Normal Form

- A relation schema R is in third normal form (3NF) if for all: α⟶β in F+ where α ⊆ R and β ⊆ R, at least one of the following holds:
  - α⟶β is trivial (i.e., β ⊆ α)
  - α is a superkey for R
  - Each attribute A in β – α is contained in a candidate key for R (NOTE: each attribute may be in a different candidate key)
- If a relation is in BCNF, it is in 3NF (since in BCNF one of the first two conditions above must hold)
- The third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later)
- Example
  - Consider a schema: dept_advisor(s_ID, i_ID, dept_name)
  - With function dependencies:
    - i_ID ⟶ dept_name
    - s_ID, dept_name ⟶ i_ID
  - Two candidate keys: {s_ID, dept_name}, {s_ID, i_ID}
  - We have seen before that dept_advisor is not in BCNF
  - R, however, is in 3NF
    - s_ID, dept_name is a superkey
    - i_ID ⟶ dept_name and i_ID is not a superkey, but:
      - {dept_name} – {i_ID} = {dept_name} and dept_name is contained in a candidate key

Redundancy in 3NF

- Consider the schema R below, which is in 3NF
  - R = (J, L, K)
  - F = {JK⟶L, L⟶K}

- An instance table:

  dept_advisor(*s_ID*, *i_ID*, *dept_name*)

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
null	l_2	k_2

- What is wrong with the table?
  - Repetition of information
  - Null values (e.g., to represent the relationship l2, k2, where there is no corresponding value for J)

Comparison of BCNF and 3NF

- Advantages to 3NF over BCNF
  - It is always possible to obtain a 3NF design without sacrificing losslessness or dependency preservation
- Disadvantages to 3NF
  - We may have to use null values to represent some of the possible meaningful relationships among data items
  - There is the problem of repetition of information

Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies
- Decide whether a relation scheme R is in “good” form
- In the case that a relation scheme R is not in “good” form, need to decompose it into a set of relation scheme {R1, R2, ..., Rn} such that:
  - Each relation scheme is in good form
  - The decomposition is a lossless decomposition
  - Preferably, the decomposition should be dependency preserving

How Good Is BCNF?

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a relation inst_info (ID, child_name, phone)
  - An instructor may have more than one phone and can have multiple children
  - An instance of inst_info:

ID	*child_name*	*phone*
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	William	512-555-4321

  - There is no non-trivial functional dependency, and therefore the relation is in BCNF
  - Insertion anomalies: if we add a phone 981-992-3443 to 99999, we need to add two tuples
    - (99999, David, 981-992-3443)
    - (99999, William, 981-992-3443)

Higher Normal Forms

- It is better to decompose inst_info into:
  - inst_child:

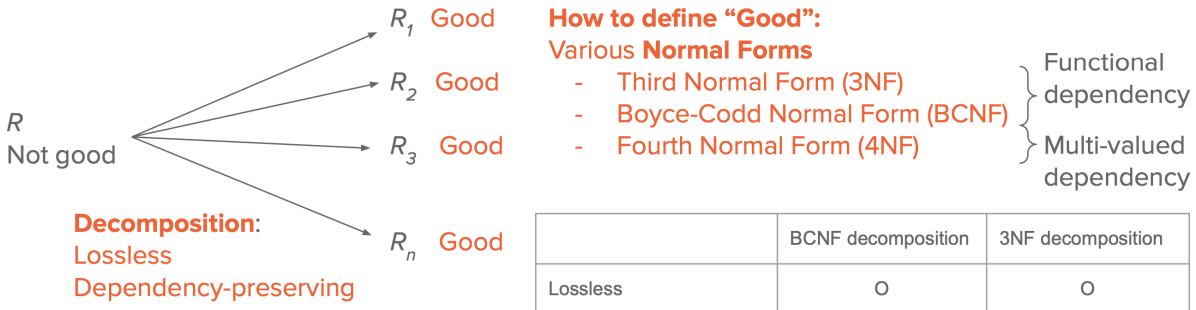
ID	*child_name*
99999	David
99999	William

  - inst_phone:

ID	*phone*
99999	512-555-1234
99999	512-555-4321

- This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later

Summary



	BCNF decomposition	3NF decomposition
Lossless	O	O
Dependency-preserving	△ (not always)	O

Functional-Dependency Theory

Functional-Dependency Theory Roadmap

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving

Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F
  - If A⟶B and B⟶C, then we can infer that A⟶C
  - etc.
- The set of all functional dependencies logically implied by F is the closure of F
- We denote the closure of F by F+
- We can compute F+, the closure of F, by repeatedly applying Armstrong's Axioms:
  - Reflexive rule: if $\beta \subseteq \alpha$, then $\alpha \longrightarrow \beta$
  - Augmentation rule: if $\alpha \longrightarrow \beta$, then $\gamma\alpha \longrightarrow \gamma\beta$
  - Transitivity rule: if $\alpha \longrightarrow \beta$ and $\beta \longrightarrow \gamma$, then $\alpha \longrightarrow \gamma$
- These rules are
  - Sound: generating only functional dependencies that actually hold
  - Complete: generating all functional dependencies that hold

Example of F+

- R = (A, B, C, G, H, I) F = {A→B, A→C, CG→H, CG→I, B→H}
- Some members of F+
  - A→H
    - by transitivity from A→B and B→H
  - AG→I
    - by augmenting A→C with G, to get AG→CG and then transitivity with CG→I
  - CG→HI
    - by augmenting CG→I to infer CG→CGI, and augmenting of CG→H to infer CGI→HI, and then transitivity
- Additional rules:
  - Union rule: If α→β holds and α→γ holds, then α→βγ holds
  - Decomposition rule: If α→βγ holds, then α→β holds and α→γ holds
  - Pseudotransitivity rule: If α→β holds and γβ→δ holds, then αγ→δ holds
- The above rules can be inferred from Armstrong's axioms
  - ***EXAM! 4.29 14min***
    - How to derive these rules
  - e.g., pseudotransitivity: αγ→γβ (augmentation), then γβ→δ (transitivity), we obtain αγ→δ

Procedure for Computing F+

```
F+ = F
repeat
        for each functional dependency f in F+
        apply reflexivity and augmentation rules on f add the resulting functional
dependencies to F+
        for each pair of functional dependencies f1 and f2 in F+
                if f1 and f2 can be combined using transitivity
                        then add the resulting functional dependency to F+
until F+ does not change any further
```

Closure of Attribute Sets

- Given a set of attributes α, we define the closure of α under F (denoted by α+) as the set of attributes that are functionally determined by α under F

```
result := α;
while (changes to result) do
        for each β→γ in F do
                begin
                        if β ⊆ result then result := result ∪ γ;
                end
```

- Example
  - R = (A, B, C, G, H, I) F = {A→B, A→C, CG→H, CG→I, B→H}
  - (AG)+
    - 1. result = AG
    - 2. result = ABCG (A→C and A→B)

        - 3. result = ABCGH (CG⟶H and CG ⊆ AGBC)
        - 4. result = ABCGHI (CG⟶I and CG ⊆ AGBCH)
    - Is AG a candidate key?
        - 1. Is AG a super key?
            - Does AG⟶R? == Is R = (AG)+ (i.e., if (AG)+ is the set of all attributes)
        - 2. Is any subset of AG a superkey?
            - Does A⟶R? == Is R = (A)+
            - Does G⟶R? == Is R = (G)+
            - In general: check for each subset of size n-1

Uses of Attribute Closure

- Testing for superkey
    - To test if α is a superkey, we compute α+ and check if α+ contains all attributes of R
- Testing functional dependencies
    - To check if a functional dependency α⟶β holds (or, in other words, is in F+), just check if β ⊆ α+
        - That is, we compute α+ by using attribute closure, and then check if it contains β, which is a simple and cheap test as well as very useful
- Computing closure of F
    - For each γ ⊆ R, we find the closure γ+, and for each S ⊆ γ+, we output a functional dependency γ⟶S

Canonical Cover

- Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; i.e., all the functional dependencies in F are satisfied in the new database state
- If an update violates any functional dependencies in the set F, the system must roll back the update
- We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set
- This simplified set is termed the canonical cover
- To define the canonical cover, we must first define extraneous attributes; an attribute of a functional dependency in F is extraneous if we can remove it without changing F+
    - smallest set G s.t. G+ = F+

Extraneous Attributes

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint
    - For example, if we have AB⟶C and remove B, we get the possibly stronger result A⟶C
    - It may be stronger because A⟶C logically implies AB⟶C, but AB⟶C does not, on its own, logically imply A⟶C
        - Due to AB⟶A (reflexivity) and A⟶C (transitivity), A⟶C logically implies AB⟶C
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from AB⟶C safely
    - For example, suppose that F = {AB⟶C, A⟶D, D⟶C}
    - Then, we can show that F logically implies A⟶C, making B extraneous in AB⟶C
        - Due to A⟶D and D⟶C (transitivity), A⟶C i.e., {AB⟶C, A⟶D, D⟶C} implies {A⟶C, A⟶D, D⟶C} while AB⟶C does not A⟶C

- Removing an attribute from the right side of a functional dependency could make it a weaker constraint
  - For example, if we have AB⟶CD and remove C, we get the possibly weaker result AB⟶D
  - It may be weaker because using just AB⟶D we can no longer infer AB⟶C
  - But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from AB⟶CD safely
  - For example, suppose that F = {AB⟶CD, A⟶C}
  - Then, we can show that even after replacing AB⟶CD by AB⟶D, we can still infer AB⟶C and thus AB⟶CD
    - Due to AB⟶A (reflexivity) and A⟶C (transitivity), AB⟶C; due to AB⟶D (union), AB⟶CD i.e., {AB⟶D, A⟶C} implies {AB⟶CD, A⟶C} while AB⟶D does not AB⟶CD
- An attribute of a functional dependency in F is extraneous if we can remove it without changing F+
- Consider a set F of functional dependencies and the functional dependency α⟶β in F
  - Remove from the left side: Attribute A is extraneous in α if
    - A ∈ α and stronger
    - F logically implies (F – {α⟶β}) ∪ {(α – A)⟶β}
  - Remove from the right side: Attribute A is extraneous in β if
    - A ∈ β and weaker
    - (F – {α⟶β}) ∪ {α⟶(β – A)} logically implies F
- Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one

Testing if an Attribute is Extraneous

- Let R be a relation schema and let F be a set of functional dependencies that hold on R; consider an attribute in the functional dependency α⟶β
- To test if attribute A ∈ β is extraneous in β
  - Consider the set: F' = (F – {α⟶β}) ∪ {α⟶(β – A)},
  - Check that α+ contains A; if it does, A is extraneous in β
- To test if attribute A ∈ α is extraneous in α
  - Let γ = α – {A}. Check if γ⟶β can be inferred from F
    - Compute γ+ using the dependencies in F
    - If γ+ includes all attributes in β, A is extraneous in α
- Example
  - Let F = {AB⟶CD, A⟶E, E⟶C}
  - To check if C is extraneous in AB⟶CD, we:
    - Compute the attribute closure of AB under F' = {AB⟶D, A⟶E, E⟶C}
    - The closure is ABCDE, which includes CD
    - This implies that C is extraneous

Canonical Cover

- A canonical cover for F is a set of dependencies Fc such that
  - F logically implies all dependencies in Fc , and
  - Fc logically implies all dependencies in F, and
  - No functional dependency in Fc contains an extraneous attribute, and
  - Each left side of functional dependency in Fc is unique, i.e., there are no two dependencies in Fc
    - α1⟶β1 and α2⟶β2 such that α1 = α2

- To compute a canonical cover for F:

```
  repeat
          Use the union rule to replace any dependencies in F of the form α1→β1 and
α1→β2 with α1→β1 β2
          Find a functional dependency α→β in Fc with an extraneous attribute either in
α or in β
          /* Note: test for extraneous attributes done using Fc, not F */
          If an extraneous attribute is found, delete it from α→β
  until (Fc not change)
```

- Note: The union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied
- Example
  - R = (A, B, C) F = {A→BC, B→C, A→B, AB→C}
  - Combine A→BC and A→B into A→BC
    - The set Fc is now {A→BC, B→C, AB→C}
  - A is extraneous in AB→C
    - Check if the result of deleting A from AB→C is implied by the other dependencies
      - Yes: in fact, B→C is already present! - The set Fc is now {A→BC, B→C}
  - C is extraneous in A→BC
    - Check if A→C is logically implied by A→B and the other dependencies
      - Yes: using transitivity on A→B and B→C
      - The attribute closure of A can be used in more complex cases
  - The canonical cover is: {A→B, B→C}

Dependency Preservation

- Let Fi be the set of dependencies F+ that include only attributes in Ri a decomposition is dependency preserving, if (F1 ∪ F2 ∪ ... ∪ Fn)+ = F+
- Using the above definition, testing for dependency preservation takes exponential time
- Not that if a decomposition is not dependency preserving then checking updates for violation of functional dependencies may require computing joins, which is expensive
- Let F be the set of dependencies on schema R and let R1, R2, ... , Rn be a decomposition of R
- The restriction of F to Ri is the set Fi of all functional dependencies in F+ that include only attributes of Ri
- Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation
- Note that the definition of restriction uses all dependencies in F+, not just those in F
- The set of restrictions F1, F2, ... , Fn is the set of functional dependencies that can be checked efficiently

Testing for Dependency Preservation

- To check if a dependency α→β is preserved in a decomposition of R into R1, R2, ... , Rn, we test the following (with the attribute closure with respect to F)

```
result = α
repeat
  for each Ri in the decomposition
```

```
            t = (result ∩ Ri)+ ∩ Ri
            result = result ∪ t
until (result does not change)
```

- If result contains all attributes in β, then the functional dependency α⟶β is preserved
- We apply the test on all dependencies in F to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute F+ and (F1 ∪ F2 ∪ ... ∪ Fn)+
- Example
  - R = (A, B, C) F = {A⟶B, B⟶C} Key = {A}
  - R is not in BCNF
  - Decomposition R1 = (A, B), R2 = (B, C)
    - R1 and R2 in BCNF
    - Lossless-join decomposition
    - Dependency preserving

Algorithm for Decomposition Using Functional Dependencies

Testing for BCNF

- To check if a non-trivial dependency α⟶β causes a violation of BCNF
  - 1. Compute α+ (the attribute closure of α), and
  - 2. Verify that it includes all attributes of R, that is, it is a superkey of R
- Simplified test: to check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F+
  - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F+ will cause a violation of BCNF either
- However, simplified test using only F is incorrect when testing a relation in a decomposition of R
  - Consider R = (A, B, C, D, E) with F = {A⟶B, BC⟶D}
    - Decompose R into R1 = (A, B) and R2 = (A, C, D, E)
    - Neither of the dependencies in F contain only attributes from (A, C, D, E) so we might be mislead into thinking R, satisfies BCNF
    - In fact, dependency AC⟶D in F+ shows R2 is not in BCNF

Testing Decomposition for BCNF

- To check if a relation Ri in a decomposition of R is in BCNF
- Either test Ri for BCNF with respect to the restriction of F+ to Ri (that is, all FDs in F+ that contain only attributes from Ri) or Use the original set of dependencies F that hold on R, with the following test:
  - For every set of attributes α ⊆ Ri, check that α+ (the attribute closure of α) either includes no attribute of Ri – α, or includes all attributes of Ri
  - If the condition is violated by some α, the dependency α⟶(α+ – α) ∩ Ri can be shown to be present in F+, and Ri violates BCNF
  - We use above dependency to decompose Ri

BCNF Decomposition Algorithm

```
result := { R }; done := false;
compute F+;
while (not done) do
        if (there is a schema Ri in result that is not in BCNF)
                then begin
                        let α→β be a nontrivial functional dependency that holds on Ri
such that α→Ri is not in F+, and α ∩ β = ∅;
                        result := (result − Ri) ∪ (Ri − β) ∪ (α, β);
                end
        else done := true;
```

- Note: each Ri is in BCNF, and decomposition is lossless-join

Example of BCNF Decomposition

- class (course_id, title, dept_name, credits, sec_id, semester, year, building, room_number, capacity, time_slot_id)
- Functional dependencies:
  - course_id→title, dept_name, credits
  - building, room_number→capacity
  - course_id, sec_id, semester, year→building, room_number, time_slot_id
- A candidate key: {course_id, sec_id, semester, year}
- BCNF decomposition:
  - course_id→title, dept_name, credits holds
    - but course_id is not a superkey
  - We replace class by:
    - course (course_id, title, dept_name, credits)
  - course is in BCNF
    - How do we know this?
      - (course_id)+ = {course_id, title, dept_name, credits}
      - (title)+ = {title}
      - (dept_name)+ = {dept_name}
      - (credits)+ = {credits}
      - For every set of attributes α ⊆ {course_id, title, dept_name, credits}, α+ either includes no attribute of {course_id, title, dept_name, credits} – α, or includes all attributes of {course_id, title, dept_name, credits} -> BCNF
  - building, room_number→capacity holds on class-1
    - but {building, room_number} is not a superkey for class-1
  - We replace class-1 by:
    - classroom (building, room_number, capacity)
    - section (course_id, sec_id, semester, year, building, room_number, time_slot_id)
    - classroom and section are in BCNF

Third Normal Form

- Efficient checking for FD violation on updates is important, but BCNF is not guaranteed to be dependency preserving

- Solution: a weaker normal form, called Third Normal Form (3NF)
  - Some redundancy is introduced (with resultant problems; we will see examples later)
  - Functional dependencies can be checked on individual relations without computing a join
  - There is always a lossless-join, dependency-preserving decomposition into 3NF
- Example
  - dept_advisor (s_ID, i_ID, dept_name) F = {s_ID, dept_name→i_ID, i_ID→dept_name}
  - Two candidate keys: {s_ID, dept_name} and {i_ID, s_ID}
  - dept_advisor is in 3NF
    - s_ID, dept_name→i_ID
      - s_ID, dept_name is a superkey
    - i_ID→dept_name
      - dept_name is contained in a candidate key

Testing for 3NF

- We need to check only FDs in F, not necessarily in F+
- For each dependency α→β, check if α is a superkey using its attribute closure
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
  - This test is rather more expensive, since it involves finding candidate keys
  - Testing for 3NF has been shown to be NP-complete (see the next slide)
    - Jiann H. Jou and Patrick C. Fischer, "The complexity of recognizing 3NF relation schemes," Information Processing Letters 14(4): 187--190, 1982 https:-www.sciencedirect.com/science/article/pii/0020019082900345
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time
- Proof: a quick summary from the paper (see the paper for details)
  - Any attribute that is a member of a key is called a prime attribute
  - To show that, for all dependencies α→β in F, either α is a superkey or β is prime
  - One way to do this is to first find all of the keys
  - Unfortunately the problem of finding all of the keys of a relation has been shown to be NP-complete; the problem of determining if a given attribute is prime is also NP-complete
- ***EXAM! 5.1 33min***
  - 3NF decomposition is available in polynomial time but testing 3NF is NP-complete

3NF Decomposition

```
Let Fc be a cononical cover for F;
i:=0;
for each functional dependency a->b in Fc do
        if none of the schemas Rj, 1<=j<=i contains ab
                then begin
                        i = i+1
                        Rj = ab
                end
if none of the schemas Rj, q<=j<=i contains a candidate jey for R
        then begin
                i = i+1
```

```
                Ri = any candidate key for R
        end
repeat
if any schema Rj is contained in another schema Rk
        then
                Rj = Ri
                i = i-1
return (R1,R2,...,Rj)
```

- The above algorithm ensures
  - Each relation schema Ri is in 3NF
  - The decomposition is dependency preserving and lossless-join
  - Proof of correctness is in Section 7.5.3 of the textbook
- Example
  - Relation schema:
    - cust_banker_branch = (customer_id, employee_id, branch_name, type)
  - The functional dependencies for this relation schema are:
    - customer_id, employee_id→branch_name, type
    - employee_id→branch_name
    - customer_id, branch_name→employee_id
  - We first compute a canonical cover
    - branch_name is extraneous in the r.h.s. of the 1st dependency
    - No other attribute is extraneous, so we get FC =
      customer_id, employee_id→type
      employee_id→branch_name
      customer_id, branch_name→employee_id
  - The for loop generates the following 3NF schema:
    - (customer_id, employee_id, type)
      (employee_id, branch_name)
      (customer_id, branch_name, employee_id)
    - Observe that (customer_id, employee_id, type) contains a candidate key of the original schema, so no further relation schema needs be added
  - At end of the for loop, detect and delete schemas, such as (employee_id, branch_name), which are subsets of other schemas
    - The result will not depend on the order in which FDs are considered
  - The resultant simplified 3NF schema is:
    - (customer_id, employee_id, type)
      (customer_id, branch_name, employee_id)

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
  - The decomposition is lossless
  - The dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
  - The decomposition is lossless
  - It may not be possible to preserve dependencies

Design Goals

- The first goal for a relational database design is:
  - BCNF
  - Lossless join
  - Dependency preservation
- If we cannot achieve this, we accept one of
  - Lack of dependency preservation
  - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys
  - FDs could be specified using assertions, but they are expensive to test (and currently not supported by any of the widely used databases)
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a FD whose left hand side is not a key

Multivalued Dependencies

Multivalued Dependencies (MVDs)

- Suppose we record names of children, and phone numbers for instructors:
  - inst_child (ID, child_name)
  - inst_phone (ID, phone_number)
- If we were to combine these schemas to get
  - inst_info (ID, child_name, phone_number)
  - Example data:
    (99999, David, 512-555-1234)
    (99999, David, 512-555-4321)
    (99999, William, 512-555-1234)
    (99999, William, 512-555-4321)
- This relation is in BCNF
  - Why?
- Let R be a relation schema and let α ⊆ R and β ⊆ R; The multivalued dependency α↠β holds on R if in any legal relation r(R), for all pairs for tuples t1 and t2 in r such that t1[α] = t2[α], there exist tuples t3 and t4 in r such that:
  - t1[α] = t2[α] = t3[α] = t4[α]
    t3[β] = t1[β]
    t3[R – β] = t1[R –β]
    t4[β] = t2[β]
    t4[R–β] = t2[R – β]
- MVD: Tabular Representation of α↠β

	α	β	$R-\alpha-\beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets Y, Z, W
- We say that Y↠Z (Y multidetermines Z) if and only if for all possible relations r(R)

    - < y1, z1, w1 > ∈ r and < y1, z2, w2 > ∈ r
      then
      < y1, z1, w2 > ∈ r and < y1, z2, w1 > ∈ r
- Note that since the behavior of Z and W are identical, it follows that Y↠Z if Y↠W
- Example
    - ID↠child_name
      ID↠phone_number
        - The above formal definition is supposed to formalize the notion that given a particular value of Y (ID) it has associated with it a set of values of Z (child_name) and a set of values of W (phone_number), and these two sets are in some sense independent of each other

Use of MVDs

- We use multivalued dependencies in two ways:
    - To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
    - To specify constraints on the set of legal relations; we shall concern ourselves only with relations that satisfy a given set of functional and multivalued dependencies
- If a relation r fails to satisfy a given multivalued dependency, we can construct a relations r' that satisfies the multivalued dependency by adding tuples to r

Theory MVDs

- From the definition of MVDs, we can derive the following rule:
    - If α⟶β, then α↠β
    - That is, every functional dependency is also a multivalued dependency
- Proof: ***EXAM! 5.1 65min***
    - Suppose α⟶β. t1 = (a, b, c) and t2 = (a, d, e) are given. By the functional dependency, b = d. Thus, t3 = t1 = (a, d, c) and t4 = t2 = (a, b, e) naturally exist without considering additional tuples.
    - Wikipedia: Every FD is an MVD because if α⟶β, then swapping β's between tuples that agree on α doesn't create new tuples.
- The closure D+ of D is the set of all functional and multivalued dependencies logically implied by D
    - We can compute D+ from D, using the formal definitions of functional dependencies and multivalued dependencies
        - We can manage with such reasoning for very simple multivalued dependencies, which seem to be most common in practice
    - For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules (Armstrong's axioms for MVD):
        - Complementation: If α↠β, then α↠R - β
        - Augmentation: If α↠β and γ⊆δ, then αδ↠βγ
        - Transitivity: If α↠β and β↠γ, then α↠γ - β
        - Replication: If α⟶β, then α↠β
        - Coalescence: If α↠β and ∃ δ s.t. δ∩β = ∅, δ⟶γ, and γ⊆β, then α⟶γ

Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D+ of the form α↠β, where α ⊆ R and β ⊆ R, at least one of the following hold:

- α↠β is trivial (i.e., β ⊆ α or α ∪ β = R)
- α is a superkey for the schema R
  - The definition of 4NF differs from that of BCNF only in the use of multivalued dependencies
- If a relation is in 4NF, it is in BCNF
  - Proof: Note that if a schema R is not in BCNF, there is a non-trivial functional dependency α⟶β holding on R, where α is not a superkey. Since α⟶β implies α↠β by the replication rule, R cannot be in 4NF.

4NF Decomposition Algorithm

```
result:={R}
done:=false
compute D+
Let Di denote the restriction of D+ to Rj
while(not done)
        if(there is a schema Ri in result that is not in 4NF) then
                begin
                        let a↠b be a nontrivial multivalued dpendency that holds on Ri
such that a⟶Ri is not in Di and anb!=ø
                        result:= (result-Ri) ∪ (Ri-b) ∪ (a,b)
                end
        else done:=true
```

Example

- R=(A,B,C,G,H,I)
  F = { A↠B, B↠HI, CG↠H }
- R is not in 4NF since A↠B and A is not a superkey for R
- Decomposition
  - a) R1 = (A, B) (R1 is in 4NF)
  - b) R2 = (A, C, G, H, I) (R2 is not in 4NF, decompose into R3 and R4)
  - c) R3 = (C, G, H) (R3 is in 4NF)
  - d) R4 = (A, C, G, I) (R4 is not in 4NF, decompose into R5 and R6)
    - A↠B and B↠HI ⇒ A↠HI (MVD transitivity) and
    - and hence A↠I (MVD restriction to R4)
  - e) R5 = (A, I) (R5 is in 4NF)
  - f) R6 = (A, C, G) (R6 is in 4NF)

Other Issues

First Normal Form (1NF)

,

not 1NF

Customer	Customer ID	Transactions		
Abraham	1	Transaction ID	Date	Amount
		12890	2003-10-14	–87
		12904	2003-10-15	–50
Isaac	2	Transaction ID	Date	Amount
		12898	2003-10-14	–21
Jacob	3	Transaction ID	Date	Amount
		12907	2003-10-15	–18
		14920	2003-11-20	–70
		15003	2003-11-27	–60

1NF

Customer	Customer ID
Abraham	1
Isaac	2
Jacob	3

Customer ID	Transaction ID	Date	Amount
1	12890	2003-10-14	–87
1	12904	2003-10-15	–50
2	12898	2003-10-14	–21
3	12907	2003-10-15	–18
3	14920	2003-11-20	–70
3	15003	2003-11-27	–60

- A domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in the first normal form if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and introduce repeating groups
- Atomicity is actually a property of how the elements of the domain are used
  - Example: Strings would normally be considered indivisible
  - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
    - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic; not recommended because encoding of information exists in an application program rather than in the database
- Typical relational databases satisfy the first normal form - Nested tables are not originally supported in relational databases

Second Normal Form (2NF)

not 2NF

Electric toothbrush models

Manufacturer	Model	Manufacturer country
Forte	X-Prime	Italy
Forte	Ultraclean	Italy
Dent-o-Fresh	EZbrush	USA
Brushmaster	SuperBrush	USA
Kobayashi	ST-60	Japan
Hoch	Toothmaster	Germany
Hoch	X-Prime	Germany

2NF

Electric toothbrush models

Manufacturer	Model
Forte	X-Prime
Forte	Ultraclean
Dent-o-Fresh	EZbrush
Brushmaster	SuperBrush
Kobayashi	ST-60
Hoch	Toothmaster
Hoch	X-Prime

Electric toothbrush manufacturers

Manufacturer	Manufacturer country
Forte	Italy
Dent-o-Fresh	USA
Brushmaster	USA
Kobayashi	Japan
Hoch	Germany

- A relation that is in the first normal form, and every non-primary-key attribute is fully functionally dependent on the primary key, then the relation is in the second normal form
- Candidate key: { Manufacturer, Model } FD: Manufacturer $\longrightarrow$ Manufacturer country

Overall Database Design Process

We have assumed a schema R is given:

- R could have been generated when converting an E-R diagram to a set of tables
- R could have been a single relation containing all attributes that are of interest (called a universal relation)
  - Normalization breaks R into smaller relations
- R could have been the result of some ad hoc design of relations, which we then test/convert to a normal form

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
  - Example: an employee entity with
    - Attributes department_name and building
    - Functional dependency department_name⟶building
  - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare; most relationships are binary

Denormalization for Performance

- One may want to use non-normalized schema for performance
- For example, displaying prerequisites along with course_id and title requires join of course with prereq
- Alternative 1: Use a denormalized relation containing the attributes of course as well as prereq with all above attributes
  - Faster lookup
  - Extra space and extra execution time for updates
  - Extra coding work for programmers and possibility of error in extra code
- Alternative 2: Use a materialized view defined as course ⋈ prereq
  - Same as above, except the third one

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided: Instead of earnings (company_id, year, amount), use
  - earnings_2023, earnings_2024, earnings_2025, etc., all on the schema (company_id, earnings)
    - In BCNF, but needs a new table for each year
  - company_year (company_id, earnings_2023, earnings_2024, earnings_2025)
    - Also in BCNF, but requires a new attribute for each year
    - Mainly used in spreadsheets

8. Complex Data Types

Semi-Structured Data

Semi-Structured Data

- Many applications require storage of complex data, whose schema changes often
- The relational model's requirement of atomic data types may be an overkill
  - e.g., storing set of interests as a set-valued attribute of a user profile may be simpler than normalizing it
- Data exchange can benefit greatly from semi-structured data
  - Exchange can be between applications, or between back-end and front-end of an application
  - Web-services are widely used today, with complex data fetched to the front-end and displayed using a mobile app or JavaScript
- JSON and XML are widely used semi-structured data models

Features of Semi-Structured Data Models

- Flexible schema
  - Wide column representation: each tuple is allowed to have a different set of attributes, and new attributes can be added at any time
  - Sparse column representation: schema has a fixed but large set of attributes, where each tuple may store only a subset of attributes
- Multivalued data types
  - Sets, multisets
    - e.g., set of interests {‘basketball’, ‘La Liga’, ‘cooking’, ‘anime’, ‘jazz’} - Key-value map (or just map for short)
    - A set of key-value pairs
    - e.g., {(brand, Apple), (ID, MacBook Air), (size, 13), (color, silver)}
    - Operations on maps: put(key, value), get(key), delete(key)
- Arrays
  - Widely used for scientific and monitoring applications
  - e.g., readings taken at regular intervals can be represented as array of values instead of (time,value) pairs
    - [5, 8, 9, 11] instead of {(1, 5), (2, 8), (3, 9), (4, 11)}
- Array database: a database that provides specialized support for arrays
  - e.g., compressed storage, query language extensions, etc.
  - Oracle GeoRaster, PostGIS, SciDB, etc.

Nested Data Types

- Hierarchical data is common in many applications
- JSON: JavaScript Object Notation
  - Being widely used today
- XML: Extensible Markup Language
  - Earlier generation notation, still used extensively

JSON

- Textual representation widely used for data exchange
  - Example of JSON data

```
{
        "ID": "22222",
        "name": {
                "firstname": "Albert",
                "lastname": "Einstein"
        },
        "deptname": "Physics",
        "children": [
                {"firstname": "Hans", "lastname": "Einstein" },
                {"firstname": "Eduard", "lastname": "Einstein" }
        ]
}
```

```
- Objects are key-value maps, i.e., sets of (attribute name, value) pairs
- Arrays are also key-value maps (from an offset to a value)
```

- JSON is ubiquitous in data exchange today
  - Especially being widely used for Web services
    - Most modern applications are architected around on Web services
- SQL extensions are available for
  - JSON types for storing JSON data
  - Extracting data from JSON objects using path expressions
    - e.g., v->ID, or v.ID
  - Generating JSON from relational data
    - e.g., json.build_object('ID', 12345, 'name', 'Einstein') - Creating JSON collections using aggregation
    - e.g., json_agg aggregate function in PostgreSQL # returns a JSON array
  - Syntax varies greatly across databases (see https:-www.sqlite.org/json1.html for SQLite3)
- JSON is verbose
  - Compressed representations such as BSON (Binary JSON) used for efficient data storage

XML

- XML uses tags to markup the text

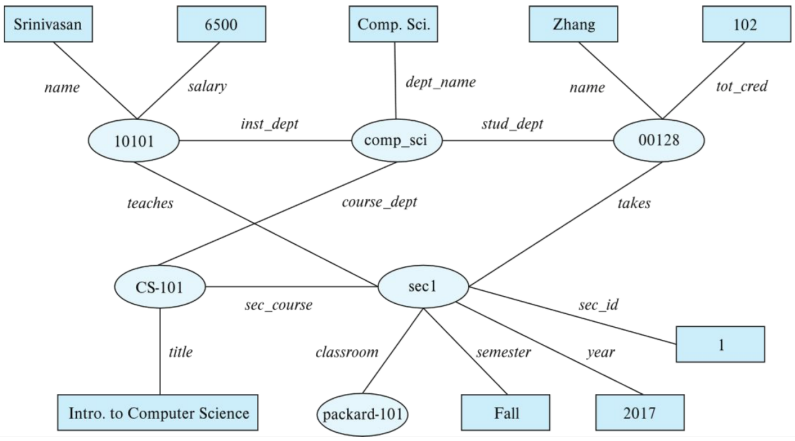
```
<course>
    <course id> CS-101 </course id>
    <title> Intro. to Computer Science </title>
    <dept name> Comp. Sci. </dept name>
    <credits> 4 </credits>
</course>
```

- Tags make the data self-documenting
- Tags can be hierarchical
- XQuery language developed to query nested XML structures - Not widely used currently
- SQL extensions to support XML
  - Storing XML data
  - Generating XML data from relational data
  - Extracting data from XML data types
    - Path expressions
- Example

```
<purchase order>
        <identifier> P-101 </identifier>
        <purchaser>
                <name> Cray Z. Coyote </name>
                <address> Route 66, Mesa Flats, Arizona 86047, USA</address>
        </purchaser>
        <supplier>
```

```
                <name> Acme Supplies </name>
                <address> 1 Broadway, New York, NY, USA </address>
        </supplier>
        <itemlist>
                <item>
                        <identifier> RS1 </identifier>
                        <description> Atom powered rocket sled </description>
                        <quantity> 2 </quantity>
                        <price> 199.95 </price>
                </item>
                <item>...</item>
        </itemlist>
        <total cost> 429.85 </total cost>
        ....
</purchase order>
```

Knowledge Representation



10101	instance-of	instructor .
10101	name	"Srinivasan" .
10101	salary	"6500" .
00128	instance-of	student .
00128	name	"Zhang" .
00128	tot_cred	"102" .
comp_sci	instance-of	department .
comp_sci	dept_name	"Comp. Sci." .
biology	instance-of	department .
CS-101	instance-of	course .
CS-101	title	"Intro. to Computer Science"
CS-101	course_dept	comp_sci .
sec1	instance-of	section .
sec1	sec_course	CS-101 .
sec1	sec_id	"1" .
sec1	semester	"Fall" .
sec1	year	"2017" .
sec1	classroom	packard-101 .
sec1	time_slot_id	"H" .
10101	inst_dept	comp_sci .
00128	stud_dept	comp_sci .
00128	takes	sec1 .
10101	teaches	sec1 .

- RDF: Resource Description Format
  - Simplified representation for facts, represented as triples (subject, predicate, object)
    - e.g., (NBA-2019, winner, Raptors)
      (Washington-DC, capital-of, USA)
      (Washington-DC, population, 6,200,000)
  - Modeling objects having attributes and relationships with other objects
    - Like the ER model, but with a flexible schema
    - (ID, attribute-name, value)
    - (ID1, relationship-name, ID2)
  - Having a natural graph representation
- RDF triples represent binary relationships
- How to represent n-ary relationships?
  - Approach 1 (from Section 6.9.4): creating an artificial entity and linking to each of the n entities
    - e.g., (Barack Obama, president-of, USA, 2008-2016) can be represented as
      (e1, person, Barack Obama), (e1, country, USA),
      (e1, president-from, 2008) (e1, president-till, 2016)

    - Approach 2: using quads instead of triples, with context entities
        - e.g., (Barack Obama, president-of, USA, c1)
          (c1, president-from, 2008) (c1, president-till, 2016)
- RDF is widely used as knowledge base representation
    - DBPedia, Yago, Freebase, WikiData, ...
- Linked Open Data project aims to connect different knowledge graphs to allow queries to span databases

Querying RDF: SPARQL

- Triple patterns
    - ?cid title "Intro. to Computer Science"
    - ?cid title "Intro. to Computer Science"
      ?sid course ?cid
- SPARQL queries
    - Select the names of the students who took the section of the course titled as "Intro. to Computer Science"
    - Aggregation, optional joins (similar to outer joins), subqueries, etc.
    - Transitive closure on paths
        - e.g., where { ?a :partOf+ ?b }

```
select ?name
where {
        ?cid title "Intro. to Computer Science" .
        ?sid course ?cid .
        ?id takes ?sid .
        ?id name ?name .
}
```

Object Orientation

Object Orientation

- Object-relational data model provides a richer type system, with complex data types and object orientation
- Applications are often written in object-oriented programming languages
    - The type system does not match the relational type system
    - Switching between an imperative language and SQL is troublesome
- Object-orientation can be incorporated into relational databases
    - Build an object-relational database, adding object-oriented features to a relational database
    - Automatically convert data between the programming language model and the relational model; data conversion specified by object-relational mapping
    - Build an object-oriented database that natively supports object-oriented data and direct access from the programming language

Object-Relational Database Systems

- User-defined types

```
create type Person
        (ID varchar(20) primary key,
        name varchar(20),
        address varchar(20)) ref from(ID); /* More on this later */
create table people of Person;
```

- Table types

```
create type interest as table (
        topic varchar(20),
        degree_of_interest int);
create table users (
        ID varchar(20),
        name varchar(20),
        interests interest);
```

- Array, multiset data types also supported by many databases
    - Syntax varies by database

Type and Table Inheritance

- Type inheritance

```
create type Student under Person
(degree varchar(20));
create type Teacher under Person
(salary integer);
```

- Table inheritance syntax in PostgreSQL and Oracle

```
create table students
        (degree varchar(20))
        inherits people;
create table teachers
        (salary integer)
        inherits people;
```

```
create table people of Person;
create table students of Student
        under people;
create table teachers of Teacher
        under people;
```

Reference Types

- Creating reference types

```
create type Person
        (ID varchar(20) primary key,
        name varchar(20),
        address varchar(20)) ref from(ID);
create table people of Person;
create type Department (
        dept_name varchar(20),
        head ref(Person) scope people);
create table departments of Department;
```

- `ref(Person)`: physical pointer to a tuple of the person type
  - join is not needed, we can directly get the name of head of department
- System generated references can be retrieved using subqueries
  - `(select ref(p) from people as p where ID = '12345')`
- Using references in path expressions

```
select head->name, head->address
from departments;
```

Object-Relational Mapping

- Object-relational mapping (ORM) systems allow
  - Specification of mapping between programming language objects and database tuples
  - Automatic creation of database tuples upon creation of objects
  - Automatic update/delete of database tuples when objects are update/deleted
  - Interface to retrieve objects satisfying specified conditions
    - Tuples in database are queried, and object are created from the tuples
- Details in Section 9.6.2
  - Hibernate ORM for Java
  - Django ORM for Python

Django's ORM

- Allowing developers to interact with the database using Python code rather than writing SQL statements
  - Model definition

```
from django.db import models
class Book(models.Model):
        title = models.CharField(max_length=200)
        author = models.ForeignKey('Author', on_delete=models.CASCADE)
        published_date = models.DateField()
        price = models.DecimalField(max_digits=6, decimal_places=2)
```

- Querying the database

```
from myapp.models import Book
books = Book.objects.all()
```

```
books = Book.objects.filter(price__lt=10) # all books that cost less than $10
```

Textual Data

Textual Data

- Information retrieval: querying of unstructured data
  - Simple models of keyword queries, given query keywords, retrieve documents containing all the keywords
  - More advanced models rank the documents according to their relevance
  - Today, keyword queries return many types of information as answers
    - e.g., a query “Son Heung-min” returns information about early life, club career, ...
- Relevance ranking
  - Essential since there are usually many documents matching keywords

Ranking using TF-IDF

- Term: keyword occurring in a document/query
- Term Frequency: TF(d, t), the relevance of a term t to a document d
  - A definition: $TF(d,t) = log(1 + \frac{n(d,t)}{n(d)})$ where
    - n(d,t) = number of occurrences of term t in document d
    - n(d) = number of terms in document d
- Inverse Document Frequency: IDF(t)
  - A definition:$IDF(t) = \frac{1}{n(t)}$, where n(t) is the number of documents containing term t
  - A definition: $r(d, Q) = \sum_{t \in Q} TF(d,t) \cdot IDF(t)$
  - Other definitions
    - Proximity of words is taken into account
    - Stop words are often ignored

Ranking Using Hyperlinks

- PageRank
  - A measure of popularity/importance based on hyperlinks to pages, developed by Google
    - Hyperlinks provide very useful clues to the importance of a web page
- Rationale (see the next slide)
  - Pages hyperlinked from many pages should have a higher PageRank
  - Pages hyperlinked from the pages with a higher PageRank should have a higher PageRank
    - Formalized by the random walk model
- Formulation
  - Let T[i, j] be the probability that a random walker who is on page i will click on the link to page j
    - Assuming all links are equal $T[i,j] = 1/N_i$ where N_i = number of links out of page i
  - Then, PageRank[j] for each page j can be defined as
    - $P[j] = \frac{\delta}{N} + (1 - \delta) \sum_{i=1}^{N} (T[i,j] \cdot P[i])$
      - N = total number of pages
      - δ = probability to directly access a page(typing in url) a constant usually set to 0.15
- The definition of PageRank is circular, but can be solved as a set of linear equations
  - A simple iterative technique works well

    - Initialize all $P[j] = \frac{1}{N}$
    - In each iteration, apply $P[j] = \frac{\delta}{N} + (1 - \delta) \sum_{i=1}^{N} (T[i, j] \cdot P[i])$ to update P
    - Stop iteration when changes are small, or a certain limit (e.g., 30 iterations) is reached
- Other measures of relevance are also important
    - Keywords in the anchor text
    - Number of times users click on a link if it is returned as an answer

Retrieval Effectiveness

$$Precision = \frac{TP}{TP+FP}$$
$$Recall = \frac{TP}{TP+FN}$$

		Predicted condition	
	Total population = P + N	Predicted Positive (PP)	Predicted Negative (PN)
Actual condition	Positive (P)[a]	True positive (TP), hit[b]	False negative (FN), type II error, miss, underestimation[c]
	Negative (N)[d]	False positive (FP), type I error, false alarm, overestimation[e]	True negative (TN), correct rejection[f]

- Measures of effectiveness
    - Precision: what percentage of returned results are actually relevant
    - Recall: what percentage of relevant results are returned
        - At some number of answers, e.g., precision@10, recall@10

Spatial Data

Spatial Data

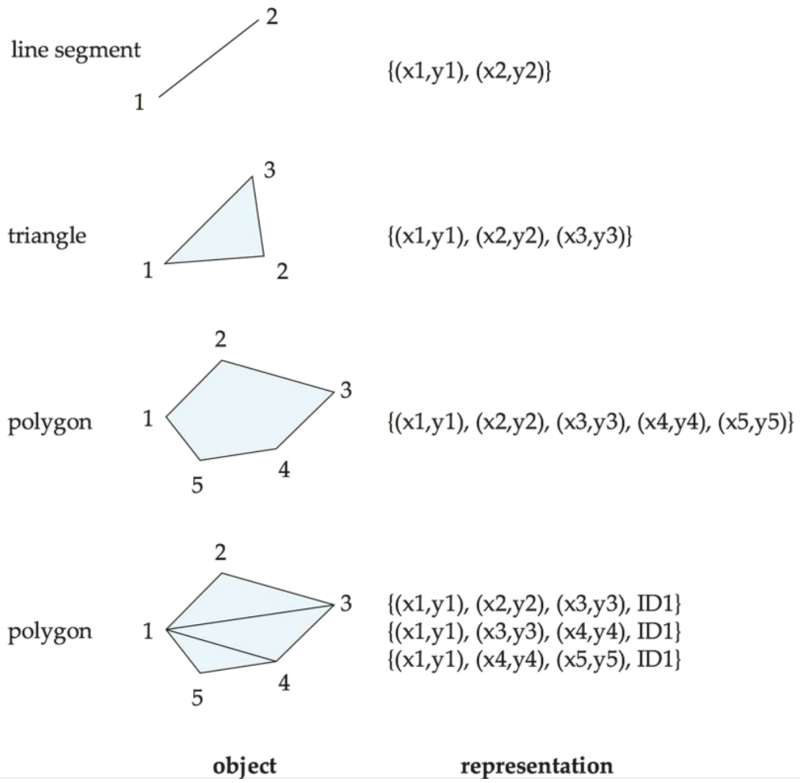
- Spatial databases store information related to spatial locations, and support efficient storage, indexing, and querying of spatial data
    - Geographic data: road maps, land-usage maps, topographic elevation maps, political maps showing boundaries, land-ownership maps, and so on
        - Geographic information systems (GISs) are special-purpose databases tailored for storing geographic data
        - A round-earth coordinate system may be used, e.g., (latitude, longitude, elevation)
    - Geometric data: design information about how objects are constructed, e.g., designs of buildings, aircraft, layouts of integrated-circuits
        - 2 or 3 dimensional Euclidean space with (X, Y, Z) coordinates

Geometric Information

- Various geometric constructs can be represented in a database in a normalized fashion (see the next slide)
- A line segment can be represented by the coordinates of its endpoints
- A polyline or linestring consists of a connected sequence of line segments and can be represented by a list containing the coordinates of the endpoints of the segments, in sequence
    - In order to approximate a curve, it is divided into a series of segments
        - Useful for two-dimensional features such as roads
- A polygon is represented by a list of vertices in order
    - The list of vertices specifies the boundary of a polygonal region

    - A polygon can also be represented as a set of triangles (triangulation)
- Representation of points and line segments in 3-D is similar to 2-D, except that points have an extra z component
- Polyhedra are represented by tetrahedrons, which is analogous to how triangulating polygons
- Geometry and geography data types are supported by many databases, e.g., SQL Server and PostGIS
    - point, linestring, curve, polygons
        - LINESTRING(1 1, 2 3, 4 4)
        - POLYGON((1 1, 2 3, 4 4, 1 1))
    - Collections: multipoint, multilinestring, multicurve, multipolygon
    - Type conversions: ST GeometryFromText() and ST GeographyFromText()
    - Operations: ST Union(), ST Intersection(), ...

Representation of Geometric Constructs



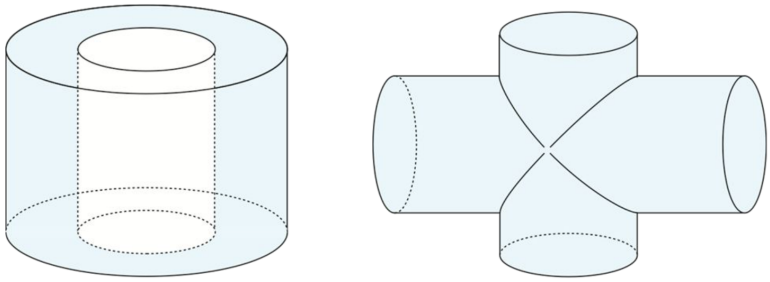
Design Databases

- Design components are represented as objects (generally geometric objects); the connections between the objects indicate how the design is structured
    - Simple two-dimensional objects: points, lines, triangles, rectangles, polygons
    - Complex two-dimensional objects: construction from simple objects via union, intersection, and difference operations
    - Complex three-dimensional objects: construction from simpler objects such as spheres, cylinders, and cuboids, by union, intersection, and difference operations
- Wireframe models represent three-dimensional surfaces as a set of simpler objects
    - e.g., see the right

Representation of Geometric Constructs

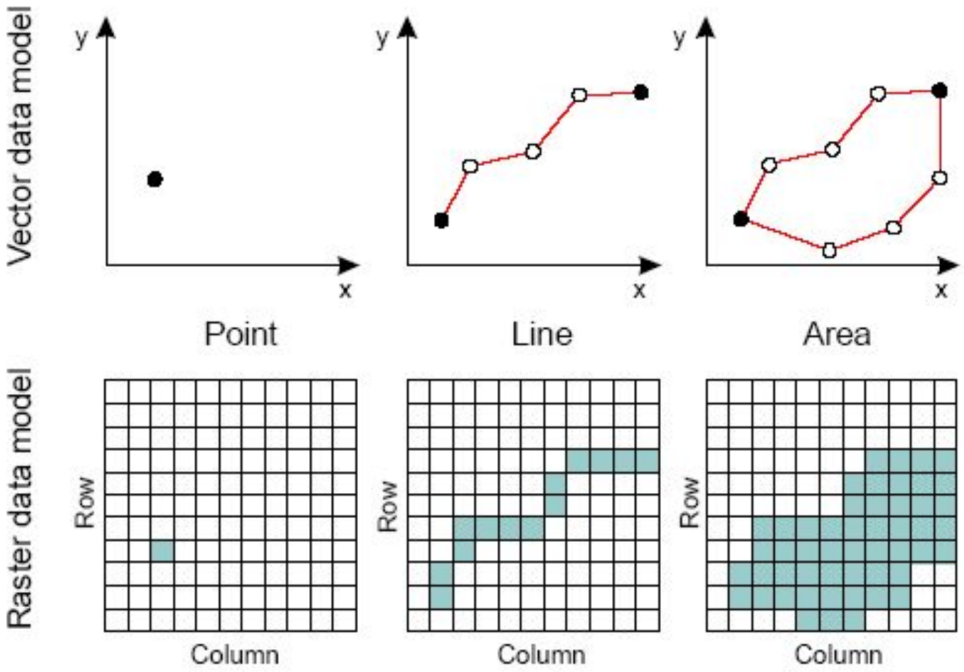
- Design databases also store non-spatial information about objects (e.g., construction material, color, etc.)
- Spatial integrity constraints are important

- e.g., pipes should not intersect, wires should not be too close to each other, etc.

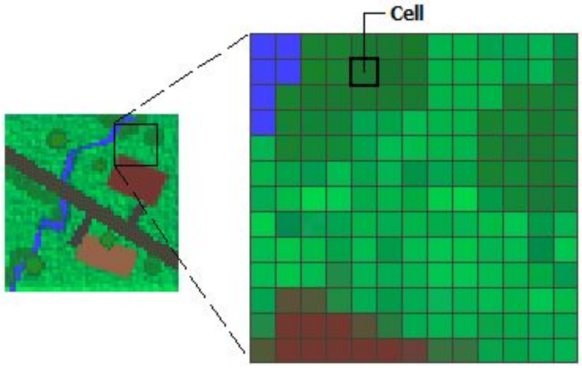


(a) Difference of cylinders (b) Union of cylinders

Geographic Data



- Raster data consist of bitmaps or pixel maps, in two or more dimensions
  - An example 2-D raster image is a satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area
    - Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time



- Design databases generally do not store raster data
- Vector data are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions
  - The vector format is often used to represent map data
    - Roads can be considered as two-dimensional and represented by lines and curves
    - Some features, such as rivers, may be represented either as complex curves or as complex polygons, depending on whether their width is relevant
    - Features such as regions and lakes can be depicted as polygons

Spatial Queries

- Region queries deal with spatial regions, e.g., ask for objects that lie partially or fully inside a specified region
  - e.g., PostGIS ST_Contains(), ST_Overlaps(), ...
- Nearness queries request objects that lie near a specified location
- Nearest neighbor queries, given a point or an object, find the nearest object that satisfies given conditions
- Spatial graph queries request information based on spatial graphs
  - e.g., shortest path between two points via a road network
- Spatial join of two spatial relations with the location playing the role of join attribute
- Queries that compute intersections or unions of regions

Vector Database

Why We Need Vector Stores for LLM Apps

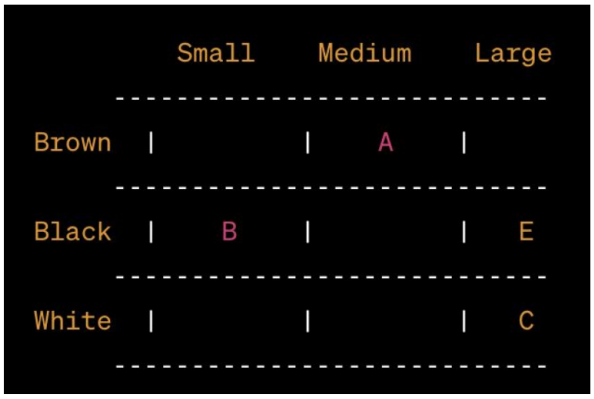
- Training a LLM is expensive. However new information such as news comes out every day.
- Fine tuning also needs a lot of computation
- But relatively easy to insert a vector data

Vector Database Fundraising

- Vector databases are a hot topic right now
- Companies keep raising money to develop their vector databases or to add vector search capabilities to their existing SQL or NoSQL databases

Understanding Vector Databases

- These databases are designed to handle data where each entry is represented as a vector in a multi-dimensional space
  - The vectors can represent a wide range of information, such as numerical features, embeddings from text or images, and even complex data like molecular structures - e.g.,



  - Image A: Brown color, Medium size
  - Image B: Black color, Small size
  - Image C: White color, Large size
  - Image E: Black color, Large size

How Vector Databases Store Data?

- Vector databases store data by using vector embeddings
- Vector embeddings in vector databases refer to a way of representing objects
    - Each object is assigned a vector that captures various characteristics or features of that object(size, color in the previous example)
    - These vectors are designed in such a way that similar objects have vectors that are closer to each other in the vector space, while dissimilar objects have vectors that are farther apart
    - For example, in a music streaming app, songs could be represented as vectors using embeddings that capture musical features like tempo, genre, and instruments used

1. First, we use the embedding model to create vector embeddings for the content we want to index
2. The vector embedding is inserted into the vector database, with some reference to the original content the embedding was created from
3. When the application issues a query, we use the same embedding model to create embeddings for the query and use those embeddings to query the database for similar vector embeddings

Which Vectors are Similar?

- e.g., Best cricket player in the world
- The vector representation of the search query is compared to the vector representations of all the player profiles in the database using cosine similarity. The more similar the vectors are, the higher the cosine similarity score.

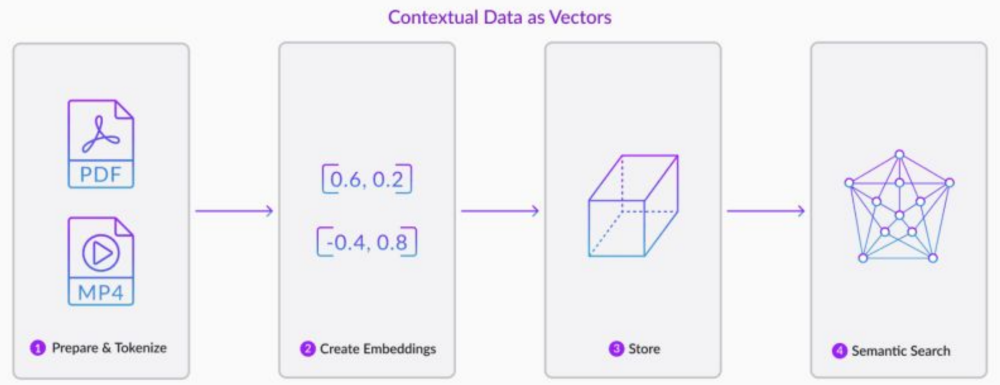
Vector Database Capabilities

- Efficient similarity search
    - Vector databases excel at performing similarity searches, where you can retrieve vectors that are most similar to a given query vector
- High-dimensional data
    - Vector databases are designed to handle high-dimensional data more efficiently, making them suitable for applications like natural language processing, computer vision, and genomics
- Machine learning and AI
    - Vector databases are often used to store embeddings generated by machine learning models
    - These embeddings capture the essential features of the data and can be used for various tasks, such as clustering, classification, and anomaly detection
- Real-time applications
    - Many vector databases are optimized for real-time or near-real-time querying, making them suitable for applications that require quick responses
- Personalization and user profiling
- Spatial and geographic data
- Healthcare and life sciences
    - In genomics and molecular biology, vector databases are used to store and analyze genetic sequences, protein structures, and other molecular data
- Data fusion and integration
- Multilingual search
- Graph data

Crucial Role of Vector Databases

- As industries increasingly adopt technologies like machine learning, artificial intelligence, and data analytics, the need to efficiently store, search, and analyze complex data representations has become paramount
- Vector databases enable businesses to harness the power of similarity search, personalized recommendations, and content retrieval, driving enhanced user experiences and improved decision-making
- Ranging from e-commerce and content platforms to healthcare and autonomous vehicles, the demand for vector databases stems from their ability to handle diverse data types and deliver accurate results in real time
- The scalability, speed, and accuracy offered by vector databases position them as a critical tool for extracting meaningful insights

SingleStore as a Vector Database



- Robust vector database capabilities of SingleStoreDB, tailored to seamlessly serve AI-driven applications, chatbots, image recognition systems, and more

Example of Face Matching with SQL

- Each row represents one image of a celebrity, and contains a unique ID number, the file name where the image is stored, and a 128-element floating point vector representing the meaning of the face
  - This vector was obtained using facenet, a pre-trained neural network for creating vector embeddings from a face image

```
create table people(
        id bigint not null primary key,
        filename varchar(255),
        vector blob
);
```

- The 1st query gets a query vector @v for Emma_Thompson_0001.jpg

```
select vector
into @v
from people
where filename = "Emma_Thompson/Emma_Thompson_0001.jpg";
```

- The 2nd query finds the top five closest matches (see the next slide)

```
select filename, dot_product(vector, @v) as score
from people where score > 0.1
order by score desc
limit 5;
```

- Emma_Thompson_0001.jpg is a perfect match for itself, so the score is close to 1; but interestingly, the next closest match is Emma_Thompson_0002.jpg

filename	score
Emma_Thompson/Emma_Thompson_0001.jpg	0.9999999403953552
Emma_Thompson/Emma_Thompson_0002.jpg	0.6130160093307495
Maria_Soledad_Alvear_Valenzuela/Maria_Soledad_Alvear_Valenzuela_0003.jpg	0.6115712523460388
Emma_Thompson/Emma_Thompson_0003.jpg	0.583335280418396
Benazir_Bhutto/Benazir_Bhutto_0005.jpg	0.5819507837295532

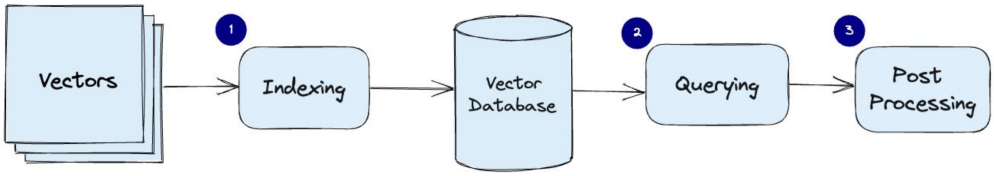
query image (Emma_Thompson_0001.jpg)

top match, score 0.613 (Emma_Thompson_0002.jpg)

Approximate Nearest Neighbor (ANN) Search

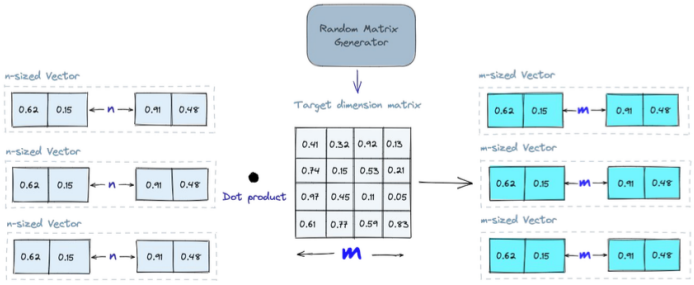
- In vector databases, we apply a similarity metric to find a vector that is the most similar to our query
- A vector database uses a combination of different algorithms that all participate in Approximate Nearest Neighbor (ANN) search
  - These algorithms optimize the search through hashing, quantization, or graph-based search
  - These algorithms are assembled into a pipeline that provides fast and accurate retrieval of the neighbors of a queried vector
    - Since the vector database provides approximate results, the main trade-offs we consider are between accuracy and speed

Pipeline



1. Indexing: The vector database indexes vectors using an algorithm such as PQ, LSH, or HNSW (more on these later)
2. Querying: The vector database compares the indexed query vector to the indexed vectors in the dataset to find the nearest neighbors (applying a similarity metric used by that index)
3. Post Processing: Sometimes, the vector database retrieves the final nearest neighbors from the dataset and post-processes them to return the final results

Indexing | Random Projection



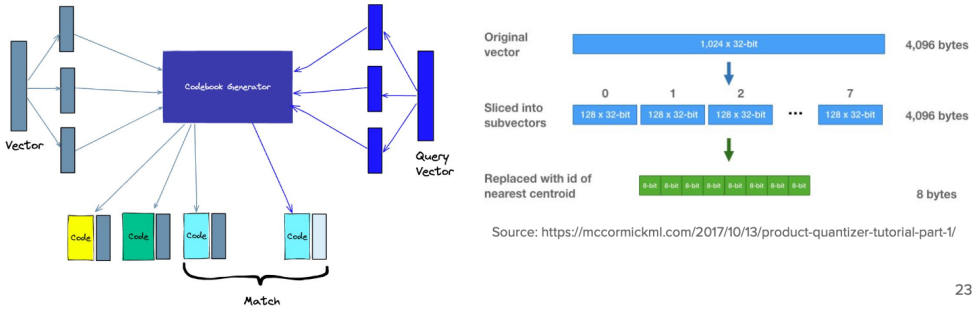
Since the dimensionality of the data is reduced, the search process is significantly faster than searching the entire high-dimensional space

2

- The basic idea behind random projection is to project the high-dimensional vectors to a lower-dimensional space using a random projection matrix

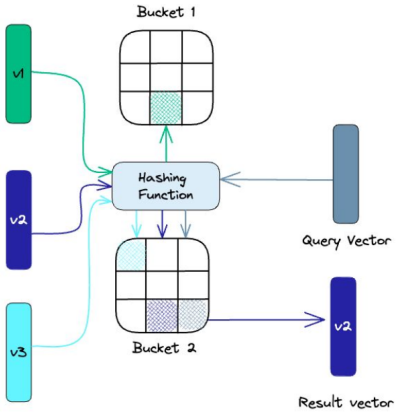
- We then calculate the dot product of the input vectors and the matrix, which results in a projected matrix of a lower-dimensionality
- When we query, we use the same projection matrix to project the query vector onto the lower-dimensional space

Indexing | Product Quantization

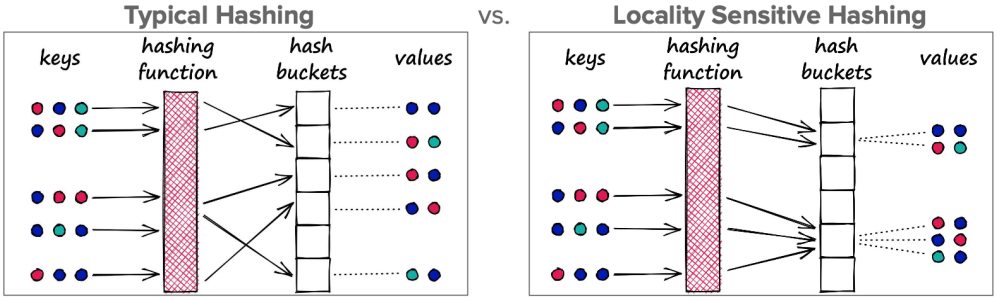


- Product quantization (PQ) takes the original vector, breaks it up into smaller chunks, simplifies the representation of each chunk by creating a representative “code” for each chunk, and then puts all the chunks back together—without losing information that is vital for similarity operations
- The process of PQ can be broken down into four steps: splitting, training, encoding, and querying.
    1. Splitting: The vectors are broken into segments
    2. Training: The algorithm generates a pool of potential “codes” that could be assigned to a vector
    3. Encoding: The algorithm assigns a specific code to each segment
    4. Querying: When we query, the algorithm breaks down the vectors into sub-vectors and quantizes them using the same codebook; then, it uses the indexed codes to find the nearest vectors to the query vector

Indexing | Locality-Sensitive Hashing



- Locality-Sensitive Hashing (LSH) maps similar vectors into “buckets” using a set of hashing functions
    - To find the nearest neighbors for a given query vector, we use the same hashing functions used to “bucket” similar vectors into hash tables
    - The query vector is hashed to a particular table and then compared with the other vectors in that same table to find the closest matches
- This method is much faster than searching through the entire dataset because there are far fewer vectors in each hash table



A typical hash function aims to place different values (no matter how similar) into separate buckets

An LSH function aims to place similar values into the same buckets

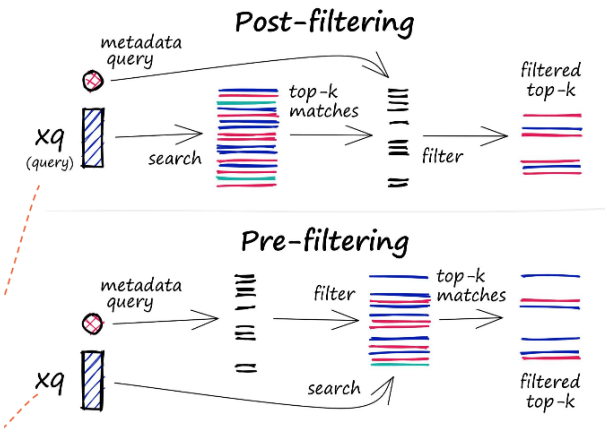
Indexing | Hierarchical Navigable Small World (HNSW)

- HNSW creates a hierarchical, tree-like structure where each node of the tree represents a set of vectors
- The edges between the nodes represent the similarity between the vectors
- The algorithm starts by creating a set of nodes, each with a small number of vectors
- This could be done randomly or by clustering the vectors with algorithms like k-means, where each cluster becomes a node
- The algorithm then examines the vectors of each node and draws an edge between that node and the nodes that have the most similar vectors to the one it has
- When we query an HNSW index, it uses this graph to navigate through the tree, visiting the nodes that are most likely to contain the closest vectors to the query vector\

Querying | Similarity Measures

- Cosine similarity: measures the cosine of the angle between two vectors in a vector space
- It ranges from -1 to 1, where 1 represents identical vectors, 0 represents orthogonal vectors, and -1 represents vectors that are diametrically opposed
- Euclidean distance: measures the straight-line distance between two vectors in a vector space
- It ranges from 0 to infinity, where 0 represents identical vectors, and larger values represent increasingly dissimilar vectors
- Dot product: measures the product of the magnitudes of two vectors and the cosine of the angle between them
- It ranges from -∞ to ∞, where a positive value represents vectors that point in the same direction, 0 represents orthogonal vectors, and a negative value represents vectors that point in opposite directions

Post Processing | Filtering

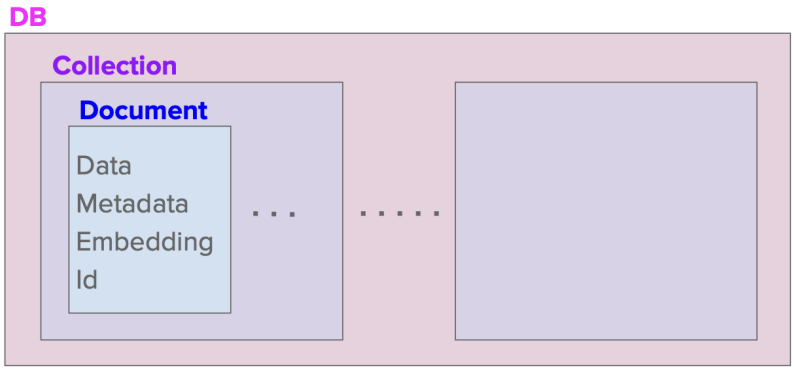


- In addition to the ability to query for similar vectors, vector databases can also filter the results based on a metadata query
- To do this, the vector database usually maintains two indexes: a vector index and a metadata index

- It then performs the metadata filtering either before or after the vector search

Chroma

Components



Four Core Commands

```
# python can also run in-memory with no server running: chromadb.PersistentClient()

import chromadb
client = chromadb.HttpClient()
collection = client.create_collection("sample_collection")

# Add docs to the collection. Can also update and delete. Row-based API coming soon!
collection.add(
        documents=["This is document1", "This is document2"],
        metadatas=[{"source": "notion"}, {"source": "google-docs"}],
        ids=["doc1", "doc2"], # must be unique for each doc
)

results = collection.query(
        query_texts=["This is a query document"],
        n_results=2,
        # where={"metadata_field": "is_equal_to_this"}, # optional filter
        # where_document={"$contains":"search_string"} # optional filter
)
```

EphemeralClient

- `def EphemeralClient(settings: Settings = Settings()) -> API`
- Creates an in-memory instance of Chroma. This is useful for testing and development, but not recommended for production use.

PersistentClient

- `def PersistentClient(path: str = "./chroma", settings: Settings = Settings()) -> API`
- Creates a persistent instance of Chroma that saves to disk. This is useful for testing and development, but not recommended for production use.
- Arguments:
    - path - The directory to save Chroma's data to. Defaults to "./chroma".

Client

- `def Client(settings: Settings = __settings) -> API`
- Return a running chroma.API instance. Defaults to EphemeralClient.

client.create_collection

```
def create_collection(
        name: str,
        metadata: Optional[CollectionMetadata] = None,
        embedding_function: Optional[EmbeddingFunction] = ef.DefaultEmbeddingFunction(),
        get_or_create: bool = False) -> Collection
```

- Create a new collection with the given name and metadata.
- Arguments:
  - name - The name of the collection to create.
  - metadata - Optional metadata to associate with the collection.
  - embedding_function - Optional function to use to embed documents. Uses the default embedding function if not provided.
  - get_or_create - If True, return the existing collection if it exists.
- Returns:
  - Collection - The newly created collection.
- Raises:
  - ValueError - If the collection already exists and get_or_create is False.
  - ValueError - If the collection name is invalid.
- Examples:

```
client.create_collection("my_collection")
# collection(name="my_collection", metadata={})
client.create_collection("my_collection", metadata={"foo": "bar"})
# collection(name="my_collection", metadata={"foo": "bar"})
```

collection.add

```
def add(ids: OneOrMany[ID],
embeddings: Optional[OneOrMany[Embedding]] = None,
metadatas: Optional[OneOrMany[Metadata]] = None,
documents: Optional[OneOrMany[Document]] = None) -> None
```

- Add embeddings to the data store.
- Arguments:
  - ids - The ids of the embeddings you wish to add
  - embeddings - The embeddings to add. If None, embeddings will be computed based on the documents using the
  - embedding_function set for the Collection. Optional.
  - metadatas - The metadata to associate with the embeddings. When querying, you can filter on this metadata. Optional.

    - documents - The documents to associate with the embeddings. Optional.
- Returns: None
- Raises:
    - ValueError: If you don't provide either embeddings or documents
    - ValueError: If the length of ids, embeddings, metadatas, or documents don't match
    - ValueError: If you don't provide an embedding function and don't provide embeddings
    - DuplicateIDError: If you provide an id that already exists

collection.query

```
def query(
      query_embeddings: Optional[OneOrMany[Embedding]] = None,
      query_texts: Optional[OneOrMany[Document]] = None,
      n_results: int = 10,
      where: Optional[Where] = None,
      where_document: Optional[WhereDocument] = None,
      include: Include = ["metadatas", "documents", "distances"]) -> QueryResult
```

- Get the n_results nearest neighbor embeddings for provided query_embeddings or query_texts.
- Arguments:
    - query_embeddings: The embeddings to get the closest neighbors of. Optional.
    - query_texts: The document texts to get the closest neighbors of. Optional.
    - n_results: The number of neighbors to return for each query_embedding or query_texts. Optional.
    - where - A Where type dict used to filter results by. E.g.
    - where_document - A WhereDocument type dict used to filter by the documents. E.g. Optional.
    - include - A list of what to include in the results. Optional.
- Returns:
    - QueryResult - A QueryResult object containing the results.
- Raises:
    - ValueError: If you don't provide either query_embeddings or query_texts
    - ValueError: If you provide both query_embeddings and query_texts

Embeddings

- Embeddings are the A.I-native way to represent any kind of data, making them the perfect fit for working with all kinds of A.I-powered tools and algorithms
    - They can represent text, images, and soon audio and video
- There are many options for creating embeddings, whether locally using an installed library, or by calling an API
- Chroma provides lightweight wrappers around popular embedding providers, making it easy to use them in your apps
    - You can set an embedding function when you create a Chroma collection, which will be used automatically, or you can call them directly yourself

Sentence Transformers

Default: all-MiniLM-L6-v2

By default, Chroma uses the Sentence Transformers all-MiniLM-L6-v2 model to create embeddings. This

embedding model can create sentence and document embeddings that can be used for a wide variety of tasks. This embedding function runs locally on your machine, and may require you download the model files (this will happen automatically).

```
from chromadb.utils import embedding_functions
default_ef = embedding_functions.DefaultEmbeddingFunction()
```

Sentence Transformers
Chroma can also use any Sentence Transformers model to create embeddings.

```
sentence_transformer_ef =
embedding_functions.SentenceTransformerEmbeddingFunction(model_name="all-MiniLM-L6-v2")
```

You can pass in an optional `model_name` argument, which lets you choose which Sentence Transformers model to use. By default, Chroma uses `all-MiniLM-L6-v2`. You can see a list of all available models here.

Pre-Trained Models for Embedding

Model Name	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
all-mpnet-base-v2	69.57	57.02	63.30	2800	420 MB
multi-qa-mpnet-base-dot-v1	66.76	57.60	62.18	2800	420 MB
all-distilroberta-v1	68.73	50.94	59.84	4000	290 MB
all-MiniLM-L12-v2	68.70	50.82	59.76	7500	120 MB
multi-qa-distilbert-cos-v1	65.98	52.83	59.41	4000	250 MB
all-MiniLM-L6-v2	68.06	49.54	58.80	14200	80 MB
multi-qa-MiniLM-L6-cos-v1	64.33	51.83	58.08	14200	80 MB
paraphrase-multilingual-mpnet-base-v2	65.83	41.68	53.75	2500	970 MB
paraphrase-albert-small-v2	64.46	40.04	52.25	5000	43 MB
paraphrase-multilingual-MiniLM-L12-v2	64.25	39.19	51.72	7500	420 MB
paraphrase-MiniLM-L3-v2	62.29	39.19	50.74	19000	61 MB
distiluse-base-multilingual-cased-v1	61.30	29.87	45.59	4000	480 MB
distiluse-base-multilingual-cased-v2	60.18	27.35	43.77	4000	480 MB

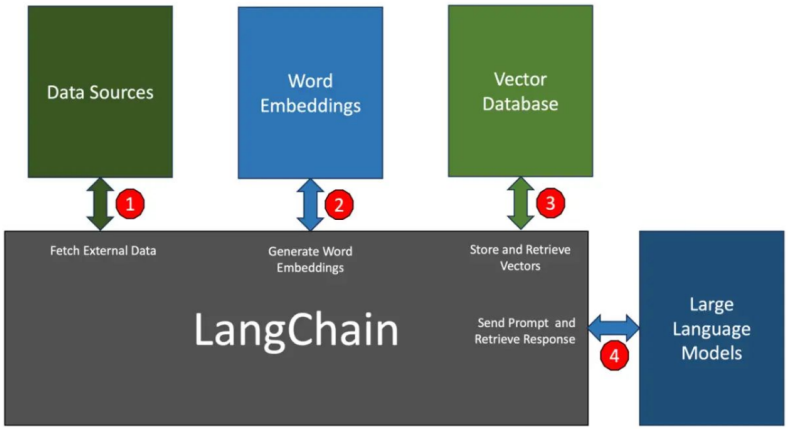
Similarity Function
`create_collection` also takes an optional `metadata` argument which can be used to customize the distance method of the embedding space by setting the value of `hnsw:space`.

```
collection = client.create_collection(
        name="collection_name",
        metadata={"hnsw:space": "cosine"} # l2 is the default
)
```

Valid options for hnsw:space are "l2", "ip, "or "cosine". The default is "l2" which is the squared L2 norm.

Distance	Equation
Squared L2	$\sum(A_i - B_i)^2$
Inner product	$1 - \sum(A_i \times B_i)$
Cosine similarity	$1 - \frac{\sum(A_i \times B_i)}{\sqrt{(\sum A_i^2 \sum B_i^2}}$

LangChain



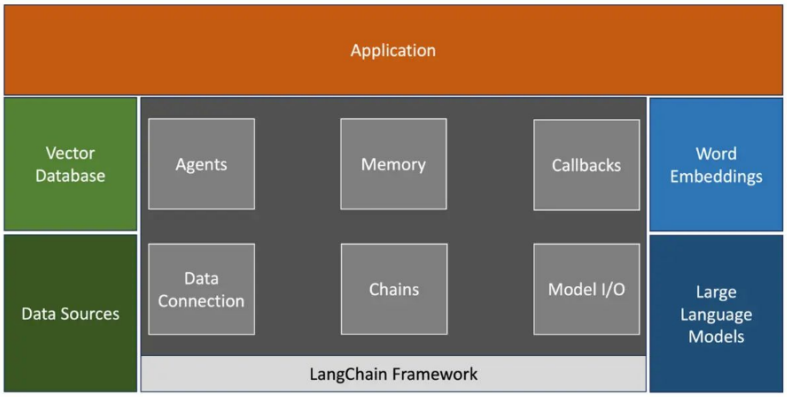
Introduction

- A framework for developing applications powered by language models
- LangChain is an framework (SDK) designed to simplify the integration of LLMs and applications
  - LangChain is similar to an ODBC or JDBC driver, which abstracts the underlying database by letting you focus on standard SQL statements
  - LangChain abstracts the implementation details of the underlying LLMs by exposing a simple and unified API
    - This API makes it easy for developers to swap in and swap out models without significant changes to the code
- LangChain is a powerful framework that integrates with external tools to form an ecosystem

Overall Flow

1. Data sources
   - Applications need to retrieve data from external sources such as PDFs, web pages, CSVs, and relational databases to build the context for the LLM
   - LangChain seamlessly integrates with modules that can access and retrieve data from disparate sources
2. Word embeddings
   - The data retrieved from some of the external sources must be converted into vectors. This is done by passing the text to a word embedding model associated with the LLM
     - For example, OpenAI's GPT-3.5 model has an associated word embeddings model that needs to be used to send the context
     - LangChain picks the best embedding model based on the chosen LLM, removing the guesswork in pairing the models
3. Vector databases
   - The generated embeddings are stored in a vector database to perform a similarity search
   - LangChain makes it easy to store and retrieve vectors from various sources ranging from in-memory arrays to hosted vector databases such as Chroma and Pinecone
4. Large language models
   - LangChain supports mainstream LLMs offered by OpenAI, Cohere, and AI21 and open source LLMs available on Hugging Face
     - The list of supported models and API endpoints is rapidly growing

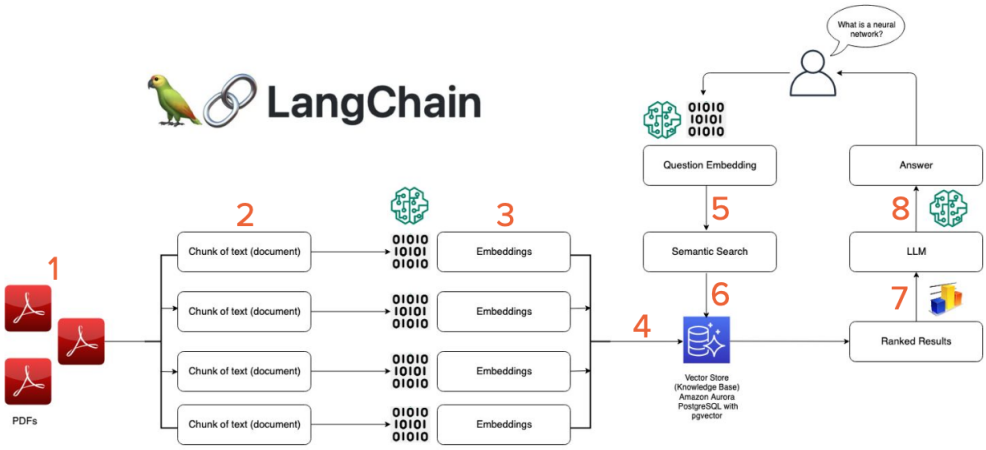
LangChain Modules



- Model I/O
  - The Model I/O module deals with the interaction with the LLM
  - It essentially helps in creating effective prompts, invoking the model API, and parsing the output; prompt engineering, which is the core of generative AI, is handled well by LangChain
  - It abstracts the authentication, API parameters, and endpoint exposed by LLM providers
  - Finally, it can parse the response sent by the model in the desired format that the application can consume
- Data connection
  - Think of the data connection module as the ETL pipeline of your LLM application
  - It deals with loading external documents such as PDF or Excel files, converting them into chunks for processing them into word embeddings in batches, storing the embeddings in a vector database, and finally retrieving them through queries
- Chains
  - Chains in LangChain are designed to build efficient pipelines that leverage the building blocks and LLMs to get an expected response
  - A simple chain may have a prompt and an LLM, but it's also possible to build highly complex chains that invoke the LLM multiple times, like recursion, to achieve an outcome
  - For example, a chain may include a prompt to summarize a document and then perform a sentiment analysis on the same
- Memory
  - LLMs are stateless but need context to respond accurately
  - The memory module makes it easy to add both short-term and long-term memory to models
  - Short-term memory maintains the history of a conversation through a simple mechanism; message history can be persisted to external sources such as Redis, representing long-term memory
- Callbacks
  - LangChain provides developers with a callback system that allows them to hook into the various stages of an LLM application
  - This is useful for logging, monitoring, streaming, and other tasks
  - It is possible to write custom callback handlers that are invoked when a specific event takes place within the pipeline
- Agents
  - Agents is by far the most powerful module of LangChain
  - LLMs are capable of reasoning and acting, called the ReAct prompting technique; the agents simplify crafting ReAct prompts that use the LLM to distill the prompt into a plan of action
  - The basic idea behind agents is to use an LLM to select a set of actions

- A sequence of actions is hard-coded in chains (in code), and LLM is used as a reasoning engine in agents to determine which actions to take and in what order

Step-by-Step Guide



1. Load the document
2. Splitting the document into chunks
3. Use embedding against the chunks and convert to vectors
4. Save the data to a vector database
5. Take a question from the user and get the embedding
6. Connect to the vector database and do a semantic search
7. Retrieve relevant responses based on user queries and send them to an LLM (e.g., ChatGPT)
8. Get an answer from the LLM and send it back to the user

There are researches to improve every step -> our assignment is simple example of one of the step

RAG

What is Retrieval-Augmented Generation(RAG)?
RAG is a technique for augmenting LLM knowledge with additional data.
LLMs can reason about wide-ranging topics, but their knowledge is limited to the public data up to a specific point in time that they were trained on. If you want to build AI applications that can reason about private data or data introduced after a model's cutoff date, you need to augment the knowledge of the model with the specific information it needs. The process of bringing the appropriate information and inserting it into the model prompt is known as Retrieval Augmented Generation (RAG).

RAG Architecture
A typical RAG application has two main components:

- Indexing: a pipeline for ingesting data from a source and indexing it
  - This usually happens offline
- Retrieval and generation: the actual RAG chain, which takes the user query at run time and retrieves the relevant data from the index, then passes that to the model
- Depending on the literature, retrieval and generation are written as separate components

Indexing Component

- Load: First we need to load our data.
  - This is done with DocumentLoaders.

- Split: Text splitters break large Documents into smaller chunks.
    - This is useful both for indexing data and for passing it in to a model, since large chunks are harder to search over and won't fit in a model's finite context window.
- Store: We need somewhere to store and index our splits, so that they can later be searched over.
    - This is often done using a VectorStore and Embeddings model.

Retrieval and generation

- Retrieve: Given a user input, relevant splits are retrieved from storage using a Retriever.
- Generate: A ChatModel / LLM produces an answer using a prompt that includes the question and the retrieved data

10. Big Data

Motivation

Motivation

- Very large volumes of data being collected
    - Driven by the growth of the Web, social media, and more recently Internet-of-Things
    - e.g., initial sources of big data: Web logs
        - Analytics on Web logs has great value for advertisements, Website structuring, what posts to show to a user, etc.
- Big Data: differentiated from data handled by earlier generation databases
    - Volume: much larger amounts of data stored
    - Velocity: much higher rates of insertions
    - Variety: many types of data, beyond relational data

Querying Big Data

- Transaction processing systems that need very high scalability
    - Many applications are willing to sacrifice ACID properties and other database features, if they can get very high scalability
        - ACID: Atomicity, Consistency, Isolation, Durability
- Query processing systems that need very high scalability and support for non-relational data

Big Data Storage Systems

- Distributed file systems
- Key-value storage systems
- Parallel and distributed databases

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides a single file-system view
- It should be highly scalable for large data-intensive applications
- e.g., 10K nodes, 100 million files, 10 PB
- It provides abundant storage of massive amounts of data on cheap and unreliable computers
    - Files are replicated to handle hardware failure

    - Failures are detected, and data is recovered from them
    - e.g., Google File System (GFS), Hadoop File System (HDFS)

Hadoop File System Architecture

- The entire cluster can been seen as a single name space
- Files are broken up into blocks
    - A block is typically a size of 64 MB
    - Each block is replicated on multiple DataNodes
- A client access a file by:
    1. Finding the locations of blocks from NameNode
    2. Accessing the data directly from DataNode

Hadoop Distributed File System (HDFS)

- NameNode
    - Mapping a filename to a list of block identifiers
    - Mapping each block identifier to DataNodes containing a replica of the block
- DataNode
    - Mapping a block identifier to a physical location on disk
- Data coherency
    - Write-once-read-many access model
    - Only append to existing files
- Good for large files, but not for a large number of small files

Key-Value Storage Systems

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are partitioned across multiple machines and queries are routed by the system to appropriate machine
- Records are also replicated across multiple machines, to ensure availability even if a machine fails
    - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are consistent
- Key-value stores may store
    - Uninterpreted bytes, with an associated key
        - e.g., Amazon S3, Amazon Dynamo
    - Wide-table (can have arbitrarily many attribute names) with an associated key
        - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
        - Allows some operations (e.g., filtering) to execute on storage node
    - JSON
        - MongoDB, CouchDB (document model)
- Document stores store semi-structured data, typically JSON
- Some key-value stores support multiple versions of data, with timestamps/version numbers
- Key-value stores support
    - put(key, value): stores values with an associated key
    - get(key): retrieves the stored value associated with the specified key
    - delete(key): removes the key and its associated value

- Some systems also support range queries on key values
- Document stores also support queries on non-key attributes
  - See books for MongoDB queries
- Key-value stores are not full database systems
  - Transactional updates are not supported or limited
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems ⇒ also called NoSQL systems

Data Representation

- An example of a JSON object is:

```
{
        "ID": "22222",
        "name": {
                "firstname: "Albert",
                "lastname: "Einstein"
        },
        "deptname": "Physics",
        "children": [
                { "firstname": "Hans", "lastname": "Einstein" },
                { "firstname": "Eduard", "lastname": "Einstein" }
        ]
}
```

Parallel and Distributed Databases

- Parallel databases were developed in 1980s, well before Big Data
- Parallel databases run on multiple machines (cluster)
  - Early generation was designed for a smaller scale (10s to 100s of machines)
- Replication is used to ensure data availability despite machine failures
  - However, query execution is typically restarted in event of a failure
    - Restarts may be frequent at a very large scale, e.g., on thousands of machines
    - MapReduce systems (coming up next) can continue query execution, working around failures
- JGL: Commercial products include Google Cloud Spanner
  - Website: https:-cloud.google.com/spanner?hl=en
  - Paper: https:-research.google/pubs/pub39966/

Replication and Consistency

- Availability (a system can run even if parts have failed) is essential for parallel/distributed databases
  - It is realized via replication; so even if a node has failed, another copy is available
- Consistency is important for replicated data
  - All live replicas have same value, and each read sees the latest version
  - It is often implemented using majority protocols
    - e.g., have 3 replicas, reads/writes must access 2 replicas
- Network partitions

- A network can break into ≥ 2 parts, each with active systems that can't talk to other parts
- In presence of partitions, both availability and consistency cannot be guaranteed
  - Brewer's CAP "Theorem" (see https:-en.wikipedia.org/wiki/CAP_theorem)

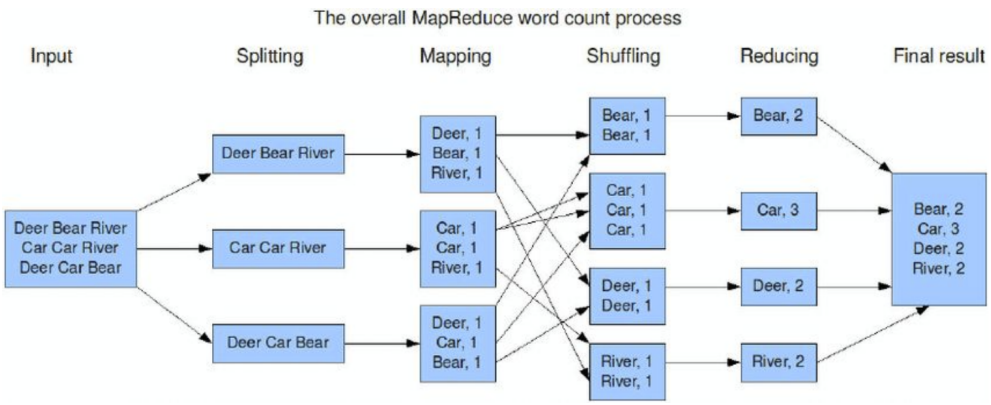
Replication and Consistency

- Explanation of CAP Theorem [Wikipedia]
  No distributed system is safe from network failures, thus network partitioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. When choosing consistency over availability, the system will return an error or a time out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.
  In the absence of a partition, both availability and consistency can be satisfied.
- Very large systems will partition at some point
  - Choose one of consistency or availability
- Traditional databases choose consistency
- Most Web applications choose availability
  - Except for specific parts such as order processing

MapReduce and Hadoop

The MapReduce Paradigm

- Providing a platform for reliable, scalable parallel computing
- Abstracting the issues of a distributed and parallel environment from programmers
  - A programmer provides only the core logic (via map() and reduce() functions)
  - A system takes care of parallelization of computation, coordination, etc.
- Typically using distributed file systems or key-value stores
- Proposed by Google and being available as Hadoop
  - Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150 (cited by around 25K times, as of November 2023)



MapReduce Example: Word Counting

- Let's consider the problem of counting the number of occurrences of each word in a large collection of documents
- How would you do it in parallel?
- Solution:

    - Divide documents among workers
    - Each worker parses document to find all words, a map function outputs (word, count) pairs
    - Partition (word, count) pairs across workers based on the word
    - For each word at a worker, a reduce function locally adds up the counts
- Given input: “One a penny, two a penny, hot cross buns.”
    - Records output by the map() function would be
        - (“One”, 1), (“a”, 1), (“penny”, 1), (“two”, 1), (“a”, 1), (“penny”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1)
    - Records output by reduce function would be
        - (“One”, 1), (“a”, 2), (“penny”, 2), (“two”, 1), (“hot”, 1), (“cross”, 1), (“buns”, 1)

Pseudocode of Word Counting

```
map(String record):
        for each word in record
                emit(word, 1);
```

- The first attribute of emit() above is called the reduce key.
- In effect, a group by is performed on the reduce key to create a list of values (all 1’s in above code).
- This requires the shuffle step across machines.
- The reduce function is called on list of values in each group.

```
reduce(String key, List value_list):
        String word = key;
        int count = 0;
        for each value in value_list:
                count = count + value;
        output(word, count);
```

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp
- Input: a set of key/value pairs
- Two user-supplied functions:
- map(k1, v1) $\longrightarrow$ list(k2, v2)
    - (k2, v2) is an intermediate key/value pair
    - k1 is the line number in a log file in our log processing example
- reduce(k2 list(v2)) $\longrightarrow$ list(k3, v3)
    - types need to match btw k1, k2, k3 and v1, v2, v3
- Output: a set of key/value (k3, v3) pairs

MapReduce Example: Log Processing

- Given a log file in the following format:
    - 2013/02/21 10:31:22.00EST /slide-dir/11.ppt
      2013/02/21 10:43:12.00EST /slide-dir/12.ppt
      2013/02/22 18:26:45.00EST /slide-dir/13.ppt

2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...

- Goal: find how many times each of the files in the slide-dir directory was accessed between 2013/01/01 and 2013/01/31
- Options:
  - Sequential program: too slow on massive datasets ⇒ parallel program
  - Database loading: expensive ⇒ direct operation on log files
  - Custom built parallel program: very laborious
  - MapReduce paradigm

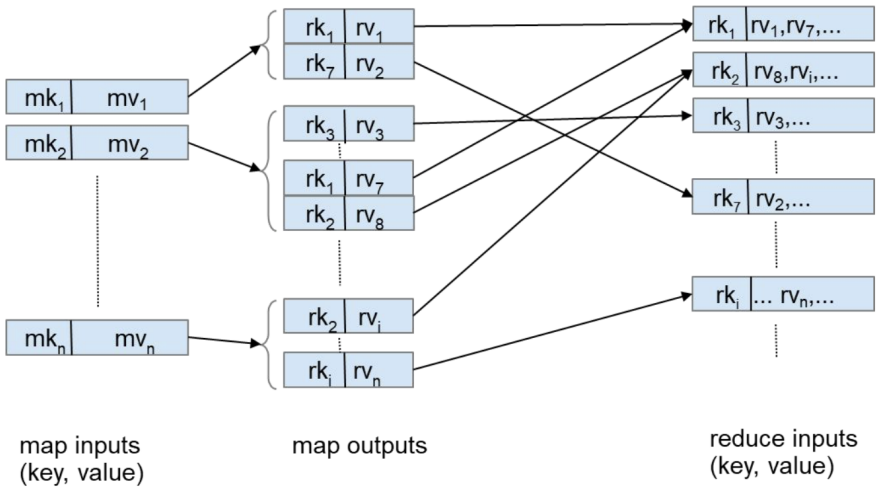
Pseudocode of Log Processing

```
map(String key, String record) {
        String attribute[3];
        .... break up a record into tokens (based on a space character), and store the
tokens in array attributes String date = attribute[0];
        String time = attribute[1];
        String filename = attribute[2];
        if date between 2013/01/01 and 2013/01/31 and filename starts with "/slide-dir/"
                emit(filename, 1);
}
```

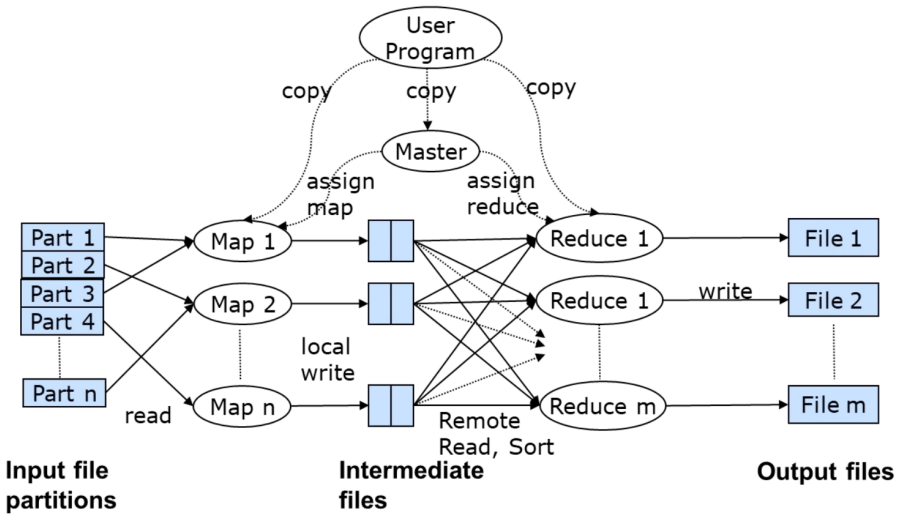
```
reduce(String key, List recordlist) {
        String filename = key;
        int count = 0;
        for each record in recordlist
                count = count + 1;
        output(filename, count);
}
```

Schematic Flow of Keys and Values

- Flow of keys and values in a MapReduce task



Parallel Processing of MapReduce Job



Hadoop MapReduce

- Google pioneered MapReduce implementations that could run on thousands of machines (nodes) and transparently handle failures of machines
- Hadoop is a widely-used open-source implementation of MapReduce written in Java
  - The map and reduce functions can be written in several different languages
- The input and output to MapReduce systems such as Hadoop must be done in parallel
  - Google uses the GFS distributed file system
  - Hadoop uses the Hadoop File System (HDFS)
  - Input files can be in several formats
    - Text/CSV
    - Compressed representation such as Avro, ORC, and Parquet
  - Hadoop also supports key-value stores such as Hbase, Cassandra, MongoDB, etc.

Types in Hadoop

- Generic Mapper and Reducer interfaces both take four type arguments, that specify the types of the input key, input value, output key, and output value
- Map class in the next slide implements the Mapper interface
  - Map input key is of type LongWritable, i.e., a long integer
  - Map input value, which is (all or part of) a document, is of type Text
  - Map output key is of type Text, since the key is a word
  - Map output value is of type IntWritable, which is an integer value

Hadoop Code in Java: Map Function

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(LongWritable key, Text value, Context context)
                throws IOException,
        InterruptedException {
                String line = value.toString();
                StringTokenizer tokenizer = new StringTokenizer(line);
                while (tokenizer.hasMoreTokens()) {
```

```
                        word.set(tokenizer.nextToken());
                        context.write(word, one); }
        }
}
```

Hadoop Code in Java: Reduce Function

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {
                sum += val.get(); }
                context.write(key, new IntWritable(sum));
        }
}
```

Hadoop Job Parameters

- The classes that contain the map and reduce functions for the job
  - Set by the methods setMapperClass() and setReducerClass()
- The types of the job's output key and values
  - Set by the methods setOutputKeyClass() and setOutputValueClass()
- The input format of the job
  - Set by the method job.setInputFormatClass()
  - Default input format in Hadoop is TextInputFormat
    - Map key whose value is a byte offset into the file
    - Map value is the contents of one line of the file
- The directories where the input files are stored and where the output files must be created
  - Set by the methods addInputPath() and addOutputPath()
- And many more parameters

Hadoop Code in Java: Overall Program

```
public class WordCount {
        public static void main(String[] args) throws Exception {
                Configuration conf = new Configuration();
                Job job = new Job(conf, "wordcount");
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(IntWritable.class);
                job.setMapperClass(Map.class);
                job.setReducerClass(Reduce.class);
                job.setInputFormatClass(TextInputFormat.class);
                job.setOutputFormatClass(TextOutputFormat.class);
                FileInputFormat.addInputPath(job, new Path(args[0]));
                FileOutputFormat.setOutputPath(job, new Path(args[1]));
                job.waitForCompletion(true);
```

```
        }
}
```

MapReduce vs. Databases

- MapReduce has been widely used for parallel processing by Google, Yahoo, and 100’s of other companies
- e.g., computing PageRank, building keyword indices, data analysis of web click logs, ...
- Many real-world uses of MapReduce cannot be expressed in SQL
- But many computations are much easier to express in SQL
- MapReduce is cumbersome for writing simple queries

MapReduce vs. Databases (Cont’d)

- Relational operations (select, project, join, aggregation, etc.) can be expressed using MapReduce
- SQL queries can be translated into the MapReduce infrastructure for execution
- Apache Hive SQL, Apache Pig Latin, Microsoft SCOPE
- Current generation execution engines support not only MapReduce, but also other algebraic operations such as joins, aggregation, etc. natively