UNIT 1

Introduction: Database systems applications, Purpose of Database Systems, view of Data, Database Languages, Relational Databases, Database Design, Data Storage and Querying, Transaction Management, Database Architecture, Data Mining and Information Retrieval, Specialty Databases, Database users and Administrators.
Introduction to Relational Model: Structure of Relational Databases, Database Schema, Keys, Schema Diagrams, Relational Query Languages, Relational Operations

Introduction

A database-management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an enterprise.
The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.
Database systems are designed to manage large bodies of information.
Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information.
In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results. Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data.

Database-System Applications

Databases are widely used. Here are some representative applications:

- Enterprise Information
  - Sales: For customer, product, and purchase information.
  - Accounting: For payments, receipts, account balances, assets and other accounting information.
  - Human resources: For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.
  - Manufacturing: For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
  - Online retailers: For sales data noted above plus online order tracking, generation of recommendation lists, and maintenance of online product evaluations.
- Banking and Finance
  - Banking: For customer information, accounts, loans, and banking transactions.
  - Credit card transactions: For purchases on credit cards and generation of monthly statements.
  - Finance: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- Universities: For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- Airlines: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- Telecommunication: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

As the list illustrates, databases form an essential part of every enterprise today, storing not only types of information that are common to most enterprises, but also information that is specific to the category of the enterprise.
Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises.
In the early days, very few people interacted directly with database systems, although without realizing it,

they interacted with databases indirectly— through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents. Then automated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice-response systems) also allowed users to deal directly with databases—a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, for example, or to register for courses in a university.
The Internet revolution of the late 1990s sharply increased direct user access to databases. Organizations converted many of their phone interfaces to databases into Web interfaces, and made a variety of services and information available online. For instance, when you access an online bookstore and browse a book or music collection, you are accessing data stored in a database. When you enter an order online, your order is stored in a database. When you access a bank Web site and retrieve your bank balance and transaction information, the information is retrieved from the bank's database system. When you access a Web site, information about you may be retrieved from a database to select which advertisements you should see. Furthermore, data about your Web accesses may be stored in a database. Thus, although user interfaces hide details of access to a database, and most people are not even aware they are dealing with a database, accessing databases forms an essential part of almost everyone's life today.

Purpose of Database Systems
In the early days, database applications were built directly on top of file system, which leads to:

- Data redundancy and inconsistency: data is stored in multiple file formats resulting induplication of information in different files
- Difficulty in accessing data
    - Need to write a new program to carry out each new task
- Data isolation
    - Multiple files and formats
- Integrity problems
    - Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones
- Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access by multiple users
    - Concurrent access needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies
        - Ex: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- Security problems
    - Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

University Database Example

- Data consists of information about:
    - Students
    - Instructors
    - Classes
- Application program examples:
    - Add new students, instructors, and courses
    - Register students for courses, and generate class rosters
    - Assign grades to students, compute grade point averages (GPA) and generate transcripts

View of Data

- A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data.
- A major purpose of a database system is to provide users with an abstract view of the data.
  - Data models
    - A collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.
  - Data abstraction
    - Hide the complexity of data structures to represent data in the database from users through several levels of data abstraction.

Data Models

- A collection of tools for describing
  - Data
  - Data relationships
  - Data semantics
  - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi-structured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

Relational Model

- All the data is stored in various tables.
- Example of tabular data in the relational model

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

(a) The *instructor* table

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Levels of Abstraction

Physical level: describes how a record (e.g., instructor) is stored.

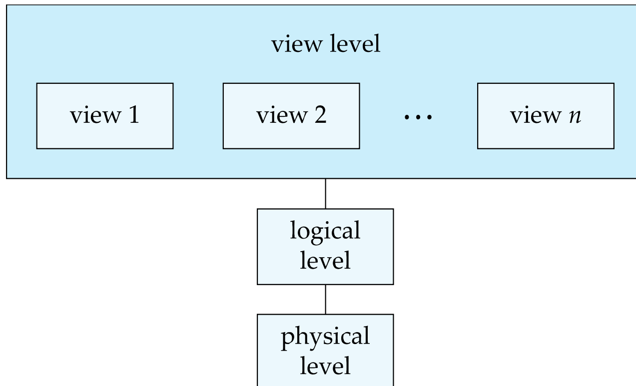
Logical level: describes data stored in database, and the relationships among the data.

```
type instructor = record
        ID : string;
        name : string;
        dept_name : string;
```

salary : integer;
end;

View level: application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

Architecture for a database system



Instances and Schemas

- Similar to types and variables in programming languages
- **Logical Schema** – the overall logical structure of the database
  - Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them
    - Analogous to type information of a variable in a program
- **Physical schema** – the overall physical structure of the database
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable

Physical Data Independence

- The ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

Data Definition Language (DDL)

- Specification notation for defining the database schema

Example:

```
create table instructor (
    ID          char(5),
    name        varchar(20),
    dept_name   varchar(20),
    salary      numeric(8,2))
```

- DDL compiler generates a set of table templates stored in a ***data dictionary***
- Data dictionary contains metadata (i.e., data about data)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
  - Authorization
    - Who can access what

Data Manipulation Language (DML)

- Language for accessing and updating the data organized by the appropriate data model
  - DML also known as query language
- There are basically two types of data-manipulation language
  - **Procedural DML** -- require a user to specify what data are needed and how to get those data.
  - **Declarative DML --** require a user to specify what data are needed without specifying how to get those data.
- Declarative DMLs are usually easier to learn and use than are procedural DMLs.
- Declarative DMLs are also referred to as non-procedural DMLs
- The portion of a DML that involves information retrieval is called a **query** language.

SQL Query Language

- SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table.
- Example to find all instructors in Comp. Sci. dept

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.'
- SQL is **NOT** a Turing machine equivalent language
- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Database Access from Application Program

- Non-procedural query languages such as SQL are not as powerful as a universal Turing machine.
- SQL does not support actions such as input from users, output to displays, or communication over the network.
- Such computations and actions must be written in a **host language**, such as C/C++, Java or Python, with embedded SQL queries that access the data in the database.
- **Application programs** -- are programs that are used to interact with the database in this fashion.

Database Design

The process of designing the general structure of the database:

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
  - Business decision – What attributes should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

Database Engine

- A database system is partitioned into modules that deal with each of the responsibilities of the overall system.
- The functional components of a database system can be divided into
  - The storage manager,
  - The query processor component,
  - The transaction management component.

Storage Manager

- A program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

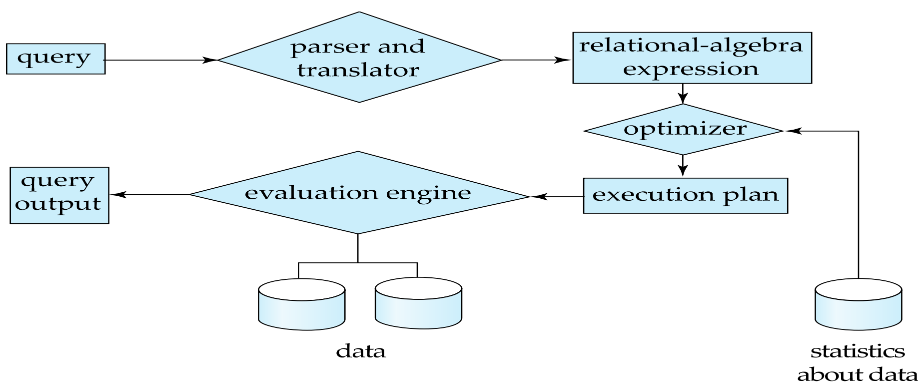
- The storage manager is responsible to the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data
- The storage manager components include:
  - Authorization and integrity manager
  - Transaction manager
  - File manager
  - Buffer manager
- The storage manager implements several data structures as part of the physical system implementation:
  - Data files -- store the database itself
  - Data dictionary -- stores metadata about the structure of the database, in particular the schema of the database.
  - Indices -- can provide fast access to data items. A database index provides pointers to those data items that hold a particular value.

Query Processor

- The query processor components include:
  - DDL interpreter -- interprets DDL statements and records the definitions in the data dictionary.
  - DML compiler -- translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.
    - The DML compiler performs query optimization; that is, it picks the lowest cost evaluation plan from among the various alternatives.
  - Query evaluation engine -- executes low-level instructions generated by the DML compiler.

Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

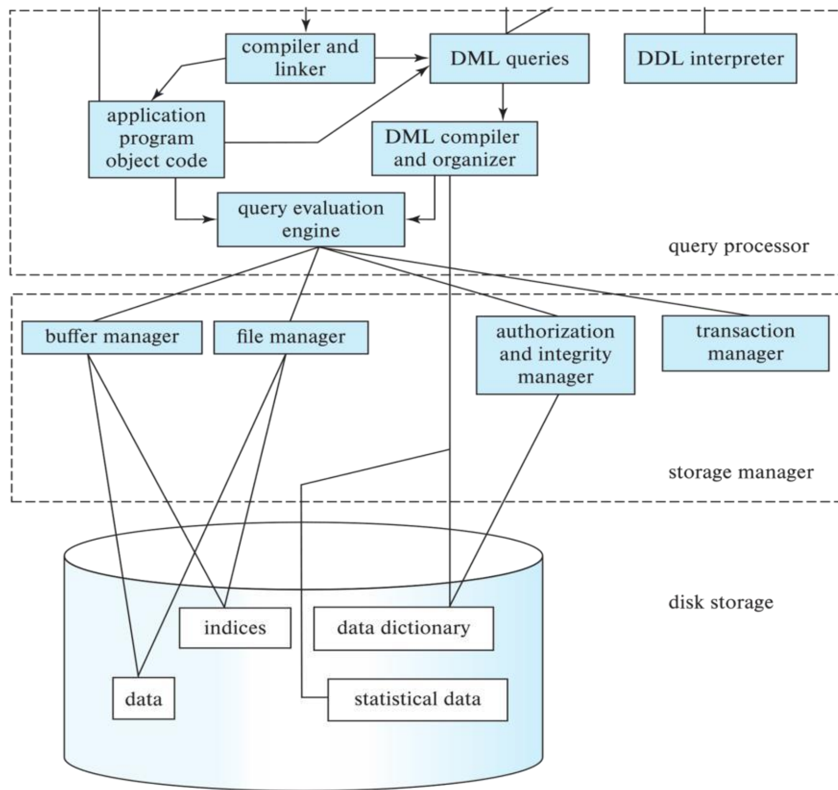


Transaction Management

- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Database Architecture

- Centralized databases
  - One to a few cores, shared memory
- Client-server,
  - One server machine executes work on behalf of multiple client machines.
- Parallel databases
  - Many core shared memory
  - Shared disk
  - Shared nothing
- Distributed databases
  - Geographical distribution
  - Schema/data heterogeneity

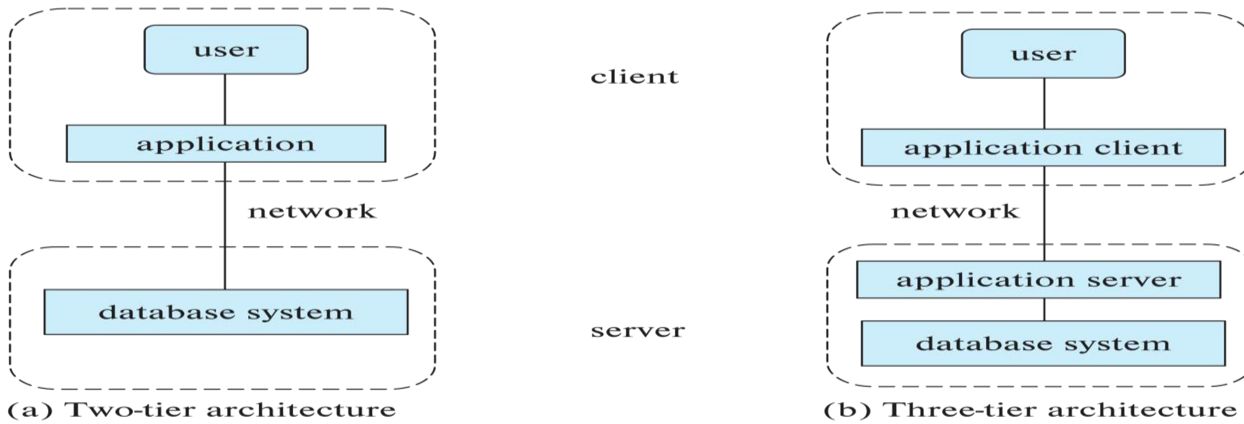


Database Applications

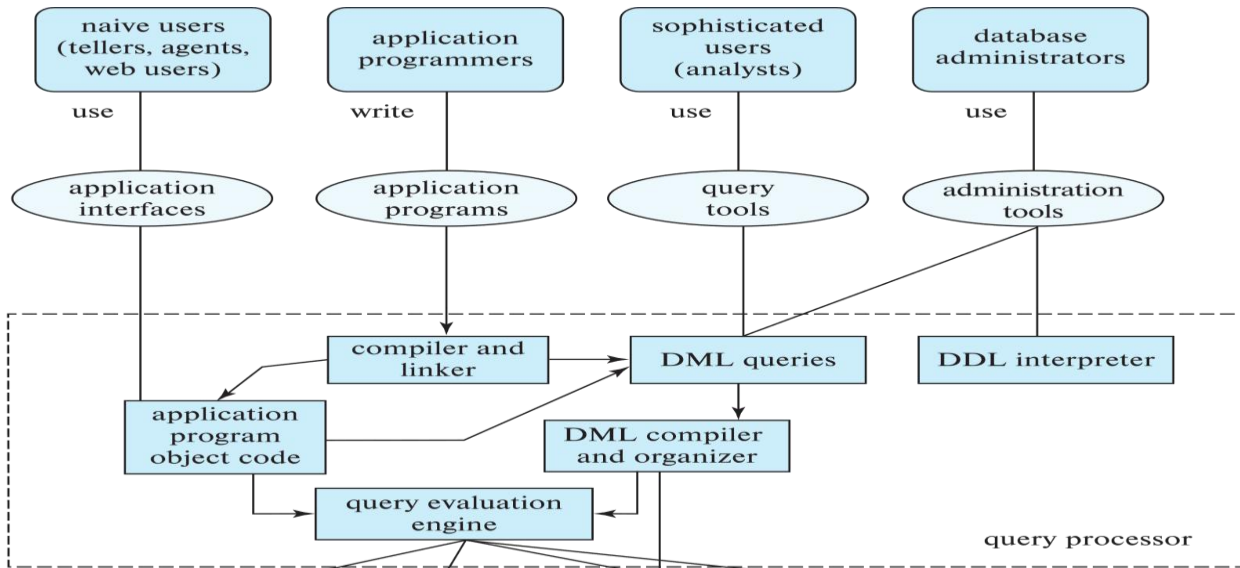
Database applications are usually partitioned into two or three parts

- Two-tier architecture -- the application resides at the client machine, where it invokes database system functionality at the server machine
- Three-tier architecture -- the client machine acts as a front end and does not contain any direct database calls.
  - The client end communicates with an application server, usually through a forms interface.
  - The application server in turn communicates with a database system to access data.

Two-tier and three-tier architectures



Database Users



Database Administrator

A person who has central control over the system is called a **database administrator (DBA).**

Functions of a DBA include:

- Schema definition
- Storage structure and access-method definition
- Schema and physical-organization modification
- Granting of authorization for data access
- Routine maintenance
- Periodically backing up the database
- Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required
- Monitoring jobs running on the database

History of Database Systems

- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provided only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allowed direct access to data

        - Network and hierarchical data models in widespread use
        - Ted Codd defines the relational data model
            - Would win the ACM Turing Award for this work
            - IBM Research begins System R prototype
            - UC Berkeley (Michael Stonebraker) begins Ingres prototype
            - Oracle releases first commercial relational database
        - High-performance (for the era) transaction processing
- 1980s:
    - Research relational prototypes evolve into commercial systems
        - SQL becomes industrial standard
    - Parallel and distributed database systems
        - Wisconsin, IBM, Teradata
    - Object-oriented database systems
- 1990s:
    - Large decision support and data-mining applications
    - Large multi-terabyte data warehouses
    - Emergence of Web commerce
- 2000s
    - Big data storage systems
        - Google BigTable, Yahoo PNuts, Amazon,
        - "NoSQL" systems.
    - Big data analysis: beyond SQL
        - Map reduce and friends
- 2010s
    - SQL reloaded
        - SQL front end to Map Reduce systems
        - Massively parallel database systems
        - Multi-core main-memory databases

Relation Schema and Instance

- $A_1, A_2, \ldots, A_n$ are *attributes*
- $R = (A_1, A_2, \ldots, A_n)$ is a *relation schema*
  Example: *instructor* = (*ID, name, dept_name, salary*)
- A relation instance *r* defined over schema *R* is denoted by *r* (*R*).
- The current values a relation are specified by a table
- An element ***t*** of relation ***r*** is called a *tuple* and is represented by a *row* in a table

Attributes

- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is "unknown"
- The null value causes complications in the definition of many operations

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *instructor* relation with unordered tuples

Database Schema

- Database schema -- is the logical structure of the database.
- Database instance -- is a snapshot of the data in the database at a given instant in time.
- Example:
    - schema: i*nstructor* (*ID, name, dept_name, salary*)

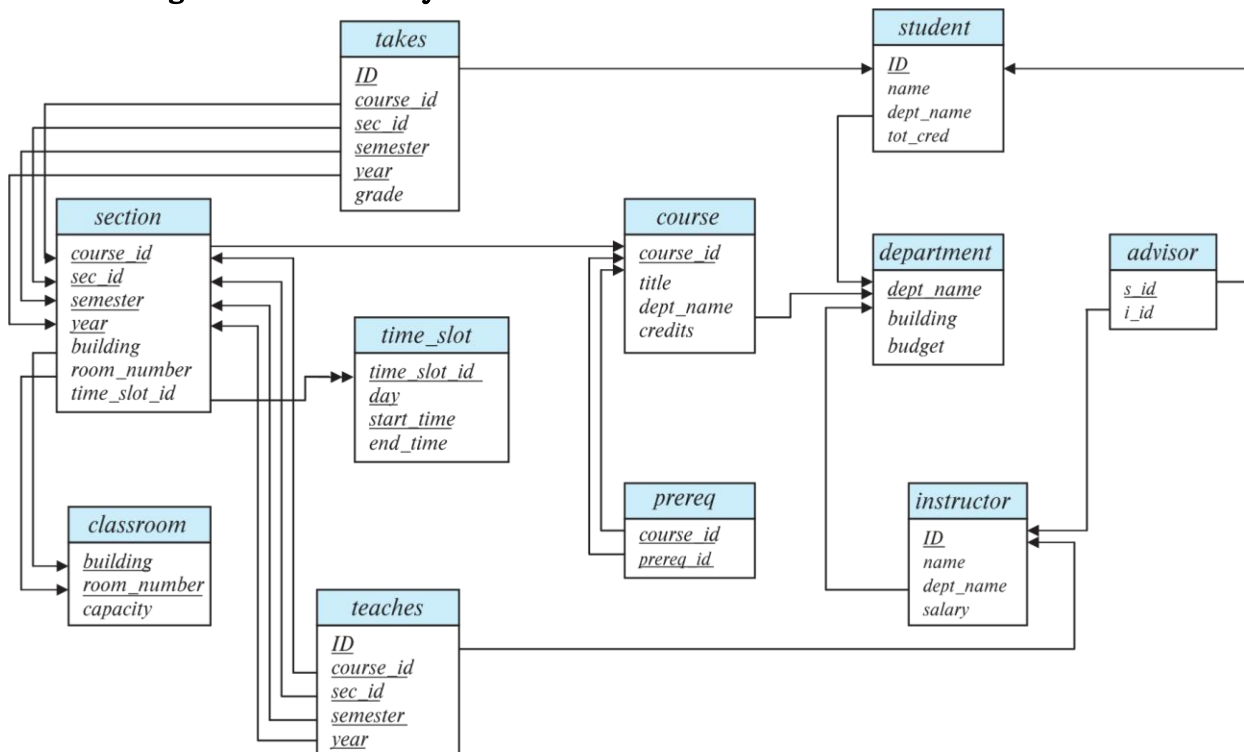
- Instance:

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

Keys

- Let K ⊆ R
- *K* is a **superkey** of *R* if values for *K* are sufficient to identify a unique tuple of each possible relation *r(R)*
  - Example: {*ID*} and {ID,name} are both superkeys of *instructor.*
- Superkey *K* is a **candidate key** if *K* is minimal
  Example: {*ID*} is a candidate key for *Instructor*
- One of the candidate keys is selected to be the **primary key**.
- **Foreign key** constraint: Value in one relation must appear in another
  - **Referencing** relation
  - **Referenced** relation
  - Example: *dept_name* in i*nstructor* is a foreign key from *instructor* referencing *department*

Schema Diagram for University Database



Relational Query Languages

- Procedural versus non-procedural, or declarative
- "Pure" languages:

- Relational algebra
- Tuple relational calculus
- Domain relational calculus

▪ The above 3 pure languages are equivalent in computing power
▪ We will concentrate in this chapter on relational algebra
  - Not Turing-machine equivalent
  - Consists of 6 basic operations

Relational Algebra

▪ A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
▪ Six basic operators:
  - select: σ
  - project: ∏
  - union: ∪
  - set difference: –
  - Cartesian product: x
  - rename: *ρ*

Select Operation

▪ The **selec**t operation selects tuples that satisfy a given predicate.
▪ Notation: $\sigma_{p}(r)$
▪ *p* is called the **selection predicate**
▪ Example: select those tuples of the *instructor* relation where the instructor is in the "Physics" department.
  - Query: $\sigma_{dept_name=\text{"Physics"}}$ (instructor)
  - Result

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

▪ We allow comparisons using =, ≠, >, ≥. <. ≤ in the selection predicate.
▪ We can combine several predicates into a larger predicate by using the connectives: ∧ (**and**), ∨ (**or**), ¬ (**not**)

Example: Find the instructors in Physics with a salary greater $90,000, we write:

$\sigma_{dept_name=\text{"Physics"} \wedge salary > 90,000}$ (instructor)

▪ The select predicate may include comparisons between two attributes.
  - Example, find all departments whose name is the same as their building name:
  - $\sigma_{dept_name=building}$ (*department*)

Project Operation

▪ A unary operation that returns its argument relation, with certain attributes left out.
▪ Notation: $\prod_{A1, A2, A3 ....Ak}(r)$ where A_1, A_2, …, A_k are attribute names and *r* is a relation name.
▪ The result is defined as the relation of *k* columns obtained by erasing the columns that are not listed
▪ Duplicate rows removed from result, since relations are sets

Example: eliminate the *dept_name* attribute of *instructor*

▪ Query: $\prod_{ID, name, salary}$ (*instructor*)
▪ Result:

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

Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query -- Find the names of all instructors in the Physics department.
  $\prod_{name}(\sigma_{dept_name = \text{“Physics”}}\ (instructor))$
- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

Cartesian-Product Operation

- The Cartesian-product operation (denoted by X) allows us to combine information from any two relations.
  Example: the Cartesian product of the relations *instructor* and t*eaches* is written as:
  instructor X *teaches*
- We construct a tuple of the result out of each possible pair of tuples: one from the *instructor* relation and one from the *teaches* relation (see next slide)
- Since the instructor *ID* appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - *instructor.ID*
  - *teaches.ID*

The *instructor* X *teaches* table

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
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Join Operation

- The Cartesian-Product
  instructor X *teaches* associates every tuple of instructor with every tuple of teaches.
  - Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of "*instructor* X *teaches* " that pertain to instructors and the courses that they taught, we write:
  $\sigma_{instructor.id\ =\ teaches.id}$ (*instructor* x *teaches*))
  - We get only those tuples of "*instructor* X *teaches*" that pertain to instructors and the courses that they taught.
- The result of this expression,

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

- The join operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations r(R) and s(S). Let "theta" be a predicate on attributes in the schema R "union" S.
- The join operation $r \bowtie_\theta s$ is defined as follows: $r \bowtie_\theta s = \sigma_\theta (r \times s)$
- Thus, $\sigma_{instructor.id\ =\ teaches.id}$ (instructor x teaches)
- Can equivalently be written as: instructor $\bowtie_{instructor.id\ =\ teaches.id}$ teaches

Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
  1. *r, s* must have the *same* **arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2nd column of *r* deals with the same type of values as does the 2nd column of *s*)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both $\prod_{course_id} (\sigma_{semester=\text{"Fall"} \land year=2017} (section)) \cup \prod_{course_id} (\sigma_{semester=\text{"Spring"} \land year=2018} (section))$
- Result of: $\prod_{course_id} (\sigma_{semester=\text{"Fall"} \land year=2017} (section)) \cup \prod_{course_id} (\sigma_{semester=\text{"Spring"} \land year=2018} (section))$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
  - *r*, *s* have the *same arity*
  - attributes of *r* and *s* are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

  $\prod_{course_id} (\sigma_{semester=\text{"Fall"} \land year=2017} (section)) \cap \prod_{course_id} (\sigma_{semester=\text{"Spring"} \land year=2018} (section))$

Result:

course_id
CS-101

Set Difference Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Set differences must be taken between **compatible** relations.
  - *r* and *s* must have the same arity
  - attribute domains of *r* and *s* must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

  $\prod_{course_id} (\sigma_{semester=\text{"Fall"} \land year=2017} (section)) - \prod_{course_id} (\sigma_{semester=\text{"Spring"} \land year=2018} (section))$

course_id
CS-347
PHY-101

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by $\leftarrow$ and works like assignment in a programming language.
- Example: Find all instructor in the "Physics" and Music department.

  $Physics \leftarrow \sigma_{dept_name=\text{"Physics"}} (instructor)$

$Music \leftarrow \sigma_{dept_name=\text{"Music"}}(instructor)$

$Physics \cup Music$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them. The rename operator, ρ, is provided for that purpose
- The expression: $\rho_x(E)$ returns the result of expression E under the name x
- Another form of the rename operation: $\rho_{x(A1,A2, .. An)}(E)$

Equivalent Queries

- There is more than one way to write a query in relational algebra.

Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000

- Query 1: $\sigma_{dept_name=\text{"Physics"} \wedge salary > 90,000}(instructor)$
- Query 2: $\sigma_{dept_name=\text{"Physics"}}(\sigma_{salary > 90.000}(instructor))$
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database.

Example: Find information about courses taught by instructors in the Physics department

- Query 1: $\sigma_{dept_name=\text{"Physics"}}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$
- Query 2: $(\sigma_{dept_name=\text{"Physics"}}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$.
- The two queries are not identical; they are, however, equivalent -- they give the same result on any database

UNIT 2

Introduction to SQL: Overview of the SQL Query Language, SQL Data Definition, Basic Structure of SQL Queries, Additional Basic Operations, Set Operations, Null Values, Aggregate Functions, Nested Sub-queries, Modification of the Database.

Intermediate SQL: Joint Expressions, Views, Transactions, Integrity Constraints, SQL Data types and schemas, Authorization.

Advanced SQL: Accessing SQL from a Programming Language, Functions and Procedures, Triggers, Recursive Queries, OLAP, Formal relational query languages.

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

SQL Parts

- DML -- provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- Integrity – the DDL includes commands for specifying integrity constraints.
- View definition -- The DDL includes commands for defining views.
- Transaction control – includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL -- define how SQL statements can be embedded within general-purpose programming languages.
- Authorization – includes commands for specifying access rights to relations and views.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The Integrity constraints
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length n.
- **varchar(n).** Variable length character strings, with user-specified maximum length n.
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of p digits, with d digits to the right of decimal point. (ex., **numeric**(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least n digits.

Create Table Construct

- An SQL relation is defined using the **create table** command:

```
create table r
        (A1 D1, A2 D2, ..., An Dn,
    (integrity-constraint1),
    ..,
    (integrity-constraintk))
```

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i
- Example:

```
create table instructor (
        ID          char(5),
        name        varchar(20),
        dept_name   varchar(20),
        salary      numeric(8,2))
```

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n)
  - **foreign key** (A_m, ..., A_n) **references** r
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.

Example:

```
create table instructor (
    ID          char(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    salary      numeric(8,2),
    primary key (ID),
    foreign key (dept_name) references department);

create table student (
    ID            varchar(5),
    name          varchar(20) not null,
    dept_name     varchar(20),
    tot_cred      numeric(3,0),
    primary key (ID),
    foreign key (dept_name) references department);
create table takes (
    ID            varchar(5),
    course_id     varchar(8),
    sec_id        varchar(8),
    semester      varchar(6),
    year          numeric(4,0),
    grade         varchar(2),
    primary key (ID, course_id, sec_id, semester, year) ,
    foreign key (ID) references  student,
    foreign key (course_id, sec_id, semester, year) references section);
create table course (
    course_id     varchar(8),
    title         varchar(50),
```

dept_name varchar(20),
credits numeric(2,0),
primary key (course_id),
foreign key (dept_name) references department);

Updates to tables

- Insert
  - insert into instructor values ('10211', 'Smith', 'Biology', 66000);
- Delete
  - Remove all tuples from the student relation
    - delete from student
- Drop Table
  - drop table r
- Alter
  - alter table r add A D
    - where A is the name of the attribute to be added to relation r and D is the domain of A.
    - All exiting tuples in the relation are assigned null as the value for the new attribute.
  - alter table r drop A
    - where A is the name of an attribute of relation r
    - Dropping of attributes not supported by many databases.

Basic Query Structure

- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P
  - A_i represents an attribute, R_i represents a relation, P is a predicate.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select name **from** instructor
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., Name ≡ NAME ≡ name
  - Some people use upper case wherever we use bold font.
- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the department names of all instructors, and remove duplicates

  select distinct dept_name **from** instructor
- The keyword **all** specifies that duplicates should not be removed.

  select all dept_name **from** instructor
- An asterisk in the select clause denotes "all attributes"

  select * **from** instructor
- An attribute can be a literal with no **from** clause

  select '437'
  - Results is a table with one column and a single row with value "437"
  - Can give the column a name using: **select** '437' **as** FOO
- An attribute can be a literal with **from** clause

  select 'A' **from** instructor
  - Result is a table with one column and N rows (number of tuples in the instructors table), each row with value "A"

- The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.
  - The query:

    select ID, name, salary/12 **from** instructor

  would return a relation that is the same as the instructor relation, except that the value of the attribute salary is divided by 12.
  - Can rename "salary/12" using the **as** clause: **select** ID, name, salary/12 **as** monthly_salary

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept

  select name **from** instructor **where** dept_name = 'Comp. Sci.'
- SQL allows the use of the logical connectives **and, or,** and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions
- To find all instructors in Comp. Sci. dept with salary > 70000

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

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select * **from** *instructor, teaches*
  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID* = *teaches.ID*

- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID* = *teaches.ID* ***and*** *instructor. dept_name* = 'Art'

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause: *old-name* **as** *new-name*
- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T.name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary* > *S.salary* **and** *S.dept_name* = *'Comp. Sci.'*
- Keyword **as** is optional and may be omitted: *instructor* **as** *T* ≡ *instructor T*

Self Join Example

- Relation *emp-super*

person	*supervisor*
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Find the supervisor of "Bob"
- Find the supervisor of the supervisor of "Bob"
- Can you find ALL the supervisors (direct and indirect) of "Bob"?

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator **like** uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all instructors whose name includes the substring "dar".
  select *name*
  from *instructor*
  where *name* **like** '%dar%'
- Match the string "100%"
  like '100 \%' **escape** '\' ; we use backslash (\) as the escape character.
- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp" as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.
- SQL supports a variety of string operations such as
  - concatenation (using "||")
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors
  select distinct *name*
  from *instructor*
  order by *name*
- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, $90,000 and £ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison

- **select** *name*, *course_id*
  from *instructor*, *teaches*
  where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

Set Operations

- Find courses that ran in Fall 2017 or in Spring 2018
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  union
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)
- Find courses that ran in Fall 2017 and in Spring 2018
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)
- Find courses that ran in Fall 2017 but not in Spring 2018
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  except
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)

- Set operations **union, intersect,** and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **union all**,
  - **intersect all**
  - **except all**.

Null Values

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving **null** is **null**
  - Example: 5 + **null** returns **null**
- The predicate is **null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.
    select *name*
    from *instructor*
    where *salary* **is null**
- The predicate **is not null** succeeds if the value on which it is applied is not null.

- It is possible for tuples to have a null value, denoted by **null**, for some of their attributes
- **null** signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving **null** is **null**
  - Example: 5 + **null** returns **null**
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.
    select *name*
    from *instructor*
    where *salary* **is null**
- The predicate **is not null** succeeds if the value on which it is applied is not null.

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value
  avg: average value
  min: minimum value
  max: maximum value

sum: sum of values
count: number of values

- Find the average salary of instructors in the Computer Science department

  select avg (*salary*)
  from *instructor*
  where *dept_name*= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2018 semester

  select count (**distinct** *ID*)
  from *teaches*
  where *semester* = 'Spring' **and** *year* = 2018;
- Find the number of tuples in the *course* relation

  select count (*)
  from *course*;
- Find the average salary of instructors in each department

  select *dept_name*, **avg** (*salary*) **as** *avg_salary*
  from *instructor*
  group by *dept_name*;
- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

  /* erroneous query */
  select *dept_name*, *ID*, **avg** (*salary*)
  from *instructor*
  group by *dept_name*;

Aggregate Functions – Having Clause

- Find the names and average salaries of all departments whose average salary is greater than 42000
  select *dept_name*, **avg** (*salary*) **as** *avg_salary*
  from *instructor*
  group by *dept_name*
  having avg (*salary*) > 42000;
- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is a **select-from-where** expression that is nested within another query.
- The nesting can be done in the following SQL query

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

  as follows:
  - **From clause:** r_i can be replaced by any valid subquery
  - **Where clause:** P can be replaced with an expression of the form: B <operation> (subquery)

  B is an attribute and <operation> to be defined later.
  - **Select clause:**

  A_i can be replaced be a subquery that generates a single value.

Set Membership

- Find courses offered in Fall 2017 and in Spring 2018
  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);

- Find courses offered 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);
```

- Name all instructors whose name is neither “Mozart” nor Einstein”

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

- Find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 10101

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in
                        (select course_id, sec_id, semester, year
                         from teaches
                         where teaches.ID= 10101);
```

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- Same query using > **some** clause

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

Definition of "some" Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<, \leq, >, =, \neq$

(5 < **some** [0, 5, 6]) = true (read: 5 < some tuple in the relation)

(5 < **some** [0, 5]) = false

(5 = **some** [0, 5]) = true

(5 ≠ **some** [0, 5]) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison – "all" Clause

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.
  select *name*
  from *instructor*
  where *salary* > **all** (**select** *salary* **from** *instructor* **where** *dept name* = 'Biology');

Definition of "all" Clause

- F <comp> **all** $r \Leftrightarrow \forall\, t \in r$ (F <comp> t)

Definition of "all" Clause

- F <comp> all $r \Leftrightarrow \forall\, t \in r$ (F <comp> t)

(5 < **all** [0, 5, 6]) = false

(5 < **all** [6, 10]) = true

(5 = **all** [4, 5]) = false

(5 ≠ **all** [4, 6]) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**
However, (= **all**) ≢ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester"

```
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);
```

- **Correlation name** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
select distinct 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));
```

- First nested query lists all courses offered in Biology
- Second nested query lists all courses a particular student took

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to "true" if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2017

```
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);
```

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from ( select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from ( select dept_name, avg (salary)
       from instructor
       group by dept_name)
       as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

With Clause

- The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
     (select max(budget)
      from department)
select department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept _total (dept_name, value) as
     (select dept_name, sum(salary)
      from instructor
      group by dept_name),
dept_total_avg(value) as
     (select avg(value)
      from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
       ( select count(*)
         from instructor
         where department.dept_name = instructor.dept_name)
       as num_instructors
from department;
```

- Runtime error if subquery returns more than one result tuple

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation
- Updating of values in some tuples in a given relation

Deletion

- Delete all instructors

```
delete from instructor
```

- Delete all instructors from the Finance department

```
delete from instructor
where dept_name= 'Finance';
```

- *Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.*

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

- Delete all instructors whose salary is less than the average salary of instructors

```
delete from instructor
    where salary < (select avg (salary)
        from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary changes
  - Solution used in SQL:
    1. First, compute **avg** (salary) and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Insertion

- Add a new tuple to *course*

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

```
insert into student
    values ('3003', 'Green', 'Finance', null);
```

- Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

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

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.
  Otherwise queries like: **insert into** *table*1 **select** * **from** *table*1 would cause problem

Updates

- Give a 5% salary raise to all instructors

```
update instructor set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70000

```
update instructor
    set salary = salary * 1.05
    where salary < 70000;
```

- Give a 5% salary raise to instructors whose salary is less than average

```
update instructor
    set salary = salary * 1.05
    where salary < (select avg (salary) from instructor);
```

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

```
update instructor
        set salary = salary * 1.03
        where salary > 100000;
update instructor
        set salary = salary * 1.05
        where salary <= 100000;
```

- The order is important
- Can be done better using the **case** statement

Case Statement for Conditional Updates

- Same query as before but with case statement

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

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
set tot_cred = (select sum(credits)
            from takes, course
          where takes.course_id = course.course_id  and
                S.ID= takes.ID and takes.grade <> 'F' and
                takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
        when sum(credits) is not null then sum(credits)
        else 0
end
```

Intermediate SQL

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join
- The join operations are typically used as subquery expressions in the **from** clause
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

Natural Join in SQL

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.
- List the names of instructors along with the course ID of the courses that they taught
  - **select** *name*, *course_id*
    from *students, takes*
    where *student.ID* = *takes.ID*;
- Same query in SQL with "natural join" construct
  - **select** *name*, *course_id*
    from *student* **natural join** *takes*;
- The **from** clause can have multiple relations combined using natural join:
  select $A_1, A_2, \ldots A_n$
  from r_1 **natural join** r_2 **natural join .. natural join** r_n
  where P ;

Student Relation

ID	name	dept_name	tot_cred
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	course_id	sec_id	semester	year	grade
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student **natural join** *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students instructors along with the titles of courses that they have taken
  - Correct version
    select *name*, *title*
    from *student* **natural join** *takes*, *course*
    where *takes.course_id* = *course.course_id*;
  - Incorrect version
    select *name*, *title*
    from *student* **natural join** *takes* **natural join** *course*;
    - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department.
    - The correct version (above), correctly outputs such pairs.

Natural Join with Using Clause

- To avoid the danger of equating attributes erroneously, we can use the "**using**" construct that allows us to specify exactly which columns should be equated.
- Query example
  select *name*, *title*
  from (*student* **natural join** *takes*) **join** *course* **using** (*course_id*)

Join Condition

- The **on** condition allows a general predicate over the relations being joined
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**
- Query example
  select * from *student* **join** *takes* **on** *student_ID = takes_ID*
  - The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:
  select * from *student , takes* **where** *student_ID = takes_ID*
- The **on** condition allows a general predicate over the relations being joined.
- This predicate is written like a **where** clause predicate except for the use of the keyword **on**.
- Query example
  select * from *student* **join** *takes* **on** *student_ID = takes_ID*
  - The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal.
- Equivalent to:
  select * from *student , takes* **where** *student_ID = takes_ID*

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.
- Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join

Outer Join Examples

course_id	title	dept_name	credits
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

Relation *course*

course_id	prereq_id
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Relation *prereq*

Left Outer Join

- *course* **natural left outer join** *prereq*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

  - In relational algebra: *course* ⟕ *prereq*

Right Outer Join

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- In relational algebra: *course* ⟖ *prereq*

Full Outer Join

- *course* **natural full outer join** *prereq*
- In relational algebra: *course* ⟗ *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- *course* **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

- *course* **inner join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- *course* **left outer join** *prereq* **on** *course.course_id* = *prereq.course_id*

course_id	title	dept_name	credits	prereq_id	course_id
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

- *course* **natural right outer join** *prereq*

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- *course* **full outer join** *prereq* **using** (*course_id*)

course_id	title	dept_name	credits	prereq_id
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name* **from** *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition and Use

- A view of instructors without their salary

  create view *faculty* as
  select *ID*, *name*, *dept_name* **from** *instructor*
- Find all instructors in the Biology department

  select *name* **from *faculty* where** *dept_name* = 'Biology'
- Create a view of department salary totals

  create view *departments_total_salary(dept_name, total_salary)* as
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be ***recursive*** if it depends on itself.

- **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';

- **create view *physics_fall_2017*_*watson* as**
  select *course_id*, *room_number*
  from *physics_fall_2017*
  where *building*= 'Watson';

View Expansion

- Expand the view :
  create view *physics_fall_2017_watson* as
  select *course_id*, *room_number*
  from *physics_fall_2017*
  where *building*= 'Watson'
- To:
  create view *physics_fall_2017_watson* as
  select *course_id*, *room_number*
  from (**select** *course.course_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')
  where *building*= 'Watson';

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:
  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date

- Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

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

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Insert the tuple

    ('30765', 'Green', 'Music', null)

  into the *instructor* relation

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name* = *department.dept_name*;
- **insert into** *instructor_info*
  values ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?
- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert
  ('25566', 'Brown', 'Biology', 100000) into *history_instructors?*

View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

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 work**. The updates performed by the transaction become permanent in the database.
  - **Rollback work**. All the updates performed by the SQL statements in the transaction are undone.
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00
  - A salary of a bank employee must be at least $4.00 an hour
  - A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

- Declare *name* and *budget* to be **not null**

  name **varchar**(20) **not null**
  budget **numeric**(12,2) **not null**

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes A_1, A_2, …, A_m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

- The **check** (P) clause specifies a predicate P that must be satisfied by every tuple in a relation.
- Example: ensure that semester is one of fall, winter, spring or summer

```
create table section
(course_id varchar (8),
sec_id varchar (8),
semester varchar (6),
year numeric (4,0),
building varchar (15),
room_number varchar (7),
time slot id varchar (4),
primary key (course_id, sec_id, semester, year),
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.
- Foreign *keys can be* specified as part of the SQL **create table** statement

  foreign key (*dept_name*) **references** *department*
- By default, a foreign key references the primary-key attributes of the referenced table.
- SQL allows a list of attributes of the referenced relation to be specified explicitly.

  foreign key (*dept_name*) **references** *department* (*dept_name*)

Cascading Actions in Referential Integrity

- When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- An alternative, in case of delete or update is to cascade

```
create table course (
(…
```

dept_name **varchar**(20),
foreign key (*dept_name*) **references** *department*
on delete cascade
on update cascade,
. . .)

- Instead of cascade we can use :
  - **set null**,
  - **set default**

Integrity Constraint Violation During Transactions

- Consider:

create table *person* (
ID **char**(10),
name **char**(40),
mother **char**(10),
father **char**(10),
primary key *ID,*
foreign key *father* **references** *person,*
foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - Insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking

Complex Check Conditions

- The predicate in the check clause can be an arbitrary predicate that can include a subquery.

check (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))

The check condition states that the time_slot_id in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation.

  - The condition has to be checked not only when a tuple is inserted or modified in *section* , but also when the relation *time_slot* changes

Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- The following constraints, can be expressed using assertions:
- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot
- An assertion in SQL takes the form:

create assertion <assertion-name> **check** (<predicate>);

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value

- Interval values can be added to date/time/timestamp values

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of character data
- When a query returns a large object, a pointer is returned rather than the large object itself.

User-Defined Types

- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

- Example:

  create table *department*
  (*dept_name* **varchar** (20),
  building **varchar** (15),
  budget Dollars);

Domains

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- Example:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Index Creation

- Many queries reference only a small proportion of the records in a table.
- It is inefficient for the system to read every record to find a record with particular value
- An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation.
- We create an index with the **create index** command

  create index <name> **on** <relation-name> (attribute);

- **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*))

- **create index** *studentID_index* **on** *student*(*ID*)
- The query:

  select * from *student* **where** *ID* = '12345; can be executed by using the index to find the required record, without looking at all records of *student*

Authorization

- We may assign a user several forms of authorizations on parts of the database.
  - **Read** - allows reading, but not modification of data.

- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

- Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.
- Forms of authorization to modify the database schema
  - **Index** - allows creation and deletion of indices.
  - **Resources** - allows creation of new relations.
  - **Alteration** - allows addition or deletion of attributes in a relation.
  - **Drop** - allows deletion of relations.

Authorization Specification in SQL

- The **grant** statement is used to confer authorization

  grant <privilege list> **on** <relation or view > **to** <user list>
- <user list> is:
  - a user-id
  - **public**, which allows all valid users the privilege granted
  - A role (more on this later)
- Example:
  - **grant select on** *department* **to** Amit, Satoshi
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
  - Example: grant users U_1, U_2, and U_3 **select** authorization on the *instructor* relation:

    grant select on *instructor* **to** U_1, U_2, U_3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

  revoke <privilege list> **on** <relation or view> **from** <user list>
- Example:

  revoke select on *student* **from** U_1, U_2, U_3
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If <revokee-list> includes **public,** all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Roles

- A **role** is a way to distinguish among various users as far as what these users can access/update in the database.
- To create a role we use:

  create a role <name>
- Example:
  - **create role** instructor
- Once a role is created we can assign "users" to the role using:
  - **grant** <role> **to** <users>

Roles Example

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - Creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;
  - Why is this required?
- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;
  - **revoke select on** *department* **from** Amit, Satoshi **cascade**;
  - **revoke select on** *department* **from** Amit, Satoshi **restrict**;

Advanced SQL

Accessing SQL from a Programming Language

A database programmer must have access to a general-purpose programming language for at least two reasons:

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server.
    - The SQL statements are translated at compile time into function calls.
    - At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
    - Open a connection
    - Create a "statement" object
    - Execute queries using the statement object to send queries and fetch results
    - Exception mechanism to handle errors

JDBC Code

```
public static void JDBCexample(String dbid, String userid, String passwd)
        {
    try (Connection conn = DriverManager.getConnection(
      "jdbc:oracle:thin:@db.yale.edu:2000:univdb", userid, passwd);
         Statement stmt = conn.createStatement();
    )
    {
         … Do Actual Work ….
    }
   catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
    }
    }
```

NOTE: Above syntax works with Java 7, and JDBC 4 onwards.
Resources opened in "try (….)" syntax ("try with resources") are automatically closed at the end of the try block

- Update to database

```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
} catch (SQLException sqle)
{
```

```
   System.out.println("Could not insert tuple. " + sqle);
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery(
                        "select dept_name, avg (salary)
                         from instructor
                         group by dept_name");
while (rset.next()) {
      System.out.println(rset.getString("dept_name") + " " +
                                    rset.getFloat(2));
}
```

JDBC SUBSECTIONS

- Connecting to the Database
- Shipping SQL Statements to the Database System
- Exceptions and Resource Management
- Retrieving the Result of a Query
- Prepared Statements
- Callable Statements
- Metadata Features
- Other Features
- Database Access from Python

JDBC Code Details

- Getting result fields:
  - rs.getString("dept_name") and rs.getString(1) equivalent if dept_name is the first argument of select result.
- Dealing with Null values
  int a = rs.getInt("a");
  if (rs.wasNull()) Systems.out.println("Got null value");

Prepared Statement

- PreparedStatement pStmt = conn.prepareStatement(
  "insert into instructor values(?,?,?,?)");
  pStmt.setString(1, "88877");
  pStmt.setString(2, "Perry");
  pStmt.setString(3, "Finance");
  pStmt.setInt(4, 125000);
  pStmt.executeUpdate();
  pStmt.setString(1, "88878");
  pStmt.executeUpdate();
- WARNING: always use prepared statements when taking an input from the user and adding it to a query
  - NEVER create a query by concatenating strings
  - "insert into instructor values(' " + ID + " ', ' " + name + " ', " + " ' + dept name + " ', " ' balance + ')"
  - What if name is "D'Souza"?

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = '" + name + "'"
- Suppose the user, instead of entering a name, enters:
  - X' or 'Y' = 'Y

- then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even used
    - X'; update instructor set salary = salary + 10000; --
- Prepared stament internally uses:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - **Always use prepared statements, with user inputs as parameters**

Metadata Features

- ResultSet metadata
- E.g. after executing query to get a ResultSet rs:
  - ResultSetMetaData rsmd = rs.getMetaData();

```
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnTypeName(i));
        }
```

- Database metadata
- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
// and Column-Pattern
// Returns: One row for each column; row has a number of attributes
// such as COLUMN_NAME, TYPE_NAME
// The value null indicates all Catalogs/Schemas.
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
while( rs.next()) {
        System.out.println(rs.getString("COLUMN_NAME"),
                                    rs.getString("TYPE_NAME");
}
```

- DatabaseMetaData dbmd = conn.getMetaData();

```
// Arguments to getTables: Catalog, Schema-pattern, Table-pattern, and Table-Type
// Returns: One row for each table; row has a number of attributes
// such as TABLE_NAME, TABLE_CAT, TABLE_TYPE, ..
// The value null indicates all Catalogs/Schemas.
// The value "" indicates current catalog/schema
// The value "%" has the same meaning as SQL like clause
// The last attribute is an array of types of tables to return.
//    TABLE means only regular tables
ResultSet rs = dbmd.getTables ("", "", "%", new String[] {"TABLES"});
while( rs.next()) {
        System.out.println(rs.getString("TABLE_NAME"));
}
```

Finding Primary Keys

- DatabaseMetaData dmd = connection.getMetaData();

```
// Arguments below are:  Catalog, Schema, and Table
// The value ""  for Catalog/Schema indicates current catalog/schema
//  The value null indicates all catalogs/schemas
ResultSet rs = dmd.getPrimaryKeys("", "", tableName);
```

```
while(rs.next()){
    // KEY_SEQ indicates the position of the attribute in
    // the primary key, which is required if a primary key has multiple
    // attributes
    System.out.println(rs.getString("KEY_SEQ"),
                        rs.getString("COLUMN_NAME");
}
```

Transaction Control in JDBC

- By default, each SQL statement is treated as a separate transaction that is committed automatically
  - bad idea for transactions with multiple updates
- Can turn off automatic commit on a connection
  - conn.setAutoCommit(false);
- Transactions must then be committed or rolled back explicitly
  - conn.commit(); or
  - conn.rollback();
- conn.setAutoCommit(true) turns on automatic commit.

Other JDBC Features

- Calling functions and procedures
  - CallableStatement cStmt1 = conn.prepareCall("{? = call some function(?)}");
  - CallableStatement cStmt2 = conn.prepareCall("{call some procedure(?,?)}");
- Handling large object types
  - getBlob() and getClob() that are similar to the getString() method, but return objects of type Blob and Clob, respectively
  - get data from these objects by getBytes()
  - associate an open stream with Java Blob or Clob object to update large objects
    - blob.setBlob(int parameterIndex, InputStream inputStream).

SQLJ

- JDBC is overly dynamic, errors cannot be caught by compiler
- SQLJ: embedded SQL in Java
  - #sql iterator deptInfoIter (String dept name, int avgSal);

```
deptInfoIter iter = null;
#sql iter = { select dept_name, avg(salary) from instructor
            group by dept name };
while (iter.next()) {
        String deptName = iter.dept_name();
    int avgSal = iter.avgSal();
    System.out.println(deptName + " " + avgSal);
}
iter.close();
```

ODBC

- Open DataBase Connectivity (ODBC) standard
  - standard for application program to communicate with a database server.
  - application program interface (API) to
    - open a connection with a database,
    - send queries and updates,
    - get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as C, C++, Java, Fortran, and PL/1,
- A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R embedding of SQL into PL/1.
- **EXEC SQL** statement is used in the host language to identify embedded SQL request to the preprocessor

  EXEC SQL <embedded SQL statement >;

Note: this varies by language:

  - In some languages, like COBOL, the semicolon is replaced with END-EXEC
  - In Java embedding uses # SQL { …. };

- Before executing any SQL statements, the program must first connect to the database. This is done using:

  EXEC-SQL **connect to** *server* **user** *user-name* **using** *password*;

  Here, *server* identifies the server to which a connection is to be established.
- Variables of the host language can be used within embedded SQL statements. They are preceded by a colon (:) to distinguish from SQL variables (e.g., :*credit_amount)*
- Variables used as above must be declared within DECLARE section, as illustrated below. The syntax for declaring the variables, however, follows the usual host language syntax.

  EXEC-SQL BEGIN DECLARE SECTION}

  int *credit-amount* ;

  EXEC-SQL END DECLARE SECTION;
- To write an embedded SQL query, we use the

  declare *c* **cursor for <SQL query>**

  statement. The variable *c* is used to identify the query
- Example:
  - From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount in the host langue
  - Specify the query in SQL as follows:

    EXEC SQL

    declare *c* **cursor for**

    select *ID, name*

    from *student*

    where tot_cred > *:credit_amount*

    END_EXEC
- The **open** statement for our example is as follows:

  EXEC SQL **open** *c* ;

  This statement causes the database system to execute the query and to save the results within a temporary relation. The query uses the value of the host-language variable *credit-amount* at the time the **open** statement is executed.
- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** :*si, :sn* END_EXEC

  Repeated calls to fetch get successive tuples in the query result
- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* ;

  Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Updates Through Embedded SQL

- Embedded SQL expressions for database modification (**update**, **insert**, and **delete**)
- Can update tuples fetched by cursor by declaring that the cursor is for update

```
EXEC SQL
declare c cursor for
        select *
        from instructor
        where dept_name = 'Music'
        for update
```

- We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code:

```
    update instructor
        set salary = salary + 1000
        where current of c
```

Functions and Procedures

- Functions and procedures allow "business logic" to be stored in the database and executed from SQL statements.
- These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++.
- The syntax we present here is defined by the SQL standard.
  - Most databases implement nonstandard versions of this syntax.

Declaring SQL Functions

- Define a function that, given the name of a department, returns the count of the number of instructors in that department.

```
  create function dept_count (dept_name varchar(20))
     returns integer
    begin
    declare d_count integer;
          select count (*) into d_count
         from instructor
         where instructor.dept_name = dept_name
    return d_count;
end
```

- The function *dept*_count can be used to find the department names and budget of all departments with more that 12 instructors.

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

Table Functions

- The SQL standard supports functions that can return tables as results; such functions are called table functions
- Example: Return all instructors in a given department

```
create function instructor_of (dept_name char(20))
        returns table  (
          ID varchar(5),
        name varchar(20),
      dept_name varchar(20),
        salary numeric(8,2))
       return table
       (select ID, name, dept_name, salary
```

```
        from instructor
        where instructor.dept_name = instructor_of.dept_name)
```

- Usage

```
select * from table (instructor_of ('Music'))
```

SQL Procedures

- The *dept_count* function could instead be written as procedure:

```
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
```

- The keywords in and out are parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.
- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.

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

- Procedures and functions can be invoked also from dynamic SQL
- SQL allows more than one procedure of the so long as the number of arguments of the procedures with the same name is different.
- The name, along with the number of arguments, is used to identify the procedure.

Language Constructs for Procedures & Functions

- SQL supports constructs that gives it almost all the power of a general-purpose programming language.
    - Warning: most database systems implement their own variant of the standard syntax below.
- Compound statement: begin … end,
    - May contain multiple SQL statements between begin and end.
    - Local variables can be declared within a compound statements
- While and repeat statements:
    - while boolean expression do
      ```
              sequence of statements ;
      end while
      ```
    - repeat
      ```
      sequence of statements ;
      until boolean expression
      end repeat
      ```
- For loop
    - Permits iteration over all results of a query
- Example: Find the budget of all departments

```
declare n integer default 0;
for r as
      select budget from department where dept_name = 'Music'
 do
            set n = n + r.budget
 end for
```

Language Constructs – if-then-else

- Conditional statements (if-then-else)

```
if boolean expression
        then statement or compound statement
elseif boolean expression
        then statement or compound statement
else statement or compound statement
        end if
```

Example procedure

- Registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if capacity is exceeded
  - See book (page 202) for details
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
…
end
```

- The statements between the begin and the end can raise an exception by executing "signal *out_of_classroom_seats*"
- The handler says that if the condition arises he action to be taken is to exit the enclosing the begin end statement.

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.
  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - For example, **after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as :** for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. For example, convert blank grades to null.

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

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger
- Risk of unintended execution of triggers, for example, when
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Recursive Queries

- SQL:1999 permits recursive view definition
- Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
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_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )
select ∗
from rec_prereq;
```

  This example view, *rec_prereq,* is called the *transitive closure* of the *prereq* relation
- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
  - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *prereq* with itself
    - This can give only a fixed number of levels of managers
    - Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work
    - Alternative: write a procedure to iterate as many times as required
      - See procedure *findAllPrereqs* in book
- Computing transitive closure using iteration, adding successive tuples to *rec_prereq*
  - The next slide shows a *prereq* relation

- Each step of the iterative process constructs an extended version of *rec_prereq* from its recursive definition.
- The final result is called the *fixed point* of the recursive view definition.

▪ Recursive views are required to be **monotonic**. That is, if we add tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more

Advanced Aggregation Features

Ranking

▪ Ranking is done in conjunction with an order by specification.

▪ Suppose we are given a relation
student_grades(ID, GPA)
giving the grade-point average of each student

▪ Find the rank of each student.

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

▪ An extra **order by** clause is needed to get them in sorted order

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

▪ Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
- **dense_rank** does not leave gaps, so next dense rank would be 2

▪ Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

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*;

▪ Ranking can be done within partition of the data.

▪ “Find the rank of students within each department.”

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

▪ Multiple **rank** clauses can occur in a single **select** clause.

▪ Ranking is done *after* applying **group by** clause/aggregation

▪ Can be used to find top-n results
- More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

▪ Other ranking functions:
- **percent_rank** (within partition, if partitioning is done)
- **cume_dist** (cumulative distribution)
  - ▪ fraction of tuples with preceding values
- **row_number** (non-deterministic in presence of duplicates)

▪ SQL:1999 permits the user to specify **nulls first** or **nulls last**

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

▪ For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.

▪ E.g.,

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

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: "Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day"
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

  select *date, **sum**(value)* **over**
  (**order by** *date* **between rows** 1 **preceding and** 1 **following**)
  from *sales*
- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**
  - **range between** 10 **preceding and current row**
    - All rows with values between current row value –10 to current value
  - **range interval** 10 **day preceding**
    - Not including current row
- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

  select *account_number, date_time*,
  sum (*value*) **over**
  (**partition by** *account_number*
  order by *date_time*
  rows unbounded preceding)
  as *balance*
  from *transaction*
  order by *account_number, date_time*

OLAP

Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

Example sales relation

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6

Cross Tabulation of *sales* **by** *item_name* **and** *color*

clothes_size **all**

color

item_name	dark	pastel	white	total
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pants	20	2	5	27
total	62	54	48	164

- The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

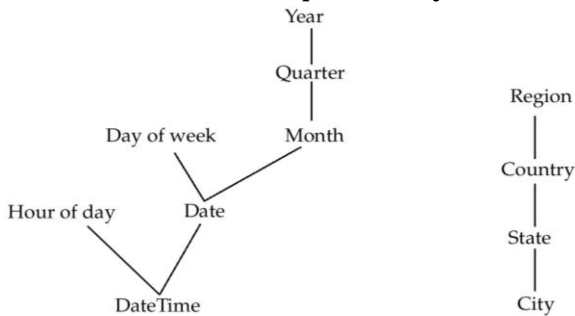
- A **data cube** is a multidimensional generalization of a cross-tab
- Can have n dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube

color	skirt	dress	shirt	pants	all
dark	8	20	14	20	62
pastel	35	10	7	2	54
white	10	8	28	5	48
all	53	38	49	27	164

item_name

Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
    - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
    - Can drill down or roll up on a hierarchy

clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
    - We use the value **all** is used to represent aggregates.
    - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.

item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	White	**all**	28
shirt	**all**	**all**	49
pant	dark	**all**	20
pant	pastel	**all**	2
pant	white	**all**	5
pant	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**'s on every subset of the specified attributes

- Example relation for this section
  sales(*item_name, color, clothes_size, quantity*)
- E.g., consider the query
  select *item_name, color, size,* **sum**(*number*)
  from *sales*
  group by cube(*item_name, color, size*)

This computes the union of eight different groupings of the *sales* relation:

{ (*item_name, color, size*), (*item_name, color*),
(*item_name, size*), (*color, size*),
(*item_name*), (*color*),
(*size*), ( ) }

where ( ) denotes an empty **group by** list.

- For each grouping, the result contains the null value
  for attributes not present in the grouping.

Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by
- **select** *item_name*, *color*, **sum**(*number*)
  from *sales*
  group by cube(*item_name, color*)
- The function **grouping()** can be applied on an attribute
  - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

  select *item_name, color, size*, **sum**(*number*),
  grouping(*item_name*) **as** *item_name_flag*,
  grouping(*color*) **as** *color_flag*,
  grouping(*size*) **as** *size_flag*,
  from *sales*
  group by cube(*item_name, color, size*)
- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item_name* in first query by
    decode(**grouping**(item_*name*), 1, 'all', *item_name*)
- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,
  select *item_name*, *color*, *size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name, color, size*)
  - Generates union of four groupings:
    { (*item_name, color, size*), (*item_name, color*), (*item_name*), ( ) }
- Rollup can be used to generate aggregates at multiple levels of a
  hierarchy.
- E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then
  select *category, item_name*, **sum**(*number*)
  from *sales, itemcategory*
  where *sales.item_name* = *itemcategory.item_name*
  group by rollup(*category, item_name*)
  would give a hierarchical summary by *item_name* and by *category.*
- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,
  select *item_name, color, size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name*), **rollup**(*color, size*)

generates the groupings
{item_name, ()} X {(color, size), (color), ()}
= { (*item_name, color, size*), (*item_name, color*), (*item_name*), (*color, size*), (*color*), ( ) }

- **Pivoting**: changing the dimensions used in a cross-tab is called
- **Slicing**: creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup**: moving from finer-granularity data to a coarser granularity
- **Drill down**: The opposite operation - that of moving from coarser-granularity data to finer-granularity data

OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.
- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - 2^n combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
      - For all but a few "non-decomposable" aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
  - Can compute aggregates on (*item_name, color, size*),
    (*item_name, color*) and (*item_name*) using a single sorting of the base data

Unit 3

Database Design and the E-R Model: Overview of the Design Process, The Entity-Relationship Model, Constraints, Removing Redundant Attributes in Entity Sets, Entity-Relationship Diagrams, Reduction to Relational Schemas, Entity-Relationship Design Issues.

Relational Database Design: Features of Good Relational Designs, Atomic Domains and First Normal Form, Decomposition Using Functional Dependencies, Functional-Dependency Theory, Algorithms for Decomposition, Decomposition Using Multi-valued Dependencies, More Normal Forms.

Design Phases

- Initial phase -- characterize fully the data needs of the prospective database users.
- Second phase -- choosing a data model
  - Applying the concepts of the chosen data model
  - Translating these requirements into a conceptual schema of the database.
  - A fully developed conceptual schema indicates the functional requirements of the enterprise.
    - Describe the kinds of operations (or transactions) that will be performed on the data.
- Final Phase -- Moving from an abstract data model to the implementation of the database
  - Logical Design – Deciding on the database schema.
    - Database design requires that we find a "good" collection of relation schemas.
    - Business decision – What attributes should we record in the database?
    - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
  - Physical Design – Deciding on the physical layout of the database

Design Alternatives

- In designing a database schema, we must ensure that we avoid two major pitfalls:
  - Redundancy: a bad design may result in repeat information.
    - Redundant representation of information may lead to data inconsistency among the various copies of information
  - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.
- Avoiding bad designs is not enough. There may be a large number of good designs from which we must choose.

Design Approaches

- Entity Relationship Model
  - Models an enterprise as a collection of *entities* and *relationships*
    - Entity: a "thing" or "object" in the enterprise that is distinguishable from other objects
      - Described by a set of *attributes*
    - Relationship: an association among several entities
  - Represented diagrammatically by an *entity-relationship diagram:*
- Normalization Theory
  - Formalize what designs are bad, and test for them

ER model -- Database Modeling

- The ER data mode was developed to facilitate database design by allowing specification of an **enterprise schema** that represents the overall logical structure of a database.
- The ER data model employs three basic concepts:
  - entity sets,
  - relationship sets,
  - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

Entity Sets

- An **entity** is an object that exists and is distinguishable from other objects.
  - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, trees, holidays
- An entity is represented by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
  - Example:

  instructor = (*ID, name, salary*)
  course= (*course_id, title, credits*)
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.

Entity Sets -- *instructor* and *student*

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

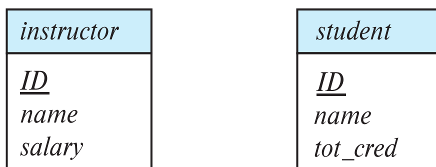
instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Representing Entity sets in ER Diagram

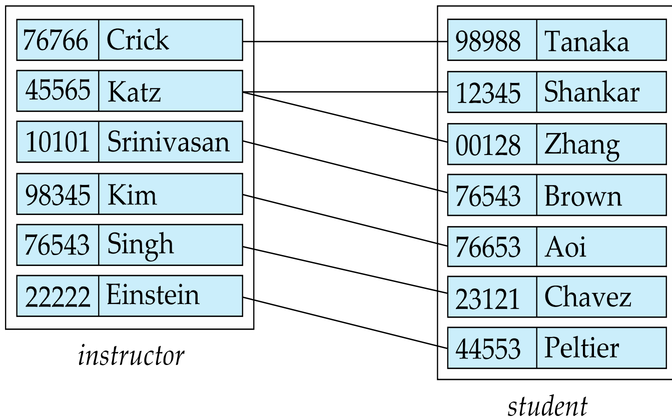
- Entity sets can be represented graphically as follows:
  - Rectangles represent entity sets.
  - Attributes listed inside entity rectangle
  - Underline indicates primary key attributes



Relationship Sets

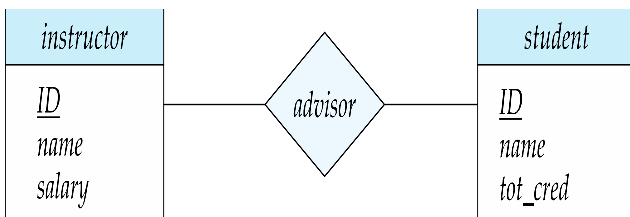
- A **relationship** is an association among several entities
  Example:
  44553 (Peltier)   *advisor*   22222 (Einstein)
  student entity   relationship set   *instructor* entity
- A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from entity sets
  $\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$
  where $(e_1, e_2, \ldots, e_n)$ is a relationship
  - Example:

    (44553,22222) $\in$ *advisor*
- Example: we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.
- Pictorially, we draw a line between related entities.



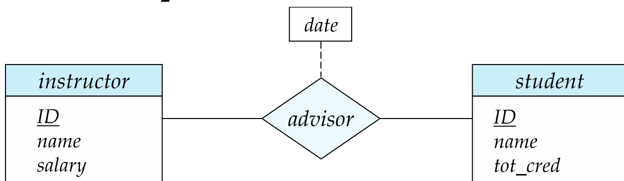
Representing Relationship Sets via ER Diagrams

Diamonds represent relationship sets.



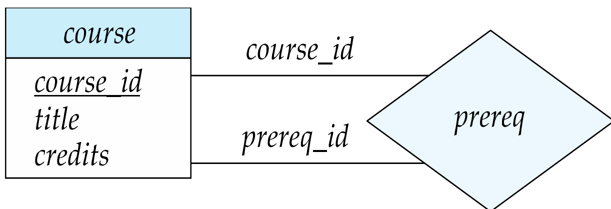
- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor

Relationship Sets with Attributes



Roles

- Entity sets of a relationship need not be distinct
  - Each occurrence of an entity set plays a "role" in the relationship
- The labels "*course_id*" and "*prereq_id*" are called **roles**.

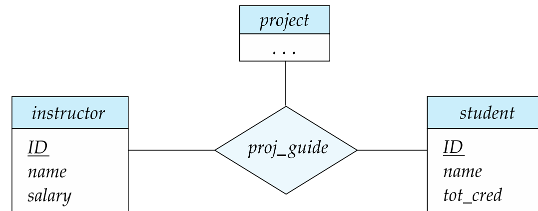


Degree of a Relationship Set

- Binary relationship
  - involve two entity sets (or degree two).
  - most relationship sets in a database system are binary.
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)
  - Example: *students* work on research *projects* under the guidance of an *instructor*.
  - relationship *proj_guide* is a ternary relationship between *instructor, student,* and *project*

Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship

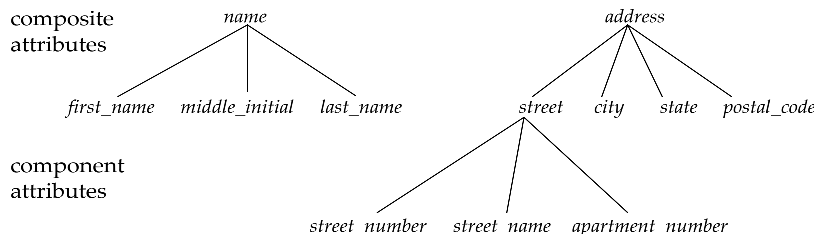


Complex Attributes

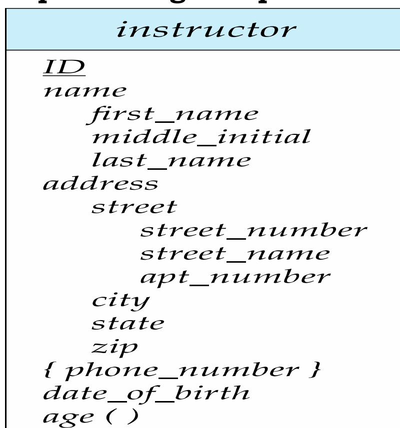
- Attribute types:
  - **Simple** and **composite** attributes.
  - **Single-valued** and **multivalued** attributes
    - Example: multivalued attribute: *phone_numbers*
  - **Derived** attributes
    - Can be computed from other attributes
    - Example: age, given date_of_birth
- **Domain** – the set of permitted values for each attribute

Composite Attributes

- Composite attributes allow us to divided attributes into subparts (other attributes).



Representing Complex Attributes in ER Diagram

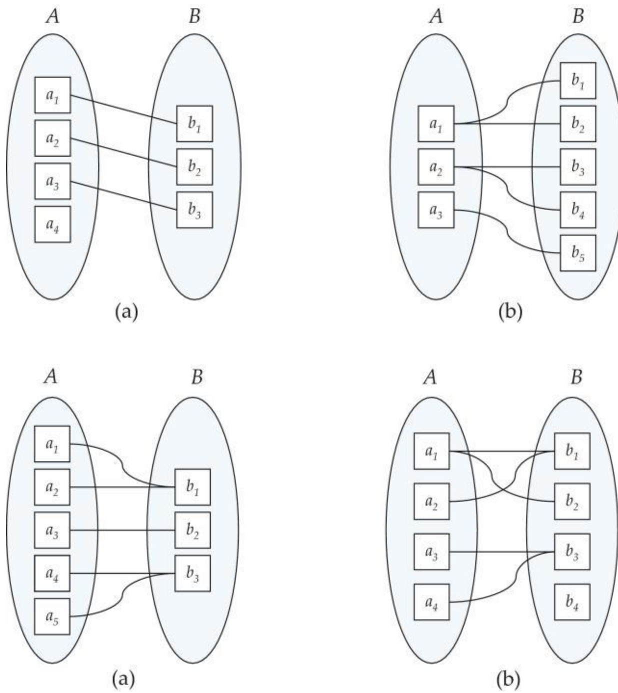


Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one

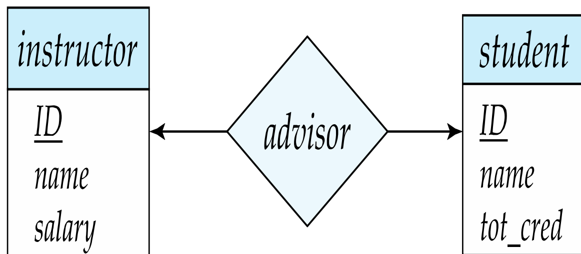
- Many to many

Mapping Cardinalities



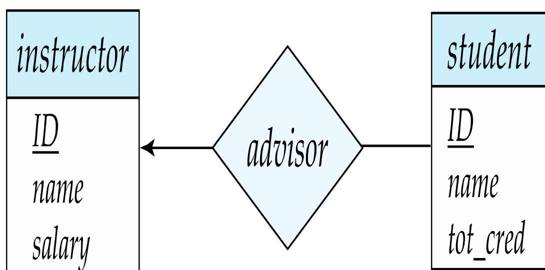
Representing Cardinality Constraints in ER Diagram

- We express cardinality constraints by drawing either a directed line (→), signifying "one," or an undirected line (—), signifying "many," between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
  - A student is associated with at most one *instructor* via the relationship *advisor*
  - A *student* is associated with at most one *department* via *stud_dept*



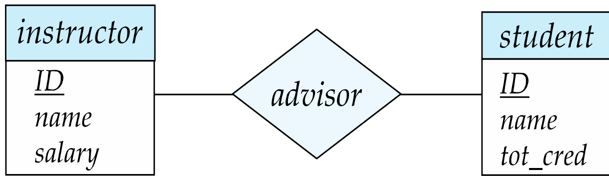
One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
  - an instructor is associated with several (including 0) students via *advisor*
  - a student is associated with at most one instructor via advisor,

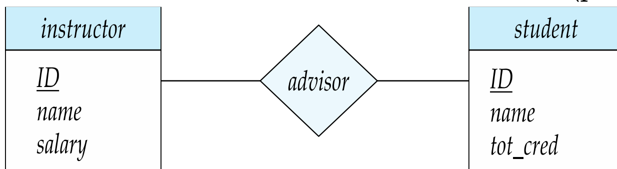


- In a many-to-one relationship between an *instructor* and a *student,*

- an instructor is associated with at most one student via *advisor*,
- and a student is associated with several (including 0) instructors via *advisor*



- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



Total and Partial Participation

- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  participation of *student* in *advisor* relation is total
    - every *student* must have an associated instructor
- **Partial participation**: some entities may not participate in any relationship in the relationship set
  - Example: participation of *instructor* in *advisor* is partial

Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form *l..h*, where *l* is the minimum and *h* the maximum cardinality
  - A minimum value of 1 indicates total participation.
  - A maximum value of 1 indicates that the entity participates in at most one relationship
  - A maximum value of * indicates no limit.
- Example
  - Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the meaning.
  - For example, a ternary relationship *R* between *A*, *B* and *C* with arrows to *B* and *C* could mean
    1. Each *A* entity is associated with a unique entity from B and *C* or
    2. Each pair of entities from (*A, B*) is associated with a unique C entity, and each pair (*A, C*) is associated with a unique *B*
  - Each alternative has been used in different formalisms
  - To avoid confusion we outlaw more than one arrow

Primary Key

- Primary keys provide a way to specify how entities and relations are distinguished. We will consider:
  - Entity sets
  - Relationship sets.
  - Weak entity sets

Primary key for Entity Sets

- By definition, individual entities are distinct.
- From database perspective, the differences among them must be expressed in terms of their attributes.
- The values of the attribute values of an entity must be such that they can uniquely identify the entity.
  - No two entities in an entity set are allowed to have exactly the same value for all attributes.
- A key for an entity is a set of attributes that suffice to distinguish entities from each other

Primary Key for Relationship Sets

- To distinguish among the various relationships of a relationship set we use the individual primary keys of the entities in the relationship set.
  - Let *R* be a relationship set involving entity sets E1, E2, .. En
  - The primary key for R is consists of the union of the primary keys of entity sets E1, E2, ..En
  - If the relationship set *R* has attributes a1, a2, .., am associated with it, then the primary key of *R* also includes the attributes a1, a2, .., am
- Example: relationship set "advisor".
  - The primary key consists of *instructor.ID* and s*tudent.ID*
- The choice of the primary key for a relationship set depends on the mapping cardinality of the relationship set.

Choice of Primary key for Binary Relationship

- Many-to-Many relationships. The preceding union of the primary keys is a minimal superkey and is chosen as the primary key.
- One-to-Many relationships . The primary key of the "Many" side is a minimal superkey and is used as the primary key.
- Many-to-one relationships. The primary key of the "Many" side is a minimal superkey and is used as the primary key.
- One-to-one relationships. The primary key of either one of the participating entity sets forms a minimal superkey, and either one can be chosen as the primary key.

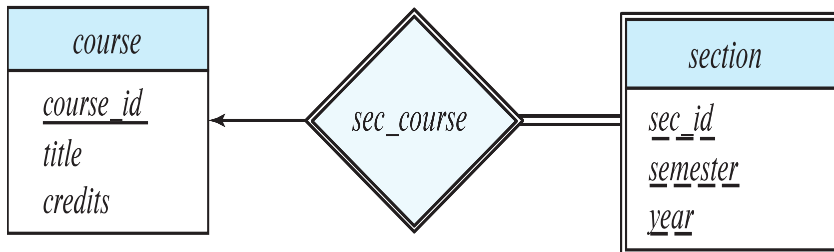
Weak Entity Sets

- Consider a *section* entity, which is uniquely identified by a *course_id*, *semester, year*, and *sec_id*.
- Clearly, section entities are related to course entities. Suppose we create a relationship set *sec_course* between entity sets *section* and *course*.
- Note that the information in *sec_course* is redundant, since *section* already has an attribute *course_id*, which identifies the course with which the section is related.
- One option to deal with this redundancy is to get rid of the relationship s*ec_course*; however, by doing so the relationship between *section* and *course* becomes implicit in an attribute, which is not desirable.
- An alternative way to deal with this redundancy is to not store the attribute *course_id* in the *section* entity and to only store the remaining attributes *section_id*, *year*, and *semester.*
  - However, the entity set *section* then does not have enough attributes to identify a particular *section* entity uniquely
- To deal with this problem, we treat the relationship *sec_course* as a special relationship that provides extra information, in this case, the *course_id*, required to identify *section* entities uniquely.
- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.
- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.

- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.
- Note that the relational schema we eventually create from the entity set *section* does have the attribute *course_id*, for reasons that will become clear later, even though we have dropped the attribute *course_id* from the entity set *section.*

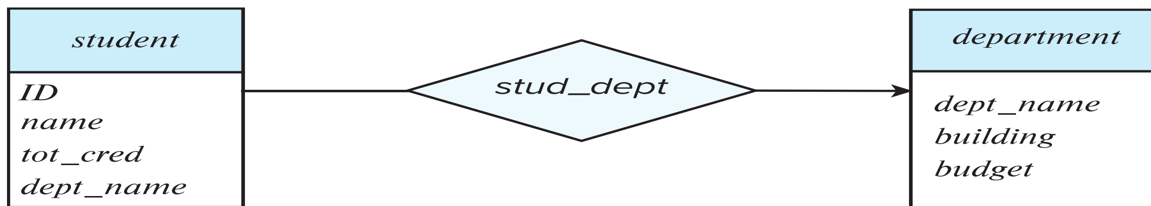
Expressing Weak Entity Sets

- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id, sec_id, semester, year*)



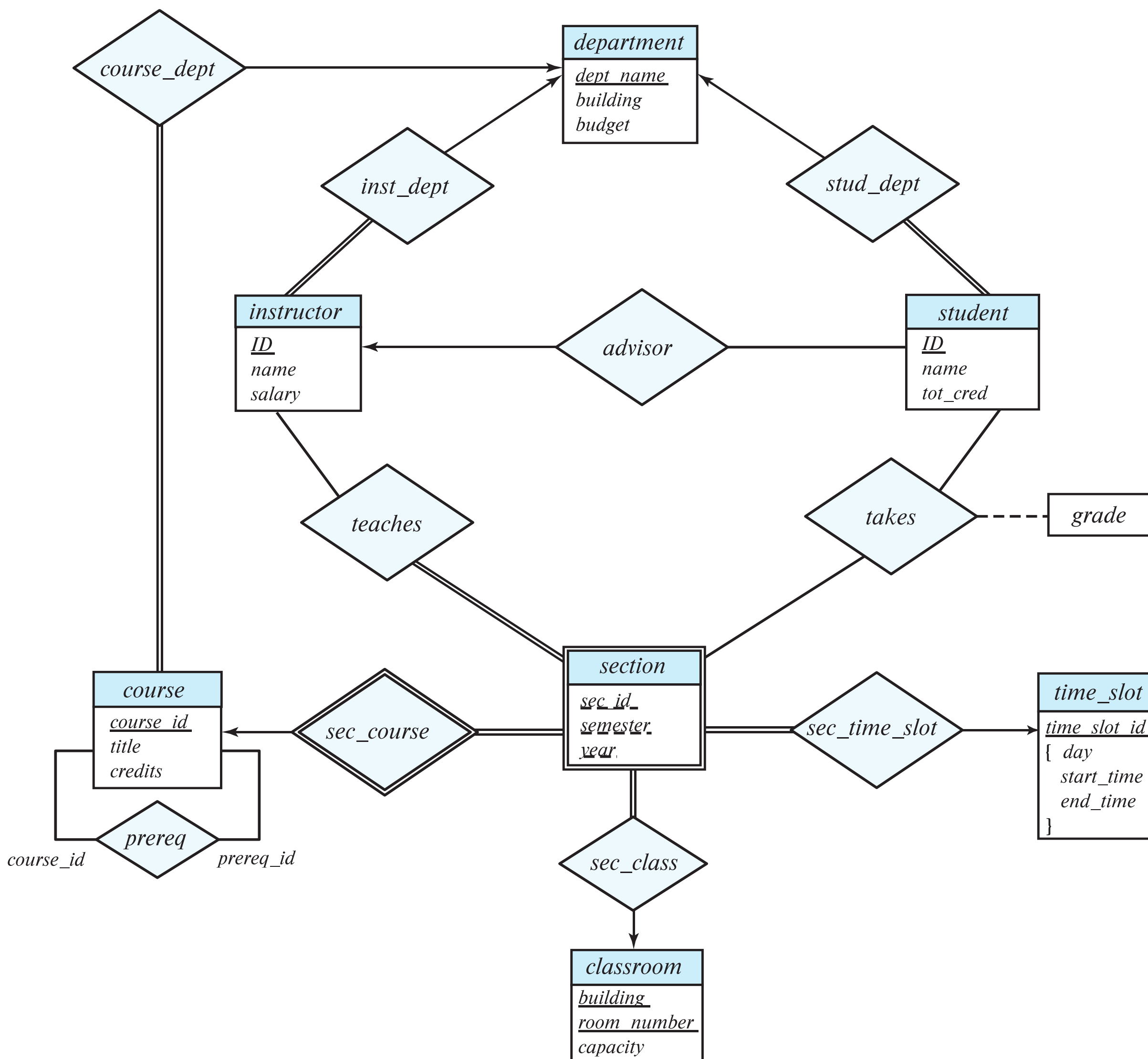
Redundant Attributes

- Suppose we have entity sets:
  - *student*, with attributes: *ID*, *name, tot_cred*, *dept_name*
  - *department,* with attributes: *dept_name, building, budget*
- We model the fact that each student has an associated department using a relationship set *stud_dept*
- The attribute *dept_name* in *student* below replicates information present in the relationship and is therefore redundant
  - and needs to be removed.
- BUT: when converting back to tables, in some cases the attribute gets reintroduced, as we will see later.



(a) Incorrect use of attribute

E-R Diagram for a University Enterprise



Reduction to Relation Schemas

- Entity sets and relationship sets can be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R 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.

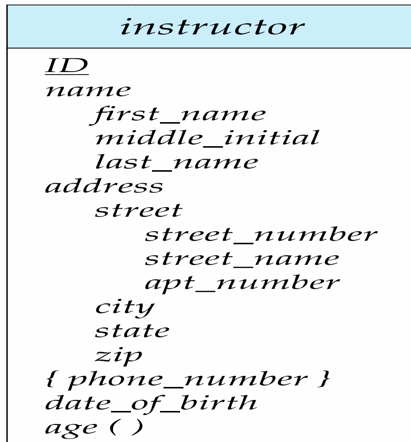
Representing Entity Sets

- 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)

Representation of Entity Sets with Composite Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*

- Prefix 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)



Representation of Entity Sets with Multivalued Attributes

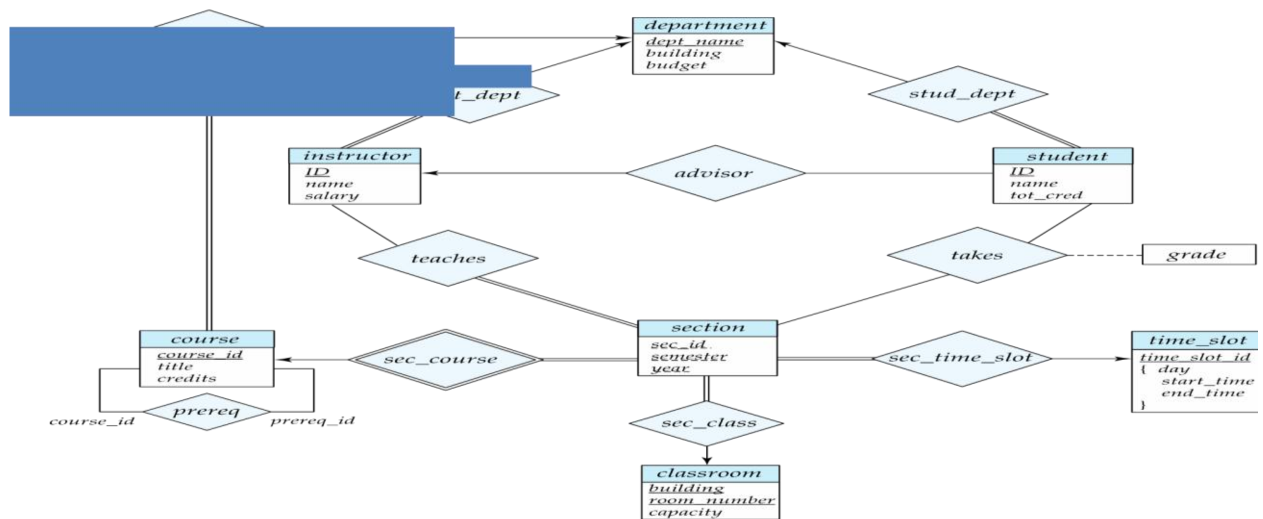
- A multivalued attribute *M* of an entity *E* is represented by a separate schema *EM*
- Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
- Example: Multivalued attribute *phone_number* of *instructor* is represented by a schema:
  inst_phone= (*ID*, *phone_number*)
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema *EM*
  - For example, an *instructor* entity with 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.
- Example: schema for relationship set *advisor*
  advisor = (*s_id, i_id*)

Redundancy of Schemas

- 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
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*
- Example



- For one-to-one relationship sets, either side can be chosen to act as the "many" side
  - That is, 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
- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
- Example: The *section* schema already contains the attributes that would appear in the *sec_course* schema

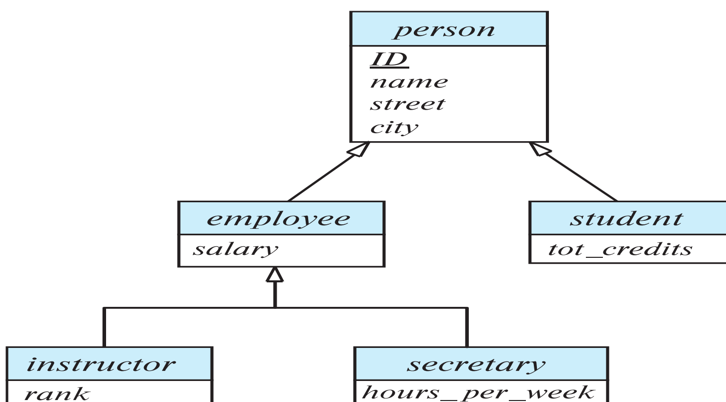
Extended E-R Features

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.
- Depicted by a *triangle* component labeled ISA (e.g., *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:
  - Form a schema for the higher-level entity
  - Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

  - 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:
  - Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

  - Drawback: *name, street* and *city* may be stored redundantly for people who are both students and employees

Generalization

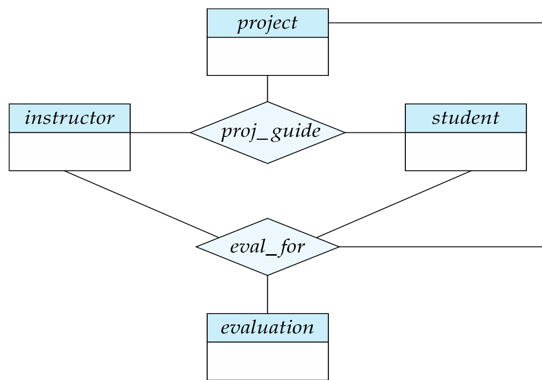
- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.

Completeness constraint

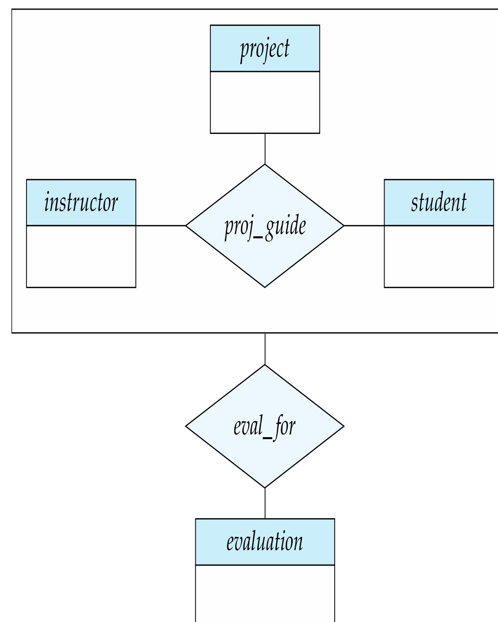
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
  - **total**: an entity must belong to one of the lower-level entity sets
  - **partial**: an entity need not belong 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 (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).
- The *student* generalization is total: All student entities must be either graduate or undergraduate. Because the higher-level entity set arrived at through generalization is generally composed of only those entities in the lower-level entity sets, the completeness constraint for a generalized higher-level entity set is usually total

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 can't discard the *proj_guide* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into 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



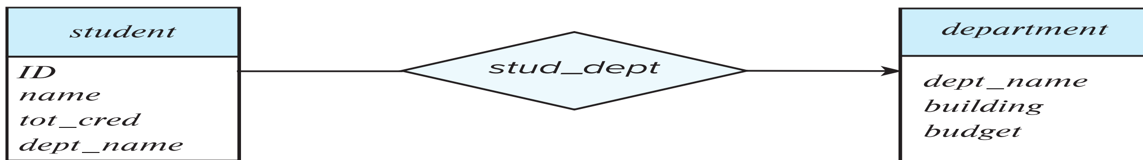
Reduction to Relational Schemas

- To represent aggregation, create a schema containing
  - Primary key of the aggregated relationship,
  - The primary key of the associated entity set
  - Any descriptive attributes
- In our example:
  - The schema *eval_for* is:

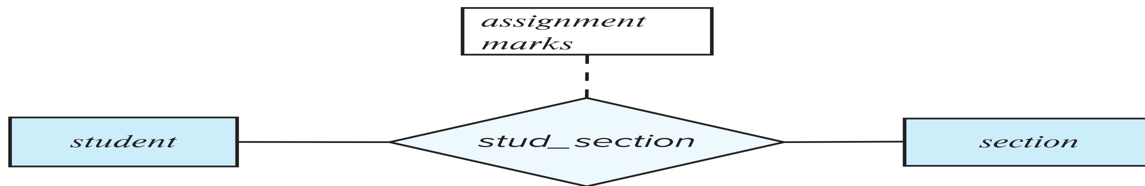
    eval_for (*s_ID, project_id, i_ID, evaluation_id*)
  - The schema *proj_guide* is redundant.

Design Issues

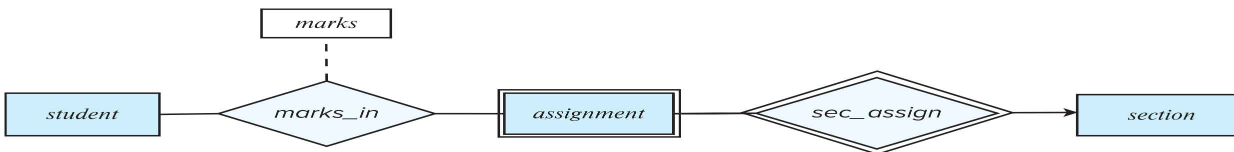
Common Mistakes in E-R Diagrams



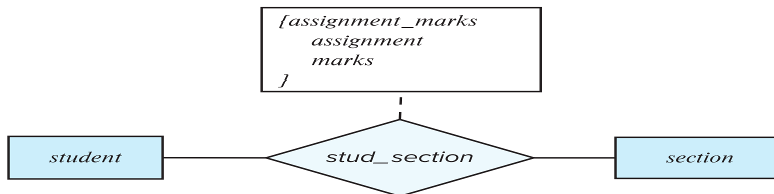
(a) Incorrect use of attribute



(b) Erroneous use of relationship attributes



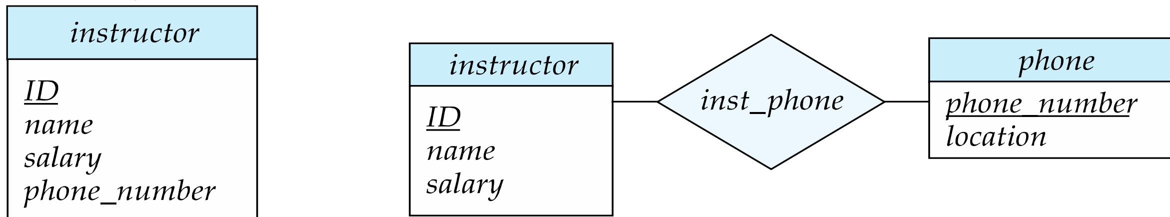
(c) Correct alternative to erroneous E-R diagram (b)



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

Entities vs. Attributes

- Use of entity sets vs. attributes

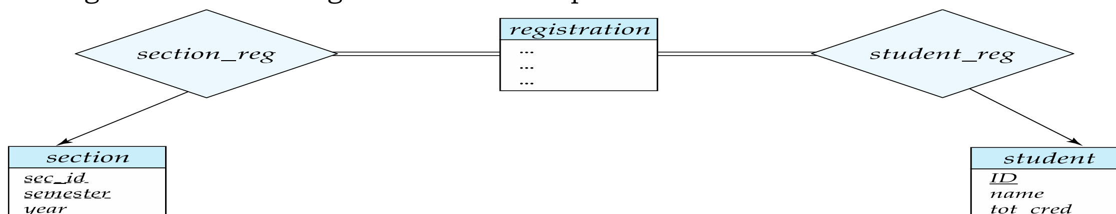


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities vs. Relationship sets

- **Use of entity sets vs. relationship sets**

Possible guideline is to designate a relationship set to describe an action that occurs between entities



- **Placement of relationship attributes**

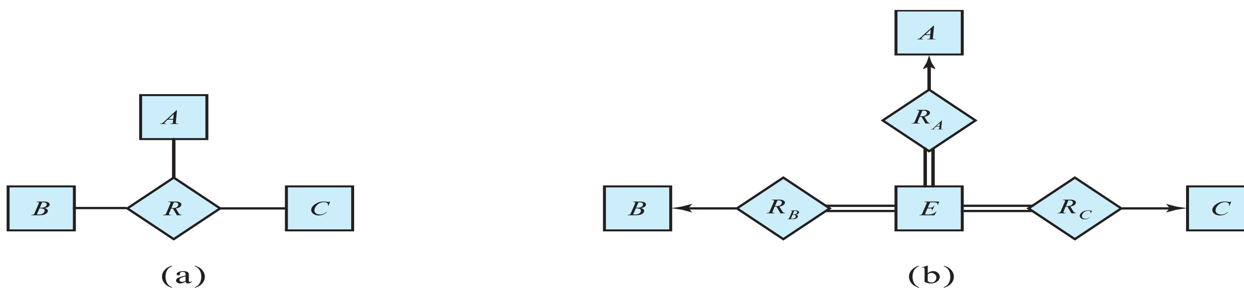
  For example, attribute date as attribute of advisor or as attribute of student

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
    - Example: *proj_guide*

Converting Non-Binary Relationships to Binary Form.

- 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. R_A, relating E and A 2. R_B, relating E and B 3. R_C, relating E and C
  - Create an identifying attribute for E *and* add any attributes of R to E
  - For each relationship (a_i , b_i , c_i) in R, create
    1. a new entity e_i in the entity set E 2. add (e_i , a_i) to R_A
    3. add (e_i , b_i) to R_B 4. add (e_i , c_i) to R_C



- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of R
    - Exercise: *add constraints to the relationships R_A, R_B and R_C* to ensure that a newly created entity corresponds to exactly one entity in each of entity sets A, B and C
  - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

E-R 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.

Summary of Symbols Used in E-R Notation

entity set

relationship set

identifying relationship set for weak entity set

total participation of entity set in relationship

many-to-many relationship

one-to-one relationship

role indicator

total (disjoint) generalization

attributes: simple (A1), composite (A2) and multivalued (A3) derived (A4)

primary key

discriminating attribute of weak entity set

many-to-one relationship

cardinality limits

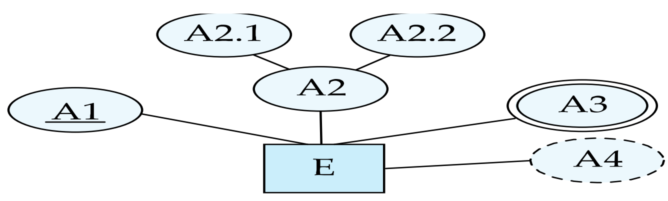
ISA: generalization or specialization

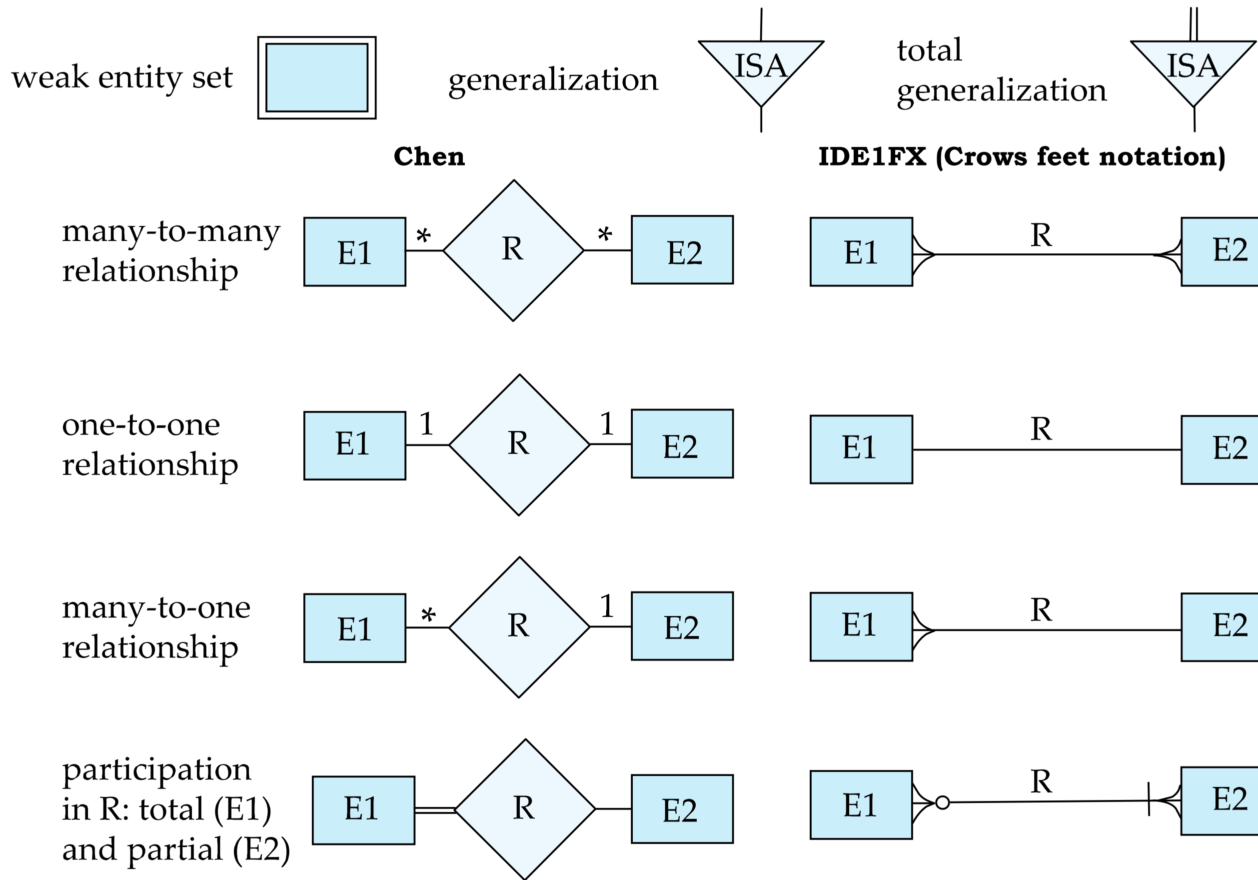
disjoint generalization

Alternative ER Notations

- Chen, IDE1FX, …

entity set E with simple attribute A1, composite attribute A2, multivalued attribute A3, derived attribute A4, and primary key A1



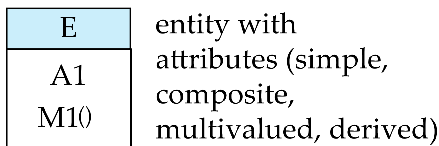


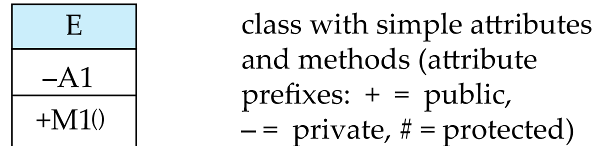
UML

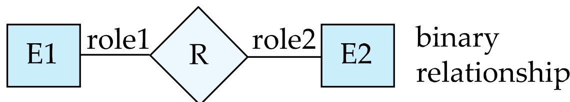
- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

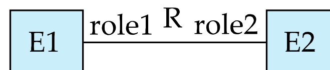
ER vs. UML Class Diagrams

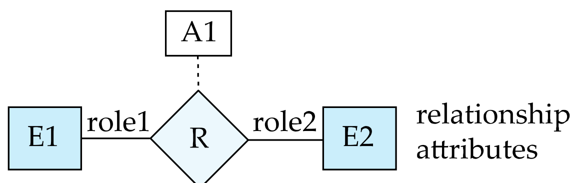
ER Diagram Notation **Equivalent in UML**

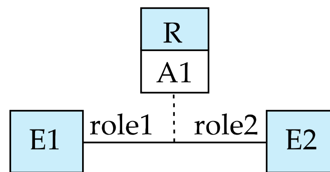


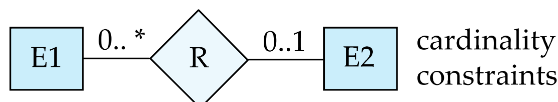


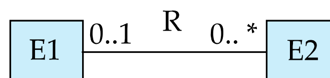


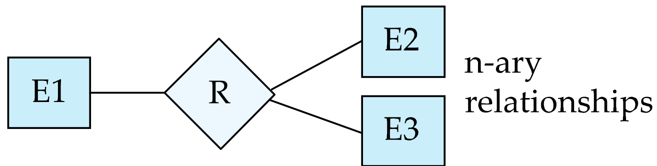


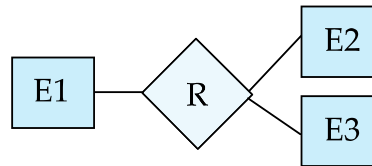


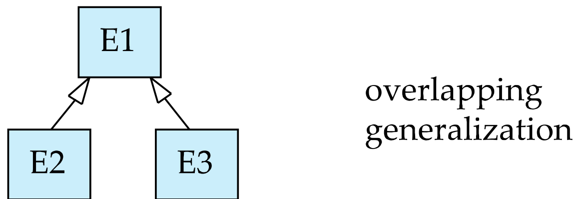


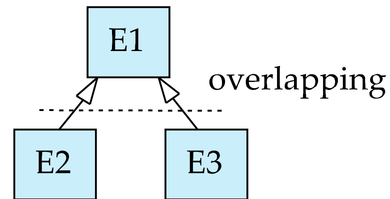


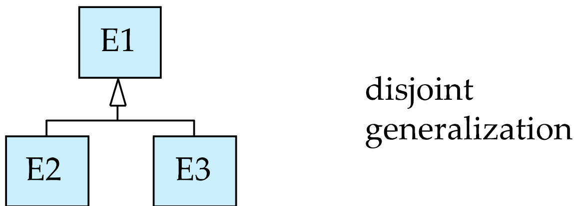


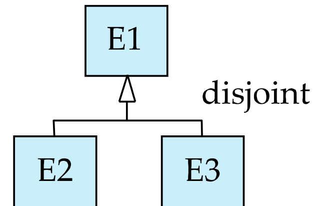












- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.

Other Aspects of Database Design

- Functional Requirements
- Data Flow, Workflow
- Schema Evolution

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
- Need to use null values (if we add a new department with no instructors)

A Combined Schema Without Repetition

Not all combined schemas result in repetition of information

- Consider combining relations
  - *sec_class(sec_id, building, room_number)* and
  - *section(course_id, sec_id, semester, year)*

into one relation

  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

Decomposition

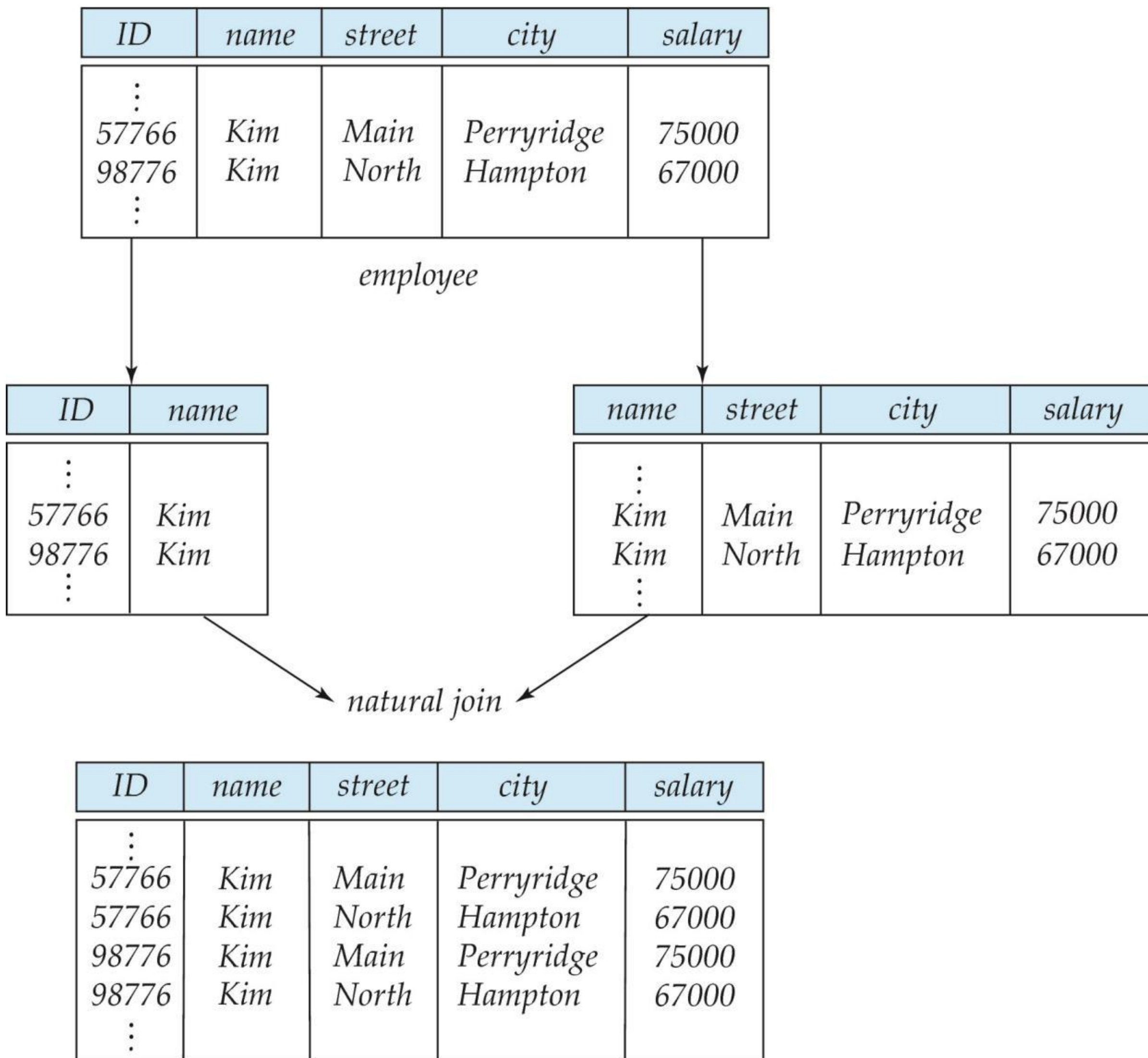
- The only way to avoid the repetition-of-information problem in the i*n_dep* schema is to decompose it into two schemas – instructor and *department* schemas.
- Not all decompositions are good. Suppose we decompose

employee(ID, name, street, city, salary)
into
employee1 (*ID, name*)
employee2 (*name, street, city, salary*)

The problem arises when we have two employees with the same name

- We cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition



ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

employee

ID	name
⋮	
57766	Kim
98776	Kim
⋮	

name	street	city	salary
⋮			
Kim	Main	Perryridge	75000
Kim	North	Hampton	67000
⋮			

natural join

ID	name	street	city	salary
⋮				
57766	Kim	Main	Perryridge	75000
57766	Kim	North	Hampton	67000
98776	Kim	Main	Perryridge	75000
98776	Kim	North	Hampton	67000
⋮				

Lossless Decomposition

- Let R be a relation schema and let R_1 and R_2 form a decomposition of R . That is R = $R_1 \cup R_2$
- We say that the decomposition is a **lossless decomposition** if there is no loss of information by replacing R with the two relation schemas $R_1 \cup R_2$
- Formally,

  $\Pi_{R1}(r) \quad \Pi_{R2}(r) = r$
- And, conversely a decomposition is lossy if

  $r \subset \Pi_{R1}(r) \quad \Pi_{R2}(r) = r$

Example of Lossless Decomposition

- Decomposition of $R = (A, B, C)$

  $R_1 = (A, B) \qquad R_2 = (B, C)$

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

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 set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - Each relation is in good form
  - The decomposition is a lossless decomposition
- Our theory is based on:
  - Functional dependencies
  - Multivalued dependencies

What is Normalization?

Normalization is the process of organizing the data in the database.

Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.

Normalization divides the larger table into smaller and links them using relationships.

Why do we need Normalization?

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure

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
- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- 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 t_1 and t_2 of *r* agree on the attributes α, they also agree on the attributes *β*. That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: 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 $\boldsymbol{F^+}$.

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. Consider the schema:

  in_dep (*ID, name, salary, dept_name, building, budget*).

  We expect these functional dependencies to hold:

  dept_name → *building*

  ID → *building*

  but would not expect the following to hold:

  dept_name → *salary*

Use of Functional Dependencies

- We use functional dependencies to:
  - 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.
  - For example, 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
- Example*:*
  - *ID, name* → *ID*
  - *name* → *name*
- In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Lossless Decomposition

- We can use functional dependencies to show when certain decomposition are lossless.
- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

  $r = \prod_{R1}(r) \quad \prod_{R2}(r)$
- A decomposition of R into R_1 and R_2 is lossless decomposition if at least one of the following dependencies is in F^+:
  - $R_1 \cap R_2 \rightarrow R_1$
  - $R_1 \cap R_2 \rightarrow R_2$
- 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

- $R = (A, B, C)$

  $F = \{A \rightarrow B, B \rightarrow C)$
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless decomposition:

  $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
- *Note:*
  - $B \rightarrow BC$

  is a shorthand notation for
  - $B \rightarrow \{B, C\}$

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 is possible that it is no longer possible to do the testing without having to perform a Cartesian Produced.
- A decomposition that makes it computationally hard to enforce functional dependency is said to be NOT **dependency preserving**.

Dependency Preservation Example

- Consider a schema:

  dept_advisor(s_ID, i_ID, department_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 composition NOT be dependency preserving

Normal Forms

Database normalization is the process of organizing the attributes of the database to reduce or eliminate data redundancy (having the same data but at different places).

Problems because of data redundancy

Data redundancy unnecessarily increases the size of the database as the same data is repeated in many places. Inconsistency problems also arise during insert, delete and update operations.

Functional Dependency

Functional Dependency is a constraint between two sets of attributes in relation to a database. A functional dependency is denoted by an arrow (→).
If an attribute A functionally determines B, then it is written as A → B.
For example, employee_id → name means employee_id functionally determines the name of the employee. As another example in a timetable database, {student_id, time} → {lecture_room}, student ID and time determine the lecture room where the student should be.

What does functionally dependent mean?
A function dependency A → B means for all instances of a particular value of A, there is the same value of B.
For example in the below table A → B is true, but B → A is not true as there are different values of A for B = 3.

A	B
1	3
2	3
4	0
1	3
4	0

Trivial Functional Dependency

X → Y is trivial only when Y is subset of X.

Examples
ABC → AB
ABC → A
ABC → ABC

Non Trivial Functional Dependencies

X → Y is a non trivial functional dependency when Y is not a subset of X.
X → Y is called completely non-trivial when X intersects Y is NULL.

Example:
Id → Name,
Name → DOB

Semi Non Trivial Functional Dependencies

X → Y is called semi non-trivial when X intersect Y is not NULL.
Examples:

AB → BC,
AD → DC

Normalization is the process of minimizing redundancy from a relation or set of relations. Redundancy in relation may cause insertion, deletion, and update anomalies. So, it helps to minimize the redundancy in relations. Normal forms are used to eliminate or reduce redundancy in database tables.

1. First Normal Form –

If a relation contain composite or multi-valued attribute, it violates first normal form or a relation is in first normal form if it does not contain any composite or multi-valued attribute. A relation is in first normal form if every attribute in that relation is singled valued attribute.

Example 1 – Relation STUDENT in table 1 is not in 1NF because of multi-valued attribute STUD_PHONE. Its decomposition into 1NF has been shown in table 2.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

Conversion to first normal form

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	INDIA
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

Example 2 –

ID	Name	Courses
1	A	c1, c2
2	E	c3
3	M	C2, c3

In the above table Course is a multi-valued attribute so it is not in 1NF.

Below Table is in 1NF as there is no multi-valued attribute

ID	Name	Course
1	A	c1
1	A	c2
2	E	c3
3	M	c2
3	M	c3

2. Second Normal Form –

To be in second normal form, a relation must be in first normal form and relation must not contain any partial dependency. A relation is in 2NF if it has **No Partial Dependency,** i.e., no non-prime attribute (attributes which are not part of any candidate key) is dependent on any proper subset of any candidate key of the table.

Partial Dependency – If the proper subset of candidate key determines non-prime attribute, it is called partial dependency.

Example 1 – Consider table-3 as following below.

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

{Note that, there are many courses having the same course fee.}
Here,
COURSE_FEE cannot alone decide the value of COURSE_NO or STUD_NO;
COURSE_FEE together with STUD_NO cannot decide the value of COURSE_NO;
COURSE_FEE together with COURSE_NO cannot decide the value of STUD_NO;
Hence,
COURSE_FEE would be a non-prime attribute, as it does not belong to the one only candidate key {STUD_NO, COURSE_NO} ;

But, COURSE_NO -> COURSE_FEE, i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF.

To convert the above relation to 2NF, we need to split the table into two tables such as :
Table 1: STUD_NO, COURSE_NO
Table 2: COURSE_NO, COURSE_FEE

Table 1

STUD_NO	COURSE_NO
1	C1
2	C2
1	C4
4	C3
4	C1
2	C5

Table 2

COURSE_NO	COURSE_FEE
C1	1000
C2	1500
C3	1000
C4	2000
C5	2000

NOTE: 2NF tries to reduce the redundant data getting stored in memory. For instance, if there are 100 students taking C1 course, we don't need to store its Fee as 1000 for all the 100 records, instead, once we can store it in the second table as the course fee for C1 is 1000.

Example 2 – Consider following functional dependencies in relation R (A, B , C, D)

AB -> C [A and B together determine C]
BC -> D [B and C together determine D]

In the above relation, AB is the only candidate key and there is no partial dependency, i.e., any proper subset of AB doesn't determine any non-prime attribute.

3. Third Normal Form –

A relation is in third normal form, if there is **no transitive dependency** for non-prime attributes as well as it is in second normal form.
A relation is in 3NF if **at least one of the following condition holds** in every non-trivial function dependency X –> Y

1. X is a super key.
2. Y is a prime attribute (each element of Y is part of some candidate key).

STUD_NO	STUD_NAME	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	HARYANA	INDIA	20
2	RAM	PUNJAB	INDIA	19
3	SURESH	PUNJAB	INDIA	21

Table 4

Transitive dependency – If A->B and B->C are two FDs then A->C is called transitive dependency.

Example 1 – In relation STUDENT given in Table 4,
FD set: {STUD_NO -> STUD_NAME, STUD_NO -> STUD_STATE, STUD_STATE -> STUD_COUNTRY, STUD_NO -> STUD_AGE}
Candidate Key: {STUD_NO}

For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true. So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY_STUD_AGE) as:
STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_AGE)
STATE_COUNTRY (STATE, COUNTRY)

Example 2 – Consider relation R(A, B, C, D, E)
A -> BC,
CD -> E,
B -> D,
E -> A

All possible candidate keys in above relation are {A, E, CD, BC} All attributes are on right sides of all functional dependencies are prime.

4. Boyce-Codd Normal Form (BCNF) –

A relation R is in BCNF if R is in Third Normal Form and for every FD, LHS is super key. A relation is in BCNF iff in every non-trivial functional dependency X –> Y, X is a super key.

Example 1 – Find the highest normal form of a relation R(A,B,C,D,E) with FD set as {BC->D, AC->BE, B->E}
Step 1. As we can see, (AC)+ ={A,C,B,E,D} but none of its subset can determine all attribute of relation, So AC will be candidate key. A or C can't be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
Step 2. Prime attributes are those attributes that are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example.
Step 3. The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attribute.
The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).
The relation is not in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or RHS should be prime attribute.
So the highest normal form of relation will be 2nd Normal form.

Example 2 –For example consider relation R(A, B, C)
A -> BC,
B ->
A and B both are super keys so above relation is in BCNF.

Key Points –

1. BCNF is free from redundancy.
2. If a relation is in BCNF, then 3NF is also satisfied.
3. If all attributes of relation are prime attribute, then the relation is always in 3NF.
4. A relation in a Relational Database is always and at least in 1NF form.
5. Every Binary Relation (a Relation with only 2 attributes) is always in BCNF.
6. If a Relation has only singleton candidate keys(i.e. every candidate key consists of only 1 attribute), then the Relation is always in 2NF(because no Partial functional dependency possible).
7. Sometimes going for BCNF form may not preserve functional dependency. In that case go for BCNF only if the lost FD(s) is not required, else normalize till 3NF only.
8. There are many more Normal forms that exist after BCNF, like 4NF and more. But in real world database systems it's generally not required to go beyond BCNF.

Exercise 1: Find the highest normal form in R (A, B, C, D, E) under following functional dependencies.
ABC --> D
CD --> AE

Important Points for solving above type of question.
1) It is always a good idea to start checking from BCNF, then 3 NF, and so on.
2) If any functional dependency satisfied a normal form then there is no need to check for lower normal form. For example, ABC –> D is in BCNF (Note that ABC is a superkey), so no need to check this dependency for lower normal forms.
Candidate keys in the given relation are {ABC, BCD}
BCNF: ABC -> D is in BCNF. Let us check CD -> AE, CD is not a super key so this dependency is not in BCNF. So, R is not in BCNF.
3NF: ABC -> D we don't need to check for this dependency as it already satisfied BCNF. Let us consider CD -> AE. Since E is not a prime attribute, so the relation is not in 3NF.
2NF: In 2NF, we need to check for partial dependency. CD is a proper subset of a candidate key and it determines E, which is non-prime attribute. So, given relation is also not in 2 NF. So, the highest normal form is 1 NF.

Problems:
1. Relation R has eight attributes ABCDEFGH. Fields of R contain only atomic values. F = {CH -> G, A -> BC, B -> CFH, E -> A, F -> EG} is a set of functional dependencies (FDs) so that F+ is exactly the set of FDs that hold for R. How many candidate keys does the relation R have?
Hint:
A+ is ABCEFGH which is all attributes except D.
B+ is also ABCEFGH which is all attributes except D.
E+ is also ABCEFGH which is all attributes except D.
F+ is also ABCEFGH which is all attributes except D.
So there are total 4 candidate keys AD, BD, ED and FD.

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

  $\alpha \rightarrow \beta$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a superkey for R
- Example schema that is ***not*** in BCNF:

  in_dep (<u>*ID,*</u> *name, salary,* <u>*dept_name,*</u> *building, budget*)

  because :

  - *dept_name*→ *building, budget*
    - holds on *in_dep*
    - but
  - *dept_name* is not a superkey
- When decompose *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 $\alpha \rightarrow \beta$ be the FD that causes a violation of BCNF.
- We decompose R into:
  - $(\alpha \cup \beta)$
  - $(R - (\beta - \alpha))$
- In our example of *in_dep*,
  - α = *dept_name*
  - β = *building, budget*

  and *in_dep* is replaced by

  - $(\alpha \cup \beta)$ = (*dept_name, building, budget*)
  - $(R - (\beta - \alpha))$ = (*ID, name, dept_name, salary*)

Example

- $R = (A, B, C)$

  $F = \{A \rightarrow B, B \rightarrow C)$
- $R_1 = (A, B)$, $R_2 = (B, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B)$, $R_2 = (A, C)$
  - Lossless-join decomposition:

    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving

    (cannot check $B \rightarrow C$ without computing R_1 R_2)

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, department_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.

- Any decomposition of *dept_advisor* will not include all the attributes in *s_ID, dept_name* $\rightarrow$ *i_ID*
- Thus, the composition is NOT be dependency preserving

Third Normal Form

- A relation schema *R* is in **third normal form (3NF)** if for all:

  $\alpha \rightarrow \beta$ in F^+

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for *R*
- Each attribute *A* in $\beta - \alpha$ 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).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation

3NF Example

- Consider a schema:
  dept_advisor(s_ID, i_ID, dept_name)
- With function dependencies:
  i_ID $\rightarrow$ *dept_name*
  s_ID, dept_name $\rightarrow$ *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* $\rightarrow$ *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
  - $R = (J, K, L)$
  - $F = \{JK \rightarrow L, L \rightarrow K\}$
  - And an instance table:

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
  - Need to use null values (e.g., to represent the relationship l_2, k_2 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 $\{R_1, R_2, ..., R_n\}$ 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)
  - where an instructor may have more than one phone and can have multiple children
  - 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 are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., 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).

Functional-Dependency Theory

- 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 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^+.
- We can compute F^+, the closure of F, by repeatedly applying **Armstrong's Axioms:**
  - **Reflexive rule:** if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$
  - **Augmentation rule:** if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$
  - **Transitivity rule:** if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$
- These rules are
  - **Sound** -- generate only functional dependencies that actually hold, and
  - **Complete** -- generate all functional dependencies that hold.

Example of F^+

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- Some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$
      and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

- Additional rules:
  - **Union rule**: If $\alpha \rightarrow \beta$ holds *and* $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\,\gamma$ holds.
  - **Decomposition rule**: If $\alpha \rightarrow \beta\,\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
  - **Pseudotransitivity rule**:If $\alpha \rightarrow \beta$ holds *and* $\gamma\,\beta \rightarrow \delta$ holds, then $\alpha\,\gamma \rightarrow \delta$ holds**.**
- The above rules can be inferred from Armstrong's axioms.

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies 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 f_1 and f_2 in F^+
    if f_1 and f_2 can be combined using transitivity
      then add the resulting functional dependency to F^+
until F^+ does not change any further

- **NOTE**: We shall see an alternative procedure for this task later

Closure of Attribute Sets

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

- Algorithm to compute a^+, the closure of a under F

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

Example of Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{ \quad A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H \}$
- $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
  3. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
  4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
- Is AG a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $R \supseteq (AG)^+$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $R \supseteq (A)^+$
     2. Does $G \rightarrow R$? == Is $R \supseteq (G)^+$
     3. In general: check for each subset of size $n-1$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- 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 $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
  - That is, we compute α^+ by using attribute closure, and then check if it contains β.
  - Is a simple and cheap test, and very useful
- Computing closure of F
  - For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover

- Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies; that is, 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 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^+

Extraneous Attributes

- Removing an attribute from the left side of a functional dependency could make it a stronger constraint.
  - For example, if we have $AB \rightarrow C$ and remove B, we get the possibly stronger result $A \rightarrow C$. It may be stronger because $A \rightarrow C$ logically implies $AB \rightarrow C$, but $AB \rightarrow C$ does not, on its own, logically imply $A \rightarrow C$
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove B from $AB \rightarrow C$ safely.
  - For example, suppose that
  - $F = \{AB \rightarrow C, A \rightarrow D, D \rightarrow C\}$
  - Then we can show that F logically implies $A \rightarrow C$, making extraneous in $AB \rightarrow C$.
- Removing an attribute from the right side of a functional dependency could make it a weaker constraint.
  - For example, if we have $AB \rightarrow CD$ and remove C, we get the possibly weaker result $AB \rightarrow D$. It may be weaker because using just $AB \rightarrow D$, we can no longer infer $AB \rightarrow C$.
- But, depending on what our set F of functional dependencies happens to be, we may be able to remove C from $AB \rightarrow CD$ safely.
  - For example, suppose that

  $F = \{ AB \rightarrow CD, A \rightarrow C.$
  - Then we can show that even after replacing $AB \rightarrow CD$ by $AB \rightarrow D$, we can still infer $AB → C and thus $AB \rightarrow CD$.

- 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 $\alpha \rightarrow \beta$ in F.
  - **Remove from the left side**: Attribute A is **extraneous** in α if
    - $A \in \alpha$ and
    - F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - **Remove from the right side**: Attribute A is **extraneous** in β if
    - $A \in \beta$ and
    - The set of functional dependencies

  $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ 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 $\alpha \rightarrow \beta$.
- To test if attribute $A \in \beta$ is extraneous in β
  - Consider the set:

    $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
  - check that α^+ contains A; if it does, A is extraneous in β
- To test if attribute $A \in \alpha$ is extraneous in α
  - Let $\gamma = \alpha - \{A\}$. Check if $\gamma \rightarrow \beta$ can be inferred from F.
    - Compute γ^+ using the dependencies in F
    - If γ^+ includes all attributes in β then , A is extraneous in α

Examples of Extraneous Attributes

- Let $F = \{AB \rightarrow CD, A \rightarrow E, E \rightarrow C\}$

- To check if C is extraneous in $AB \rightarrow CD$, we:
    - Compute the attribute closure of AB under $F = \{AB \rightarrow D, A \rightarrow E, E \rightarrow C\}$
    - 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 F_c such that

- F logically implies all dependencies in F_c , and
- F_c logically implies all dependencies in F, and
- No functional dependency in F_c contains an extraneous attribute, and
- Each left side of functional dependency in F_c is unique. That is, there are no two dependencies in F_c
    - $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ such that
    - $\alpha_1 = \alpha_2$

To compute a canonical cover for F:

repeat

Use the union rule to replace any dependencies in F of the form

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$

Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous attribute either in α or in β

/* Note: test for extraneous attributes done using F_c, not F*/

If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$ in F_c

until (F_c does not change)

Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

- $R = (A, B, C)$
  $F = \{A \rightarrow BC$
  $B \rightarrow C$
  $A \rightarrow B$
  $AB \rightarrow C\}$
- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
    - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
    - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
        - Yes: in fact, $B \rightarrow C$ is already present!
    - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
    - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
        - Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
            - Can use attribute closure of A in more complex cases
- The canonical cover is: $A \rightarrow B$
  $B \rightarrow C$

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
    - A decomposition is **dependency preserving**, if
  $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
- Using the above definition, testing for dependency preservation take 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 $R_1, R_2, .., R_n$ be a decomposition of R.

- The restriction of F to R_i is the set F_i of all functional dependencies in F^+ that include **only** attributes of R_i.
- 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 in F^+, not just those in F.
- The set of restrictions $F_1, F_2, .., F_n$ is the set of functional dependencies that can be checked efficiently.

Testing for Dependency Preservation

- To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of R into $R_1, R_2, \ldots, R_n$, we apply the following test (with attribute closure done with respect to F)
  - *result* = α

    repeat

    for each R_i in the decomposition

    $t = (result \cap R_i)^+ \cap R_i$

    $result = result \cup t$

    until (*result* does not change)
  - If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ 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 $(F_1 \cup F_2 \cup \ldots \cup F_n)^+$

Example

- $R = (A, B, C)$

  $F = \{A \rightarrow B$

  $B \rightarrow C\}$

  Key = $\{A\}$
- R is not in BCNF
- Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$
  - R_1 and R_2 in BCNF
  - Lossless-join decomposition
  - Dependency preserving

Algorithm for Decomposition Using Functional Dependencies

Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ 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 $F = \{ A \rightarrow B, BC \rightarrow D\}$
    - Decompose R into $R_1 = (A,B)$ and $R_2 = (A,C,D, E)$
    - Neither of the dependencies in F contain only attributes from (A,C,D,E) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.

Testing Decomposition for BCNF

To check if a relation R_i in a decomposition of R is in BCNF

- Either test R_i for BCNF with respect to the **restriction** of F^+ to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
- Or use the original set of dependencies *F* that hold on *R*, but with the following test:
  - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.
  - If the condition is violated by some $\alpha \rightarrow \beta$ in F^+, the dependency

    $\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$

    can be shown to hold on R_i, and R_i violates BCNF.
  - We use above dependency to decompose R_i

BCNF Decomposition Algorithm

result := {*R*};
done := false;
compute F^+;
while (not *done)* **do**
    if (there is a schema R_i in *result* that is not in BCNF)
        then begin
            let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that
        holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+,
                and $\alpha \cap \beta = \emptyset$;
            $result := (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta);$
        end
        else *done* := **true;**

Note: each R_i 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*)
    - *class-1* (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *capacity*, *time_slot_id*)
- *course* is in BCNF
  - How do we know this?
- *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

- There are some situations where
  - BCNF is not dependency preserving, and

- efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems; we will see examples later)
  - But functional dependencies can be checked on individual relations without computing a join.
  - There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example -- Relation *dept_advisor*

- *dept_advisor* (*s_ID, i_ID, dept_name)*
  $F = \{s_ID, dept_name \rightarrow i_ID,\ i_ID \rightarrow dept_name\}$
- Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*
- *R* is in 3NF
  - *s_ID, dept_name* $\rightarrow$ *i_ID s_ID*
    - *dept_name* is a superkey
  - *i_ID* $\rightarrow$ *dept_name*
    - *dept_name* is contained in a candidate key

Testing for 3NF

- Need to check only FDs in *F*, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- 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 involve finding candidate keys
  - Testing for 3NF has been shown to be NP-hard
  - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

3NF Decomposition Algorithm

Let F_c be a canonical cover for *F;*
i := 0;
for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**
    if none of the schemas R_j, $1 \leq j \leq i$ contains $\alpha\ \beta$
        then begin
            $i := i + 1$;
            $R_i := \alpha\ \beta$
        end
if none of the schemas R_j, $1 \leq j \leq i$ contains a candidate key for *R*
    then begin
        $i := i + 1$;
        R_i := any candidate key for *R;*
    end
/* Optionally, remove redundant relations */
repeat
if any schema R_j is contained in another schema R_k
  then /* delete R_j ***/**
    $R_j = R_i$;
    i=i-1;
return $(R_1, R_2, ..., R_i)$

Above algorithm ensures:

- Each relation schema R_i is in 3NF
- Decomposition is dependency preserving and lossless-join
- Proof of correctness is at end of this presentation (click here)

3NF Decomposition: An 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 F_C =
    customer_id, employee_id → *type*
    employee_id → *branch_name*
    customer_id, branch_name → *employee_id*
- The **for** loop generates 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 for loop, detect and delete schemas, such as (*employee_id, branch_name*), which are subsets of other schemas
  - 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

- 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.

Can specify FDs 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 functional dependency whose left hand side is not a key.

Multivalued Dependencies

- 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?

Multi-valued Dependencies

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multi-valued dependency**

$$\alpha \rightarrow\rightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$
$t_3[\beta] = t_1[\beta]$
$t_3[R - \beta] = t_2[R - \beta]$
$t_4[\beta] = t_2[\beta]$
$t_4[R - \beta] = t_1[R - \beta]$

- Tabular representation of $\alpha \rightarrow\rightarrow \beta$

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

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.
  Y, Z, W
- We say that $Y \rightarrow\rightarrow Z$ (Y **multidetermines** Z)
  if and only if for all possible relations $r(R)$
  $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$
  then
  $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$
- Note that since the behavior of Z and W are identical it follows that
  $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$

Example

- In our example:
  $ID \rightarrow\rightarrow$ *child_name*
  $ID \rightarrow\rightarrow$ *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.
- Note:
  - If $Y \rightarrow Z$ then $Y \rightarrow\rightarrow Z$
  - Indeed we have (in above notation) $Z_1 = Z_2$
    The claim follows.

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:

1. To test relations to **determine** whether they are legal under a given set of functional and multivalued dependencies
2. 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 does satisfy the multivalued dependency by adding tuples to *r.*

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
  - If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$

  That is, every functional dependency is also a multivalued dependency
- 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 (Appendix C).

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 $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a superkey for schema *R*
- If a relation is in 4NF it is in BCNF

Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i
  - All multivalued dependencies of the form

$\alpha \rightarrow\rightarrow (\beta \cap R_i)$ where $\alpha \subseteq R_i$ and $\alpha \rightarrow\rightarrow \beta$ is in D^+

4NF Decomposition Algorithm

result: = {*R*};
done := false;
compute D^+;
Let D_i denote the restriction of D^+ to R_i
while (**not** *done*)
if (there is a schema $\mathbf{R}_i$ in *result* that is not in 4NF) **then**
begin
let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds
on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
result := $(result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$;
end
else *done*:= true;
Note: each R_i is in 4NF, and decomposition is lossless-join

Example

- $R = (A, B, C, G, H, I)$

$F = \{ A \rightarrow\rightarrow B$

$B \rightarrow\rightarrow HI$

$CG \rightarrow\rightarrow H \}$

- R is not in 4NF since $A \rightarrow\rightarrow B$ and A is not a superkey for R
- Decomposition

a) $R_1 = (A, B)$ (R_1 is in 4NF)
b) $R_2 = (A, C, G, H, I)$ (R_2 is not in 4NF, decompose into R_3 and R_4)
c) $R_3 = (C, G, H)$ (R_3 is in 4NF)
d) $R_4 = (A, C, G, I)$ (R_4 is not in 4NF, decompose into R_5 and R_6)

- $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$ ➔ $A \rightarrow\rightarrow HI$, (MVD transitivity), and
- and hence $A \rightarrow\rightarrow I$ *(MVD restriction to R_4)*

e) $R_5 = (A, I)$ (R_5 is in 4NF)
f) $R_6 = (A, C, G)$ (R_6 is in 4NF)

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Overall Database Design Process

We have assumed schema R is given

- R could have been generated when converting E-R diagram to a set of tables.
- R could have been a single relation containing *all* attributes that are of interest (called **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 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

- May want to use non-normalized schema for performance
- For example, displaying *prereqs* along with *course_id,* and *title* requires join of *course* with *prereq*
- Alternative 1: Use denormalized relation containing 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 programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined a *course* *prereq*
  - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors

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_2004, earnings_2005, earnings_2006*, etc., all on the schema (*company_id, earnings*).
    - Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company_year* (*company_id, earnings_2004, earnings_2005, earnings_2006*)
    - Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
    - Is an example of a **crosstab**, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools

Modeling Temporal Data

- **Temporal data** have an association time interval during which the data are *valid.*
- A **snapshot** is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
  - attributes, e.g., address of an instructor at different points in time
  - entities, e.g., time duration when a student entity exists
  - relationships, e.g., time during which an instructor was associated with a student as an advisor.
- But no accepted standard
- Adding a temporal component results in functional dependencies like
  $ID \rightarrow street, city$
  not holding, because the address varies over time
- A **temporal functional dependency** $X \rightarrow Y$ holds on schema R if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances r (R).
- In practice, database designers may add start and end time attributes to relations
  - E.g., *course*(*course_id, course_title*) is replaced by *course*(*course_id, course_title, start, end*)
  - Constraint: no two tuples can have overlapping valid times
    - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
  - E.g., student transcript should refer to course information at the time the course was taken

Additional Notes/Problems:

Functional Dependency and Attribute Closure

A functional dependency A->B in a relation holds if two tuples having same value of attribute A also have same value for attribute B. For Example, in relation STUDENT shown in table 1, Functional Dependencies

STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE **hold**
but
STUD_NAME->STUD_STATE **do not hold**

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

How to find functional dependencies for a relation?

Functional Dependencies in a relation are dependent on the domain of the relation. Consider the STUDENT relation given in Table 1.

- We know that STUD_NO is unique for each student. So STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE, STUD_NO->STUD_STATE, STUD_NO->STUD_COUNTRY and STUD_NO -> STUD_AGE all will be true.
- Similarly, STUD_STATE->STUD_COUNTRY will be true as if two records have same STUD_STATE, they will have same STUD_COUNTRY as well.
- For relation STUDENT_COURSE, COURSE_NO->COURSE_NAME will be true as two records with same COURSE_NO will have same COURSE_NAME.

Functional Dependency Set: Functional Dependency set or FD set of a relation is the set of all FDs present in the relation.

For Example, FD set for relation STUDENT shown in table 1 is:
{STUD_NO -> STUD_NAME,
STUD_NO -> STUD_PHONE,
STUD_NO -> STUD_STATE,
STUD_NO -> STUD_COUNTRY,
STUD_NO -> STUD_AGE,
STUD_STATE -> STUD_COUNTRY}

Attribute Closure: Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.

How to find attribute closure of an attribute set?

To find attribute closure of an attribute set:

- Add elements of attribute set to the result set.
- Recursively add elements to the result set which can be functionally determined from the elements of the result set.

Using FD set of table 1, attribute closure can be determined as:

(STUD_NO)+ = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}

(STUD_STATE)+ = {STUD_STATE, STUD_COUNTRY}

How to find Candidate Keys and Super Keys using Attribute Closure?

- If attribute closure of an attribute set contains all attributes of relation, the attribute set will be super key of the relation.
- If no subset of this attribute set can functionally determine all attributes of the relation, the set will be candidate key as well. For Example, using FD set of table 1,

(STUD_NO, STUD_NAME)+ = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}

(STUD_NO)+ = {STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY, STUD_AGE}

(STUD_NO, STUD_NAME) will be super key but not candidate key because its subset (STUD_NO)+ is equal to all attributes of the relation. So, STUD_NO will be a candidate key.

GATE Question: Consider the relation scheme R = {E, F, G, H, I, J, K, L, M, N} and the set of functional dependencies {{E, F} -> {G}, {F} -> {I, J}, {E, H} -> {K, L}, K -> {M}, L -> {N} on R. What is the key for R? (GATE-CS-2014)
A. {E, F}
B. {E, F, H}
C. {E, F, H, K, L}
D. {E}
Answer: Finding attribute closure of all given options, we get:
{E,F}+ = {EFGIJ}
{E,F,H}+ = {EFHGIJKLMN}
{E,F,H,K,L}+ = {{EFHGIJKLMN}
{E}+ = {E}
{EFH}+ and {EFHKL}+ results in set of all attributes, but EFH is minimal. So it will be candidate key. So correct option is (B).

How to check whether an FD can be derived from a given FD set?
To check whether an FD A->B can be derived from an FD set F,

1. Find (A)+ using FD set F.
2. If B is subset of (A)+, then A->B is true else not true.

GATE Question: In a schema with attributes A, B, C, D and E following set of functional dependencies are given
{A -> B, A -> C, CD -> E, B -> D, E -> A}
Which of the following functional dependencies is NOT implied by the above set? (GATE IT 2005)
A. CD -> AC
B. BD -> CD
C. BC -> CD
D. AC -> BC
Answer: Using FD set given in question,
(CD)+ = {CDEAB} which means CD -> AC also holds true.
(BD)+ = {BD} which means BD -> CD can't hold true. So this FD is no implied in FD set. So (B) is the required option.
Others can be checked in the same way.

Prime and non-prime attributes

Attributes which are parts of any candidate key of relation are called as prime attribute, others are non-prime attributes. For Example, STUD_NO in STUDENT relation is prime attribute, others are non-prime attribute.

GATE Question: Consider a relation scheme R = (A, B, C, D, E, H) on which the following functional dependencies hold: {A–>B, BC–> D, E–>C, D–>A}. What are the candidate keys of R? [GATE 2005]
(a) AE, BE
(b) AE, BE, DE
(c) AEH, BEH, BCH
(d) AEH, BEH, DEH
Answer: (AE)+ = {ABECD} which is not set of all attributes. So AE is not a candidate key. Hence option A and B are wrong.
(AEH)+ = {ABCDEH}
(BEH)+ = {BEHCDA}
(BCH)+ = {BCHDA} which is not set of all attributes. So BCH is not a candidate key. Hence option C is wrong.
So correct answer is D.

Finding Attribute Closure and Candidate Keys using Functional Dependencies
What is Functional Dependency?
A functional dependency X->Y in a relation holds if two tuples having same value for X also have same value for Y i.e X uniquely determines Y.
In EMPLOYEE relation given in Table 1,

- FD E-ID->E-NAME holds because for each E-ID, there is a unique value of E-NAME.
- FD E-ID->E-CITY and E-CITY->E-STATE also holds.
- FD E-NAME->E-ID does not hold because E-NAME 'John' is not uniquely determining E-ID. There are 2 E-IDs corresponding to John (E001 and E003).

EMPLOYEE

E-ID	E-NAME	E-CITY	E-STATE
E001	John	Delhi	Delhi
E002	Mary	Delhi	Delhi
E003	John	Noida	U.P.

Table 1

The FD set for EMPLOYEE relation given in Table 1 are:
{E-ID->E-NAME, E-ID->E-CITY, E-ID->E-STATE, E-CITY->E-STATE}

Trivial versus Non-Trivial Functional Dependency: A trivial functional dependency is the one which will always hold in a relation.
X->Y will always hold if X ⊇ Y

In the example given above, E-ID, E-NAME->E-ID is a trivial functional dependency and will always hold because {E-ID,E-NAME} ⊃ {E-ID}. You can also see from the table that for each value of {E-ID, E-NAME}, value of E-ID is unique, so {E-ID, E-NAME} functionally determines E-ID.

If a functional dependency is not trivial, it is called Non-Trivial Functional Dependency. Non-Trivial functional dependency may or may not hold in a relation. e.g; E-ID->E-NAME is a non-trivial functional dependency which holds in the above relation.

Properties of Functional Dependencies

Let *X*, *Y*, and *Z* are sets of attributes in a relation *R*. There are several properties of functional dependencies which always hold in R also known as Armstrong Axioms.

1. Reflexivity: If *Y* is a subset of *X*, then $X \rightarrow Y$. e.g.; Let X represents {E-ID, E-NAME} and Y represents {E-ID}. {E-ID, E-NAME}->E-ID is true for the relation.
2. Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$. e.g.; Let X represents {E-ID}, Y represents {E-NAME} and Z represents {E-CITY}. As {E-ID}->E-NAME is true for the relation, so { E-ID,E-CITY}->{E-NAME,E-CITY} will also be true.
3. Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. e.g.; Let X represents {E-ID}, Y represents {E-CITY} and Z represents {E-STATE}. As {E-ID} ->{E-CITY} and {E-CITY}->{E-STATE} is true for the relation, so { E-ID }->{E-STATE} will also be true.
4. Attribute Closure: The set of attributes that are functionally dependent on the attribute A is called Attribute Closure of A and it can be represented as A^+.

Steps to Find the Attribute Closure of A

Q. Given FD set of a Relation R, The attribute closure set S be the set of A

1. Add A to S.
2. Recursively add attributes which can be functionally determined from attributes of the set S until done.

From Table 1, FDs are

Given R(E-ID, E-NAME, E-CITY, E-STATE)

FDs = { E-ID->E-NAME, E-ID->E-CITY, E-ID->E-STATE, E-CITY->E-STATE }

The attribute closure of E-ID can be calculated as:

1. Add E-ID to the set {E-ID}
2. Add Attributes which can be derived from any attribute of set. In this case, E-NAME and E-CITY, E-STATE can be derived from E-ID. So these are also a part of closure.
3. As there is one other attribute remaining in relation to be derived from E-ID. So result is:

$(E\text{-}ID)^+$ = {E-ID, E-NAME, E-CITY, E-STATE }

Similarly,

$(E\text{-}NAME)^+$ = {E-NAME}

$(E\text{-}CITY)^+$ = {E-CITY, E_STATE}

Q. Find the attribute closures of given FDs R(ABCDE) = {AB->C, B->D, C->E, D->A} To find $(B)^+$,we will add attribute in set using various FD which has been shown in table below.

Attributes Added in Closure	FD used
{B}	Triviality
{B,D}	B->D
{B,D,A}	D->A
{B,D,A,C}	AB->C
{B,D,A,C,E}	C->E

- We can find $(C, D)^+$ by adding C and D into the set (triviality) and then E using(C->E) and then A using (D->A) and set becomes. $(C,D)^+$ = {C,D,E,A}
- Similarly we can find $(B,C)^+$ by adding B and C into the set (triviality) and then D using (B->D) and then E using (C->E) and then A using (D->A) and set becomes $(B,C)^+$ ={B,C,D,E,A}

Candidate Key

Candidate Key is minimal set of attributes of a relation which can be used to identify a tuple uniquely. For Example, each tuple of EMPLOYEE relation given in Table 1 can be uniquely identified by E-ID and it is minimal as well. So it will be Candidate key of the relation.
A candidate key may or may not be a primary key.
Super Key
Super Key is set of attributes of a relation which can be used to identify a tuple uniquely.For Example, each tuple of EMPLOYEE relation given in Table 1 can be uniquely identified by E-ID or (E-ID, E-NAME) or (E-ID, E-CITY) or (E-ID, E-STATE) or (E_ID, E-NAME, E-STATE) etc. So all of these are super keys of EMPLOYEE relation.
Note: A candidate key is always a super key but vice versa is not true.

Q. Finding Candidate Keys and Super Keys of a Relation using FD set The set of attributes whose attribute closure is set of all attributes of relation is called super key of relation. For Example, the EMPLOYEE relation shown in Table 1 has following FD set. {E-ID->E-NAME, E-ID->E-CITY, E-ID->E-STATE, E-CITY->E-STATE} Let us calculate attribute closure of different set of attributes:
$(E\text{-}ID)^+$ = {E-ID, E-NAME,E-CITY,E-STATE}
$(E\text{-}ID,E\text{-}NAME)^+$ = {E-ID, E-NAME,E-CITY,E-STATE}
$(E\text{-}ID,E\text{-}CITY)^+$ = {E-ID, E-NAME,E-CITY,E-STATE}
$(E\text{-}ID,E\text{-}STATE)^+$ = {E-ID, E-NAME,E-CITY,E-STATE}
$(E\text{-}ID,E\text{-}CITY,E\text{-}STATE)^+$ = {E-ID, E-NAME,E-CITY,E-STATE}
$(E\text{-}NAME)^+$ = {E-NAME}
$(E\text{-}CITY)^+$ = {E-CITY,E-STATE}
As $(E\text{-}ID)^+$, $(E\text{-}ID, E\text{-}NAME)^+$, $(E\text{-}ID, E\text{-}CITY)^+$, $(E\text{-}ID, E\text{-}STATE)^+$, $(E\text{-}ID, E\text{-}CITY, E\text{-}STATE)^+$ give set of all attributes of relation EMPLOYEE. So all of these are super keys of relation.
The minimal set of attributes whose attribute closure is set of all attributes of relation is called candidate key of relation. As shown above, $(E\text{-}ID)^+$ is set of all attributes of relation and it is minimal. So E-ID will be candidate key. On the other hand $(E\text{-}ID, E\text{-}NAME)^+$ also is set of all attributes but it is not minimal because its subset $(E\text{-}ID)^+$ is equal to set of all attributes. So (E-ID, E-NAME) is not a candidate key.

Types of Functional dependencies in DBMS

A functional dependency is a constraint that specifies the relationship between two sets of attributes where one set can accurately determine the value of other sets. It is denoted as **X → Y**, where X is a set of attributes that is capable of determining the value of Y. The attribute set on the left side of the arrow, **X** is called **Determinant**, while on the right side, **Y** is called the **Dependent**.

Example:

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

- roll_no → { name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, dept_name and dept_building, hence a valid Functional dependency
- roll_no → dept_name , Since, roll_no can determine whole set of {name, dept_name, dept_building}, it can determine its subset dept_name also.
- dept_name → dept_building , Dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies: roll_no → name, {roll_no, name} ⇢ {dept_name, dept_building}, etc.

Here are some invalid functional dependencies:

- name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name There can be multiple departments in the same building, For example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.
- More invalid functional dependencies: name → roll_no, {name, dept_name} → roll_no, dept_building → roll_no, etc.

Armstrong's axioms/properties of functional dependencies**:**

1. **Reflexivity:** If Y is a subset of X, then X→Y holds by reflexivity rule
   For example, {roll_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
   For example, If {roll_no, name} → dept_building is valid, hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.→
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
   For example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

Types of Functional dependencies in DBMS:

1. Trivial functional dependency
2. Non-Trivial functional dependency
3. Multivalued functional dependency
4. Transitive functional dependency

1. Trivial Functional Dependency

In Trivial Functional Dependency, a dependent is always a subset of the determinant.
i.e. If X → Y and Y is the subset of X, then it is called trivial functional dependency

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, {roll_no, name} → name is a trivial functional dependency, since the dependent name is a subset of determinant set {roll_no, name}
Similarly, roll_no → roll_no is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In Non-trivial functional dependency, the dependent is strictly not a subset of the determinant.
i.e. If X → Y and Y is not a subset of X, then it is called Non-trivial functional dependency.
For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, roll_no → name is a non-trivial functional dependency, since the dependent name is not a subset of determinant roll_no
Similarly, {roll_no, name} → age is also a non-trivial functional dependency, since age is not a subset of {roll_no, name}

3. Multivalued Functional Dependency

In Multivalued functional dependency, entities of the dependent set are not dependent on each other.
i.e. If a → {b, c} and there exists no functional dependency between b and c, then it is called a multivalued functional dependency.
For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, roll_no → {name, age} is a multivalued functional dependency, since the dependents name & age are not dependent on each other(i.e. name → age or age → name doesn't exist !)

4. Transitive Functional Dependency

In transitive functional dependency, dependent is indirectly dependent on determinant. i.e. If a → b & b → c, then according to axiom of transitivity, a → c. This is a transitive functional dependency
For example,

enrol_no	name	dept	building_no
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, enrol_no → dept and dept → building_no,
Hence, according to the axiom of transitivity, enrol_no → building_no is a valid functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

Number of Possible Super Keys in DBMS
Any set of attributes of a table that can uniquely identify all the tuples of that table is known as a Super key. It's different from the primary and candidate keys in the sense that only the minimal superkeys are the candidate/primary keys.

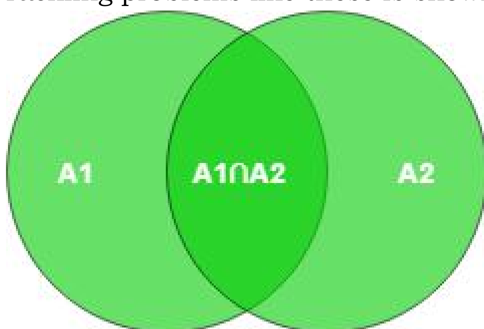
This means that from a super key when we remove all the attributes that are unnecessary for its uniqueness, only then it becomes a primary/candidate key. So, in essence, all primary/candidate keys are super keys but not all super keys are primary/candidate keys. By the formal definition of a Relation(Table), we know that the tuples of a relation are all unique. So, the set of all attributes itself is a super key.

Example-1: Let a Relation R have attributes {a1,a2,a3} and a1 is the candidate key. Then how many super keys are possible?
Here, any superset of a1 is the super key.
Super keys are = {a1, a1 a2, a1 a3, a1 a2 a3}
Thus we see that 4 Super keys are possible in this case.
In general, if we have 'N' attributes with one candidate key then the number of possible superkeys is $2^{(N-1)}$.

Example-2 : Let a Relation R have attributes {a1, a2, a3,…,an}. Find Super key of R.
Maximum Super keys = $2^n - 1$.
If each attribute of relation is candidate key.

Example-3: Let a Relation R have attributes {a1, a2, a3,…, an} and the candidate key is "a1 a2 a3" then the possible number of super keys?
Following the previous formula, we have 3 attributes instead of one. So, here the number of possible super keys is $2^{(N-3)}$.

Example-4: Let a Relation R have attributes {a1, a2, a3,…, an} and the candidate keys are "a1", "a2" then the possible number of super keys?
This problem now is slightly different since we now have two different candidate keys instead of only one. Tackling problems like these is shown in the diagram below:

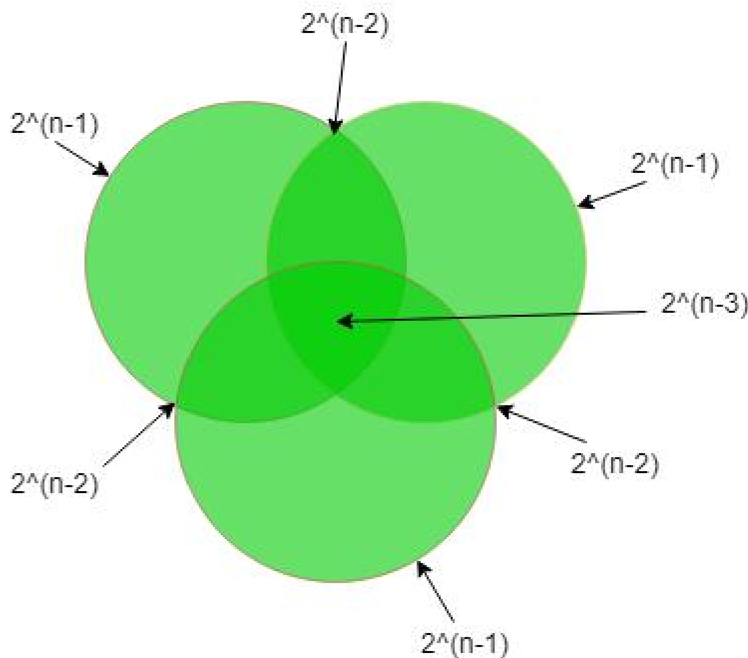


→ $|A1 \cup A2| = |A1| + |A2| - |A1 \cap A2|$
= (super keys possible with candidate key A1) + (super keys possible with candidate key A2) – (common superkeys from both A1 and A2)
= $2^{(n-1)} + 2^{(n-1)} - 2^{(n-2)}$

Example-5: Let a Relation R have attributes {a1, a2, a3,…, an} and the candidate keys are "a1", "a2 a3" then the possible number of super keys?
Super keys of (a1) + Super keys of (a2 a3) – Super keys of (a1 a2 a3)
⇒ $2^{(n-1)} + 2^{(n-2)} - 2^{(n-3)}$

Example-6: Let a Relation R have attributes {a1, a2, a3,…, an} and the candidate keys are "a1 a2", "a3 a4" then the possible number of super keys?
Super keys of(a1 a2) + Super keys of(a3 a4) – Super keys of(a1 a2 a3 a4)
$\Rightarrow 2^{(n-2)} + 2^{(n-2)} - 2^{(n-4)}$
Example-7: Let a Relation R have attributes {a1, a2, a3,…, an} and the candidate keys are "a1 a2", "a1 a3" then the possible number of super keys?
Super keys of (a1 a2) + Super keys of (a1 a3) – Super keys of(a1 a2 a3)
$\Rightarrow 2^{(n-2)} + 2^{(n-2)} - 2^{(n-3)}$

Example-8 : Let a Relation R have attributes {a1, a2, a3,…,an} and the candidate keys are "a1", "a2", "a3" then the possible number of super keys?
In this question, we have 3 different candidate keys. Tackling problems like these are shown in the diagram below.



→ $|A1 \cup A2 \cup A3| = |A1| + |A2| + |A3| - |A1 \cap A2| - |A1 \cap A3| - |A2 \cap A3| + |A1 \cap A2 \cap A3|$
= (super keys possible with candidate key A1) + (super keys possible with candidate key A2) + (super keys possible with candidate key A3) – (common super keys from both A1 and A2) – (common super keys from both A1 and A3) – (common super keys from both A2 and A3) + (common super keys from both A1, A2, and A3)
$= 2^{(n-1)} + 2^{(n-1)} + 2^{(n-1)} - 2^{(n-2)} - 2^{(n-2)} - 2^{(n-2)} + 2^{(n-3)}$

Example-9: A relation R (A, B, C, D, E, F, G, H)and set of functional dependencies are
CH → G,
A → BC,
B → CFH,
E → A,
F → EG
Then how many possible super keys are present?
Step 1:- First of all, we have to find what the candidate keys are:-
as we can see in the given functional dependency D is missing but in relation, D is given so D must be a prime attribute of the Candidate key.
$A^+ = E^+ = B^+ = F^+$ = all attributes of a relation except D
So, Candidate keys are = AD, BD, ED, FD
Step 2:-Find super keys due to a single candidate key there is a two possibilities of attribute either we select or not hence there will be 2 chances so,
A_ _D_ _ _ _ = _ B_ D_ _ _ _ = _ _ _ DE _ _ _ = _ _ _ D_F_ _ = 2^6

Step 3:-Find superkeys due to a combination of two Candidate Keys. So,
$n(AD \cap BD) = n(AD \cap ED) = n(AD \cap FD) = n(BD \cap ED) = n(BD \cap FD) = n(ED \cap FD) = 2^5$
Step 4:-Find super keys due to a combination of 3 Candidate Keys
So,
$n(AD \cap BD \cap ED) = n(AD \cap ED \cap FD) = n(ED \cap BD \cap FD) = n(BD \cap FD \cap AD) = 2^4$
Step 5:-Find super keys due to all. So,
$n(AD \cap BD \cap ED \cap FD)$ = AB_DEF_ _ = 2^3
So, According to the inclusion-exclusion principle :-
$|W \cup X \cup Y \cup Z| = |W| + |X| + |Y| + |Z| - |W \cap X| - |W \cap Y| - |W \cap Z| - |X \cap Y| - |X \cap Z| - |Y \cap Z| + |W \cap X \cap Y| + |W \cap X \cap Z| + |W \cap Y ? Z| + |X \cap Y \cap Z| - |W \cap X \cap Y \cap Z|$
Super keys = $4 * 2^6 - 6 * 2^5 + 4 * 2^4 - 2^3 = 120$
So the number of super keys is 120.

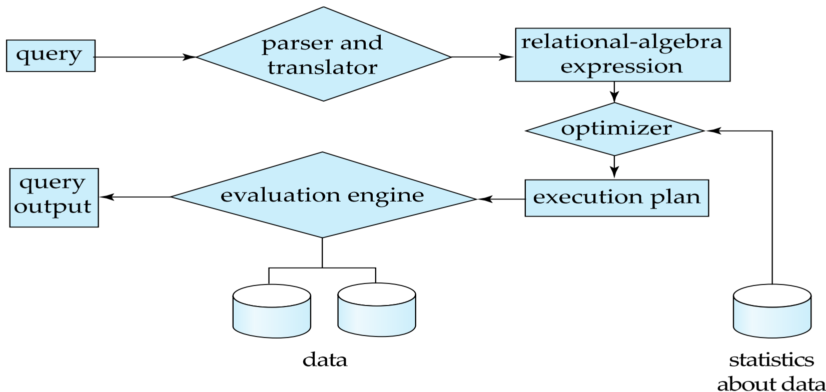
Example 10 : Let a Relation R have attributes {a1,a2,a3______ an} and {a1a2a3____ak} as the candidate key where k<=n. Then how many super keys are possible?
The possible number of super keys is $2^{(n-k)}$.

Example 11: Let a relation R have attributes {a1,a2,a3______ an} such that any k of the attributes at a time determines all other attributes. Find the value of k such that the number of candidate keys in the relation will be maximum.
Any k attributes at a time constitute one candidate key. These k attributes are randomly chosen from the n attributes. So for some k, the possible no of candidate keys is nC_k, i.e., n!/(n-k)!k!. For the number of members to be maximum k must be ⌊n/2⌋ so that nCk is the maximum for that value.

Query Processing: Overview, Measures of Query cost, Selection operation, sorting, Join Operation, other operations, Evaluation of Expressions.
Query optimization: Overview, Transformation of Relational Expressions, Estimating statistics of Expression results, Choice of Evaluation Plans, Materialized views, Advanced Topics in Query Optimization.

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to
    $\prod_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**. E.g.,:
  - Use an index on *salary* to find instructors with salary < 75000,
  - Or perform complete relation scan and discard instructors with salary ≥ 75000
- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g.. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account

- Network costs must be considered for parallel systems
- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk
- Disk cost can be estimated as:
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

    $b * t_T + S * t_S$
- t_S and t_T depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
  - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
  - But more optimistic estimates are used in practice

Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - cost = (b_r /2) block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A2** (**clustering index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = $(h_i + 1) * (t_T + t_S)$
- **A3** (**clustering index, equality on nonkey**) Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - *Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$
- **A4** (**secondary index, equality on key/non-key**).

- Retrieve a single record if the search-key is a candidate key
  - $Cost = (h_i + 1) * (t_T + t_S)$
- Retrieve multiple records if search-key is not a candidate key
  - each of *n* matching records may be on a different block
  - Cost = $(h_i + n) * (t_T + t_S)$
    - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**clustering index, comparison**). (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple ≥ v and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple > v; do not use index
- **A6** (**clustering index, comparison**).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry ≥ v and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > v
  - In either case, retrieve records that are pointed to
  - requires an I/O per record; Linear file scan may be cheaper!

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \theta_n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A8** (**conjunctive selection using composite index**).
  - Use appropriate composite (multiple-key) index if available.
- **A9** (**conjunctive selection by intersection of identifiers**).
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \theta_n}(r)$.
- **A10** (**disjunctive selection by union of identifiers**).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Bitmap Index Scan

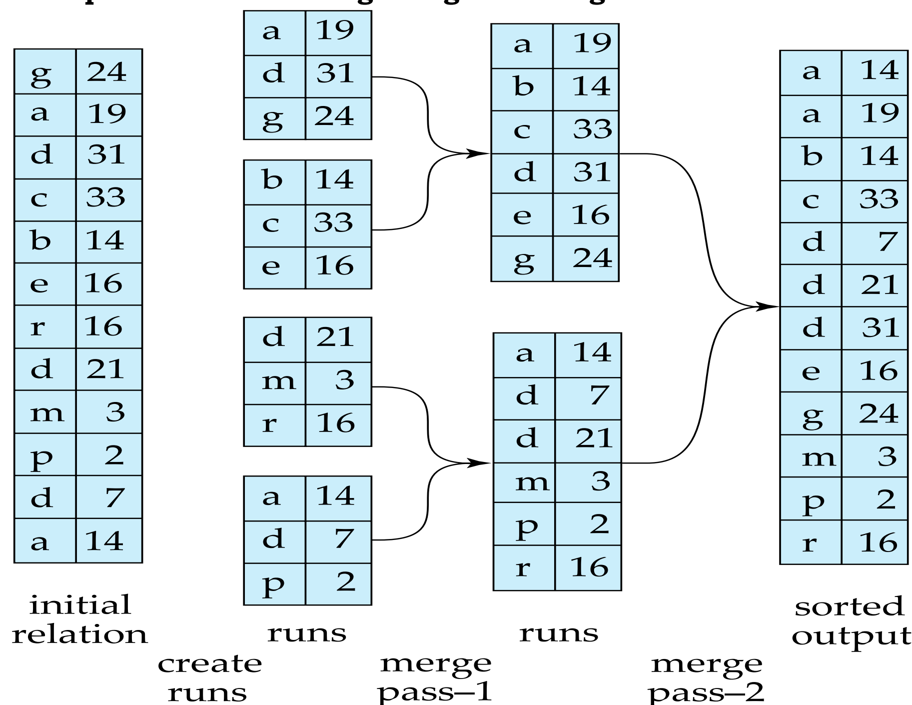
- The **bitmap index scan** algorithm of PostgreSQL
  - Bridges gap between secondary index scan and linear file scan when number of matching records is not known before execution

- Bitmap with 1 bit per page in relation
- Steps:
  - Index scan used to find record ids, and set bit of corresponding page in bitmap
  - Linear file scan fetching only pages with bit set to 1
- Performance
  - Similar to index scan when only a few bits are set
  - Similar to linear file scan when most bits are set
  - Never behaves very badly compared to best alternative

Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used.
  - For relations that don't fit in memory, **external sort-merge** is a good choice.

Example: External Sorting Using Sort-Merge



External Sort-Merge

Let *M* denote memory size (in pages).

1. **Create sorted runs**. Let *i* be 0 initially.
   Repeatedly do the following till the end of the relation:
   (a) Read *M* blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run R_i; increment *i*.

Let the final value of *i* be *N*

2. **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.
   1. Use *N* blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer. If the output buffer is full write it to disk.

3. Delete the record from its input buffer page.
   If the buffer page becomes empty **then**
   read the next block (if any) of the run into the buffer.

3. until all input buffer pages are empty:

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of M - 1 runs are merged.
  - A pass reduces the number of runs by a factor of M -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

- Cost analysis:
  - 1 block per run leads to too many seeks during merge
    - Instead use b_b buffer blocks per run
      ➔ read/write b_b blocks at a time
    - Can merge $\lfloor M/b_b \rfloor - 1$ runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M/bb \rfloor -1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:

$$b_r (2 \lceil \log_{\lfloor M/bb \rfloor -1} (b_r / M) \rceil + 1) \lceil$$

  - Seeks: next slide

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:

$$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{\lfloor M/bb \rfloor -1}(b_r / M) \rceil -1)$$

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000   *takes*: 10,000
  - Number of blocks of *student*: 100   *takes*: 400

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

for each tuple t_r **in** r **do begin**
  for each tuple t_s **in** s **do begin**

test pair (t_r,t_s) to see if they satisfy the join condition θ
if they do, add $t_r \bullet t_s$ to the result.
end
end

- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - with *student* as outer relation:
    - 5000 ∗ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

for each block B_r **of** r **do begin**
for each block B_s **of** s **do begin**
for each tuple t_r **in** B_r **do begin**
for each tuple t_s **in** B_s **do begin**
Check if (t_r,t_s) satisfy the join condition
if they do, add $t_r \bullet t_s$ to the result.
end
end
end
end

- Worst case estimate: $b_r * b_s + b_r$ block transfers + $2 * b_r$ seeks
  - Each block in the inner relation s is read once for each *block* in the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
    - Cost = $\lceil b_r / (M\text{-}2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M\text{-}2) \rceil$ seeks
  - If equi-join attribute forms a key or inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute

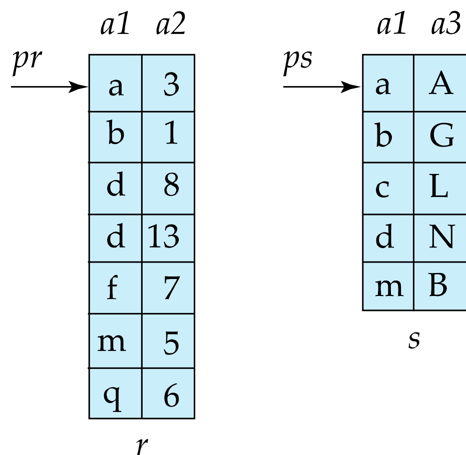
- Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one page of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r (t_T + t_S) + n_r * c$
    - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *r*
    - *c* can be estimated as cost of a single selection on *s* using the join condition.
- If indices are available on join attributes of both *r* and *s,* use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *student* ⋈ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B^+-tree index on the attribute *ID,* which contains 20 entries in each index node.
- Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
    - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
        - assuming worst case memory
        - may be significantly less with more memory
- Cost of indexed nested loops join
    - 100 + 5000 * 5 = 25,100 block transfers and seeks.
    - CPU cost likely to be less than that for block nested loops join

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
1. Join step is similar to the merge stage of the sort-merge algorithm.
2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
3. Detailed algorithm in book

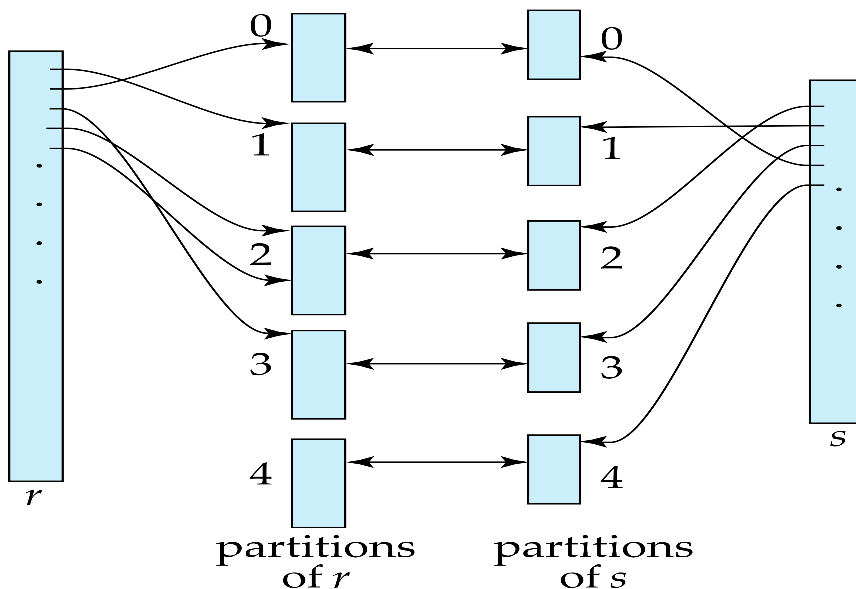


- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks + the cost of sorting if relations are unsorted.

- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree .
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - $r_0, r_1, . . ., r_n$ denote partitions of *r* tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r [JoinAttrs])$.
  - $r_0,, r_1. . ., r_n$ denotes partitions of *s* tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s [JoinAttrs])$.
- *Note:* In book, Figure 12.10 r_i is denoted as H_{ri}, s_i is denoted as H_{si} and *n* is denoted as n_h.



- *r* tuples in r_i need only to be compared with *s* tuples in s_i Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.

Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*

(a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h*.

(b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation s is called the **build input** and r is called the **probe input**.

- The value n and the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "**fudge factor**", typically around 1.2
  - The probe relation partitions s_i need not fit in memory
- **Recursive partitioning** required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

Handling of Overflows

- Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others
- **Hash-table overflow** occurs in partition s_i if s_i does not fit in memory. Reasons could be
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution** can be done in build phase
  - Partition s_i is further partitioned using different hash function.
  - Partition r_i must be similarly partitioned.
- **Overflow avoidance** performs partitioning carefully to avoid overflows during build phase
  - E.g., partition build relation into many partitions, then combine them
- Both approaches fail with large numbers of duplicates
  - Fallback option: use block nested loops join on overflowed partitions

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $3(b_r + b_s) + 4 * n_h$ block transfers +
  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
- If recursive partitioning required:
  - number of passes required for partitioning build relation s to less than M blocks per partition is $\lceil log_{\lfloor M/bb \rfloor -1}(b_s/M) \rceil$
  - best to choose the smaller relation as the build relation.
  - Total cost estimate is:
    $2(b_r + b_s)\lceil log_{\lfloor M/bb \rfloor -1}(b_s/M) \rceil + b_r + b_s$ block transfers +
    $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)\lceil log_{\lfloor M/bb \rfloor -1}(b_s/M) \rceil$ seeks
- If the entire build input can be kept in main memory no partitioning is required
  - Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

instructor ⋈ *teaches*

- Assume that memory size is 20 blocks
- $b_{instructor}$= 100 and $b_{teaches}$ = 400.
- *instructor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *teaches* into five partitions, each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - 3(100 + 400) = 1500 block transfers +
    $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil)$ = 336 seeks

Hybrid Hash–Join

- Useful when memory sized are relatively large, and the build input is bigger than memory.
- **Main feature of hybrid hash join:**

Keep the first partition of the build relation in memory.

- E.g. With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
  - Division of memory:
    - The first partition occupies 20 blocks of memory
    - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- *teaches* is similarly partitioned into five partitions each of size 80
  - the first is used right away for probing, instead of being written out
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if M >>

Complex Joins

- Join with a conjunctive condition:

  $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta_i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

  $\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$
- Join with a disjunctive condition

  $r \bowtie_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

  $(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s)$

Joins over Spatial Data

- No simple sort order for spatial joins
- Indexed nested loops join with spatial indices
  - R-trees, quad-trees, k-d-B-trees

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.
- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization*:* **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values

        - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
            - When combining partial aggregate for count, add up the partial aggregates
        - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ and —): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
      1. Using a different hashing function, build an in-memory hash index on r_i.
      2. Process s_i as follows
          - $r \cup s$:
              1. Add tuples in s_i to the hash index if they are not already in it.
              2. At end of s_i add the tuples in the hash index to the result.
- E.g., Set operations using hashing:
  1. as before partition *r* and *s*,
  2. as before, process each partition *i* as follows
      1. build a hash index on r_i
      2. Process s_i as follows
          1. $r \cap s$:
              - output tuples in s_i to the result if they are already there in the hash index
          2. $r - s$:
              - for each tuple in s_i, if it is there in the hash index, delete it from the index.
              - At end of s_i add remaining tuples in the hash index to the result.

Answering Keyword Queries

- Indices mapping keywords to documents
    - For each keyword, store sorted list of document IDs that contain the keyword
        - Commonly referred to as a **inverted index**
        - E.g.,: database: d1, d4, d11, d45, d77, d123
          distributed: d4, d8, d11, d56, d77, d121, d333
    - To answer a query with several keywords, compute intersection of lists corresponding to those keywords
- To support ranking, inverted lists store extra information
    - "**Term frequency**" of the keyword in the document
    - "**Inverse document frequency**" of the keyword
    - **Page rank** of the document/web page

Other Operations : Outer Join

- **Outer join** can be computed either as
    - A join followed by addition of null-padded non-participating tuples.
    - by modifying the join algorithms.
- Modifying merge join to compute *r* ⟕ *s*
    - In *r* ⟕ *s*, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
    - Modify merge-join to compute *r* ⟕ *s*:
        - During merging, for every tuple t_r from *r* that do not match any tuple in *s*, output t_r padded with nulls.
    - Right outer-join and full outer-join can be computed similarly.
- Modifying hash join to compute *r* ⟕ *s*
    - If *r* is probe relation, output non-matching *r* tuples padded with nulls
    - If *r* is build relation, when probing keep track of which *r* tuples matched *s* tuples. At end of s_i output non-matched *r* tuples padded with nulls

Evaluation of Expressions

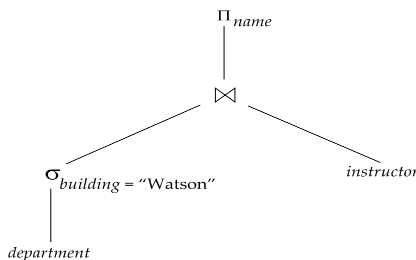
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{building="Watson"}(department)$$

then compute the store its join with *instructor,* and finally compute the projection on *name.*



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="Watson"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents

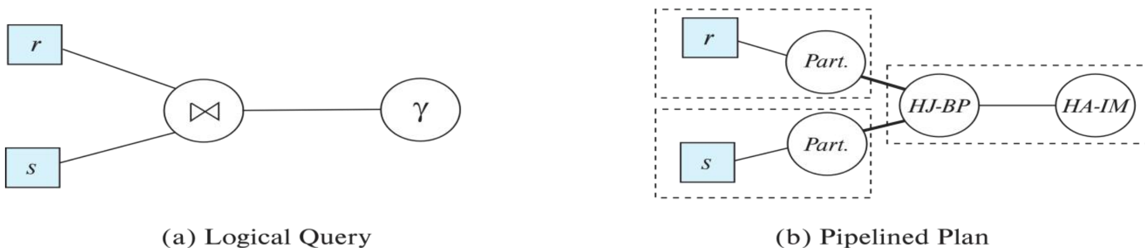
- Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
- if buffer is full, child waits till there is space in the buffer, and then generates more tuples

- System schedules operations that have space in output buffer and can process more input tuples

- Alternative name: **pull** and **push** models of pipelining
- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g., file scan: initialize file scan
        - state: pointer to beginning of file
      - E.g., merge join: sort relations;
        - state: pointers to beginning of sorted relations
    - **next()**
      - E.g., for file scan: Output next tuple, and advance and store file pointer
      - E.g., for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**

Blocking Operations

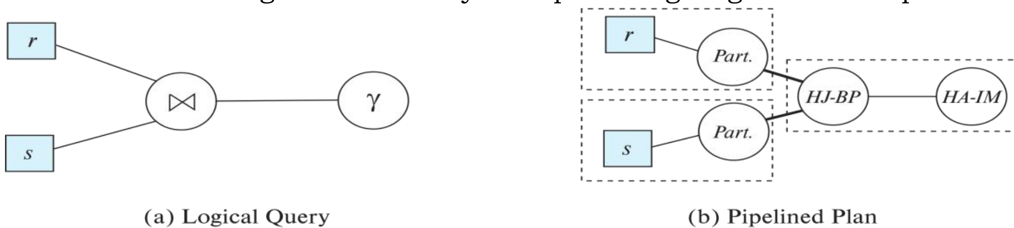
- **Blocking operations**: cannot generate any output until all input is consumed
  - E.g., sorting, aggregation, …
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - E.g., for sort: run generation and merge
  - For hash join: partitioning and build-probe
- Treat them as separate operations



(a) Logical Query (b) Pipelined Plan

Pipeline Stages

- **Pipeline stages**:
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution



(a) Logical Query (b) Pipelined Plan

Evaluation Algorithms for Pipelining

- Some algorithms are not able to output results even as they get input tuples
  - E.g., merge join, or hash join
  - intermediate results written to disk and then read back
- Algorithm variants to generate (at least some) results on the fly, as input tuples are read in

    - E.g., hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
    - **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
        - When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
        - Symmetrically for s_0 tuples

Pipelining for Continuous-Stream Data

- **Data streams**
    - Data entering database in a continuous manner
    - E.g., Sensor networks, user clicks, ...
- **Continuous queries**
    - Results get updated as streaming data enters the database
    - Aggregation on windows is often used
        - E.g., **tumbling windows** divide time into units, e.g., hours, minutes
- Need to use pipelined processing algorithms
    - **Punctuations** used to infer when all data for a window has been received

Query Processing in Memory

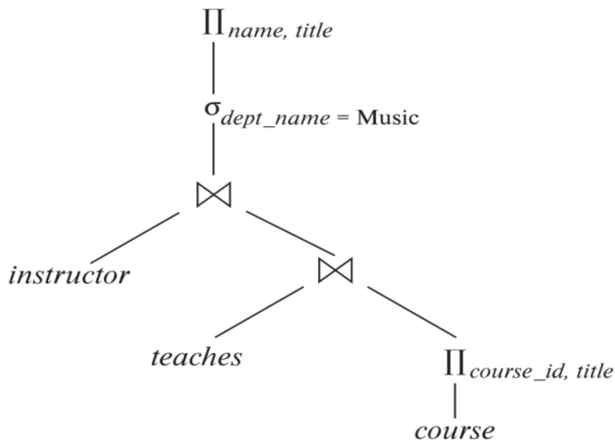
- Query compilation to machine code
    - Overheads of interpretation
        - E.g., repeatedly finding attribute location within tuple, from metadata
        - Overhead of expression evaluation
    - Compilation can avoid many such overheads and speed up query processing
    - Often via generation of Java byte code / LLVM, with just-in-time (JIT) compilation
- Column-oriented storage
    - Allows vector operations (in conjunction with compilation)
- Cache conscious algorithms

Cache Conscious Algorithms

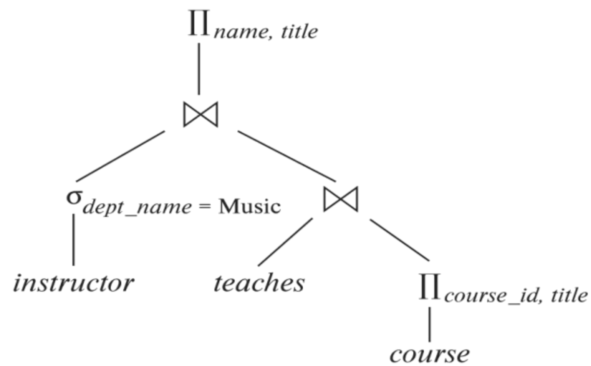
- Goal: minimize cache misses, make best use of data fetched into the cache as part of a cache line
- For sorting:
    - Use runs that are as large as L3 cache (a few megabytes) to avoid cache misses during sorting of a run
    - Then merge runs as usual in merge-sort
- For hash-join
    - First create partitions such that build+probe partitions fit in memory
    - Then subpartition further s.t. build subpartition+index fits in L3 cache
        - Speeds up probe phase significantly by avoiding cache misses
- Lay out attributes of tuples to maximize cache usage
    - Attributes that are often accessed together should be stored adjacent to each other
- Use multiple threads for parallel query processing
    - Cache misses leads to stall of one thread, but others can proceed

Query Optimization

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation

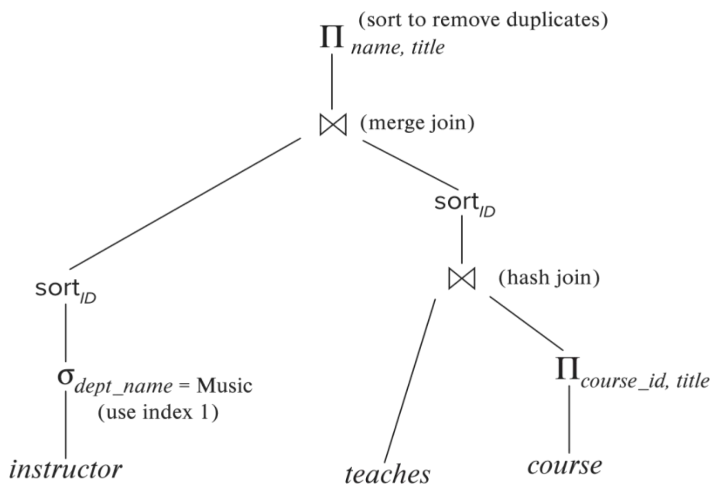


(a) Initial expression tree



(b) Transformed expression tree

An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated



- Cost difference between evaluation plans for a query can be enormous
  - E.g., seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Viewing Query Evaluation Plans

- Most database support **explain** <query>
  - Displays plan chosen by query optimizer, along with cost estimates

- Some syntax variations between databases
  - Oracle: **explain plan for** <query> followed by **select** * **from** table (*dbms_xplan.display*)
  - SQL Server: **set showplan_text on**
- Some databases (e.g. PostgreSQL) support **explain analyse** <query>
  - Shows actual runtime statistics found by running the query, in addition to showing the plan
- Some databases (e.g. PostgreSQL) show cost as $f..l$
  - f is the cost of delivering first tuple and l is cost of delivering all results

Generating Equivalent Expressions

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
  - we don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

   $\sigma_{\theta 1 \wedge \theta 2}(E) \equiv \sigma_{\theta 1}(\sigma_{\theta 2}(E))$

2. Selection operations are commutative.

   $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) \equiv \sigma_{\theta 2}(\sigma_{\theta 1}(E))$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E))\ldots)) \equiv \prod_{L1}(E)$ where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$

   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) \equiv E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

5. Theta-join operations (and natural joins) are commutative.

   $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$

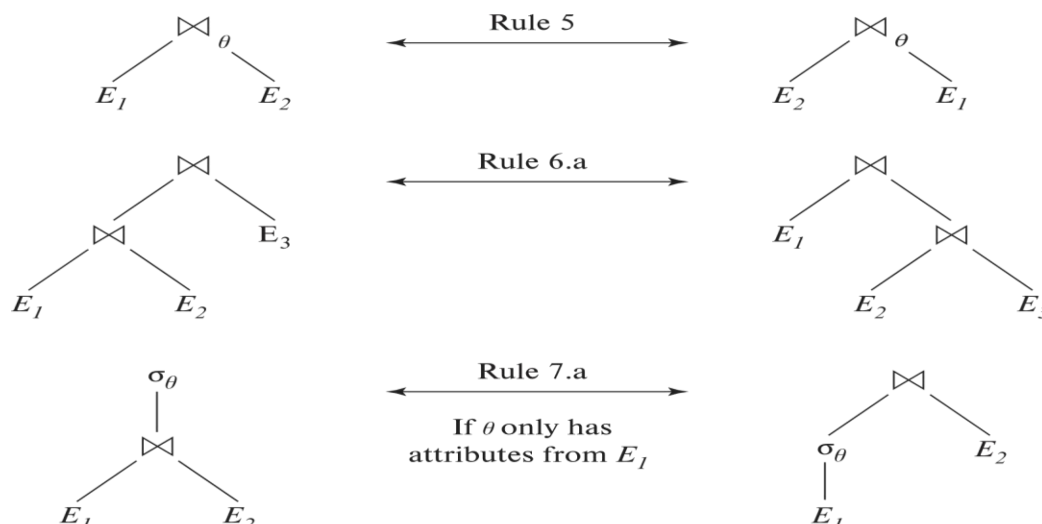
6. (a) Natural join operations are associative:

   $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$

   (b) Theta joins are associative in the following manner:

   $(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$

   where θ_2 involves attributes from only E_2 and E_3.



7. The selection operation distributes over the theta join operation under the following two conditions:

(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0} (E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

(b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2} (E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

8. The projection operation distributes over the theta join operation as follows:

(a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L1}(E_1) \bowtie_\theta \Pi_{L2}(E_2)$$

(b) In general, consider a join $E_1 \bowtie_\theta E_2$.

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
- let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \Pi_{L1 \cup L2}(\Pi_{L1 \cup L3}(E_1) \bowtie_\theta \Pi_{L2 \cup L4}(E_2))$$

Similar equivalences hold for outerjoin operations: ⟕, ⟖, and ⟗

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$

b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$

c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$

d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$

e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

preceding equivalence does not hold for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

13. Selection distributes over aggregation as below

$$\sigma_\theta({}_G\gamma_A(E)) \equiv {}_G\gamma_A(\sigma_\theta(E))$$

provided θ only involves attributes in G

14. a. Full outerjoin is commutative:

$$E_1 ⟗ E_2 \equiv E_2 ⟗ E_1$$

b. Left and right outerjoin are not commutative, but:

$$E_1 ⟕ E_2 \equiv E_2 ⟖ E_1$$

15. Selection distributes over left and right outerjoins as below, provided θ_1 only involves attributes of E_1

a. $\sigma_{\theta 1} (E_1 ⟕_\theta E_2) \equiv (\sigma_{\theta 1} (E_1)) ⟕_\theta E_2$

b. $\sigma_{\theta 1} (E_1 ⟖_\theta E_2) \equiv E_2 ⟕_\theta (\sigma_{\theta 1} (E_1))$

16. Outerjoins can be replaced by inner joins under some conditions

a. $\sigma_{\theta 1}$ (E_1 ⟖$_\theta$ E_2) ≡ $\sigma_{\theta 1}$ (E_1 $\bowtie_\theta$ E_2)
b. $\sigma_{\theta 1}$ (E_1 ⟕$_\theta$ E_1) ≡ $\sigma_{\theta 1}$ (E_1 $\bowtie_\theta$ E_2)
provided θ_1 is null rejecting on E_2

Note that several equivalences that hold for joins do not hold for outerjoins

- $\sigma_{year=2017}$(*instructor* ⟖ *teaches*) ≢ $\sigma_{year=2017}$(*instructor* ⋈ *teaches*)
- Outerjoins are not associative
  (r ⟖ s) ⟖ t ≢ r ⟖ (s ⟖ t)
    - e.g. with r(A,B) = {(1,1), s(B,C) = { (1,1)}, t(A,C) = { }

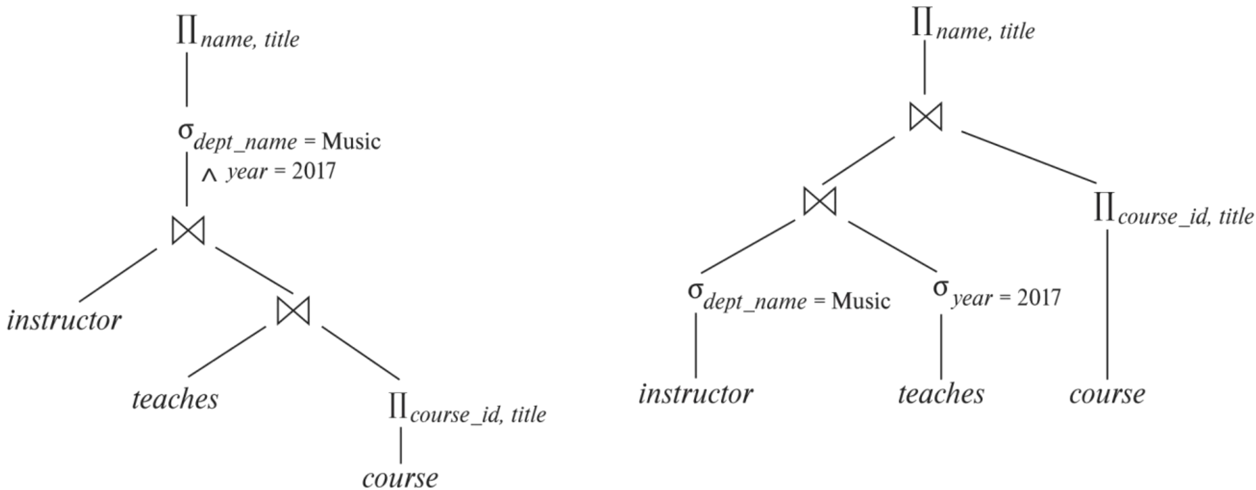
Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
    - $\Pi_{name,\ title}(\sigma_{dept_name=\text{'Music'}}$
      (*instructor* ⋈ *(teaches* ⋈ $\Pi_{course_id,\ title}$ (*course*))))
- Transformation using rule 7a.
    - $\Pi_{name,\ title}((\sigma_{dept_name=\text{'Music'}}$(*instructor*)) ⋈
      (teaches ⋈ $\Pi_{course_id,\ title}$ (*course*)))
- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
    - $\Pi_{name,\ title}(\sigma_{dept_name=\text{"Music"} \wedge year = 2017}$
      (*instructor* ⋈ *(teaches* ⋈ $\Pi_{course_id,\ title}$ (*course*))))
- Transformation using join associatively (Rule 6a):
    - $\Pi_{name,\ title}(\sigma_{dept_name=\text{"Music"} \wedge year = 2017}$
      ((*instructor* ⋈ *teaches)* ⋈ $\Pi_{course_id,\ title}$ (*course*)))
- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

$\sigma_{dept_name = \text{"Music"}}$ (*instructor*) ⋈ $\sigma_{\ year = 2017}$ (*teaches*)



(a) Initial expression tree (b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept_name=\text{"Music"}}$ (*instructor)* ⋈ *teaches*)
  ⋈ $\Pi_{course_id,\ title}$ (*course*))))
- When we compute
  ($\sigma_{dept_name = \text{"Music"}}$ (*instructor* ⋈ *teaches*)

we obtain a relation whose schema is:
(*ID, name, dept_name, salary, course_id, sec_id, semester, year)*

- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$$\Pi_{name,\ title}(\Pi_{name,\ course_id}(\sigma_{dept_name=\text{"Music"}}(instructor) \bowtie teaches)) \bowtie \Pi_{course_id,\ title}(course))))$$

- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,
  $(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$
  (Join Associativity) $\bowtie$
- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose
  $(r_1 \bowtie r_2) \bowtie r_3$
  so that we compute and store a smaller temporary relation.

- Consider the expression

$$\Pi_{name,\ title}(\sigma_{dept_name=\text{"Music"}}(instructor) \bowtie teaches) \bowtie \Pi_{course_id,\ title}(course))))$$

- Could compute $teaches \bowtie \Pi_{course_id,\ title}(course)$ first, and join result with
  $\sigma_{dept_name=\text{"Music"}}(instructor)$
  but the result of the first join is likely to be a large relation.
- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute
    $\sigma_{dept_name=\text{"Music"}}(instructor) \bowtie teaches$ first.

Enumeration of Equivalent Expressions

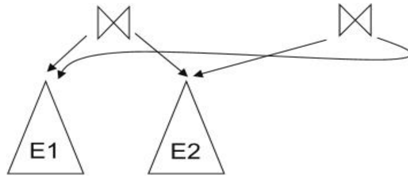
- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers

- E.g., when applying join commutativity



- Same sub-expression may get generated multiple times
  - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computer as described in Chapter 15
  - Need statistics of input relations
    - E.g., number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g., number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ . \ . \ . \bowtie r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of
  $\{r_1, r_2, . . . r_n\}$ is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of n relations:
  - To find best plan for a set S of n relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of S.
  - Recursively compute costs for joining subsets of S to find the cost of each plan. Choose the cheapest of the $2^n - 2$ alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Dynamic programming

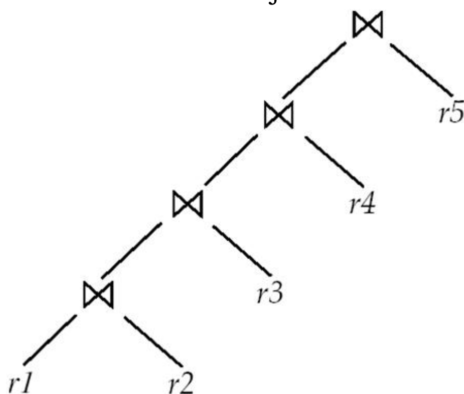
Join Order Optimization Algorithm

```
procedure findbestplan(S)
if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
// else bestplan[S] has not been computed earlier, compute it now
if (S contains only 1 relation)
   set bestplan[S].plan and bestplan[S].cost based on the best way
   of accessing S using selections on S and indices (if any) on S else for each non-empty subset S1 of S such
that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        for each algorithm A for joining results of P1 and P2
          ... compute plan and cost of using A (see next page) ..
            if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = plan;
return bestplan[S]
```

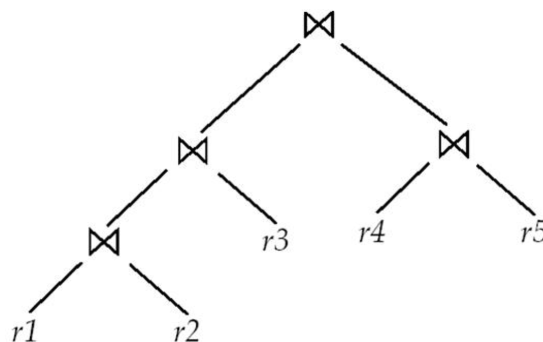
```
for each algorithm A for joining results of P1 and P2
 // For indexed-nested loops join, the outer could be P1 or P2
 // Similarly for hash-join, the build relation could be P1 or P2
 //  We assume the alternatives are considered as separate algorithms
     if algorithm A is indexed nested loops
    Let Pi and Po denote inner and outer inputs
    if Pi has a single relation ri and ri has an index on the join attribute
            plan = "execute Po.plan; join results of Po and ri using A",
                  with any selection conditions on Pi performed as part of
                   the join condition
         cost = Po.cost + cost of A
     else  cost = ∞; /* cannot use indexed nested loops join */
else
     plan = "execute P1.plan; execute P2.plan;
                      join results of P1 and P2 using A;"
     cost = P1.cost + P2.cost + cost of A
```

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree



(b) Non-left-deep join tree

c

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With n = 10, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "**for each** non-empty subset $S1$ of S such that $S1 \neq S$"
    - By: **for each** relation r in S
      let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could make a later operation (join/group by/order by) cheaper
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
- Not sufficient to find the best join order for each subset of the set of n given relations
  - must find the best join order for each subset, **for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.
- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on **memoization**, which stores the best plan for a subexpression the first time it is optimized, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans
- Pioneered by the Volcano project and implemented in the SQL Server optimizer

Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.

- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g., nested subqueries
- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
  - **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Statistics for Cost Estimation

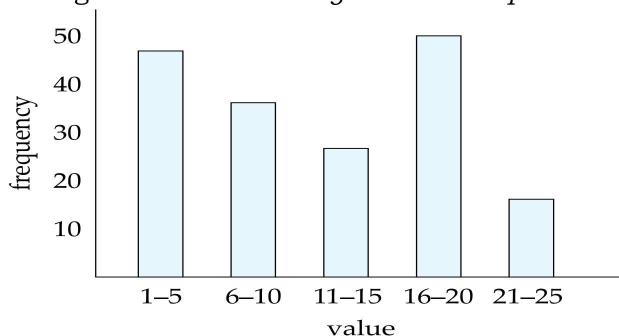
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- $V(A, r)$: number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Histograms

- Histogram on attribute *age* of relation *person*



- Equi-width histograms

- Equi-depth histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* most-frequent values and their counts
  - Histogram is built on remaining values only
- Histograms and other statistics usually computed based on a random sample
- Statistics may be out of date
  - Some database require a analyze command to be executed to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .

- If s_i is the number of satisfying tuples in *r*, the selectivity of θ_i is given by s_i / n_r.

- **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_n}(r)$. *Assuming independence,* estimate of tuples in the result is:

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_n}(r)$. Estimated number of tuples: $n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$
- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:
  $n_r - size(\sigma_\theta(r))$

Join Operation: Running Example

Running example: *student* ⋈ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*
  - *V(ID, student)* = 5000 (*primary key!*)

Estimation of the Size of Joins

- The Cartesian product *r* x *s* contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.

- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *student* ⋈ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000
- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in R S, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:

  - $V(ID, takes) = 2500$, and
    $V(ID, student) = 5000$
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r) = V(A,r)$
- Aggregation : estimated size of ${}_G\gamma_A(r) = V(G,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g., $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \text{ or } \theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s = r$.
    - <u>All the three estimates may be quite inaccurate, but provide upper bounds on the sizes</u>.
- Outer join:
  - Estimated size of r ⟕ s = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of r ⟗ s = *size of* $r \bowtie s$ + size of r + size of s

Estimation of Number of Distinct Values

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
    - e.g., $A = 3$

- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
    - (e.g., ($A = 1 \lor A = 3 \lor A = 4$)),
- If the selection condition θ is of the form *A op r*
  estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
    - where *s* is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  $\min(V(A,r),\ n_{\sigma\theta(r)})$
    - More accurate estimate can be got using probability theory, but this one works fine generally

Joins: $r \bowtie s$
- If all attributes in *A* are from *r*
  estimated $V(A,\ r \bowtie s) = \min(V(A,r),\ n_{r \bowtie s})$
- If *A* contains attributes *A*1 from *r* and *A*2 from *s*, then estimated
  $V(A,r \bowtie s) =$
  $\min(V(A1,r)*V(A2 - A1,s),\ V(A1 - A2,r)*V(A2,s),\ n_{r \bowtie s})$
    - More accurate estimate can be got using probability theory, but this one works fine generally

- Estimation of distinct values are straightforward for projections.
    - They are the same in $\prod_{A}(r)$ as in *r*.
- The same holds for grouping attributes of aggregation.
- For aggregated values
    - For min(*A*) and max(*A*), the number of distinct values can be estimated as min(V(*A,r*), *V*(*G,r*)) where G denotes grouping attributes
    - For other aggregates, assume all values are distinct, and use *V*(*G,r*)

Optimizing Nested Subqueries**
- Nested query example:
  select *name*
  from *instructor*
  where exists (**select** *
  from *teaches*
  where *instructor.ID* = *teaches.ID* **and** *teaches.year* = *2019*)
- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
    - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
    - Such evaluation is called **correlated evaluation**
    - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery
- Correlated evaluation may be quite inefficient since
    - a large number of calls may be made to the nested query
    - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.,: earlier nested query can be rewritten as
  $\prod_{name}(instructor \bowtie_{instructor.ID=teaches.ID \land teaches.year=2019} teaches)$
- Note: the two queries generate different numbers of duplicates (why?)
    - Can be modified to handle duplicates correctly using semijoins
- The **semijoin** operator ⋉ is defined as follows

- A tuple r_i appears n times in $r \ltimes_\theta s$ if it appears n times in r, and there is at least one matching tuple s_i in s

- E.g.: earlier nested query can be rewritten as
  $\Pi_{name}(instructor \ltimes_{instructor.ID=teaches.ID \wedge teaches.year=2019} teaches)$
  - Or even as: $\Pi_{name}(instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{teaches.year=2019}\ teaches))$
  - Now the duplicate count is correct!
- The above relational algebra query is also equivalent to
  from *instructor*
  where *ID* **in** (**select** *teaches.ID*
  from *teaches*
  where *teaches.year = 2019*)

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** *A*
  from $r_1, r_2, \ldots, r_n$
  where P_1 **and exists** (**select** *
  from $s_1, s_2, \ldots, s_m$
  where P_2^1 **and** P_2^2)
- To: $\Pi_A(\sigma_{P1}(r_1 \times r_2 \times \ldots \times r_n) \ltimes_{P22} \sigma_{P21}(s_1 \times s_2 \times \ldots \times s_m))$
  - P_2^1 contains predicates that do not involve any correlation variables
  - P_2^2 contains predicates involving correlation variables
- The process of replacing a nested query by a query with a join/semijoin (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated in several cases, e.g.
  - The nested subquery uses aggregation, or
  - The nested subquery is a scalar subquery
  - Correlated evaluation used in these cases
- Decorrelation of scalar aggregate subqueries can be done using groupby/aggregation in some cases
- **select** *name*
  from *instructor*
  where 1 < (**select count**(*)
  from *teaches*
  where *instructor.ID = teaches.ID*
  and *teaches.year = 2019*)
- $\Pi_{name}(instructor \ltimes_{instructor.ID=TID \wedge 1 < cnt} ({}_{ID\ as\ TID}\gamma_{\mathbf{count}(*)\ as\ cnt}(\sigma_{teaches.year=2019}(teaches))))$

Materialized Views

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  c**reate view** *department_total_salary*(*dept_name, total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*
- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update

- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to the materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Incremental maintenance supported by many database systems
    - Avoids manual effort/correctness issues

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - $R = (A,B)$, and r(R) = { $(a,2)$, $(a,3)$}
  - $\prod_A(r)$ has a single tuple (a).
  - If we delete the tuple $(a,2)$ from r, we should not delete the tuple (a) from $\prod_A(r)$, but if we then delete $(a,3)$ as well, we should delete the tuple
- For each tuple in a projection $\prod_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\prod_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\prod_A(r)$
    - if the count becomes 0, we delete the tuple from $\prod_A(r)$

Aggregation Operations

- **Count** : $v = {}_A\gamma_{count(B)}(r)$.
  - When a set of tuples i_r is inserted
    - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted

- for each tuple t in i_r we look for the group *t.A* in *v*, and subtract 1 from the count for the group.
  - If the count becomes 0, we delete from *v* the tuple for the group *t.A*

- **Sum**: $v = {}_A\gamma_{sum(B)}(r)$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)
- **Avg**: How to handle average?
  - Maintain **sum** and **count** separately, and divide at the end
- **min**, **max**: $v = {}_A\gamma_{min(B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of *r* that are in the same group to find the new minimum

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in *r* we check if it is present in *s*, and if so we add it to *v*.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to *s* are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g., consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that *s* has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace *v* by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: "What is the best set of views to materialize?"
- **Index selection**: "what is the best set of indices to create"

        - closely related, to materialized view selection
            - but simpler
- Materialized view selection and index selection based on typical system **workload** (queries and updates)
    - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
    - One of the steps in database tuning
        - more on tuning in later chapters
- Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

Top-K Queries

select *
from r, s
where r.B = s.B
order by r.A **ascending**
limit 10

- Alternative 1: Indexed nested loops join with r as outer
- Alternative 2: estimate highest r.A value in result and add selection (**and** r.A <= H) to where clause
    - If < 10 results, retry with larger H

Optimization of Updates

- **Halloween problem**

  update R **set** A = 5 * A
  where A > 10
- If index on A is used to find tuples satisfying A > 10, and tuples updated immediately, same tuple may be found (and updated) multiple times
- Solution 1: *Always defer updates*
    - collect the updates (old and new values of tuples) and update relation and indices in second pass
    - Drawback: extra overhead even if e.g. update is only on R.B, not on attributes in selection condition
- Solution 2: *Defer only if required*
    - Perform immediate update if update does not affect attributes in where clause, and deferred updates otherwise.

Join Minimization

select r.A, r.B
from r, s
where r.B = s.B

- Check if join with s is redundant, drop it
    - E.g., join condition is on foreign key from r to s, r.B is declared as not null, and no selection on s
    - Other sufficient conditions possible

      select r.A, s2.B
      from r, s **as** s1, s **as** s2
      where r.B=s1.B **and** r.B = s2.B **and** s1.A < 20 **and** s2.A < 10

        - join with s1 is redundant and can be dropped (along with selection on s1)
    - Lots of research in this area since 70s/80s!

Multiquery Optimization

- Example

  Q1: **select** * **from** (r **natural join** t) **natural join** s
  Q2: **select** * **from** (r **natural join** u) **natural join** s

    - Both queries share common subexpression (r natural join s)
    - May be useful to compute (r natural join s) once and use it in both queries
        - But this may be more expensive in some situations
            - e.g. (r natural join s) may be expensive, plans as shown in queries may be cheaper
- **Multiquery optimization**: find best overall plan for a set of queries, exploiting sharing of common subexpressions between queries where it is useful
- Simple heuristic used in some database systems:
    - optimize each query separately
    - detect and exploiting common subexpressions in the individual optimal query plans
        - May not always give best plan, but is cheap to implement
    - **Shared scans**: widely used special case of multiquery optimization
- Set of materialized views may share common subexpressions
    - As a result, view maintenance plans may share subexpressions
    - Multiquery optimization can be useful in such situations

Parametric Query Optimization

- Example

  select *
  from r **natural join** s
  where r.a < $1

    - value of parameter $1 not known at compile time
        - known only at run time
    - different plans may be optimal for different values of $1
- Solution 1: optimize at run time, each time query is submitted
        - can be expensive
- Solution 2: **Parametric Query Optimization**:
    - optimizer generates a set of plans, optimal for different values of $1
        - Set of optimal plans usually small for 1 to 3 parameters
        - Key issue: how to do find set of optimal plans efficiently
    - best one from this set is chosen at run time when $1 is known
- Solution 3: **Query Plan Caching**
    - If optimizer decides that same plan is likely to be optimal for all parameter values, it caches plan and reuses it, else re-optimize each time
    - Implemented in many database systems

Plan Stability Across Optimizer Changes

- What if 95% of plans are faster on database/optimizer version N+1 than on N, but 5% are slower?
  - Why should plans be slower on new improved optimizer?
    - Answer: Two wrongs can make a right, fixing one wrong can make things worse!
- Approaches:
  - Allow hints for tuning queries
    - Not practical for migrating large systems with no access to source code
  - Set optimization level, default to version N (Oracle)
    - And migrate one query at a time after testing both plans on new optimizer
  - Save plan from version N, and give it to optimizer version N+1
    - Sybase, XML representation of plans (SQL Server)

Adaptive Query Processing

- Some systems support adaptive operators that change execution algorithm on the fly
  - E.g., (indexed) nested loops join or hash join chosen at run time, depending on size of outer input
- Other systems allow monitoring of behavior of plan at run time and adapt plan
  - E.g., if statistics such as number of rows is found to be very different in reality from what optimizer estimated

Unit 5

Transaction Management: Transactions: Concept, A Simple Transactional Model, Storage Structures, Transaction Atomicity and Durability, Transaction Isolation, Serializability, Isolation and Atomicity, Transaction Isolation Levels, Implementation of Isolation Levels, Transactions as SQL Statements.

Concurrency Control: Lock-based Protocols, Deadlock Handling, Multiple granularity, Timestamp-based Protocols, and Validation-based Protocols.

Recovery System: Failure Classification, Storage, Recovery and Atomicity, Recovery Algorithm, Buffer Management, Failure with Loss of Nonvolatile Storage, Early Lock Release and Logical Undo Operations.

A transaction is a unit of program execution that accesses and possibly updates various data items.

E.g., transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Example of Fund Transfer

Transaction to transfer $50 from account A to account B:

1. read(A)
2. A := A – 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

- Atomicity requirement - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state

Failure could be due to software or hardware

The system should ensure that updates of a partially executed transaction are not reflected in the database

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
- Consistency requirement in above example:

The sum of A and B is unchanged by the execution of the transaction

In general, consistency requirements include:

Explicitly specified integrity constraints such as primary keys and foreign keys

Implicit integrity constraints - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

A transaction must see a consistent database.

During transaction execution the database may be temporarily inconsistent.

When the transaction completes successfully the database must be consistent - Erroneous transaction logic can lead to inconsistency

- Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum A + *B* will be less than it should be).

	T1	T2
1.	**read**(*A*)	
2.	*A* := *A* – 50	

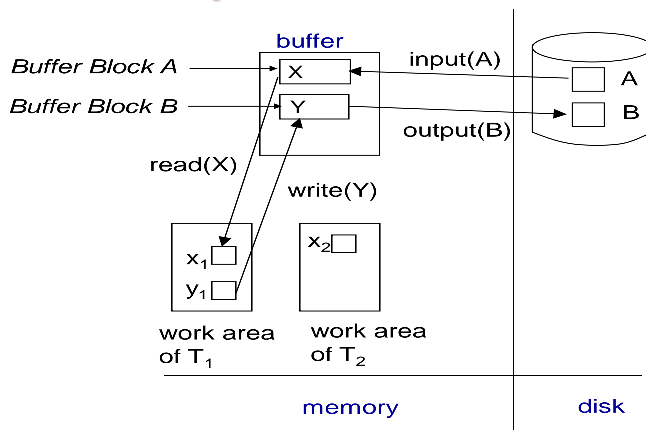
3. **write**(*A*)

read(A), read(B), print(A+B)

4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits.

Example of Data Access



Database System Concepts - 6th Edition 16.7 ©Silberschatz, Korth and Sudarshan

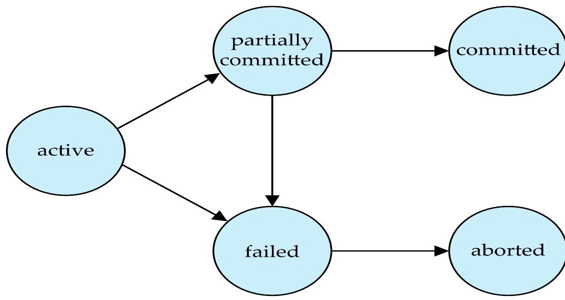
ACID Properties

A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - Restart the transaction - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp$:= A * 0.1 A := A - $temp$ write (A) read (B) B := B + $temp$ write (B) commit
read (A) A := A – 50 write (A) read (B) B := B + 50 write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A) A := A – 50 write (A)	
	read (A) $temp$:= A * 0.1 A := A - $temp$ write (A)
read (B) B := B + 50 write (B) commit	
	read (B) B := B + $temp$ write (B) commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of (A + B).

T_1	T_2
read (A) A := A – 50	
	read (A) $temp$:= A * 0.1 A := A - $temp$ write (A) read (B)
write (A) read (B) B := B + 50 write (B) commit	
	B := B + $temp$ write (B) commit

Serializability

Basic Assumption – Each transaction preserves database consistency.

Thus, serial execution of a set of transactions preserves database consistency.

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. Conflict serializability
2. View Serializability

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
    1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
    2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
    3. l_i = **write**(Q), l_j = **read**(Q). They conflict
    4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Anomalies with Interleaved Execution

1. Reading Uncommitted Data (WR Conflicts, "**dirty reads**"):

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
	C
read (B)	
write (B)	
Abbort	

2. Unrepeatable Reads (RW Conflicts):

T_1	T_2
read (A)	
	read (A)
	write (A)
	C
read (A)	
write (B)	
C	

3. Overwriting Uncommitted Data (WW Conflicts):

T_1	T_2
write (A)	
	write (A)
	write (B)
	C
write (B)	
C	

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

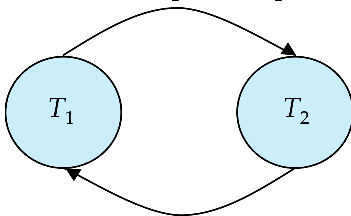
- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A) $A := A - 50$ write (A)	
	read (B) $B := B - 10$ write (B)
read (B) $B := B + 50$ write (B)	
	read (A) $A := A + 10$ write (A)

- Determining such equivalence requires analysis of operations other than read and write.

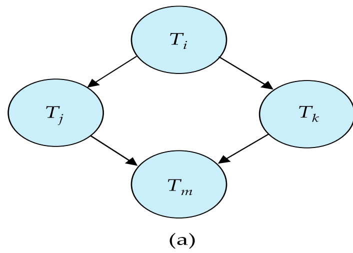
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph:

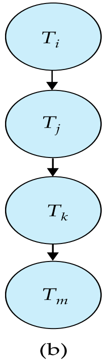


Test for Conflict Serializability

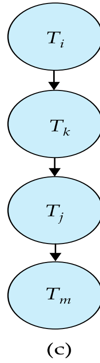
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which takes order n^2 time, where n is the number of vertices in the graph.
    - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
    - This is a linear order consistent with the partial order of the graph.
    - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
        - Are there others?



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A) read (B) write (A)		
	read (A) write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.
- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
  - E.g. In SQL **set transaction isolation level serializable**
  - E.g. in JDBC - connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?

- Instance of the **phantom phenomenon**

- Also consider T3 below, with Wu's salary = 90000

  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'

- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

Concurrency Control

Purpose of Concurrency Control

To enforce Isolation (through mutual exclusion) among conflicting transactions.
To preserve database consistency through consistency preserving execution of transactions.
To resolve read-write and write-write conflicts.

Example:
In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques: Essential components

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

	S	X
S	true	false
X	false	false

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item
Data items can be locked in two modes :

1. exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. shared (S) mode. Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.
Lock-compatibility matrix

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

Any number of transactions can hold shared locks on an item,

But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

```
T2: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
```

Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B, the displayed sum would be wrong.

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

The Two-Phase Locking Protocol

This protocol ensures conflict-serializable schedules.

Phase 1: Growing Phase

Transaction may obtain locks

Transaction may not release locks

Phase 2: Shrinking Phase

Transaction may release locks

Transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction Ti that does not follow two-phase locking, we can find a transaction Tj that uses two-phase locking, and a schedule for Ti and Tj that is not conflict serializable.

Lock Conversions

Two-phase locking with lock conversions:

- First Phase:
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (upgrade)
- Second Phase:
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

A transaction Ti issues the standard read/write instruction, without explicit locking calls.

The operation read(D) is processed as:

```
if Ti has a lock on D
   then
        read(D)
   else begin
           if necessary wait until no other
              transaction has a lock-X on D
           grant Ti a  lock-S on D;
           read(D)
        end
```

write(D) is processed as:

```
if Ti has a  lock-X on D
   then
     write(D)
  else begin
      if necessary wait until no other transaction has any lock on D,
      if Ti has a lock-S on D
           then
             upgrade lock on D  to lock-X
          else
             grant Ti a lock-X on D
          write(D)
    end;
```

All locks are released after commit or abort

Deadlocks

Consider the partial schedule

T_3	T_4
lock-x (*B*)	
read (*B*)	
B := *B* - 50	
write (*B*)	
	lock-s (*A*)
	read (*A*)
	lock-s (*B*)
lock-x (*A*)	

Neither T3 nor T4 can make progress — executing lock - S(B) causes T4 to wait for T3 to release its lock on B, while executing lock-X(A) causes T3 to wait for T4 to release its lock on A.
Such a situation is called a deadlock.
To handle a deadlock one of T3 or T4 must be rolled back and its locks released.

Two-phase locking does not ensure freedom from deadlocks.
In addition to deadlocks, there is a possibility of starvation.
Starvation occurs if the concurrency control manager is badly designed.

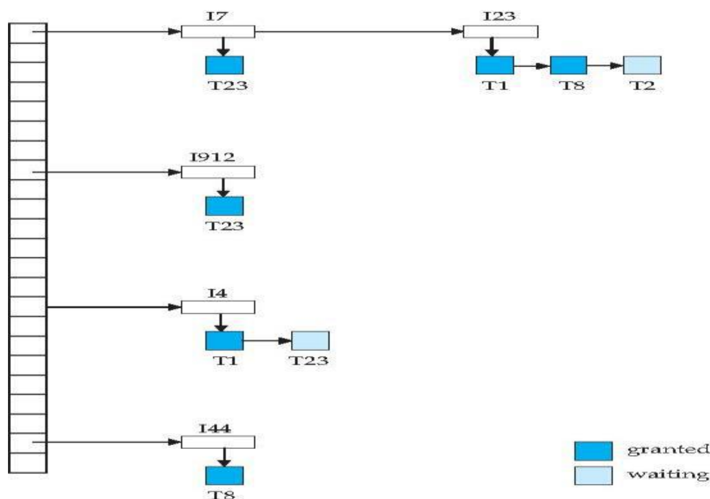
For example:
A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
The same transaction is repeatedly rolled back due to deadlocks.
Concurrency control manager can be designed to prevent starvation.

The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
When a deadlock occurs there is a possibility of cascading roll-backs.
Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking -- a transaction must hold all its exclusive locks till it commits/aborts.
Rigorous two-phase locking is even stricter. Here, all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Implementation of Locking

- A lock manager can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a lock table to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
- lock manager may keep a list of locks held by each transaction, to implement this efficiently

Deadlock Handling

System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
Deadlock prevention protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (predeclaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order.

More Deadlock Prevention Strategies

Following schemes use transaction timestamps for the sake of deadlock prevention alone.

wait-die scheme — non-preemptive

- older transaction may wait for younger one to release data item. (older means smaller timestamp) Younger transactions never Younger transactions never wait for older ones; they are rolled back instead.
- a transaction may die several times before acquiring needed data item

wound-wait scheme — preemptive

- older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
- may be fewer rollbacks than wait-die scheme.

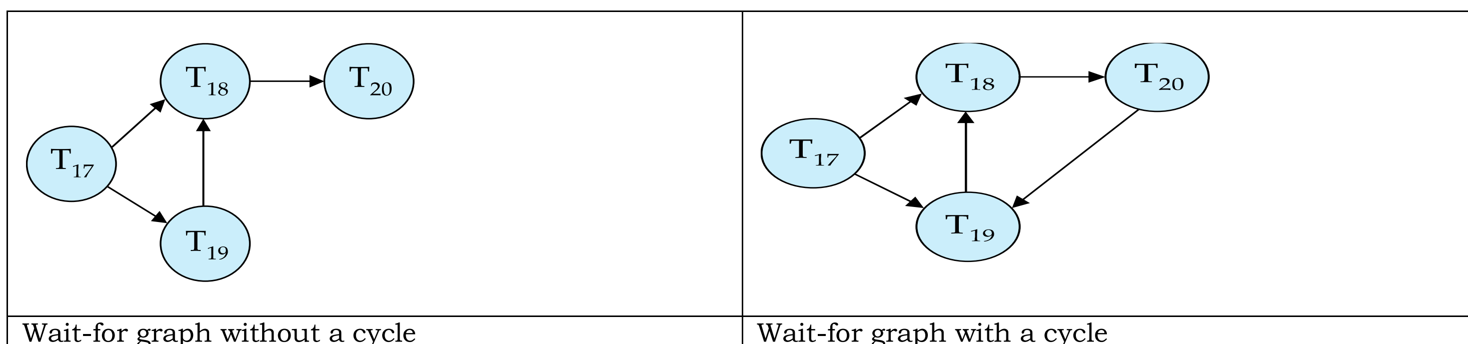
Both in wait-die and in wound-wait schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Timeout-Based Schemes:

- A transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
- Thus, deadlocks are not possible
- simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

- Deadlocks can be described as a wait-for graph, which consists of a pair G = (V,E),
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair Ti -> Tj.
- If Ti -> Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- When Ti requests a data item currently being held by Tj, then the edge Ti -> Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle	Wait-for graph with a cycle

Deadlock Recovery

When deadlock is detected: Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.

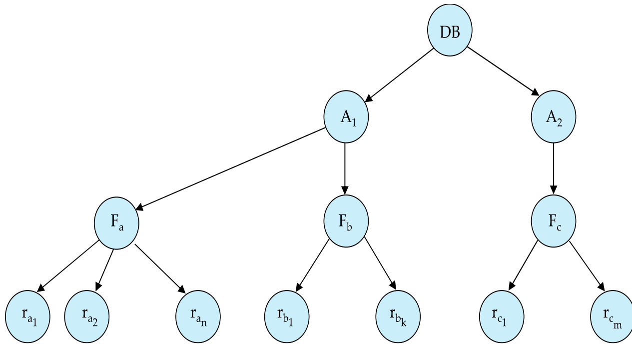
Rollback -- determine how far to roll back transaction

- Total rollback: Abort the transaction and then restart it.
- More effective to roll back transaction only as far as necessary to break deadlock.

Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree.
- When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - fine granularity (lower in tree): high concurrency, high locking overhead
  - coarse granularity (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
  - The lock compatibility matrix must be observed.
  - The root of the tree must be locked first, and may be locked in any mode.

- A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
- A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
- T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

* Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
* **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Timestamp-Based Protocols

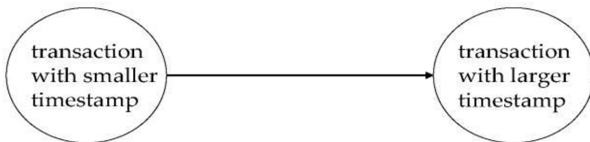
* Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that $TS(T_i) < TS(T_j)$.
* The protocol manages concurrent execution such that the time-stamps determine the serializability order.
* In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
    - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
    - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
* The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
* Suppose a transaction T_i issues a **read**(Q)
    - If $TS(T_i) \leq$ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
        - Hence, the **read** operation is rejected, and T_i is rolled back.
    - If $TS(T_i) \geq$ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).
* Suppose that transaction T_i issues **write**(Q).
    - If $TS(T_i) <$ R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
        - Hence, the **write** operation is rejected, and T_i is rolled back.
    - If $TS(T_i) <$ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
        - Hence, this **write** operation is rejected, and T_i is rolled back.
    - Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

* A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

* The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i
  - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by T_j must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a "validation test'' to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - Assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation
- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase

- Serializability order is determined by timestamp given at validation time; this is done to increase concurrency.
    - Thus, TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
    - because the serializability order is not pre-decided, and
    - relatively few transactions will have to be rolled back.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
    - **finish**(T_i) < **start**(T_j)
    - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.
- then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
    - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
    - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

Schedule Produced by Validation

T_{25}	T_{26}
read (B)	
	read (B) $B := B$ 50 read (A) $A := A + 50$
read (A) ⟨ *validate* ⟩ display ($A + B$)	
	⟨ *validate* ⟩ write (B) write (A)

Recovery System

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Consider transaction T_i that transfers $50 from account A to account B
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after one of these modifications have been made but before both of them are made.
  - Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
- Recovery algorithms have two parts
  - Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  - Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
  - but may still fail, losing data
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media
  - See book for more details on how to implement stable storage

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical block.

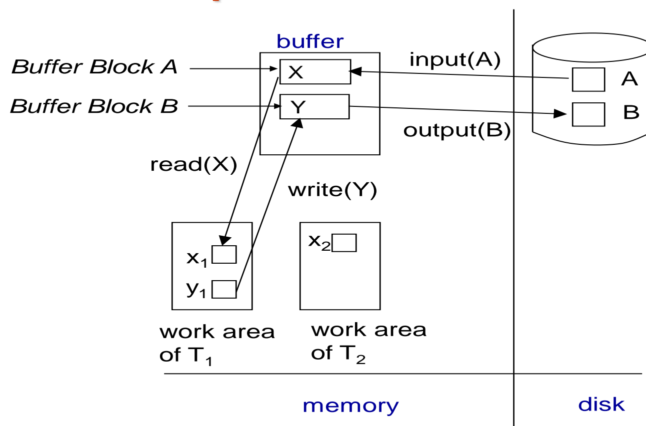
2. When the first write successfully completes, write the same information onto the second physical block.
3. The output is completed only after the second write successfully completes.

- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
     1. *Expensive solution*: Compare the two copies of every disk block.
     2. *Better solution*:
        - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
        - Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these.
        - Used in hardware RAID systems
  2. If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



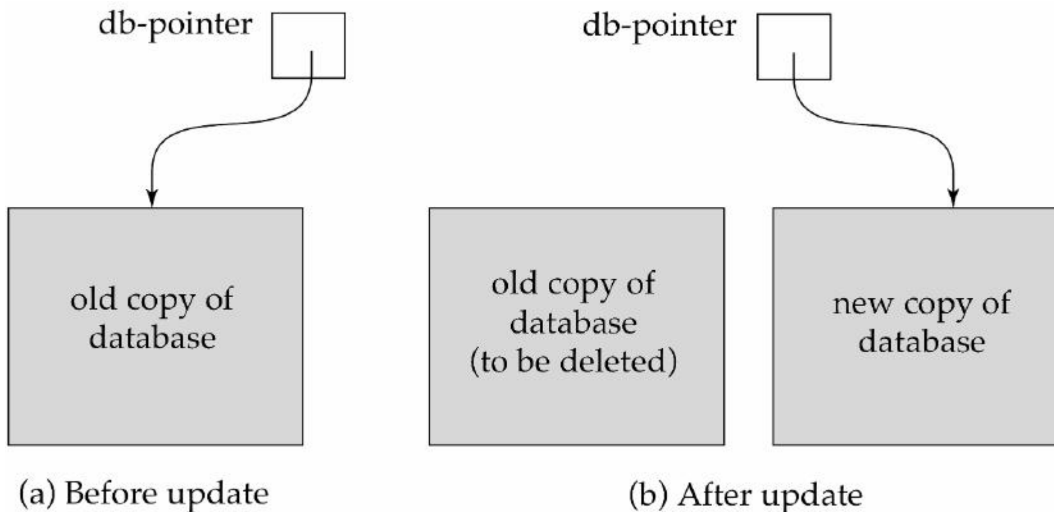
 b

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item *X* is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(*X*) assigns the value of data item *X* to the local variable x_i.
  - **write**(*X*) assigns the value of local variable x_i to data item {*X*} in the buffer block.
  - **Note: output**(B_X) need not immediately follow **write**(*X*). System can perform the **output** operation when it deems fit.
- Transactions

- Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
- **write**(X) can be executed at any time before the transaction commits

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study **log-based recovery mechanisms** in detail
  - We first present key concepts
  - And then present the actual recovery algorithm
- Less used alternative: **shadow-copy** and **shadow-paging**



(a) Before update (b) After update

Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  <T_i **start**>log record
- *Before* T_i executes **write**(X), a log record
  <T_i, X, V_1, V_2>
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record <T_i **commi**t> is written.
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Immediate Database Modification

- The **immediate-modification** scheme allows updates of an uncommitted transaction to be made to the buffer, or the disk itself, before the transaction commits
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - (Will see later that how to postpone log record output to some extent)
- Output of updated blocks to stable storage can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

- The **deferred-modification** scheme performs updates to buffer/disk only at the time of transaction commit
    - Simplifies some aspects of recovery
    - But has overhead of storing local copy

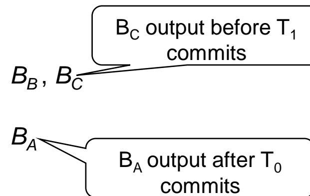
Transaction Commit

- A transaction is said to have committed when its commit log record is output to stable storage
    - all previous log records of the transaction must have been output already
- Writes performed by a transaction may still be in the buffer when the transaction commits, and may be output later

Immediate Database Modification Example

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_o, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A



- Note: B_X denotes block containing X.

Database System Concepts - 6th Edition 16.15 ©Silberschatz, Korth and Sudarshan

Concurrency Control and Recovery

- With concurrent transactions, all transactions share a single disk buffer and a single log
    - A buffer block can have data items updated by one or more transactions
- We assume that *if a transaction T_i has modified an item, no other transaction can modify the same item until T_i has committed or aborted*
    - i.e. the updates of uncommitted transactions should not be visible to other transactions
        - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
    - Can be ensured by obtaining exclusive locks on updated items and holding the locks till end of transaction (strict two-phase locking)
- Log records of different transactions may be interspersed in the log.

Undo and Redo Operations

- **Undo** of a log record <T_i, X, V_1, V_2> writes the **old** value V_1 to X
- **Redo** of a log record <T_i, X, V_1, V_2> writes the **new** value V_2 to X
- **Undo and Redo of Transactions**
    - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i

        - each time a data item X is restored to its old value V a special log record <T_i , X, V> is written out
        - when undo of a transaction is complete, a log record <T_i **abort**> is written out.
    - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
        - No logging is done in this case

Undo and Redo on Recovering from Failure

- When recovering after failure:
    - Transaction T_i needs to be undone if the log
        - contains the record <T_i **start**>,
        - but does not contain either the record <T_i **commit**> *or* <T_i **abort**>.
    - Transaction T_i needs to be redone if the log
        - contains the records <T_i **start**>
        - and contains the record <T_i **commit**> *or* <T_i **abort**>
- Note that If transaction T_i was undone earlier and the <T_i **abort**> record written to the log, and then a failure occurs, on recovery from failure T_i is redone
    - **such a redo redoes all the original actions *including the steps that restored old values***
        - Known as **repeating history**
        - Seems wasteful, but simplifies recovery greatly

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, abort> are written out

(b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, abort> are written out.

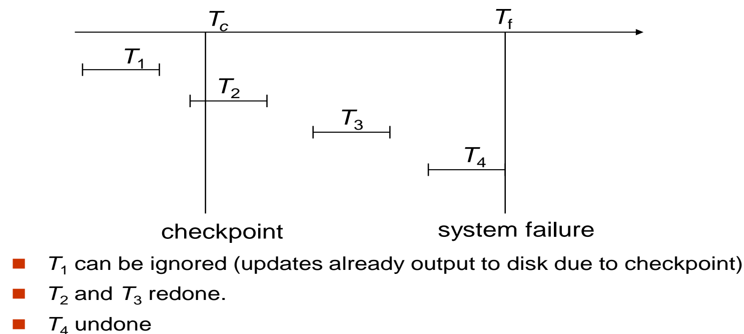
(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Checkpoints

- Redoing/undoing all transactions recorded in the log can be very slow
    - processing the entire log is time-consuming if the system has run for a long time
    - we might unnecessarily redo transactions which have already output their updates to the database.
- Streamline recovery procedure by periodically performing **checkpointing**
    - Output all log records currently residing in main memory onto stable storage.
    - Output all modified buffer blocks to the disk.
    - Write a log record < **checkpoint** L> onto stable storage where L is a list of all transactions active at the time of checkpoint.

- All updates are stopped while doing checkpointing
- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
    - Scan backwards from end of log to find the most recent <**checkpoint** L> record
    - Only transactions that are in L or started after the checkpoint need to be redone or undone
    - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
    - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in L.
    - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



Database System Concepts - 6th Edition 16.22 ©Silberschatz, Korth and Sudarshan

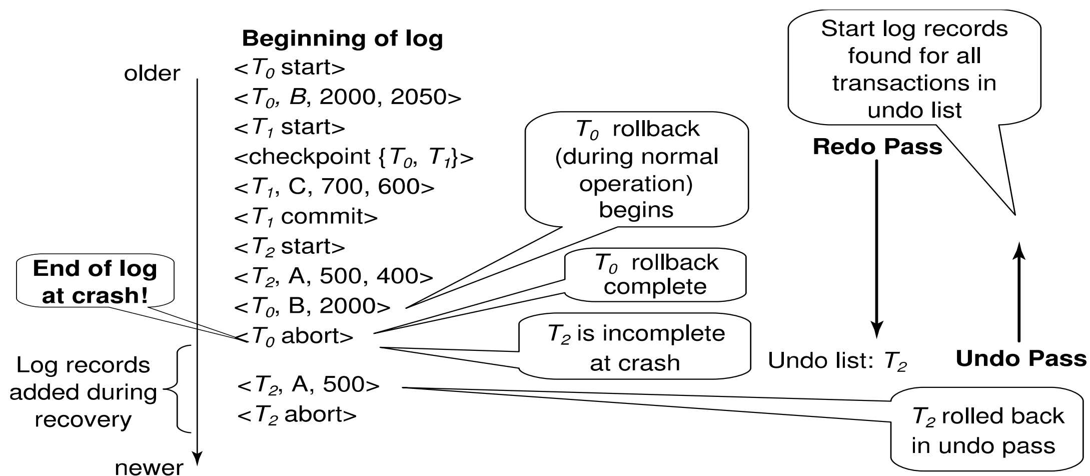
Recovery Algorithm

- **Logging** (during normal operation):
    - <T_i **start**> at transaction start
    - <T_i, X_j, V_1, V_2> for each update, and
    - <T_i **commit**> at transaction end
- **Transaction rollback (during normal operation)**
    - Let T_i be the transaction to be rolled back
    - Scan log backwards from the end, and for each log record of T_i of the form <T_i, X_j, V_1, V_2>
        - perform the undo by writing V_1 to X_j,
        - write a log record <T_i , X_j, V_1>
            - such log records are called **compensation log records**
    - Once the record <T_i **start**> is found stop the scan and write the log record <T_i **abort**>
- **Recovery from failure**: Two phases
    - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
    - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
    - Find last <**checkpoint** L> record, and set undo-list to L.
    - Scan forward from above <**checkpoint** L> record
        - Whenever a record <T_i, X_j, V_1, V_2> or <T_i, X_j, V_2> is found, redo it by writing V_2 to X_j
        - Whenever a log record <T_i **start**> is found, add T_i to undo-list
        - Whenever a log record <T_i **commit**> *or* <T_i **abort**> is found, remove T_i from undo-list
- **Undo phase:**

- Scan log backwards from end
  - Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
    - perform undo by writing V_1 to X_j.
    - write a log record <T_i , X_j, V_1>
  - Whenever a log record <T_i **start**> is found where T_i is in undo-list,
    - Write a log record <T_i **abort**>
    - Remove T_i from undo-list
  - Stop when undo-list is empty
    - i.e. <T_i **start**> has been found for every transaction in undo-list

- After undo phase completes, normal transaction processing can commence

Example of Recovery



Database System Concepts - 6th Edition 16.27 ©Silberschatz, Korth and Sudarshan

Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.
- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
      - Strictly speaking WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks

- When a new block is needed, if buffer is full an existing block needs to be removed from buffer
- If the block chosen for removal has been updated, it must be output to disk

- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**:i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits
- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  - First acquire an exclusive latch on the block
    - Ensures no update can be in progress on the block
  - Then perform a **log flush**
  - Then output the block to disk
  - Finally release the latch on the block

Buffer Management

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    - Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use

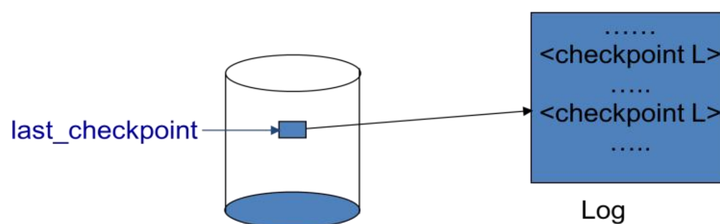
    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Fuzzy Checkpointing

- To avoid long interruption of normal processing during checkpointing, allow updates to happen during checkpointing
- **Fuzzy checkpointing** is done as follows:
  1. Temporarily stop all updates by transactions

2. Write a <**checkpoint** *L*> log record and force log to stable storage
3. Note list *M* of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list *M*
   - blocks should not be updated while being output
   - Follow WAL: all log records pertaining to a block must be output before the block is output
6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk

- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely



Failure with Loss of Nonvolatile Storage

- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure
  - restore database from most recent dump.
  - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**
  - Similar to fuzzy checkpointing

Recovery with Early Lock Release and Logical Undo Operations

Recovery with Early Lock Release

- Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  - Supports "logical undo"
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing

Logical Undo Logging

- Operations like B^+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+-tree.

- Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - Operations are called **logical operations**
  - Other examples:
    - delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

Physical Redo

- Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
  - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
  - Physical redo logging does not conflict with early lock release

Operation Logging

- Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.
- If crash/rollback occurs before operation completes:
  - the **operation-end** log record is not found, and
  - the physical undo information is used to undo operation.
- If crash/rollback occurs after the operation completes:
  - the **operation-end** log record is found, and in this case
  - logical undo is performed using U; the physical undo information for the operation is ignored.
- Redo of operation (after crash) still uses physical redo information.

Transaction Rollback with Logical Undo

Rollback of transaction T_i is done as follows:

Scan the log backwards

1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a al <T_i, X, V_1>.
2. If a <T_i, O_j, **operation-end**, U> record is found
   - Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record <T_i, O_j, **operation-abort**>.
   - Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found

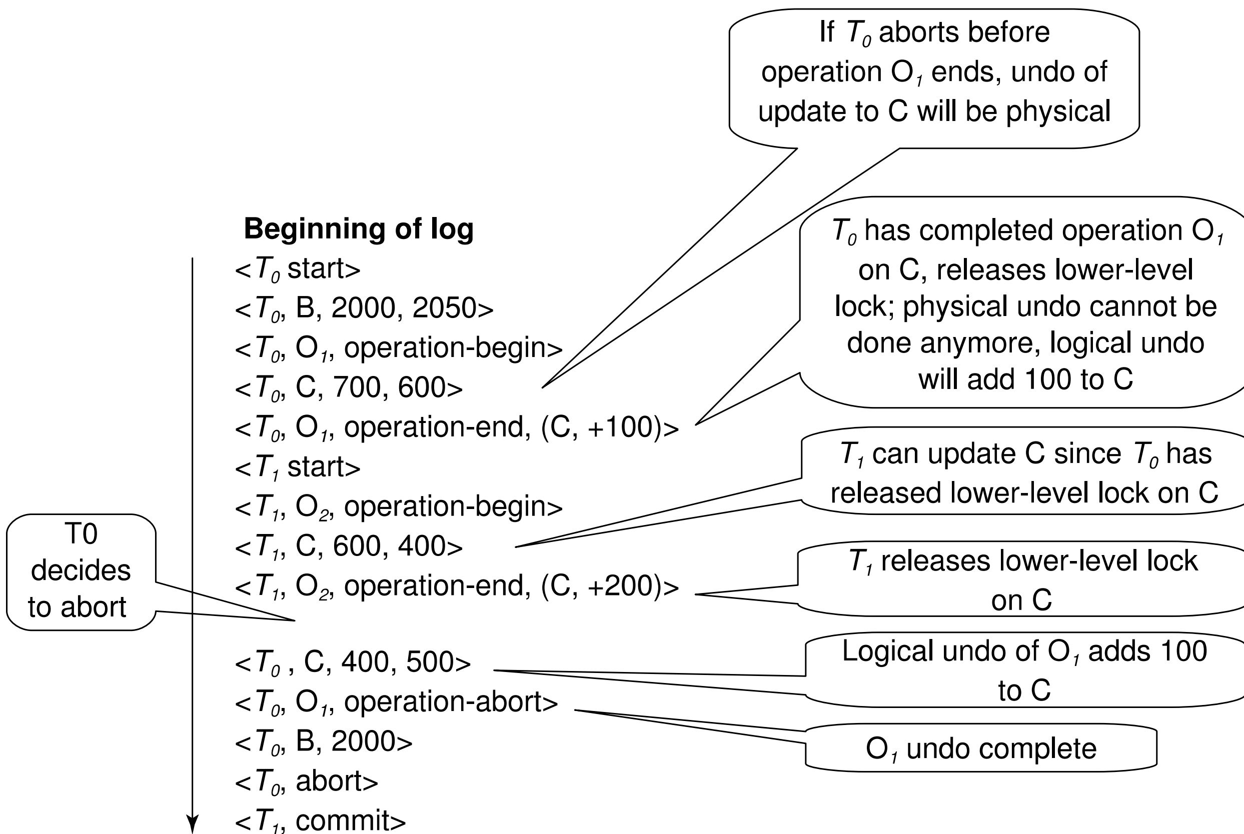
Transaction rollback, scanning the log backwards (cont.):

3. If a redo-only record is found ignore it
4. If a <T_i, O_j, **operation-abort**> record is found:
   H skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
5. Stop the scan when the record <T_i, start> is found
6. Add a <T_i, **abort**> record to the log

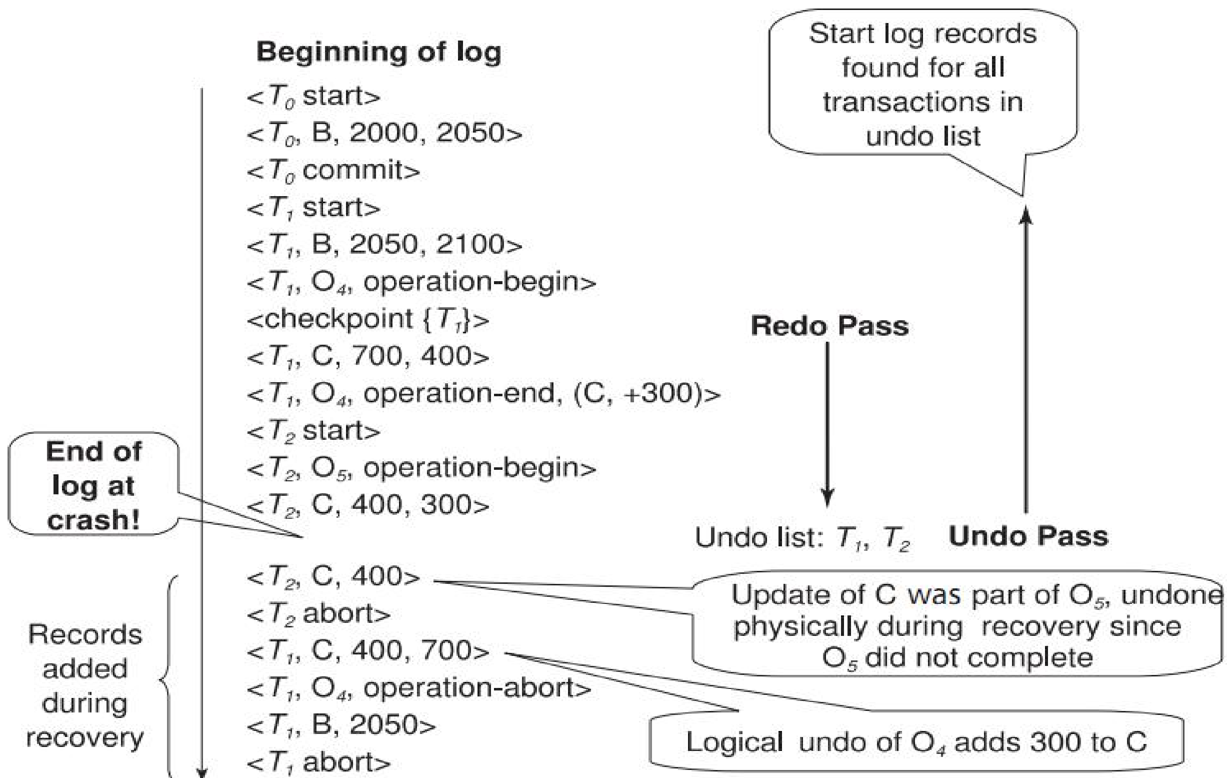
Some points to note:

a. Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.

b. Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.



Failure Recovery with Logical Undo



Recovery Algorithm with Logical Undo

Basically same as earlier algorithm, except for changes described earlier for transaction rollback

1. (**Redo phase**): Scan log forward from last < **checkpoint** *L*> record till end of log
    1. **Repeat history** by physically redoing all updates of all transactions,
    2. Create an undo-list during the scan as follows
        - *undo-list* is set to *L* initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

    This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

    Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.
2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*.
    - Log records of transactions being rolled back are processed as described earlier, as they are found
        - Single shared scan for all transactions being undone
    - When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
    - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records).
Recovery is now complete.