Week 1 Lecture 1

Class	BSCCS2001
Created	@August 19, 2021 1:46 PM
Materials	https://drive.google.com/drive/folders/19FhdYYKeH3ZshWhoZIJlP_MC1nVnUUmU?usp=sharing
Module #	1
Type	Lecture
Week #	1

Database Management Systems (DBMS)

DBMS: A database management system (or DBMS) is essentially nothing more than a computerized data-keeping system. (via IBM)

DBMS contains info about a particular enterprise

- Collection of interrelated data
- Set of programs to access the data
- An environment that is both *convenient* and *efficient* to use

Database Applications:

- Banking: transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases
- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- HR: employee records, salaries, tax deductions

Databases can be very large

Databases touch various aspects of our lives

University Database Example

Application program examples

- Add new students, instructors and courses
- Register students for courses and generate class rosters
- Assign grades to students, compute Grade Point Average (GPA) and generate transcripts

In early days, database applications were built directly on top of file systems

Drawbacks of using file systems to store data

- Data redundancy and inconsistency
  - Multiple file formats, duplication 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 (eg: account balance > 0) become "buried" in program code rather than being stated explicity
  - Hard to add new constraints or change existing ones
- Atomicity of updates
  - Failures may leave databases in an inconsistent state with partial updates carries 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
    - **Example:** 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 the above problems

Course pre-requisites:

Set Theory

- Definition of a set
  - Intensional definition
  - Extensional definition
  - Set-builder notation
- Membership, Subset, Superset, Power set, Universal set
- Operations on sets:
  - Unions, Intersections, Complement, Difference, Cartesian product
- De-Morgan's Law

Relations and Functions

- Definition of Relations
- Ordered pairs and Binary relations
  - Domain and Range

    - Image, Pre-image, Inverse
    - Properties: Reflexive, Symmetric, Anti-symmetric, Transitive, Total
- Definition of functions
- Properties of functions: Injective, Surjective, Bijective
- Composition of functions
- Inverse of functions

Propositional Logic

- Truth values and Truth tables
- Operators: conjunction (and), disjunction (or), negation (not), implication, equivalence
- Closure under Operations

Predicate Logic

- Predicates
- Quantification
    - Existential
    - Universal

Python

Algorithms and Programming in C

- Sorting
    - Merge sort
    - Quick sort
- Search
    - Linear search
    - Binary search
    - Interpolation search

Data Structures

- Arrays
- List
- Binary Search Tree
    - Balanced Tree
- B - Tree
- Hash table/map

Object-Oriented analysis and design

- **Refresher material**
    - Discrete Mathematics by Brilliant: https://brilliant.org/wiki/discrete-mathematics
    - Python
        - IITM online book: https://pypod.github.io
        - Cheatsheet: https://www.pythoncheatsheet.org
        - DataCamp Cheatsheet: https://www.datacamp.com/community/tutorials/python-data-science-cheat-sheet-basics

- C Language: https://www.youtube.com/watch?v=zYierUhIFNQ&list=PLhQjrBD2T382_R182iC2gNZI9HzWFMC_8&index=2 (part of CS50 2020 Lectures)

Week 1 Lecture 2

Class	BSCCS2001
Created	@August 19, 2021 3:48 PM
Materials	
Module #	2
Type	Lecture
Week #	1

Why DBMS?

Data Management

- Storage
- Retrieval
- Transaction
- Audit
- Archival

For

- Individuals
- Small / Big Enterprises
- Global

There has been 2 major approaches in this practice:

1. **Physical:**

   Physical Data or Records Management, more formally known as ***Book Keeping,*** has been using physical ledgers and journals for centuries

   The most significant development happened when Henry Brown patented a "receptacle for storing and preserving papers" on November 2, 1886

   Herman Hollerith adapted the punch cards used for weaving looms to act as the memory for a mechanical tabulating machine in 1890

2. **Electronic:**

   Electronic Data or Records management moves with the advances in technology, especially of memory, storage, computing and networking

   - 1950s: Computer programming started
   - 1960s: Data Management with punch cards / tapes and magnetic tapes
   - 1970s:
     - COBOL and CODASYL approach was introduced in 1971
     - On October 14, 1979, Apple II platform shipped VisiCalc, marking the birth of spreadsheets
     - Magnetic disks became prevelant
     - 1980s: RDBMS changed the face of data management
     - 1990s: With internet, data management started becoming global
     - 2000s: e-Commerce boomed, NoSQL was introduced for unstructured data management
     - 2010s: Data Science started riding high

Electronic Data Management Params

Electronic Data or Records management depends on various params including ...

- Durability
- Scalability
- Security
- Retrieval
- Ease of Use
- Consistency
- Efficiency
- Cost

Book Keeping

A book register was maintained on which the shop owner wrote the amount received from customers, the amount due for any customer, inventory details and so on ...

Problems with such an approach of book keeping:

- **Durability:** Physical damage to these registers is a possibility due to rodents, humidity, wear and tear
- **Scalability:** Very difficult to maintain over the years, some shops have numerous registers spanning over the years
- **Security:** Susceptible to tampering by the outsiders
- **Retrieval:** Time consuming process to search for previous entry
- **Consistency:** Prone to human errors

Not only small shops but large orgs also used to maintain their transactions in book registers

Spreadsheet files - A better solution

Mostly useful for single user or small enterprise applications

Spreadsheet software like Google Sheets: Due to disadvantages of maintaining ledger registers, organizations dealing with huge amount of data shifted to using spreadsheets for maintaining records in files

- **Durability:** These are computer applications and hence data is less prone to physical damage
- **Scalability:** Easier to search, insert and modify records as compared to book ledgers
- **Security:** Can be password protected
- **Easy to Use:** Computer applications are used to search and manipulate records in the spreadsheets leading to reduction in manpower needed to perform routing computations

- **Consistency:** Not guaranteed but spreadsheets are less prone to mistakes registers

Why leave filesystems?

Lack of efficiency in meeting growing needs

- With rapid scale up of data, there has been considerable increase in the time required to perform most operations
- A typical spreadsheet file may have an upper limit on the number of rows
- Ensuring consistency of data is a big challenge
- No means to check violations of constraints in the face of concurrent processing
- Unable to give different permissions to different people in a centralized manner
- A system crash could be catastrophic

The above mentioned limitations of filesystems paved the way for a comprehensive platform dedicated to management of data - the **Database Management System**

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 model in widespread use
  - **Ted Codd** defines the relational data model
    - Would win the ACM Turin Award for his work
    - IBM Research begins in System R prototype
    - UC Berkeley begins Ingres prototype
  - High-performance (for the era) transaction processing
- 1980s
  - Research relational prototypes evolve into commercial systems - SQL becomes industrial standard
  - Parallel and distributed database systems
  - Object oriented database systems
- 1990s
  - Large decision support and data mining applications
  - Large multi-terabyte data warehouses
  - Emergence of Web commerce
- Early 2000s
  - XML and XQuery standards
  - Automated database administration
- Later 2000s
  - Giant data storage systems - Google BigTable, Yahoo PNuts, Amazon, ...

Week 1 Lecture 3

Class	BSCCS2001
Created	@August 19, 2021 4:47 PM
Materials	
Module #	3
Type	Lecture
Week #	1

Why DBMS? (part 2)

Case study of a Bank Transaction

Consider a simple banking system where a person can open a bank account, transfer funds to an existing account and check the history of all her transactions till date

The application performs the following checks

- If the account balance is not enough, it will now allow the fund transfer
- If the account numbers are not correct, it will flash a message and terminate the transaction
- If a transaction is successful, it prints a confirmation message

We will use this banking transaction system to compare various features of a file-based (.csv file) implementation viz-a-viz a DBMS-based implementation

- Account details are stored in
  - Accounts.csv for file-based implementation
  - Accounts table for DBMS implementation
- The transaction details are stored in
  - Ledger.csv for file-based implementation
  - Ledger table for DBMS implementation

Source: https://github.com/bhaskariitm/transition-from-files-to-db

Initiating a transaction

Python

```
def begin_Transaction(credit_account, debit_account, amount):
    temp = []
    success = 0

    # Open file handles to retrieve and store transaction data
    f_obj_Account1 = open('Accounts.csv', 'r')
    f_reader1 = csv.DictReader(f_obj_Account1)
    f_obj_Account2 = open('Accounts.csv', 'r')
    f_reader2 = csv.DictReader(f_obj_Account2)
    f_obj_Ledger = open('Ledger.csv', 'a+')
    f_writer = csv.DictWriter(f_obj_Ledger, fieldnames=col_name_Ledger)
```

SQL

```
-- Handled implicitly by the DBMS
```

Transaction

Python

```
try:
  for sRec in f_reader1:
    # CONDITION CHECK FOR ENOUGH BALANCE
    if sRec['AcctNo'] == debitAcc and int(sRec['Balance']) > int(amt):
      for rRec in f_reader2:
        if rRec['AcctNo'] == creditAcc:
          sRec['Balance'] = str(int(sRec['Balance']) - int(amt))    # DEBIT
          temp.append(sRec)
          # CRITICAL POINT
          f_writer.writerow({
            'Acct1':sRec['AcctNo'],
            'Acct2':rRec['AcctNo'],
            'Amount':amt,
            'D/C':'D'
          })
          rRec['Balance'] = str(int(rRec['Balance']) + int(amt))    # CREDIT
          temp.append(rRec)
          f_writer.writerow({'Account1': r_record['Account_no'], 'Account2': s_record['Account_no'], 'Amount': amount,'D/C': 'C'})
          success = success + 1
          break
        f_obj_Account1.seek(0)
        next(f_obj_Account1)
        for record in f_reader1:
            if record['Account_no'] != temp[0]['Account_no'] and record['Account_no'] != temp[1]['Account_no']:
                temp.append(record)
    except:
        print('\nWrong input entered !!!')
```

SQL

```
do $$
begin
amt = 5000
sendVal = '1800090';
recVal = '1800100';
select balance from accounts
into sbalance
where account_no = sendVal;
if sbalance < amt then
raise notice "Insufficient balance";
else
update accounts
  set balance = balance - amt
  where account_no = sendVal;
insert into ledger(sendAc, recAc, amnt, ttype)
values(sendVal, recVal, amt, 'D')
update accounts
  set balance = balance + amt
  where account_no = recVal;
insert into ledger(sendAc, recAc, amnt, ttype)
values(sendVal, recVal, amt, 'C')
commit;
raise notice "Successful";
end if;
end; $$
```

Closing a transaction

Python

```
f_obj_Account1.close()
f_obj_Account2.close()
f_obj_Ledger.close()
if success == 1:
    f_obj_Account = open('Accounts.csv', 'w+', newline='')
    f_writer = csv.DictWriter(f_obj_Account, fieldnames=col_name_Account)
    f_writer.writeheader()
    for data in temp:
        f_writer.writerow(data)
    f_obj_Account.close()
    print("\nTransaction is successfull !!")
else:
    print('\nTransaction failed : Confirm Account details')
```

SQL

```
-- Handled implicitly by the DBMS
```

Comparison

Parameter	File handling via Python	DBMS
Scalability with respect to amount of data	Very difficult to handle insert, update and querying of records	In-built features to provide high scalability for a large number of records
Scalability with respect to changes in structure	Extremely difficult to change the structure of records as in the case of adding or removing attributes	Adding or removing attributes can be done seamlessly using simple SQL queries
Time of execution	in seconds	in milliseconds
Persistence	Data processed using temporary data structures have to be manually updated to the file	Data persistence is ensured via automatic, system induced mechanisms
Robustness	Ensuring robustness of data has to be done manually	Backup, recovery and restore need minimum manual intervention
Security	Difficult to implement in Python (Security at OS level)	User-specific access at database level
Programmer's productivity	Most file access operations involve extensive coding to ensure persistence, robustness and security of data	Standard and simple built-in queries reduce the effort involved in coding thereby increasing a programmer's throughput
Arithmetic operations	Easy to do arithmetic computations	Limited set of arithmetic operations are available
Costs	Low costs for hardware, software and human resources	High costs of hardware, software and human resources

Parameterized Comparison

Scalability

File Handling in Python

- **Number of records:** As the # of records increases, the efficiency of flat files reduces:
  - the time spent in searching for the right records
  - the limitations of the OS in handling huge files
- **Structural Change:** To add an attribute, initializing the new attribute of each record with a default value has to be done by program. It is very difficult to detect and maintain relationships between entities if and when an attribute has to be removed

DBMS

- **Number of records:** Databases are built to efficiently scale up when the # of records increase drastically.
  - In-built mechanisms, like indexing, for quick access of right data

- **Structural Changes:** During adding an attribute, a default value can be defined that holds for all existing records - the new attribute gets initialized with default value. During deletion, constraints are used either not to allow the removal on ensure its safe removal

Time and Efficiency

- If the number of records is very small, the overhead in installing and configuring a database will be much more than the time advantage obtained from executing the queries
- However, in the number of records is really large, then the time required in the initialization process of a database will be negligible as compared to that of using SQL queries

File Handling in Python

- The effort needed to implement a file handler is quite less in Python
- In order to process a 1GB file, a program in Python would typically take a few seconds

DBMS

- The effort to install and configure a DB in a DB server in expensive and time consuming
- In order to process a 1GB file, an SQL query would typically take a few milliseconds

Programmer's Productivity

File Handling in Python

- **Building a file handler:** Since the constraints within and across entities have to be enforced manually, the effort involved in building a file handling application is huge
- **Maintenance:** To maintain the consistency of data, one must regularly check for sanity of data and the relationships between entities during inserts, updates and deletes
- **Handling huge data:** As the data grows beyond the capacity of the file handler, more efforts are needed

DBMS

- **Configuring the database:** The installation and configuration of a database is a specialized job of a DBA. A programmer, on the other hand, is saved the trouble
- **Maintenance:** DBMS has built-in mechanisms to ensure consistency and sanity of data being inserted, updated or deleted. The programmer does not need to do such checks
- **Handling huge data:** DBMS can handle even terabytes of data - Programmer does not have to worry

Arithmetic Operations

File Handling in Python

- Extensive support for arithmetic and logical operations on data using Python. These include complex numerical calculations and recursive computations

DBMS

- SQL provides limited support for arithmetic and logical operations. Any complex computation has to be done outside of SQL

Costs and Complexity

File Handling in Python

- File systems are cheaper to install and use. No specialized hardware, software or personnel are required to maintain filesystems

DBMS

- Large databases are served by dedicated database servers which need large storage and processing power
- DBMSs are expensive software that have to be installed and regularly updated
- Databases are inherently complex and need specialized people to work on it - like DBA (Database System Administrator)
- The above factors lead to huge costs in implementing and maintaining database management systems

Week 1 Lecture 4

Class	BSCCS2001
Created	@August 19, 2021 5:55 PM
Materials	
Module #	4
Type	Lecture
Week #	1

Introduction to DBMS

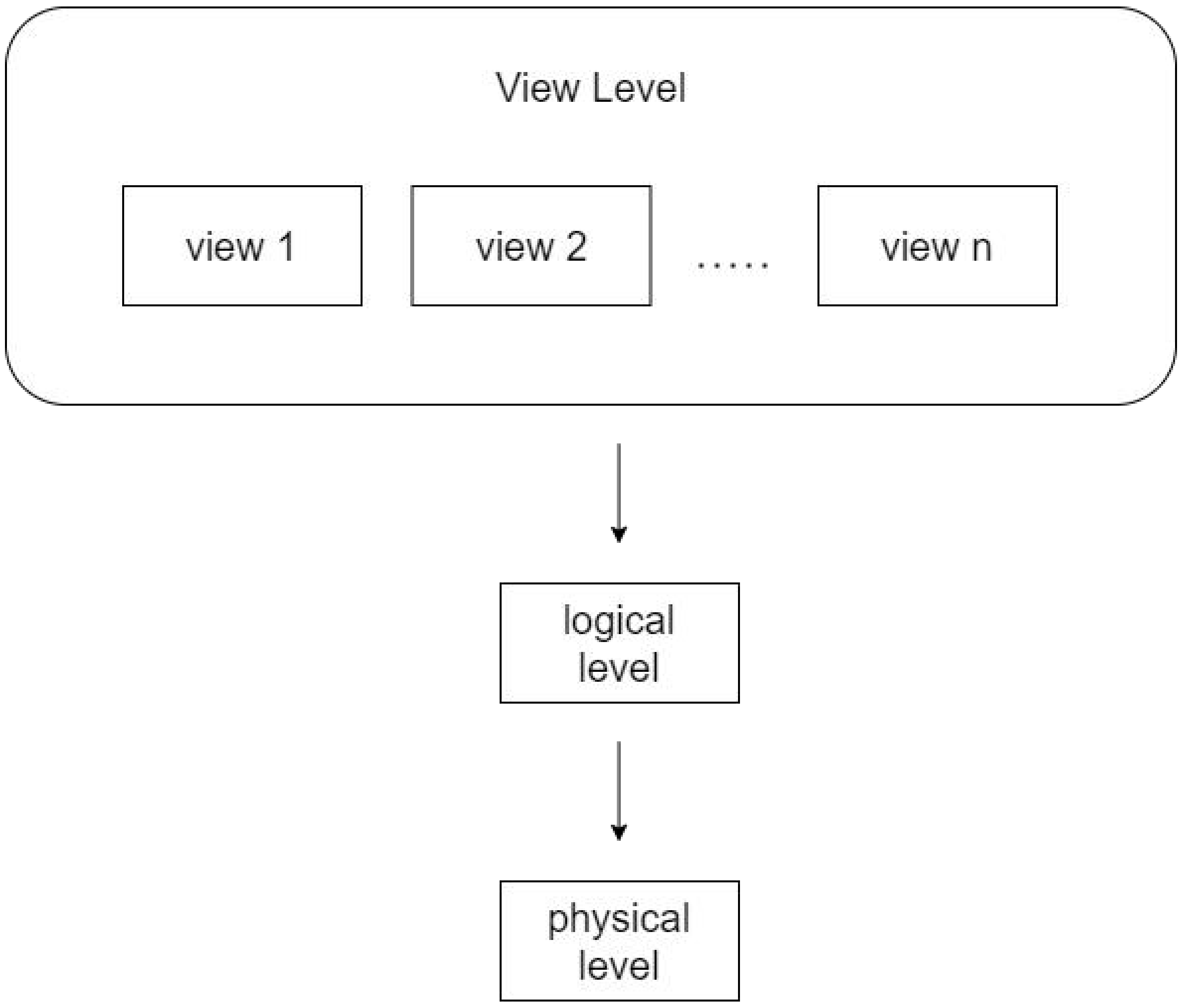
Levels of Abstraction

- **Physical Level:** describes how a record (eg: instructor) is stored
- **Logical Level:** describes data stored in a database and the relationships among the data fields

```
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 employee's salary) for security purposes

An architecture for a database system



Schema and Instances

TLDR: *Schema* is the way in which data is organized and *Instance* is the actual value of the data

- **Schema**
  - **Logical Schema** - the overall logical structure of the database
    - Analogous to type information of a variable in a program (eg: int x = 5)
    - *Example:* The database consists of information about a set of customers and accounts in a bank and the relationship between them

  Customer Schema

Name	Customer ID	Account #	Aadhaar ID	Mobile #
Untitled				

  Account Schema

Account #	Account Type	Interest Rate	Min. Bal.	Balance
Untitled				

  - **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

Customer Instance

Name	Customer ID	Account #	Aadhaar ID	Mobile #
Pavan Lakha	6728	917322	182719289372	9830100291
Lata Kala	8912	827183	918291204829	7189203928
Nand Prabhu	6617	372912	127837291021	8892021892

Account Instance

Account #	Account Type	Interest Rate	Min. Bal.	Balance

Account #	Account Type	Interest Rate	Min. Bal.	Balance
917322	Savings	4.0%	5000	7812
372912	Current	0.0%	0	291820
827183	Term Deposit	6.75%	10000	100000

- **Physical Data Independence** - the ability to modify the physical schema without changing the logical schema
    - Analogous to independence of *Interface* and *Implementation* in object-oriented systems
    - Applications depend on the logical schema
    - In general, the interfaces between various levels and components should be well defined so that changes in some parts do not seriously influence others.

Data Models

- A collection of tools that describe the following ...
    - Data
    - Data relationships
    - Data semantics
    - Data constraints
- **Relational model** (our focus in this course)
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Other older models
    - Network model
    - Hierarchical model
- Recent models for Semi-structured or Unstructured data
    - Converted to easily manageable formats
    - Content Addressable Storage (CAS) with metadata descriptors
    - XML format
    - RDBMS which support BLOBs

Relational Model

- All the data is stored in various tables
    - Tables are also called Relations
    - Columns are called attributes
    - They have particular names which tells us the schema
    - Rows are records that are the values

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 (that is, data about the data)
    - Database schema

    - Integrity constraints
        - Primary key (ID uniquely identifies instructors)
    - Authorization
        - Who can access what

Data Manipulation Language (DML)

Language for accessing and manipulating the data organized by the appropriate data model

- **DML:** also know as *Query Language*
- Two classes of languages
    - **Pure** - used for proving properties about computational power and for optimization
        - *Relational Algebra* (our focus in this course)
        - Tuple relational calculus
        - Domain relational calculus
    - **Commercial** - used in commercial systems
        - SQL is the most widely used commercial language

Structured Query Language (SQL)

- Most widely used commercial language
- ***SQL is NOT a Turing Machine equivalent language***. Read more here
    - Cannot be used to solve all problems that a C program, for example, can solve
- To be able to compute complex 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 Programming Interfaces or APIs (eg: ODBC / JDBC) which allow SQL queries to be sent to the databases

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 schema
    - Business decision
        - What attributes should we record in the databases?
    - 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

Week 1 Lecture 5

Class	BSCCS2001
Created	@August 20, 2021 11:13 AM
Materials	
Module #	5
Type	Lecture
Week #	1

Introduction to DBMS (part 2)

Database Design

Design Approaches

- Need to come up with a methodology to ensure that each relation in the database is ***good***
- Two ways of doing so:
    - Entity Relationship Model *(primarily tries to capture the business requirements)*
        - Models an enterprise as a collection of entities and relationships
        - Represented diagrammatically by an entity-relationship diagram
    - Normalization Theory *(this is the Computer Science perspective)*
        - Formalize what designs are bad and test for them

Object-Relational Data Models

- Relational model: flat, atomic values
- Object Relational Data Models
    - Extend the relational data model by including object orientation and constructs to deal with added data types
    - Allow attributes of tuples to have complex types, including non-atomic values such as nested relations
    - Preserve relational foundations, in particular the declarative access to data, while extending modeling power
    - Provide upward compatibility with existing relational language

XML: eXtensible Markup Language

- Defined by the ***WWW Consortium (W3C)***
- What XML primarily says; *XML is a description of name-value pair*
  - It talks about a tag, so you can put a value on that
- Originally intended as a document markup language not a database language
- The ability to specify new tags and to create tag structures made XML a great way to exchange data, not just documents
- XML has become the basis for all new generation data interchange formats
- A wide variety of tools are available for parsing, browsing and querying XML documents

Database Engine

3 major components are:

- Storage Manager
- Query processing
- Transaction Manager

Storage Management

Storage Manager is 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 for the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data
- Issues:
  - Storage access
  - File organization
  - Indexing and hashing

Query Processing

- Parsing and Translation
- Optimization
- Evaluation

How a query is processed?

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
  - Depends critically on statistical information about relations which the database must maintain
  - Need to estimate statistics for intermediate results to compute cost of complex expressions

Transaction Management

- What is the system fails?
- What if more than one user is concurrently updating the same file?
- A **transaction** is a collection of operations that perform single logical function in a database application
- **Transaction-Management component** ensure that the database remains in a consistent (correct) state despite system failures (eg: power failures and operating system crashes) and transaction failures

- **Concurrency-control manager** controls the interaction among the concurrent transactions to ensure consistency of the database

Database Architecture

The architecture of a database system is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-Server
- Parallel (multi-processor)
- Distributed
- Cloud

Week 2 Lecture 1

Class	BSCCS2001
Created	@August 21, 2021 12:32 PM
Materials	
Module #	6
Type	Lecture
Week #	2

Introduction to Relational Model

Attribute Types

- Consider

  Student = Roll #, First Name, Last Name, DoB, Passport #, Aadhaar #, Department

  relation
- The set of allowed values for each attribute is called the **domain** of the attribute
  - **Roll #** - Alphanumeric string
  - **First Name, Last Name** - Alpha string
  - **DoB** - Date
  - **Passport #** - String (Letter followed by 7 digits) - nullable (Optional)
  - **Aadhaar #** - 12-digit number
  - **Department** - Alpha string
- Attribute values are (normally) required to be **atomic**; this is, indivisible
- The special value **null** is a member of every domain. Indicates that the value is *unknown*
- the *null* value may cause complications in the definition of many operations

Roll #	First Name	Last Name	DoB	Passport	Aadhaar	Dept.
15CS10026	Lalit	Dubey	27-Mar-1997	L4032464	172861749239	Computer

Roll #	First Name	Last Name	DoB	Passport	Aadhaar	Dept.
16EE30029	Jatin	Chopra	17-Nov-1996	null	391718363816	Electrical

Relational Schema and Instance

- $A_1, A_2, ..., A_n$ are the attributes
- R = $(A_1, A_2, ..., A_n)$ is a relation schema

  Example: `instructor = (ID, name, dept_name, salary)`
- Formally, given as $D_1, D_2, ..., D_n$ a relation r is a subset of

  $D_1 \times D_2 \times ... D_n$

  Thus, a relation is a set of n-tuples $(a_1, a_2, ..., a_n)$ where each $a_i \in D_i$
- The current values (**relation instance**) of a relation are specified by a table
- An element t or r is a tuple, represented by a row in a table
- Example

  instructor ≡ (String(5) × String × String × Number+), where ID ∈ String(5), name ∈ String, dept_name ∈ String and salary ∈ Number+

Keys

- Let $K \subseteq R$, where R is the set of attributes in the relation
- K is a **superkey** of R if values of 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**
- A **surrogate key** (or synthetic key) in a database is a unique identifier for either an entity in the modeled world or an object in the database
  - The surrogate key is not derived from application data, unlike a natural (or business) key which is derived from application data

Keys: Examples

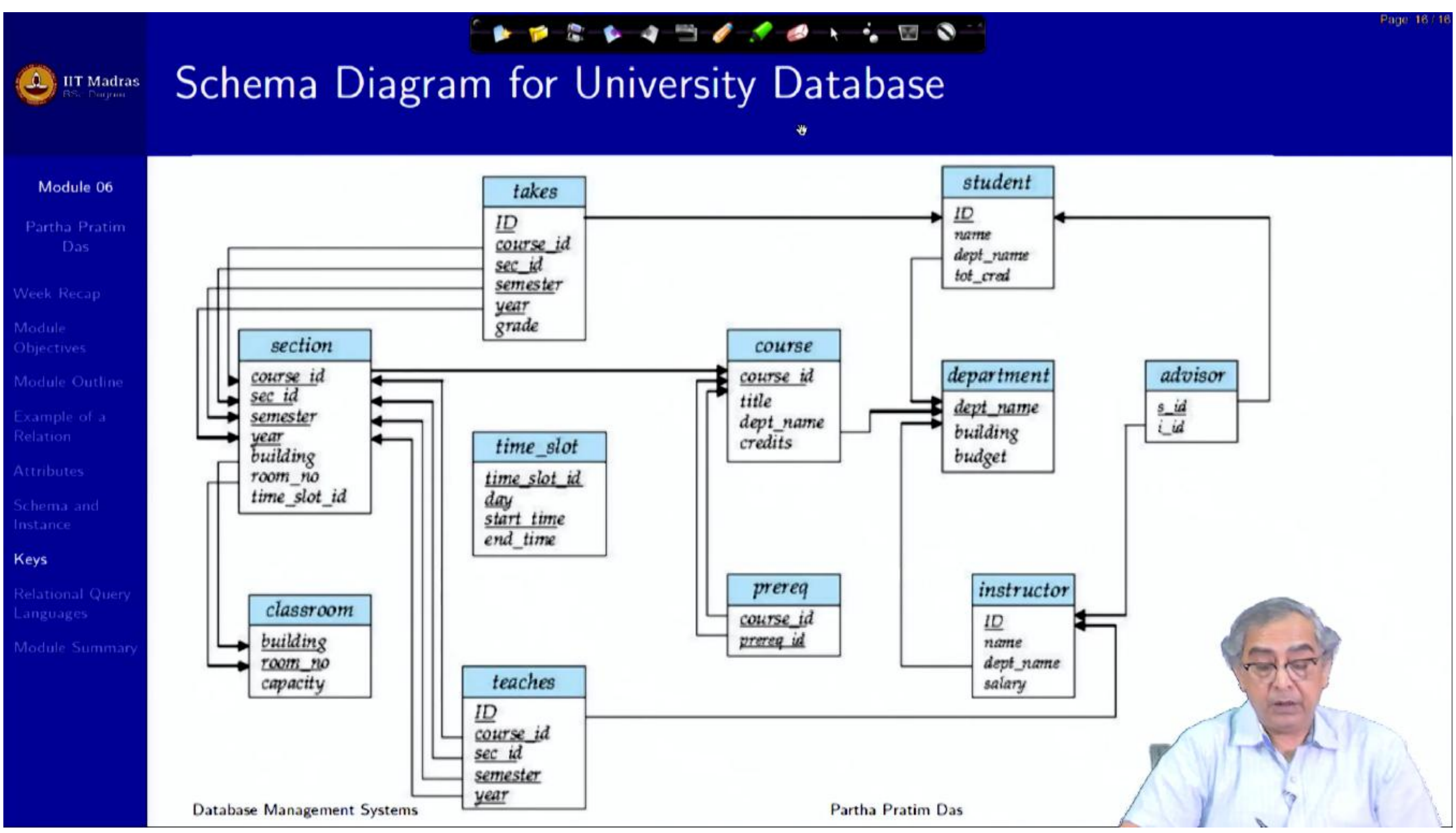
- Students = Roll #, First Name, Last Name, DoB, Passport #, Aadhaar #, Department
- **Super Key**: Roll #, {Roll #, DoB}
- **Candidate Keys**: Roll #, {First Name, Last Name}, Aadhaar #
  - Passport # cannot be a key because it is an optional field and can take null values, but an ID can never be null
- **Primary Key**: Roll #
  - Can Aadhaar # be a key?
  - It may suffice for unique identification, but Roll # may have additional useful information.
  - For example: 14CS92P01
    - Read it as 14-CS-92-P-01
    - 14 - Admission in 2014
    - CS - Department: Computer Science
    - 92 - Category of the Student
    - P - Type of admission: *Project*
    - 01 - Serial Number
- **Secondary / Alternate Key**: {First Name, Last Name}, Aadhaar #
- **Simple Key**: Consists of a single attribute

- **Composite Key**: {First Name, Last Name}
  - Consists of more than one attribute to uniquely identify an entity occurrence
  - One or more of the attributes, which make up the key are not simple keys in their own right

Roll #	First Name	Last Name	DoB	Passport	Aadhaar	Dept
15CS10026	Lalit	Dubey	27-Mar-1997	L4032464	172861749239	Computer
16EE30029	Jatin	Chopra	17-Nov-1996	null	391718363816	Electrical
15EC10016	Smriti	Mongra	23-Dec-1996	G5432849	204592710914	Electronics
16CE10038	Dipti	Dutta	02-Feb-1997	null	571919482918	Civil
15CS30021	Ramdin	Minz	10-Jan-1997	X8811623	492849275924	Computer

- **Foreign key constraint**: Value in one relation must appear in another (in other words, when a particular attribute is a key in a different table)
  - **Referencing** relation
    - Enrolment: Foreign Keys - Roll #, Course #
  - **Referenced** relation
    - Students, Courses
- A **compound key** consists of more than one attribute to uniquely identify an entity occurence
  - Each attribute, which makes up the key, is a simple key in its own right
  - {Roll #, Course #}

Schema Diagram for University Database

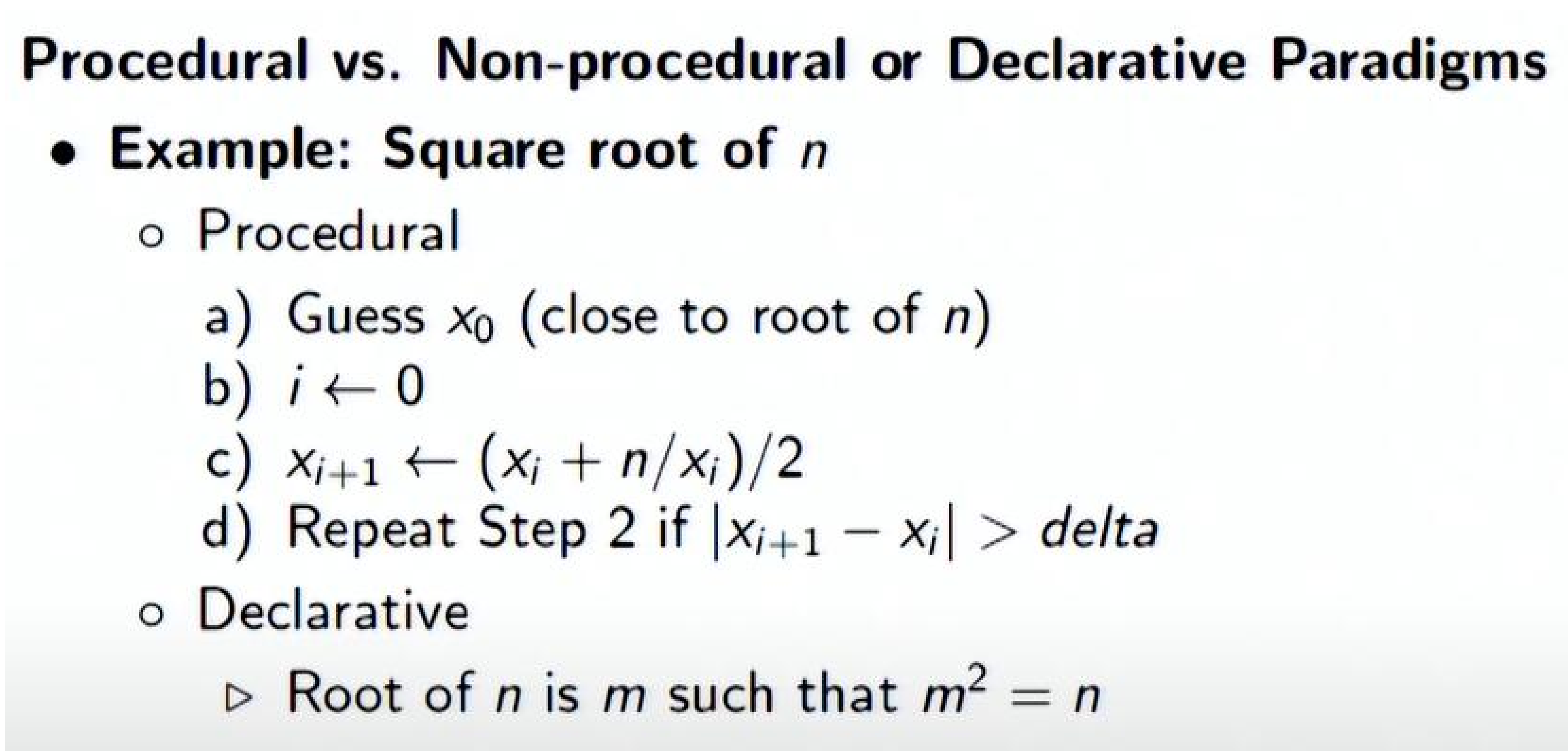


Relational Query Languages

Procedural viz-a-viz Non-procedural or Declarative Paradigms

- Procedural programming requires that the programmer tell the computer what to do
  - That is, how to get the output for the range of required inputs
  - The programmer must know an appropriate algorithm
- Declarative programming requires a more descriptive style
  - The programmer must know what relationships hold between various entities

Relational Query Language: Example



- "Pure" languages:
  - Relational Algebra
  - Tuple relational calculus
  - Domain relational calculus
- The above 3 pure languages are equivalent in computing power
- We will concentrate on relational algebra
  - Not Turing-macine equivalent
    - Not all algorithms can be expressed in Relational Algebra
  - Consists of 6 basic operations

Week 2 Lecture 2

Class	BSCCS2001
Created	@August 22, 2021 6:57 PM
Materials	https://www.caam.rice.edu/~heinken/latex/symbols.pdf
Module #	7
Type	Lecture
Week #	2

Introduction to Relational Model (part 2)

Relational Operators

Basic properties of relations

- A relation is a set. Hence,
- Ordering of rows / tuples is inconsequential
- All rows / tuples must be distinct

Select operation - selection of rows (tuples)

- Relation r on the following table

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- The select operation is defined as

$$\sigma_{A=B \wedge D>5}(r)$$

- And it returns the following table as a result

A	B	C	D
α	α	1	7
β	β	23	10

Project operation - selection of columns (Attributes)

- Relation r

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- The projection operation is defined as

$$\pi_{A,C}(r)$$

- And it returns the following table as a result

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

Partha Pratim Das

Union of two relations

- Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- The union of two relation is defined as

$$r \cup s$$

- And it returns the following result

A	B
α	1
α	2
β	1
β	3

Set difference of two relations

- Relation r, s

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- The set difference of two relations is defined as

$$r - s$$

- And it returns the following result

A	B
α	1
β	1

$$\textbf{Note: } r \cap s = r - (r - s)$$

Joining two relations - Cartesian-product

- Relation r, s

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- The cartesian product is defined as

$$r \times s$$

- And it returns the following result

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Cartesian-product - Naming issue

A	B
α	1
β	2

r

B	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

A	r.B	s.B	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Renaming a Table

- Allows us to refer to a relation, say E, by more than one name

$$\rho_X(E)$$

  returns the expression E under the name X

- Relations r

A	B
α	1
β	2

r

- Self product

$$r \times \rho_s(r)$$

r.A	r.B	s.A	s.B
α	1	α	1
α	1	β	2
β	2	α	1
β	2	β	2

Composition of Operations

- Can build expressions using multiple operations
- Example:

$$\sigma_{A=C}(r \times s)$$

- $r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$$\sigma_{A=C}(r \times s)$$

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b

Joining two relations - Natural Join

- Let r and s be relations on schemas R and S respectively. Then, the "natural join" of relations R and S is a relation on schema $R \cup S$
  - Consider each pair of tuples t_r from r and t_s from s
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
    - t has the same value as t_r on r
    - t has the same value as t_s on s

Natural join example

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ε

s

- Natural join

$$r \bowtie s$$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

$$\pi_{A,r.B,C,r.D,E}(\sigma_{r.B=s.B \wedge r.D=s.D}(r \times s))$$

Aggregation Operators

- Can we compute:
  - SUM
  - AVG
  - MAX
  - MIN

Notes about Relational Languages

- Each query input is a table (or a set of tables)
- Each query output is a table
- All data in the output table appears in one of the input tables
- Relational Algebra is not Turing complete

Week 2 Lecture 3

Class	BSCCS2001
Created	@August 22, 2021 8:38 PM
Materials	
Module #	8
Type	Lecture
Week #	2

Introduction to Structured Query Language (SQL)

History of SQL

- IBM developed *Structured English Query Language* (**SEQUEL**) as a part of System R project.
- Renamed Structured Query Language (SQL: *still pronounced as* **SEQUEL**)

ANSI and ISO standard SQL:

Name	Description
SQL - 86	First formalized by **ANSI**
SQL - 89	+ **Integrity Constraints**
SQL - 92	Major revision (**ISO/IEC 9075 standard**), **De-facto Industry Standard**
SQL : 1999	+ **Regular Expression Matching, Recursive Queries, Triggers, Support for Procedural and Control Flow Statements**, Non-scalar types (Arrays) and some OO features (structured types), **Embedding SQL in Java (SQL/OLB)** and **Embedding Java in SQL (SQL/JRT)**
SQL : 2003	+ **XML features (SQL/XML)**, Window functions, Standardized sequences and columns with auto-generated values (identity columns)
SQL : 2006	+ Way of **importing and storing XML data** in a SQL database, **manipulating it** within the database, and **publishing both XML and conventional SQL-data in XML form**
SQL : 2008	Legalizes **ORDER BY** outside Cursor Definitions + **INSTEAD OF** Triggers, **TRUNCATE** statements and **FETCH** clause

Name	Description
SQL : 2011	+ Temporal data (**PERIOD FOR**) Enhancements for Window functions and **FETCH** clause
SQL : 2016	+ Row Pattern Matching, Polymorphic Table Functions and **JSON**
SQL : 2019	+ Multidimensional Arrays (MDarray type and operators)

Compliance

- SQL is the de facto industry standard today for relational or structured data systems
- Commercial system as well as open system may be fully or partially compliant to one or more standards from SQL-92 onward
  - Not all examples here may work on your particular system. Check your system's SQL docs.

Alternatives

- There aren't any alternatives to SQL for speaking to relational databases (i.e. SQL as a protocol)
  - There are alternatives to writing SQL in the applicaions
- These alternatives have been implemented in the form of front-ends for working with relational databases. Some examples of a front-end include (for a section of languages):
  - **SchemeQL** and **CLSQL**
    - Probably the most flexible, thanks to their Lisp heritage
    - They also look a lot more like SQL than other front-ends
  - **LINQ** (in .NET)
  - **ScalaQL** and **ScalaQuery** (in Scala)
  - **SqlStatement**, **ActiveRecord** and many others in Ruby
  - **HaskellDB**
  - ... the list goes on for many other languages

Derivatives

- There are several query languages that are derived from or inspired by SQL.
- Out of these, the most popular and effective is **SPARQL**.
- **SPARQL** (pronounced *sparkle,* a recursive acronym for *SPARQL Protocol and RDF Query Language*) is an RDF query language
  - A semantic query language for databases - able to retrieve and manipulate data stored in ***Resource Description Framework*** (***RDF***) format.
  - It has been standardized by the W3C Consortium as key technology of the semantic web
  - Versions
    - SPARQL 1.0 (Jan. 2008)
    - SPARQL 1.1 (Mar. 2013)
  - Used as the query languages for several NoSQL systems - particularly the Graph Databases that use RDF as store

Data Definition Language (DDL)

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

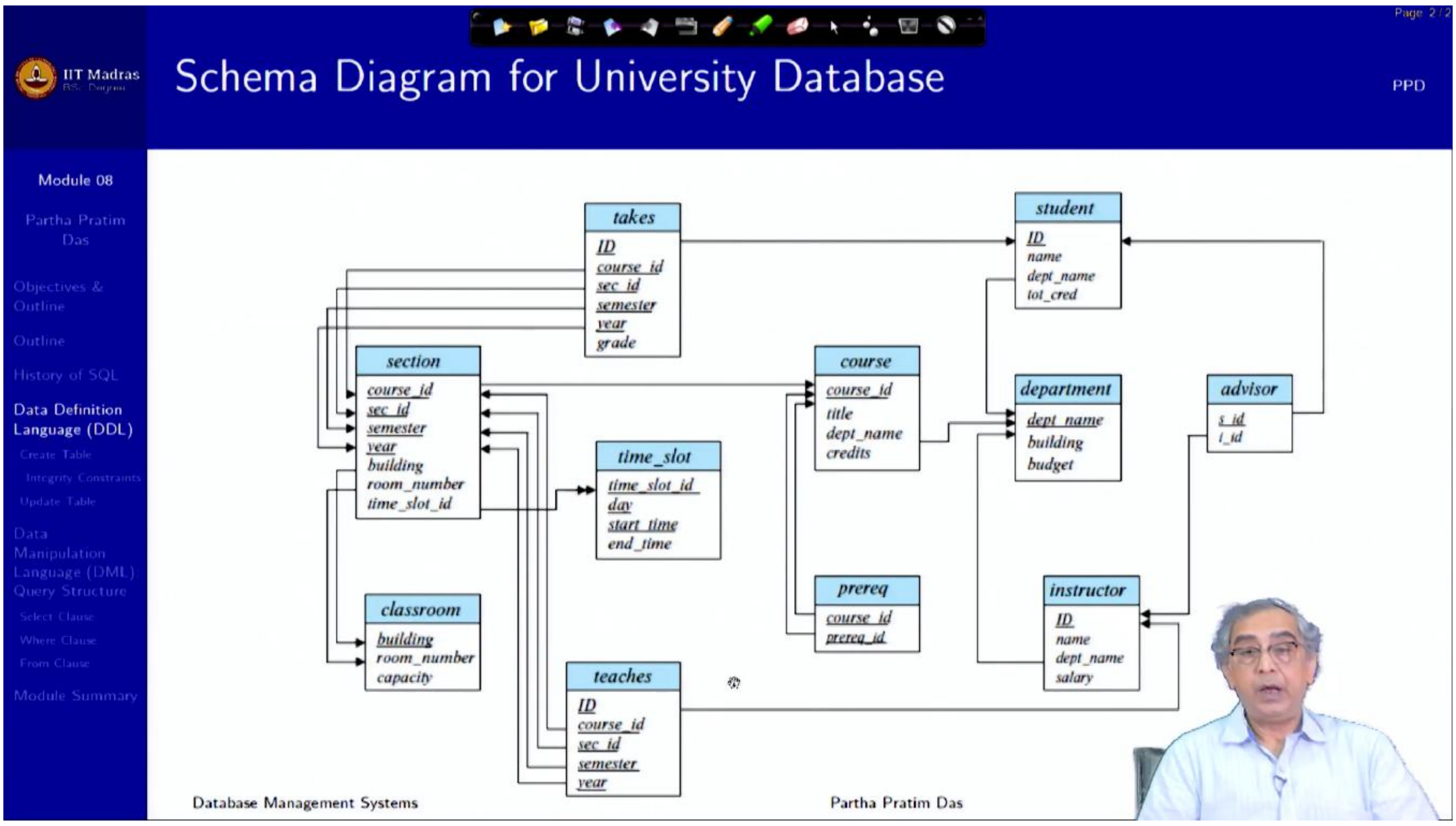
- The *Schema* for each *Relation*
- The *Domain* of values associated with each *Attribute*
- *Integrity Constraints*

- And, as we will see later, also other information such as ...
  - The set of *Indices* to be maintained for each relations
  - *Security and Authorization* information for each relation
  - The *Physical Storage Structure* of each relation on disk

Domain types (or Data types) in SQL

- ***char(n)*** - Fixed length character string, with user-specified length *n*
- ***varchar(n)*** - Variable length character strings, with user-specified max length *n*
- ***int*** - Integer (a finite subset of the integers that is machine-dependent)
- ***smallint(n)*** - 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 stored 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

Schema diagram for a University database



Create Table construct

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

  create table *r* $(A_1 D_1, A_2 D_2, ..., A_n D_n),$

  $(integrity - constraint_1),$

  ...

  $(integrity - constraint_k));$

- ***r*** is the name of the relation (table)
- 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));
```

University DB

instructor
ID
name
dept_name
salary

Create Table constructs: Integrity constraints

- **not null**
- **primary key** $(A_1, ..., A_n)$
- **foreign key** $(A_m, ..., A_n)$ **references *r***

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

```
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));
```

primary key declaration on an attribute automatically ensures ***not null***

Create Table construct: More relations

```
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 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);
```

```
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 (course_id, sec_id, semester, year) references section);
```

- **NOTE:** *sec_id* can be dropped from primary key above to ensure a student cannot register for two sections of the same course in the same semester

Update Tables

- **Insert** (DML command)

```
insert into instructor values ('10211', 'Smith', 'Biology', 66000);
```

- **Delete** (DML command)
  - Remove all tuples from the *student* relation

    ```
    delete from student
    ```

- **Drop Table** (DDL command)

```
drop table r
```

- **Alter** (DDL command) # to edit the schema

```
alter table r add A D
```

  - Where *A* is the name of the attribute to be added to relation to *r* and *D* is the domain of *A*
  - All existing 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 the attribute of relation *r*
  - Dropping of attributes not supported by many databases

Data Manipulation Language (DML): Query Structure

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 from r_i's
  - r_i represents a relation
  - *P* is a predicate
- The result of an SQL query is a relation

SELECT clause

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

```
select name from instructor
```

- **NOTE:** SQL names are case insensitive
  - Name = NAME = name
  - Some people prefer to use UPPER CASE wherever we use the **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 denotes *all attributes*

```
select *
from instructor
```

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

```
select '437'
```

  - Result is a table with one column and a single row with the 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
```

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 the Computer Science department

```
select name
from instructor
where dept_name = 'Comp. Sci.'
```

- Comparison results can be combined using the logical connectives **and, or, not**

- To find all instructors in Comp. Sci. department with salary > 80000

```
select name
from instructor
where dept_name = 'Comp. Sci.' and salary > 80000
```

- Comparisons can be applied to results of arithmetic expressions

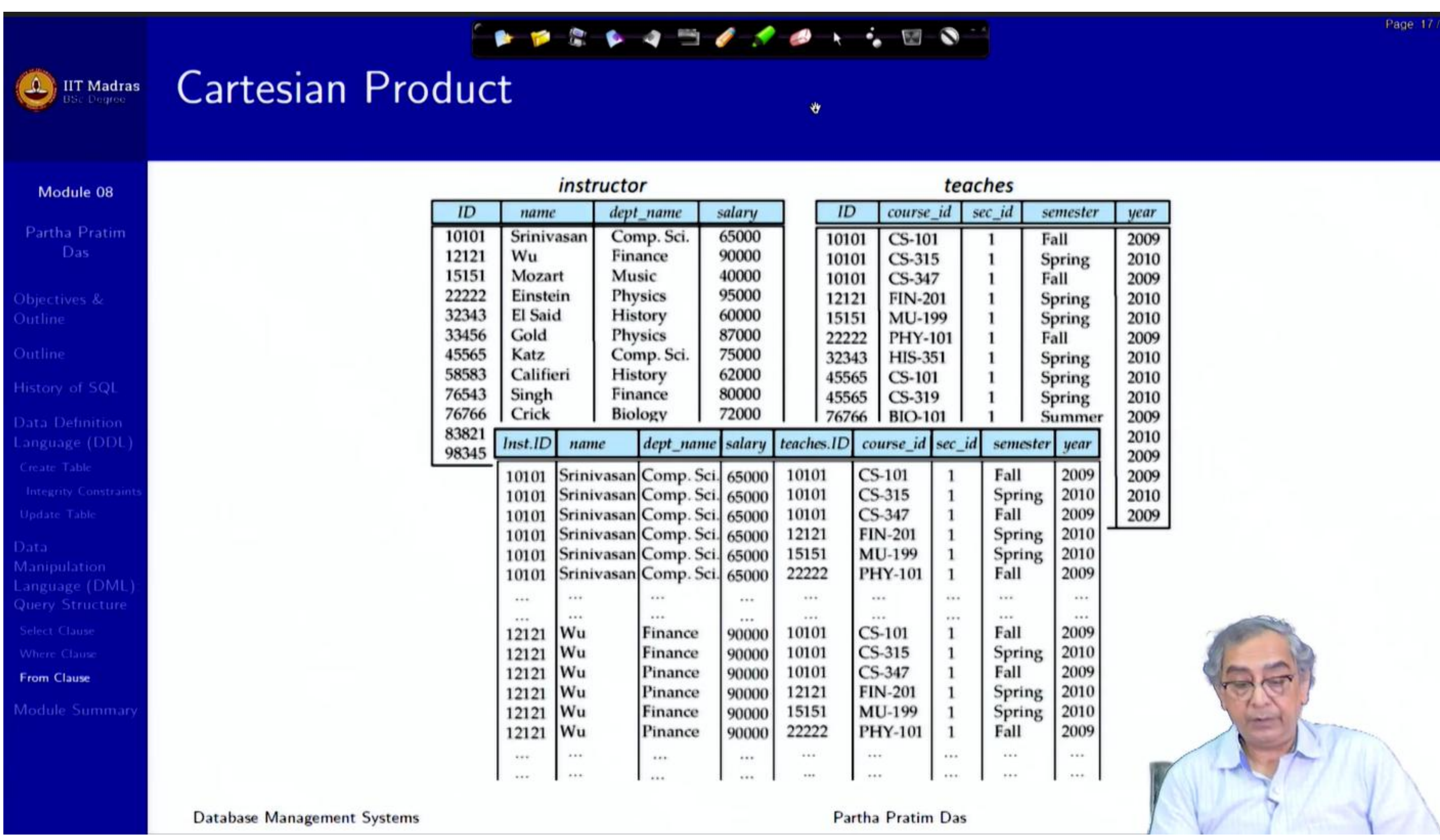
FROM clause

- 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 (for eg: ID), the attributes in the resulting table are renamed using the relation name (for eg: instructor.ID)
- Cartesian product is not very useful directly, but useful when combined with the where-clause condition (selection operation in relational algebra)

Cartesian product



Week 2 Lecture 4

Class	BSCCS2001
Created	@September 3, 2021 11:26 AM
Materials	
Module #	9
Type	Lecture
Week #	2

Introduction to Structured Query Language (SQL) (part 2)

Cartesian product (cont. from the previous lecture's end)

Example

- Find the names of all instructors who have taught some courses and the course_id

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
```

  - Equi-Join, Natural Join

instructor

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

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
				2010
				2009
				2009
				2010
				2009

Inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~12121~~	~~FIN-201~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~10101~~	~~Srinivasan~~	~~Comp. Sci.~~	~~65000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-101~~	~~1~~	~~Fall~~	~~2009~~
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-315~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~10101~~	~~CS-347~~	~~1~~	~~Fall~~	~~2009~~
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~15151~~	~~MU-199~~	~~1~~	~~Spring~~	~~2010~~
~~12121~~	~~Wu~~	~~Finance~~	~~90000~~	~~22222~~	~~PHY-101~~	~~1~~	~~Fall~~	~~2009~~
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

- Here in this table, we do not have the names of the courses
- If we want the name, we will again have to do a similar join operation with a table that has the names of the courses
  - This operations is known as **Natural Join**
- Example

  Find the names of all the instructors in the Art dept. who have taught some courses and the *course_id*

  ```
  select name, course_id
  from instructor, teaches
  where instructor.ID = teaches.ID and instructor.dept_name = 'Art'
  ```

Rename AS operation

- The SQL allows renaming relations and attributes using the **as** clause:

  ```
  old_name as new_name
  ```

- Find the names of all the 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.'
  ```

- The keyword **as** is optional and may be omitted

  instructor **as** *T* $\equiv$ *instructor T*

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 sub-string

- underscore (_)

  The _ character matches any character

- Find the names of all instructors whose name includes the sub-string "dar"

```
select name
from instructor
where name like '%dar%'
```

- Match the string "100%"

```
like '100%' escape '\'
```

in the above example, we use the backslash (\ ) as the escape character

and **'%dar%'** could match **Dar**win, Majum**dar,** Sar**dar** or Ud**dar**in

meanwhile, '%dar___' (**dar** followed by 3 underscores), it will match **Dar**win, but not the others

- Patterns are _case sensitive_
- Pattern matching example
  - 'Intro%' matches any string beginning with "Intro"
  - '%Comp%' matches any string containing "Comp" as a substring
  - '___' (3 underscores) many any string of exactly 3 characters
  - '___%' (3 underscores and then a %) matches any string of at least 3 characters
- SQL supports variety of string operations such as
  - Concatenation (using "||") [double pipe symbol]
  - Converting from upper to lower case (and vice-versa)
  - Finding the string length, extracting substrings, etc...

Ordering the display of tuples (ORDER BY clause)

- List in alphabetic order the names of all the 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*

Selecting number of tuples in output

- The **Select Top** clause is used to specify the number of records to return
- The **Select Top** clause is useful on large tables with thousands of records.
  - Returning a large number of records can impact performance

  ```
  select top 10 distinct name
  from instructor
  ```

- Not all database systems support the **SELECT TOP** clause.
  - SQL Server & MS Access support **select top**
  - MySQL supports the **limit** clause

- Oracle uses **fetch first** *n* **rows only** and **rownum**

  ```
  select distinct name
  from instructor
  order by name
  fetch first 10 rows only
  ```

WHERE clause predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all the instructors with salary between $90,000 and $100,000

  (that is, $\geq$ $90,000 and $\leq$ $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');
  ```

IN operator

- The **in** operator allows you to specify multiple values in a **where** clause
- The **in** operator is a shorthand for multiple **or** conditions

  ```
  select name
  from instructor
  where dept_name in ('Comp. Sci.', 'Biology')
  ```

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result
- **Multiset** versions of some of the relational algebra operators - given multiset relations r_1 and r_2:

  a) **SELECT** $\sigma_\theta(r_1)$ **:** If there are c_1 copies of tuple t_1 in r_1 and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$

  b) **PROJECTION** $\Pi_A(r)$ **:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1

  c) $r_1 \times r_2$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuples t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1 \cdot t_2$ in $r_1 \times r_2$

- Example: Suppose multiset relations $r_1(A, B)$ and $r_2(C)$ are as follows:

  $r_1 = \{(1, a)(2, a)\}$; $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$ while $\Pi_B(r_1) \times r_2$ would be

  $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$

- SQL duplicate semantics:

  select $A_1, A_2, ..., A_n$

  from $r_1, r_2, ..., r_m$

  where *P*

  is equivalent to the multiset version of the expression:

  $$\Pi_{A_1, A_2, ..., A_n}(\sigma_P(r_1 \times r_2 \times ... \times r_m))$$

Week 2 Lecture 5

Class	BSCCS2001
Created	@September 4, 2021 6:05 PM
Materials	
Module #	10
Type	Lecture
Week #	2

Introduction to Structured Query Language (SQL) (part 3)

Set operations

Example

- Find the courses that ran in Fall 2009 **or** in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
union
(select course_id from section where sem = 'Spring' and year = 2010)
```

- Find the courses that ran in Fall 2009 **and** in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
intersect
(select course_id from section where sem = 'Spring' and year = 2010)
```

- Find the courses that ran in Fall 2009 **but not** in Spring 2010

```
(select course_id from section where sem = 'Fall' and year = 2009)
except
(select course_id from section where sem = 'Spring' and year = 2010)
```

- Find the salaries of all the instructors that are less than the largest salary

```
select distinct T.salary
from instructor as T, instructor as S
where T.salary < S.salary
```

- Find the salaries of all the instructors

```
select distinct salary
from instructor
```

- Find the largest salary of all the instructors

```
(select distinct salary from instructor)
except
(select distinct T.salary from instructor as T, instructor as S where T.salary < S.salary)
```

- Set operations such as **union**, **intersect** and **except** automatically eliminate the duplicates
- To retain all the duplicates, use the corresponding multiset versions **union all**, **intersect all** and **except all**
- Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then it occurs ...
  - $m + n$ times in *r* **union all** *s*
  - min(*m*, *n*) times in *r* **intersect all** *s*
  - max(0, *m* - *n*) times in *r* **except all** *s*

NULL values

- What is a NULL value?

  A NULL value is something unknown or a value that does not exist yet
- Why is NULL value so important?
  - Certain values may not exist for everyone

    For eg: Every student may not have a passport at the time of registration
  - Often times while we are creating/inserting a record, we may not know all the values of all the fields

    For eg: When a student joins, the student does not have any credit assigned to him/her, so the total credit is NULL

    We can say 0 (zero), but 0 (zero) and NULL are different

    0 (zero) means the student has not taken a credit

    NULL means the credit has not been given yet
  - Naturally, when we add an attribute to all the existing rows of a table, the value of the particular field cannot be known, cannot be set, so it will have to initialized as a NULL value
- It is possible for tuples to have a *null* value, denoted by **null**, for some of their attributes
- The predicate **is null** can be used to check for *null* values
  - Example: Find all the instructors whose salary is *null*

    ```
    select name
    from instructor
    where salary is null
    ```
- It is not possible to test for *null* values with comparison operators such as $=, <, > or <>$

  We need to use the **is null** and **is not null** operators instead

NULL values: Three valued logic

- Three values - **true, false, unknown**
- Any comparison with *null* returns *unknown*
  - Example: $5 < null$ or $null <> null$ or $null = null$

- Three-valued logic using the value unknown:
  - **OR:**

    (*unknown* **or** *true*) = *true*

    (*unknown* **or** *false*) = *unknown*

    (*unknown* **or** *unknown*) = *unknown*
  - **AND:**

    (*true* **and** *unknown*) = *unknown*

    (*false* **and** *unknown*) = *false*

    (*unknown* **and** *unknown*) = *unknown*
  - **NOT:**

    (**not** *unknown*) = *unknown*
  - "*P* **is unknown**" evaluates to *true* if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate functions

- These functions operate on the multiset of values of a column of a relation (table) and return a value

  avg: average value

  min: minimum value

  max: maximum value

  sum: sum of the values

  count: number of values

Examples

- 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 2010 semester

```
select count(distinct ID)
from teaches
where semester = 'Spring' and year = 2010
```

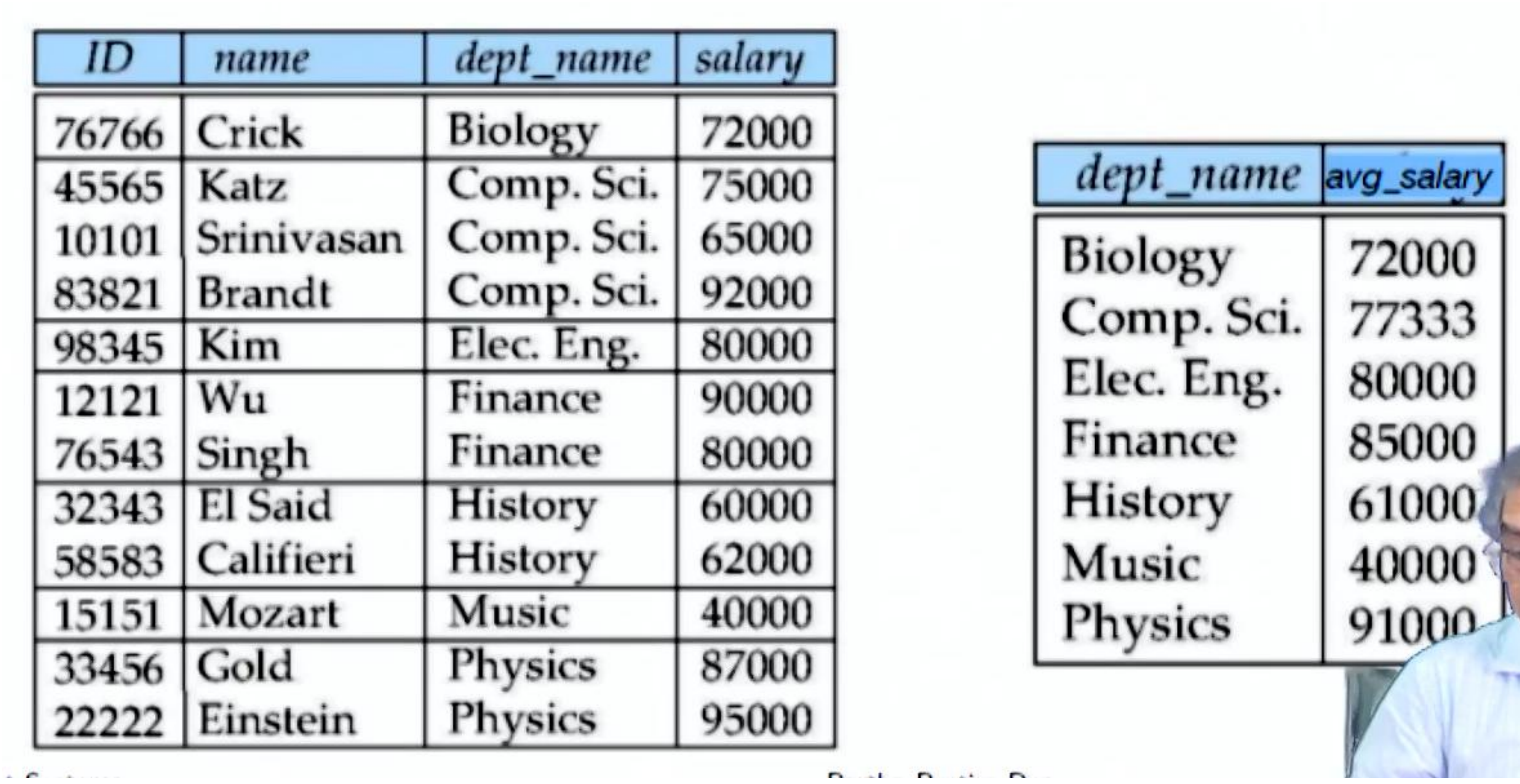
- Find the number of tuples in the *course* relation (table)

```
select count(*)
from courses;
```

Example (GROUP BY)

- Find the average salary of instructors in each department

```
select dept_name. avg(salary) as avg_salary
from instructor
group by dept_name;
```



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

dept_name	avg_salary
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

So, **group by** takes a column and makes sub-tables of all those records which have the same value on that particular group by attribute

It then applies the aggregate function on the column based on this sub-table

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list

```
-- The following query is incorrect because of the 'ID' attribute
select dept_name, ID, avg(salary)
from instructor
group by dept_name;
```

HAVING clause

- Find the names and average salaries of all departments whose average salary is greater than 42,000

```
select dept_name, ID, 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

NULL values and aggregates

- Total all salaries

```
select sum(salary)
from instructor;
```

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0 (zero)
  - all other aggregates return null

Week 3 Lecture 1

Class	BSCCS2001
Created	@September 25, 2021 9:05 AM
Materials	
Module #	11
Type	Lecture
Week #	3

SQL Examples

SELECT DISTINCT

- From the classroom relation, find the names of buildings in which every individual classroom has capacity less than 100 (removing the duplicates).
    - Relation:

      classroom

building	room_number	capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

    - Query:

      ```
      SELECT DISTINCT building
      FROM classroom
      WHERE capacity < 100;
      ```

    - **Output:**

building

building
Painter
Taylor
Watson

SELECT ALL

- From the classroom relation, find the names of buildings in which every individual classroom has capacity less than 100 (**without** removing the duplicates).
  - Relation:

    classroom

building	room_number	capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

  - Query:

    ```
    SELECT ALL building
    FROM classroom
    WHERE capacity < 100;
    ```

  - **Output:**

building
Painter
Taylor
Watson
Watson

NOTE: The duplicate retention is default and hence it is a common practice to skip ***ALL*** immediately after ***SELECT***

Cartesian Product

- Find the list of all students of departments which have a budget < $100K

  ```
  SELECT name, budget
  FROM student, department
  WHERE student.dept_name = department.dept_name AND budget < 100000;
  ```

name	budget
Brandt	50000
Peltier	70000
Levy	70000
Sanchez	80000
Snow	70000
Aoi	85000
Bourikas	85000
Tanaka	90000

- The above query generates every possible student-department pair, which is the Cartesian product of student and department.
- Then, it filters all the rows with `student.dept_name = department.dept_name AND budget < 100000`
- The common attribute `dept_name` in the resulting table are renamed using the relation name - `student.dept_name` and `department.dept_name`

RENAME AS Operation

- The same query in the above case can be framed by renaming the table as shown below:

```
SELECT S.name AS studentname, budget AS deptbudget
FROM student AS S, department AS D
WHERE S.dept_name = D.dept_name AND budget < 100000;
```

studentname	deptbudget
Brandt	50000
Peltier	70000
Levy	70000
Sanchez	80000
Snow	70000
Aoi	85000
Bourikas	85000
Tanaka	90000

- The above query renames the relation `student AS S` and the relation `department AS D`
- It also displays the attribute `name` as `StudentName` and the `budget` as `DeptBudget`
- **NOTE:** The budget attribute does not have any prefix because it occurs only in the department relation

---

SELECT: AND and OR

- From the *`instructor`* and *`department`* relations in the figure, find out the names of all the instructors whose department is Finance or whose department is in any of the following buildings: Watson, Taylor

  instructor

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

  department

dept_name	building	budget
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000

dept_name	building	budget
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

- Query:

```
SELECT name
FROM instructor I, department D
WHERE D.dept_name = I.dept_name
AND (I.dept_name = 'Finance' OR building IN ('Watson', 'Taylor'));
```

- Output:

name
Srinivasan
Wu
Einstein
Gold
Katz
Singh
Crick
Brandt
Kim

String Operations

- From the `course` relation in the figure, find the titles of all the courses whose `course_id` has 3 alphabets indicating the department

course

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

- Query:

```
SELECT title
FROM course
WHERE course_id LIKE '___-%';  -- 3 underscores
```

- Output:

title
Intro. to Biology
Genetics
Computational Biology
Investment Banking
World History
Physical Principles

- The `course_id` of each department has either 2 or 3 alphabets in the beginning followed by a hyphen and then followed by a 3-digit number. The above query returns the names of those departments that have 3 alphabets in the beginning

---

ORDER BY

- From the `student` relation in the figure, obtain the list of all students in alphabetic order of departments and within each department, in decreasing order of total credits.

  student

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

  - Query:

    ```
    SELECT name, dept_name, tot_cred
    FROM student
    ORDER BY dept_name ASC, tot_cred DESC;
    ```

  - Output:

name	dept_name	tot_cred
Tanaka	Biology	120
Zhang	Comp. Sci.	102
Brown	Comp. Sci.	58
Williams	Comp. Sci.	54
Shankar	Comp. Sci.	32
Bourikas	Elec. Eng.	98
Aoi	Elec. Eng.	60
Chavez	Finance	110
Brandt	History	80
Sanchez	Music	38
Peltier	Physics	56
Levy	Physics	46

name	dept_name	tot_cred
Snow	Physics	0

How is this sort happening?

- The list is first sorted in alphabetic order of `dept_name`
- Within each department, it is sorted in decreasing order of total credits

---

IN Operator

- From the `teaches` relation in the figure, find the IDs of all the courses taught in the Fall or Spring of 2018

teaches

id	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

- Query:

```
SELECT course_id
FROM teaches
WHERE semester IN ('Fall', 'Spring')
AND year = 2018;
```

- Output:

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

- **NOTE:** Now we can use **DISTINCT** to remove duplicates

Set Operations: UNION

- For the same question in the above table, we can find the solution using **UNION** operator as follows:
  - Query:

```
SELECT course_id
FROM teaches
WHERE semester = 'Fall'
```

```
AND year = 2018
UNION
SELECT course_id
FROM teaches
WHERE semester = 'Spring'
AND year = 2018
```

- Output:

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

- **NOTE: UNION** removes all the duplicates. If we use **UNION ALL** instead of **UNION**, we get the same set of tuples as in the above example

---

Set Operations: INTERSECT

- From the `instructor` relation in the figure, find the names of all the instructors who taught in either Computer Science department or the Finance department and whose salary is > 80,000

  instructor

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

  - Query:

```
SELECT name
FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
INTERSECT
SELECT name
FROM instructor
WHERE salary > 80000;
```

  - Output:

name
Srinivasan
Katz

- **NOTE:** The same thing can be achieved by using the query:

```
SELECT name FROM instructor WHERE dept_name IN ('Comp. Sci.', 'Finance') AND salary < 80000;
```

Set Operation: EXCEPT

- From the `instructor` relation in the figure, find the names of all the instructors who taught in either the Computer Science department or the Finance department and whose salary is either $\geq 90,000$ or $\leq 70,000$

instructor

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

  - Query:

```
SELECT name
FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
EXCEPT
SELECT name
FROM instructor
WHERE salary < 90000 AND salary > 70000;
```

  - Output:

name
Srinivasan
Brandt
Wu

- **NOTE:** The same can be achieved by using the following query

```
SELECT name FROM instructor
WHERE dept_name IN ('Comp. Sci.', 'Finance')
AND (salary >= 90000 OR salary <= 70000);
```

Aggregate function: AVG

- From the `classroom` relation given in the figure, find the names and the average capacity of each building whose average capacity is greater than 25

classroom

building	room_number	capacity
Packard	101	500
Painter	514	10
Taylor	3128	70
Watson	100	30
Watson	120	50

- Query:

```
SELECT building, AVG(capacity)
FROM classroom
GROUP BY building
HAVING AVG(capacity) > 25;
```

- Output:

bulding	avg
Taylor	70.00
Packard	500.00
Watson	40.00

Aggregate function: MIN

- From the `instructor` relation given in the figure, find the least salary drawn by any instructor among all the instructors

instructor

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

- Query:

```
SELECT MIN(salary) AS least_salary FROM instructor;
```

- Output:

least_salary
40000

Aggregate function: MAX

- From the `instructor` relation given above, find the highest salary drawn by any instructor among all the instructors
  - Query:

```
SELECT MAX(salary) AS highest_salary FROM instructor;
```

  - Output:

highest_salary
95000

Aggregate function: COUNT

- From the `instructor` relation given above, find the number of instructors in each department
    - Query:

```
SELECT dept_name, COUNT(id) AS ins_count
FROM instructor
GROUP BY dept_name;
```

    - Output:

dept_name	ins_count
Comp. Sci.	3
Finance	2
Music	1
Physics	2
History	2
Biology	1
Elec. Eng.	1

Aggregate function: SUM

- From the `course` relation given in the figure, find the total credits offered by each department

course

course_id	title	dept_name	credits
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

    - Query:

```
SELECT dept_name, SUM(credits) AS sum_credits
FROM course
GROUP BY dept_name;
```

    - Output:

dept_name	sum_credits
Finance	3
History	3
Physics	4
Music	3
Comp. Sci.	17
Biology	11
Elec. Eng.	3

Week 3 Lecture 2

Class	BSCCS2001
Created	@September 25, 2021 5:30 PM
Materials	
Module #	12
Type	Lecture
Week #	3

Intermediate SQL

Nested sub-queries

- SQL provides a mechanism for the nesting of sub-queries
- A **sub-query** 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:

  - A_i can be replaced by a sub-query that generates a single value
  - r_i can be replace by any valid sub-query
  - P can be replaced with an expression of the form:

    B <operation> (sub-query)

    where B is an attribute and <operation> is to be defined later
- Input of a query → One or more relations
- Output of a query → Always a single relation

Subqueries in WHERE clause

- Typical use of subqueries is to perform tests

- For set membership
- For set comparisons
- For set cardinality

Set Membership

- Find the courses offered in Fall 2009 and in Spring 2010 (**INTERSECT** example)

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall'
AND year = 2009
AND course_id IN (
  SELECT course_id
  FROM section
  WHERE semester = 'Spring' AND year = 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010 (**EXCEPT** example)

```
SELECT DISTINCT course_id
FROM section
WHERE semester = 'Fall'
AND year = 2009
AND course_id NOT IN (
  SELECT course_id
  FROM section
  WHERE semester = 'Spring' AND year = 2010);
```

- 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);
```

NOTE: Above query can be written in a simple manner. The formulation above is just to simply illustrate SQL features

Set comparison - "SOME" clause

- 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';
```

- The same above 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: $<, \leq, >, \geq, =, \neq$

- **SOME** represents existential quantification [The entity in `"()"` is a tuple here]

  5 $<$ **SOME** (0, 5, 6) $\rightarrow$ true

  5 $<$ **SOME** (0, 5) $\rightarrow$ false

  5 $=$ **SOME** (0, 5) $\rightarrow$ true

  5 $\neq$ **SOME** (0, 5) $\rightarrow$ true # as 0 $\neq$ 5

($=$ **SOME**) $\equiv$ **IN**

However, ($\neq$ **SOME**) $\not\equiv$ **NOT IN**

Set Comparison - "ALL" clause

- Find the names of all the instructors whose salary is greater than the salary of <u>all</u> 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** $r \Leftrightarrow \forall t \in r$ such that (F <comp> t)

  where <comp> can be: $<, \leq, >, \geq, =, \neq$

- **ALL** represents universal quantification [The entity in `"()"` is a tuple here]

  5 $<$ **ALL** (0, 5, 6) $\rightarrow$ false

  5 $<$ **ALL**(6, 10) $\rightarrow$ true

  5 $=$ **ALL**(4, 5) $\rightarrow$ false

  5 $\neq$ **ALL**(4, 5) $\rightarrow$ true

  ($\neq$ **ALL**) $\equiv$ **NOT IN**

  However, ($=$ **ALL**) $\not\equiv$ **IN**

Test for empty relations: "EXISTS"

- The **EXISTS** construct returns the value <u>true</u> if the argument subquery is non-empty
  - **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 the courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
SELECT course_id
FROM section AS S
WHERE semester = 'Fall' AND year = 2009
AND EXISTS (
  SELECT * FROM section AS T
  WHERE semester = 'Spring' AND year = 2010
  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 by 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 the courses offered by the Biology department
    - Second nested query lists all the courses a particular student has taken
- **NOTE:** $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- **NOTE:** Cannot write this query string $=$ **ALL** and its variants

Test for absence of duplicate tuples: "UNIQUE"

- The **UNIQUE** construct tests whether a subquery has any duplicate tuples in its results
- The **UNIQUE** construct evaluates to "true" if a given subquery contains no duplicates
- Find all the courses that were offered at most once in 2009

```
SELECT T.course_id
FROM course AS T
WHERE UNIQUE (
  SELECT R.course_id
  FROM course AS R
  WHERE T.course_id = R.course_id
  AND R.year = 2009);
```

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:** We do not need a **HAVING** clause
- Another way to write the 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 the 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;
```

Subqueries in the SELECT clause

- Scalar subquery: 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 occurs if subquery returns more than one result tuple

Modifications 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 instructors;
```

- 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 deposit, the average salary changes
- **Solution:**
  - First, compute `AVG ( salary )` and find all the tuples to delete
  - Next, delete all the tuples found above (without recomputing **AVG** or retesting the tuples)

Insertion

- Add a new tuple to the 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);
```

- Add all instructors to the student relation with `tot_creds` set to 0

```
INSERT INTO student
SELECT id, name, dept_name, 0
FROM instructor;
```

- The **SELECT FROM WHERE** statement is evaluated fully before any of its results are inserted into the relation
- Otherwise queries like

```
INSERT INTO table1 SELECT * FROM table1;
```

  would cause problems

Updates

- Increase salaries of instructors whose salary is over $100,000 by 3% and all other by 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 the students

```
UPDATE student S
SET tot_creds = (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);
```

- Set `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;
```

Week 3 Lecture 3

Class	BSCCS2001
Created	@September 26, 2021 11:24 AM
Materials	
Module #	13
Type	Lecture
Week #	3

Intermediate SQL (part 2)

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 conditions)
- 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

Types of JOIN relations

- Cross join
- Inner join
    - Equi-join
        - Natural join
- Outer join
    - Left outer join
    - Right outer join
    - Full outer join
- Self-join

Cross JOIN

- CROSS JOIN returns the Cartesian product of rows from tables in the join
    - Explicit

    ```
    SELECT *
    FROM employee CROSS JOIN department;
    ```

    - Implicit

    ```
    SELECT *
    FROM employee, department;
    ```

JOIN Operations - Example

- Relation course

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 *prereq*

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

- Observe that

  prereq information is missing from CS-315 and

  course information is missing from CS-347

Inner JOIN

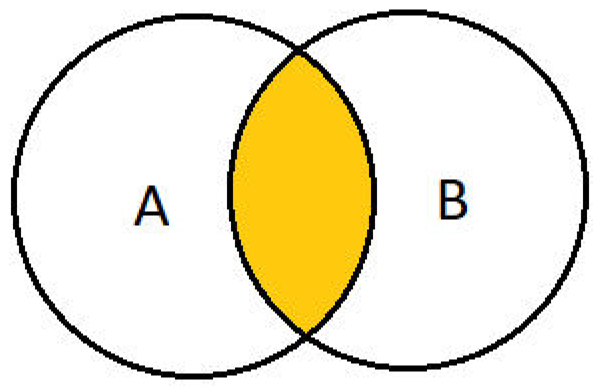
- *course* **INNER JOIN** *prereq*

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

- If specified as **NATURAL,** the 2^{nd} *course_id* field is skipped

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

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



Outer JOIN

- An extension of the join operation that avoids loss of information
- Computes the join and then adds tuples, from one relation that does not match tuples in the other relation, to the results of the join
- Uses *null* values

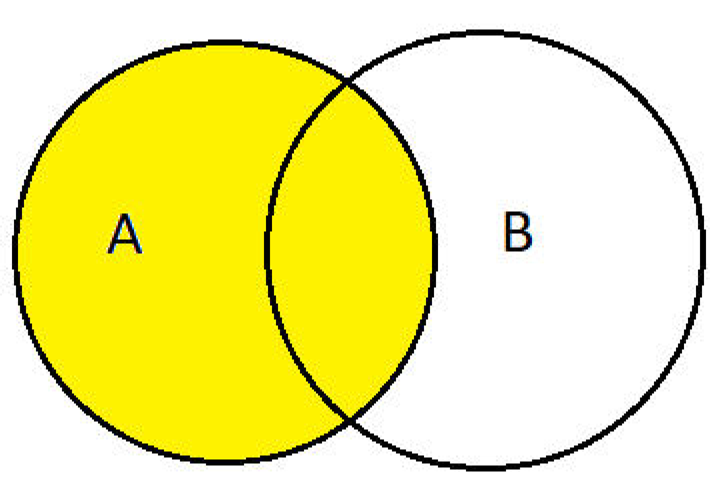
Left Outer JOIN

- *course* **NATURAL LEFT OUTER JOIN** *prereq*

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

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

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



Right Outer JOIN

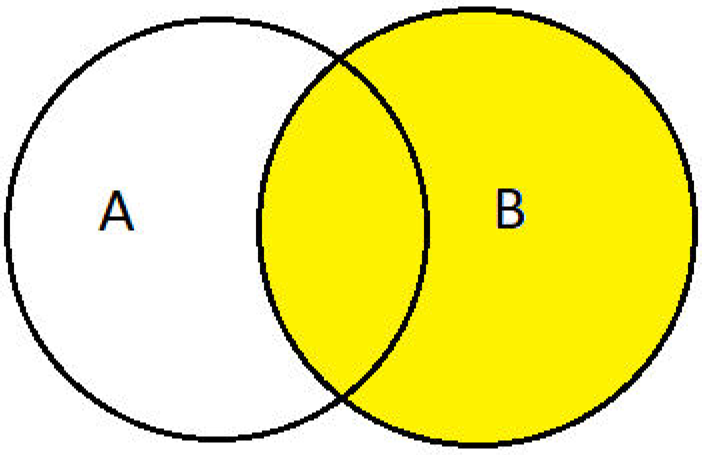
- *course* **NATURAL RIGHT OUTER JOIN** *prereq*

course_id	title	dept_name	credits	prere_id
BIO-301	Genetics	Biology	4	BIO-101

course_id	title	dept_name	credits	prere_id
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	null	null	null	CS-101

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

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



Joined relations

- **Join operations** take two relations and return a relation as the result
- These additional operations are typically used as subquery expressions in the **FROM** clause
- **Join condition** - defines which tuples in the two relations match, and what attributes are present in the result of the join
- **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 $(A_1, A_2, ..., A_n)$

Full outer JOIN

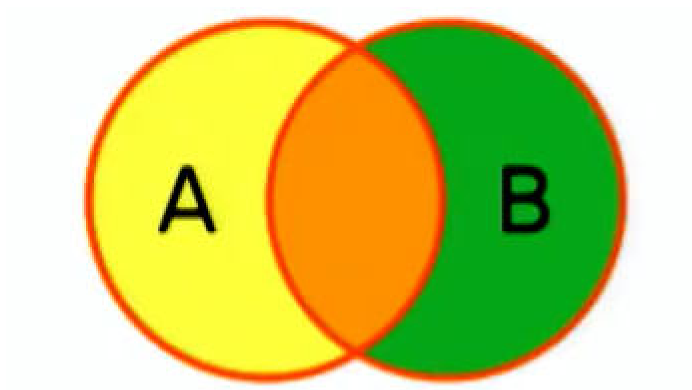
- *course* **NATURAL FULL OUTER JOIN** *prereq*

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

Aa course_id	title	dept_name	credits	prereq_id
CS-347	null	null	null	CS-101

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

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



Joined Relations - Example

- *course* **INNER JOIN** *prereq* **ON**

  course.course_id = prereq.course_id

Aa course_id	title	dept_name	credits	prere_id	courseid
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- What is the difference between the above (equi_join) and a natural join?
- *course* **LEFT OUTER JOIN** *prereq* **ON**

  course.course_id = prereq.course_id

Aa course_id	title	dept_name	credits	prere_id	courseid
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*

Aa course_id	title	dept_name	credits	prere_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*)

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

course_id	title	dept_name	credits	prere_id
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

Example views

- A view of instructors without their salary

```
CREATE VIEW faculty AS
  SELECT id, name, dept_name
  FROM instructor;
```

- Find all the 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;
```

View defined using other views

```
CREATE VIEW physics_fall_2009 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 = '2009';
```

```
CREATE VIEW physics_fall_2009_watson AS
  SELECT course_id, room_number
  FROM phsics_fall_2009
  WHERE building = 'Watson';
```

View expansion

- Expand use of a view in a query / another view

  ```
  CREATE VIEW physics_fall_2009_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 = '2009')
     WHERE building = 'Watson');
  ```

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 on 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

View expansion

- 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 with terminate

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 of 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 VALUE('69987', 'White', 'Taylor');
```

- Which department, if multiple departments in Taylor?

- What if no department is present in Taylor?
- 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

And some not at all

```
CREATE VIEW history_instructors AS
  SELECT * FROM instructor
  WHERE dept_name = 'History';
```

- What happens when we insert `('25566', 'Brown', 'Biology', 100000)` into the `history_instructors`?

Materialized views

- **Materializing a view:** Create a physical table containing all the tuples in the result of the query defining the view
- If relations used in the query are updated, the materialized view result becomes out of data
  - Need to maintain the view, by updating the view whenever the underlying relations are updated

Week 3 Lecture 4

Class	BSCCS2001
Created	@September 26, 2021 3:18 PM
Materials	
Module #	14
Type	Lecture
Week #	3

Intermediate SQL (part 3)

Transactions

- It is a unit of work
- Atomic transaction
    - Either something is fully executed or it is rolled back as if it never occurred
    - **Example:** Bank account transactions, when transferring money from one account to another, the transaction should either happen or not happen at all.

      It should not fail at a stage where money is deducted from one account and not added to the other account
- Isolation from concurrent transactions
- Transactions begin implicitly
    - Ended by **COMMIT WORK** or **ROLLBACK WORK**
- But default on most databases: each SQL statement commits automatically
    - Can turn off auto-commit for a session (for example, using API)
    - In SQL:1999, can use: **BEGIN ATOMIC ... END**
        - Not supported on most databases

Integrity Constraints

- Integrity constraints guard against accidental damage to the database by ensuring that the authorized changes to the database do not result in a loss of data consistency
    - A checking account must have a balance greater than Rs. 10,000.00

- A salary of a bank employee must be at least Rs. 250.00 an hour
- A customer must have a (non-null) phone number

Integrity constraints on a single relation

- **NOT NULL**
- **PRIMARY KEY**
- **UNIQUE**
- **CHECK(P)**, where P is a predicate

NOT NULL and UNIQUE constraints

- **NOT NULL**
  - 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 clause

- **CHECK(P),** where P is a predicate
- 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 appeals 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 than contain attributes A.
  - Here, 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

Cascading Actions in Referential Integrity

- With cascading, you can define the actions that the Database Engine takes when a user tries to delete or update a key to which existing foreign keys point

  ```
  CREATE TABLE course (
    course_id CHAR(5) PRIMARY KEY,
    title VARCHAR(20),
    dept_name VARCHAR(20) REFERENCES department
  )
  ```

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

- Alternative actions to cascade: **NO ACTION, SET NULL, SET DEFAULT**

Integrity constraint violation during transactions

```
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

SQL Data Types and Schemas

Built-in data types in SQL

- **DATE:** Dates, containing an (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 the 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

Index creation

```
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);
```

- Indices are data structures used to speed up access to records with specified values for index attributes

```
SELECT * FROM student
WHERE id = '12345';
```

- Can be executed by using the index to find the required record, without looking at all records of students

User-defined types

- **CREATE TYPE** construct in SQL creates user-defined type (alias, like typedef in C)

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

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

Domains

- **CREATE TYPE** 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

```
CREATE DOMAIN degree_level VARCHAR(10)
CONSTRAINT degree_level_test
CHECK (VALUE IN('Bachelors', 'Masters', 'Doctorate'));
```

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 than the large object itself

---

Authorization

- Forms of authorization 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
- 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 of SQL

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

```
GRANT <privilege list>
ON <relation name or view name> TO <user list>
```

- `<user list>` is:
  - A user-id

- **PUBLIC**, which allows all valid users the privilege granted
- A role

- 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 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 name or view name> FROM <user list>
```

- Example:

  `REVOKE SELECT ON branch 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

- `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 is
  - *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;
  ```

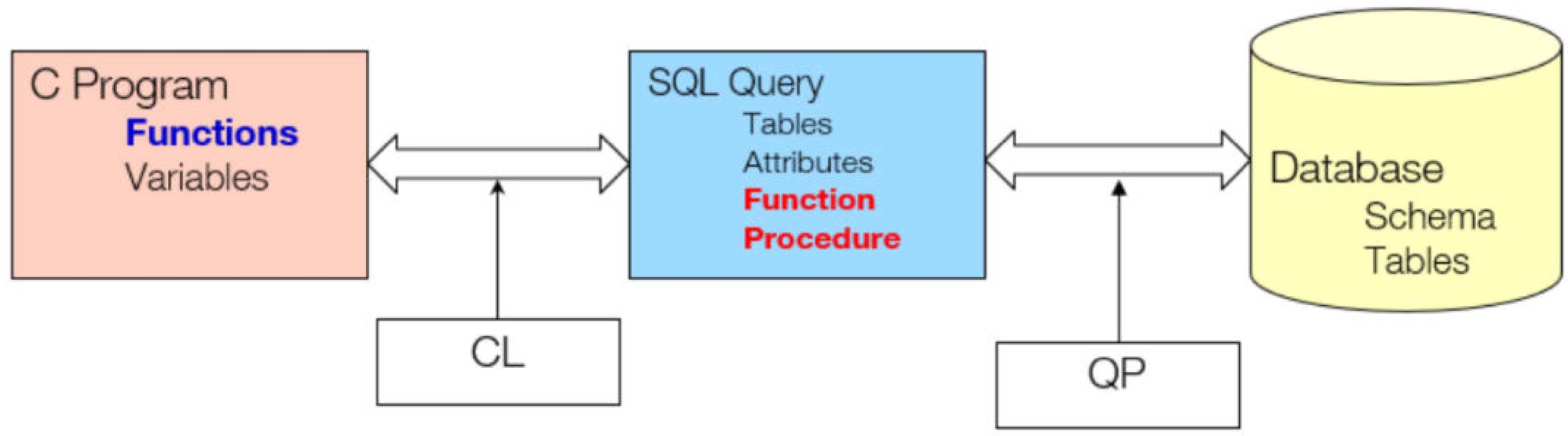
Week 3 Lecture 5

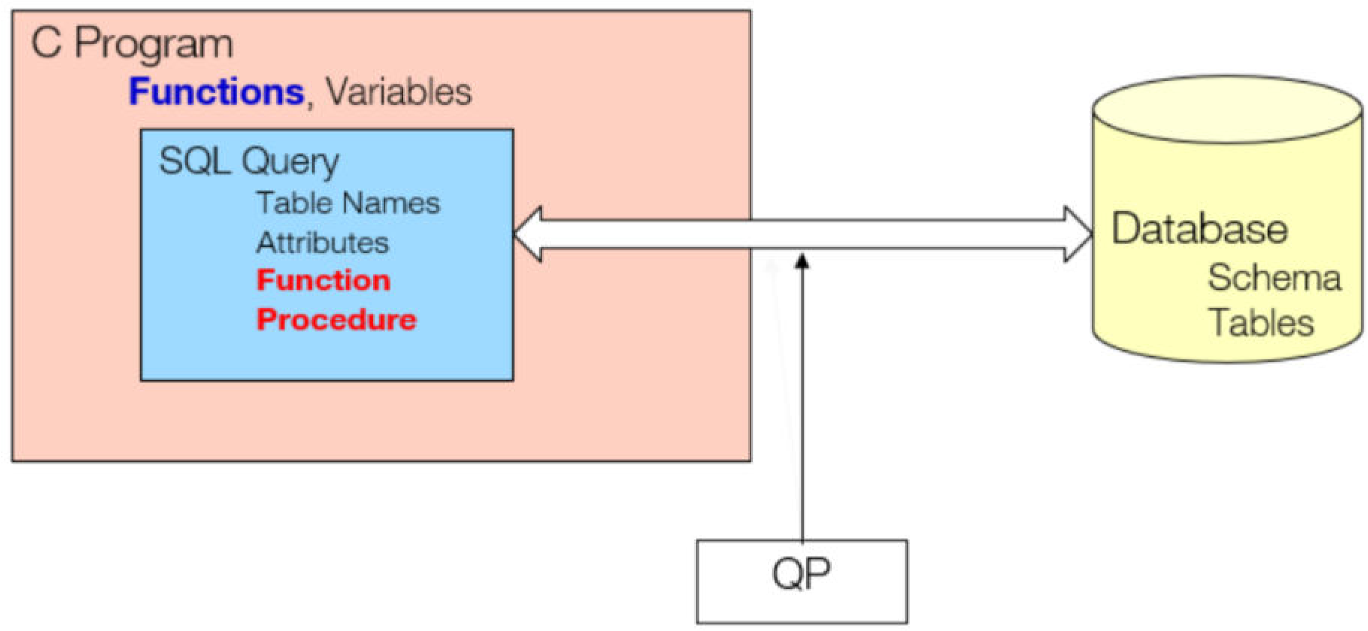
Class	BSCCS2001
Created	@September 26, 2021 10:24 PM
Materials	
Module #	15
Type	Lecture
Week #	3

Advanced SQL

Functions and Procedural Constructs

Native Language ← → Query Language





Functions and Procedures

- Functions / Procedures and Control Flow statements were added in SQL:1999
    - **Functions/Procedures** can be written in SQL itself or in an external programming language like C, Java, etc
    - Functions written in an external language are particularly useful with specialized data types such as images and geometric objects
        - Example: Functions to check if polygons overlap or to compare images for similarity
    - Some database systems support **table-valued functions** which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including `loops`, `if-then-else` and `assignment`
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

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))
  RETURN INTEGER
  BEGIN
  DECLARE d_count integer;
    SELECT COUNT(*) INTO d_count
    FROM instructor
    WHERE instructor.dept_name = dept_name
  RETURN d_cont;
  END
```

- The function dept_count can be used to find the department names and budget of all departments with more than 12 instructors:

```
SELECT dept_name, budget
FROM department
WHERE dept_count (dept_name) > 12;
```

- Compound statement: **BEGIN … END**

May contain multiple SQL statements between **BEGIN** and **END**

- **RETURNS:** indicates the variable-type that is returned (eg: integer)
- **RETURN:** specifies the values are to be returned as result of invoking the function
- SQL function are in fact parameterized views that generalize the regular notion of views by allowing parameters

Table functions

- **Functions that return a relation as a result** added in SQL:2003
- 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 instrutor.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
```

- 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:1999 allows **overloading** - more than one function/procedure of the same name as long as the number of arguments and/or the types of the arguments differ

Language constructs for procedures and 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
- Compound statement: **BEGIN … END**
  - May contain multiple SQL statements between **BEGIN** and **END**
  - Local variables can be declared within a compound statements
- **WHILE** loop:

```
WHILE boolean expression DO
  sequence of statements;
END WHILE;
```

- **REPEAT** loop:

```
REPEAT
  sequence of statements;
UNTIL boolean expression
END REPEAT;
```

- **FOR** loop:
  - Permits iteration over all results of a query
- Find the budget of all departments

```
DECLARE n INTEGER DEFAULT 0;
FOR r AS
  SELECT budget FROM department
DO
  SET n = n + r.budget
END FOR;
```

- Conditional statements
  - **if-then-else**
  - **case**
- **if-then-else** statement

```
IF boolean expression THEN
  sequence of statements;
ELSEIF boolean expression THEN
  sequence of statements;
...
ELSE
  sequence of statements;
END IF;
```

- The **IF** statement supports the use of optional **ELSEIF** clauses and a default **ELSE** clause
- Example procedure: registers student after ensuring classroom capacity is not exceeded
  - Returns 0 on success and -1 if the capacity is exceeded
- Simple **CASE** statement

```
CASE variable
  WHEN value1 THEN
    sequence of statements;
  WHEN value2 THEN
    sequence of statements;
  ...
  ELSE
    sequence of statements;
END CASE;
```

- The **WHEN** clause of the **CASE** statement defines the value that when satisfied determines the flow of control
- Searched **CASE** statement

```
CASE
  WHEN sql-expression = value1 THEN
    sequence of statements;
  WHEN sql-expression = value2 THEN
    sequence of statements;
  ...
  ELSE
    sequence of statements;
END CASE;
```

- Any supported SQL expression can be used here. These expressions can contain references to variables, parameters, special registers and more.
- Signaling of exception conditions and declaring handlers for exceptions

```
DECLARE out_of_classroom_seats CONDITION
DECLARE EXIT HANDLER FOR out_of_classroom_seats
BEGIN
  ...
  SIGNAL out_of_classroom_seats
  ...
END
```

- The handler here is **EXIT** - causes enclosing **BEGIN ... END** to terminate and exit
- Other actions possible on exception

External Language Routines

- SQL:1999 allows the definition of functions/procedures in an imperative programming language (Java, C#, C or C++) which can be invoked from SQL queries
- Such functions can be more efficient than functions defined in SQL. The computations that cannot be carried out in SQL can be executed by these functions
- Declaring external language procedures and functions

```
CREATE PROCEDURE dept_count_proc(
  IN dept_count VARCHAR(20),
  OUT count INTEGER
)
LANGUAGE C
EXTERNAL NAME '/usr/avi/bin/dept_count_proc'
```

```
CREATE FUNCTION dept_count(dept_name VARCHAR(20))
RETURNS integer
  LANGUAGE C
  EXTERNAL NAME '/usr/avi/bin/dept_count'
```

- Benefits of external language functions/procedures:
  - More efficient for many operations and more expressive power
- Drawbacks:
  - Code to implement function may need to be loaded into the DB system and executed in the DB system's address space
    - Risk of accidental corruption of the DB structures
    - Security risk, allowing users access to unauthorized data
  - There are alternatives, which provide good security at the cost of performance
  - Direct execution in the DB system's space is used when efficiency is more important than security

External Language Routines: Security

- To deal with security problems, we can do one of the following:
  - Use **sandbox** techniques:
    - That is, use a safe language like Java, which cannot be used to access/damage other parts of the DB code
  - Run external language functions/procedures in a separate process, with no access to the DB process' memory
    - Parameters and results communicated via the inter-process communication
- Both have performance overheads
- Many DB systems support both above approaches as well as direct executing in DB system address space

Triggers

- A **TRIGGER** defines a set of actions that are performed in response to an **INSERT**, **UPDATE** or **DELETE** operation on a specified table
  - When such an SQL operation is executed, the trigger is said to have been activated
  - Triggers are optional

    - Triggers are defined using the **CREATE TRIGGER** statement
- Triggers can be used
    - To enforce data integrity rules via referential constraints and check constraints
    - To cause updates to other tables, automatically generate or transform values for inserted or updated rows, or invoke functions to perform tasks such as issuing alerts
- To design a trigger mechanism, we must:
    - Specify the events / (like **UPDATE**, **INSERT** or **DELETE**) for the trigger to executed
    - Specify the time (**BEFORE** or **AFTER**) of execution
    - Specify the actions to be taken when the trigger executes
- Syntax of triggers may vary across systems

Types of Triggers: BEFORE

- **BEFORE** triggers
    - Run before an **UPDATE** or **INSERT**
    - Values that are being updated or inserted can be modified before the DB is actually modified.

      You can use triggers that run before an **UPDATE** or **INSERT** to ...
        - Check or modify the values before they are actually updated or inserted in the DB
            - Useful if user-view and internal DB format differs
        - Run other non-DB operations coded in user-defined functions
- **BEFORE DELETE** triggers
    - Run before a **DELETE**
        - Checks value (and raises an error, if necessary)

Types of Triggers: AFTER

- **AFTER** triggers
    - Run after an **UPDATE**, **INSERT** or **DELETE**
    - You can use triggers than run after an update or insert to:
        - Update data in other tables
            - Useful to maintain relationships between data or keep audit trail
        - Check against other data in the table or in other tables
            - Useful to ensure data integrity when referential integrity constraints aren't appropriate
            - When table check constraints limit checking to the current table only
        - Run non-DB operations coded in user-defined functions
            - Useful when issuing alerts or to update information outside the DB

Row level and Statement level Triggers

There are two types of triggers based on the level at which the triggers are applied:

- **Row level triggers** are executed whenever a row is affected by the event on which the trigger is defined
    - Let *Employee* be a table with 100 rows.
    - Suppose an **UPDATE** statement is executed to increase the salary of each employee by 10%
    - Any row level **UPDATE** trigger configured on the table *Employee* will affect all the 100 rows in the table during this update
- **Statement level triggers** perform a single action for all the rows affected by a statement, instead of executing a separate action for each affected row
    - Used for each statement instead of for each row

- Uses 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

Triggering Events and Actions in SQL

- Triggering event can be an **INSERT, DELETE** or **UPDATE**
- Triggers on update can 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;
  ```

Trigger to maintain `credits_earned` value

```
CREATE TRIGGER credits_earned AFTER UPDATE OF takes ON (grade)
REFERENCING NEW ROW AS nrow
REFERENCING OLD ROW AS orow
FOR EACH ROW
WHEN nrow.grade <> 'F' AND nrow.grade IS NOT NULL
  AND (orow.grade = 'F' OR orow.grade IS NULL)
BEGIN ATOMIC
  UPDATE student
  SET tot_cred = tot_cred +
    ( SELECT credits
      FROM course
      WHERE course.course_id = nrow.course_id)
    WHERE student.id = nrow.id;
END;
```

How to use triggers?

- The optimal use of DML triggers is for short, simple and easy to maintain write operations that act largely independent of an application business logic
- Typical and recommended uses of triggers include:
  - Logging changes to a history table
  - Auditing users and their actions against sensitive tables
  - Adding additional values to a table that may not be available to an application (due to security restrictions or other limitations), such as:
    - Login/user name
    - Time an operation occurs
    - Server/database name
  - Simple validation

Source: *SQL Server triggers: The good and the scary*

How not to use triggers?

- Triggers are like Lays: *Once you pop, you cannot stop*

- One of the greatest challenges for architects and developers is to ensure that
  - triggers are used only as needed, and
  - to not allow them to become a one-size-fits-all solution for any data needs that happen to come along
- Adding triggers is often seen as faster and easier than adding code to an application, but the cost of doing so is compounded over time with each added line of code.

Source: *SQL Server triggers: The good and the scary*

Alright then, how to use triggers?

- Trigger can become dangerous when:
  - There are too many
  - Trigger code becomes complex
  - Triggers go cross-server - across DBs over networks
  - Triggers call other triggers
  - Recursive triggers are set to ON. The DB-level setting is set to off by default
  - Functions, stored procedures or views are in triggers
  - Iteration occurs

Source: *SQL Server triggers: The good and the scary*

Week 4 Lecture 1

Class	BSCCS2001
Created	@September 29, 2021 11:42 AM
Materials	
Module #	16
Type	Lecture
Week #	4

Formal Relational Query Languages

- Relational Algebra
    - Procedural and Algebra based
- Tuple Relational Calculus
    - Non-procedural and Predicate Calculus based
- Domain Relational Calculus
    - Non-procedural and Predicate Calculus based

Relational Algebra

- Created by Edgar F. Codd at IBM in 1970
- Procedural Language
- Six basic operators
    - Select: σ
    - Project: Π
    - Union: $\cup$
    - Set difference: $-$
    - Cartesian product: $\times$
    - Rename: ρ
- The operators take one or two relations as inputs and produce a new relation as the result

SELECT operation

- Notation: $\sigma_p(r)$
- p is called the **selection predicate**
- Defined as:

  $\sigma_p(r) = \{t | t \in r \text{ and } p(t)\}$

  where p is a formula in propositional calculus consisting of terms connected by

  $\wedge$ (**and**)

  $\vee$ (**or**)

  $\neg$ (**not**)

  Each term is one of:

  $< attribute > op < attribute > or < constant >$

  where op is one of: $=, \neq, >, \geq . < . \leq$

- Example of selection:

  $\sigma_{dept\_name} =_{'Physics'} (instructor)$

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

A	B	C	D
α	α	1	7
β	β	23	10

$$\sigma_{A=B \wedge D > 5}(r)$$

PROJECT operation

- Notation: $\Pi_{A_1, A_2, \ldots A_k}(r)$

  where A_1, A_2 are attribute names and r is a relation

- 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:** To eliminate the `dept_name` attribute of `instructor`

  $\Pi_{ID,\ name,\ salary}(instructor)$

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

A	C
α	1
α	1
β	1
β	2

=

A	C
α	1
β	1
β	2

UNION operation

- Notation: $r \cup s$
- Defined as: $r \cup s = \{t | t \in r \ or \ t \in s\}$
- For $r \cup s$ to be valid:
    - $r, \ s$ must have the same **arity** (same number of attributes)
    - The attribute domains must be compatible (ie: same data type)
    - **Example:** To find all the courses taught in the Fall 2009 semester or in the Spring 2010 semester or in both
      $\Pi_{course\_id}(\sigma_{semester="Fall" \wedge \ year=2009}(section)) \cup \Pi_{course\_id}(\sigma_{semester="Spring" \wedge \ year=2010}(section))$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	1
α	2
β	1
β	3

r ∪ s

DIFFERENCE operation

- Notation: $r - s$
- Defined as: $r - s = \{t | t \in r \; and \; t \notin s\}$
- 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 the courses taught in the Fall 2009 semester, but not in the Spring 2010 semester
  $\Pi_{course\_id}(\sigma_{semester=\text{"}Fall\text{"} \wedge \, year=2009}(section)) - \Pi_{course\_id}(\sigma_{semester=\text{"}Spring\text{"} \wedge \, year=2010}(section))$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	1
β	1

$r - s$

INTERSECTION operation

- Notation: $r \cap s$
- Defined as:
  $r \cap s = \{t | t \in r \; and \; t \in s\}$
- Assume:
  - $r, \; s$ have the same ability
  - Attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

A	B
α	2

$r \cap s$

CARTESIAN-PRODUCT operation

- Notation: $r \times s$
- Defined as:

  $r \times s = \{t\ q | t \in r\ and\ q \in s\}$
- Assume that attributes of $r(R)$ and $s(S)$ are disjoint

  That is, $R \cap S = \phi$
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used.

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

$r \times s$

RENAME operation

- Allows us to name and, therefore, refer to the results of relational-algebra expressions
- Allows us to refer to a relation by more than one name
- **Example:**

  $\rho_x(E)$

  returns the expression E under the name X
- If a relational algebra expression E has arity n, then

  $\rho_{x(A_1,A_2,...,A_n)}(E)$

  returns the result of the expression E under the name X and with the attributes renamed to

  $A_1, A_2, ..., A_n$

DIVISION operation

- The division operation is applied to two relations
- $R(Z) \div S(X)$, where X subset Z
- Let $Y = Z - X$ (and hence $Z = X \cup Y$)

  that is, let Y be the set of attributes of R that are not attributes of S
- The result of **DIVISION** is a relation $T(Y)$ that includes a tuple t if tuples t_R appear in R with $t_R[Y] = t$, and with
  - $t_R[X] = t_s$ for every tuple t_S in S
- For a tuple t to appear in the result T of the **DIVISION**, the value in t must appear in R in combination with every tuple in S
- Division is a derived operation and can be expressed in terms of other operations
- $r \div s \equiv \Pi_{R-S}(r) - \Pi_{R-S}(r)((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$

DIVISION Example #1

R

Lecturer	Module
Brown	Compilers
Brown	Databases
Green	Prolog
Green	Databases
Lewis	Prolog
Smith	Databases

S

Subject
Prolog

R | S

Lecturer
Green
Lewis

DIVISION Example #2

R

Lecturer	Module
Brown	Compilers
Brown	Databases
Green	Prolog
Green	Databases
Lewis	Prolog
Smith	Databases

S

Subject
Databases
Prolog

R | S

Lecturer
Green

DIVISION Example #3

A

sno	pno

B1

pno

A / B1

sno

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

sno
s1
s2
s3
s4

A / B2

sno
s1
s4

A / B3

sno
s1

DIVISION Example #4

- Relation r, s

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
∈	6
∈	1
β	2

s

B
1
2

r ÷ s

A
α
β

DIVISION Example #5

- Relation r, s:

r

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

s

D	E
a	1
b	1

r ÷ s

A	B	C
α	a	γ
γ	a	γ

eg: Students who have taken both "a" and "b" courses, with instructor "1"

(Find all the students who have taken all courses given by the instructor 1)

Week 4 Lecture 2

Class	BSCCS2001
Created	@September 30, 2021 10:23 AM
Materials	
Module #	17
Type	Lecture
Week #	4

Formal Relational Query Languages (part 2)

Predicate Logic

Predicate Logic or **Predicate Calculus** is an extension of **Propositional Logic** or **Boolean Algebra**

It adds the concept of predicates and quantifiers to better capture the meaning of statements that cannot be adequately expressed by propositional logic

Tuple Relational Calculus and **Domain Relational Calculus** are based on **Predicate Calculus**

Predicate

- Consider the statement: "x is greater than 3"
    - It has 2 parts
    - The first part is the variable **x**
        - It is the subject of the statement
    - The second part **"is greater than 3"**
        - It is the predicate of the statement
        - This refers to the property that the subject of the statement can have
- The statement "x is greater than 3" can be denoted by $P(x)$ where P denotes the predicate "is greater than 3" and x is the variable
- The predicate P can be considered as a function. It tells the truth value of the statement $P(x)$ at x
    - Once a value has been assigned to the variable x, the statement $P(x)$ becomes a proposition and has a $truth$ or $false$ value

- In general, a statement involving n variables $x_1, x_2, x_3, ..., x_n$ can be denoted by $P(x_1, x_2, x_3, ..., x_n)$
  - Here, P is also referred to as the n-place predicate or an n-ary predicate

Quantifiers

In predicate logic, predicates are used alongside quantifiers to express the extent to which a predicate is true over a range of elements

Using ***quantifiers*** to create such propositions is called ***quantification***

There are 2 types of quantifiers:

- **Universal Quantifier**
- **Existential Quantifier**

Universal Quantifier

Universal Quantification: Mathematical statements sometimes assert that a property is true for all the values of a variable in a particular domain, called the **Domain of Discourse**

- Such a statement is expressed using universal quantification
- The universal quantification of $P(x)$ for a particular domain is the proposition that assert that $P(x)$ is $true$ for all values of x in this domain
- The domain is very important here since it decides the possible values of x
- Formally, the universal quantification of $P(x)$ is the statement "$P(x)$ for all values of x in the domain"
- The notation $\forall P(x)$ denotes the universal quantification of $P(x)$
  - Here, $\forall$ is called the **universal quantifier**
  - $\forall P(x)$ is read as ***"for all x P(x)"***
- **Example:** Let $P(x)$ be the statement $"x + 2 > x"$
  - What is the truth value of the statement $\forall xP(x)$?
- **Solution:** As $x + 2$ is greater than x for any real number, so $P(x) \equiv T$ for all x or $\forall xP(x) \equiv T$

Existential Quantifier

Existential Quantification: Some mathematical statements assert that there is an element with a certain property

Such statements are expressed by existential quantification

Existential quantification can be used to form a proposition that is true if and only if $P(x)$ is $true$ for at least one value of x in the domain

- Formally, the existential quantification of $P(x)$ is the statement "There exists an element x in the domain such that $P(x)$"
- The notation $\exists P(x)$ denotes the existential quantification of $P(x)$
  - Here $\exists$ is called the existential quantifier
  - $\exists P(x)$ is read as "There is at least one such x such that $P(x)$"
- **Example:** Let $P(x)$ be the statement $"x > 5"$
  - What is the truth value of the statement $\exists xP(x)$?
- **Solution:** $P(x)$ is $true$ for all real numbers greater than 5 and $false$ for all real numbers less than 5
  - So, $\exists xP(x) \equiv T$

Tuple Relational Calculus

TRC is a non-procedural query language, where each query is of the form

$\{t|P(t)\}$

where t = resulting tuples

$P(t)$ = known as predicate and these are the conditions that are used to fetch t

$P(t)$ may have various conditions logically combined with **OR($\vee$), AND($\wedge$), NOT($\neg$)**

It also uses quantifiers:

$\exists t \in r(Q(t))$ = "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true

$\forall t \in r(Q(t))$ = $Q(t)$ is true "for all" tuples in relation r

- $\{P | \exists S \in Students \ and \ (S.CGPA > 8 \wedge P.name = S.name \wedge P.age = S.age)\}$:
  returns the name and age of students with a CGPA above 8

Predicate Calculus Formula

- Set of attributes and constants
- Set of comparison operators: (eg: $<, \leq, =, \neq, >, \geq$)
- Set of connectives: and($\wedge$), or($\vee$), not($\neg$)
- Implication ($\Rightarrow$)**:** $x \Rightarrow y$, if x is true, then y is true
  $x \Rightarrow y \equiv \neg x \vee y$
- Set of quantifiers:
  - $\exists t \in r(Q(t)) \equiv$ "there exists" a tuple in t in relation r such that predicate $Q(t)$ is true
  - $\forall t \in r(Q(t)) \equiv Q$ is true "for all" tuples t in relation r

TRC Example #1

Student

Fname	Lname	Age	Course
David	Sharma	27	DBMS
Aaron	Lilly	17	JAVA
Sahil	Khan	19	Python
Sachin	Rao	20	DBMS
Varun	George	23	JAVA
Simi	Verma	22	JAVA

Q. 1: Obtain the first name of students whose age is greater than 21

Solution:

$$\{t.Fname \mid Student \wedge t.age > 21\}$$

$$\{t.Fname \mid t \in Student \wedge t.age > 21\}$$

$$\{t \mid \exists s \in Student(s.age > 21 \wedge t.Fname = s.Fname)\}$$

Fname
David
Varun
Simi

TRC Example #2

Consider the relational schema

```
student(rollNo, name, year, courseId)
course(courseId, cname, teacher)
```

Q. 2: Find out the names of all the students who have taken the course named 'DBMS'

- $\{t \mid \exists s \in student \ \exists c \in course(s.courseId = c.courseId \wedge c.cname =' DBMS' \wedge t.name = s.name)\}$
- $\{s.name \mid s \in student \wedge \exists c \in course(s.courseId = c.courseId \wedge c.cname =' DBMS')\}$

Q. 3: Find out the names of all students and their rollNo who have taken the course named 'DBMS'

- $\{s.name, s.rollNo \mid s \in student \wedge \exists c \in course(s.courseId = c.courseId \wedge c.cname =' DBMS')\}$
- $\{t \mid \exists s \in student\ \exists c \in course(s.courseId = c.courseId \wedge c.cname =' DBMS' \wedge t.name = s.name \wedge t.rollNo = s.rollNo)\}$

TRC Example #3

Consider the following relations:

```
Flights(flno, from, to, distance, departs, arrive)
Aircraft(aid, aname, cruisingrange)
Certified(eid, aid)
Employees(eid, ename, salary)
```

Q. 4: Find the eids of pilots certified for Boeing aircraft

RA

$\Pi_{eid}(\sigma_{aname='Boeing'}(Aircraft \bowtie Certified))$

TRC

- $\{C.eid \mid C \in Certified \wedge \exists A \in Aircraft(A.aid = C.aid \wedge A.aname =' Boeing')\}$
- $\{T \mid \exists C \in Certified\ \exists A \in Aircraft(A.aid = C.aid \wedge A.aname =' Boeing' \wedge T.eid = C.eid)\}$

TRC Example #4

Consider the following relations:

```
Flights (flno, from, to, distance, departs, arrives)
Aircraft (aid, aname, cruisingrange)
Certified (eid, aid)
Employees (eid, ename, salary)
```

Q. 5: Find the names and salaries of certified pilots working on Boeing aircrafts

RA

$\Pi_{ename,salary}(\sigma_{aname='Boeing'}(Aircraft \bowtie Certified \bowtie Employees))$

TRC

$\{P \mid \exists E \in Employees\ \exists C \in Certified\ \exists A \in Aircraft(A.aid = C.aid \wedge A.aname =' Boeing' \wedge E.eid = C.eid \wedge P.ename = E.ename \wedge P.salary = E.salary)\}$

TRC Example #5

Consider the following relations:

```
Flights (flno, from, to, distance, departs, arrive)
Aircraft (aid, aname, cruisingrange)
Certified (eid, aid)
Employees (eid, ename, salary)
```

Q. 6: Identify the flights that can be piloted by every pilot whose salary is more than $\$100,000$

- $\{Fl.flno \mid F \in Flights \wedge \exists C \in Certified\ \exists E \in Employees(E.salary > 100,000 \wedge E.eid = C.eid)\}$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations
- For example, $\{t \mid \neg t \in r\}$ results in an infinite relation if the domain of any attribute of the relation r is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is safe if every component of t appears in one of the relations, tuples or constants that appear in P
  - **NOTE:** This is more than just a syntax condition
  - Eg: $\{t \mid t[A] = 5 \vee true\}$ is not safe → it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in P

Domain Relational Calculus

- A non-procedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

  $\{< x_1, x_2, ..., x_n > | P(x_1, x_2, ..., x_n)\}$

  - $x_1, x_2, ..., x_n$ represents domain variables
  - P represents a formula similar to that of the predicate calculus

Equivalence of Relational Algebra, Tuple Relational Calculus & Domain Relational Calculus

SELECT operation

$R = (A,\ B)$

Relational Algebra: $\sigma_{B=17}(r)$

Tuple Calculus: $\{t \mid t \in r \wedge B = 17\}$

Domain Calculus: $\{< a, b > \mid < a, b > \in r \wedge b = 17\}$

PROJECT operation

$R = (A,\ B)$

Relational Algebra: $\Pi_A(r)$

Tuple Calculus: $\{t \mid \exists p \in r(t[A] = p[A])\}$

Domain Calculus: $\{< a > \mid \exists\ b\ (< a, b > \in r)\}$

COMBINING operation

$R = (A,\ B)$

Relational Algebra: $\Pi_A(\sigma_{B=17}(r))$

Tuple Calculus: $\{t \mid \exists p \in r(t[A] = p[A] \wedge p[B] = 17)\}$

Domain Calculus: $\{< a > \mid \exists\ b\ (< a, b > \in r \wedge b = 17)\}$

UNION

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r \cup s$

Tuple Calculus: $\{t \mid t \in r \vee t \in s\}$

Domain Calculus: $\{< a, b, c > \mid < a, b, c > \in r \vee < a, b, c > \in s\}$

SET DIFFERENCE

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r - s$

Tuple Calculus: $\{t \mid t \in r \wedge t \notin s\}$

Domain Calculus: $\{< a, b, c > \mid < a, b, c > \in r \wedge < a, b, c > \notin s\}$

INTERSECTION

$R = (A, B, C) \quad S = (A, B, C)$

Relational Algebra: $r \cap s$

Tuple Calculus: $\{t \mid t \in r \wedge t \in s\}$

Domain Calculus: $\{< a, b, c > \mid < a, b, c > \in r \wedge < a, b, c > \in s\}$

CARTESIAN / CROSS PRODUCT

$R = (A, B) \quad S = (C, D)$

Relational Algebra: $r \times s$

Tuple Calculus: $\{t \mid \exists p \in r \exists q \in s(t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = q[C] \wedge t[D] = q[D])\}$

Domain Calculus: $\{< a, b, c, d > \mid < a, b > \in r \wedge < c, d > \in s\}$

NATURAL JOIN

$R = (A, B, C, D) \quad S = (B, D, E)$

Relational Algebra:

$r \bowtie s$

$\Pi_{r.A, r.B, r.C, r.D, s.E}(\sigma_{r.B = s.B \wedge r.D = s.D}(r \times s))$

Tuple Calculus:

$\{t \mid \exists\ p \in r\ \exists\ q \in s(t[A] = p[A] \wedge t[B] = p[B] \wedge t[C] = p[C] \wedge t[D] = p[D] \wedge t[E] = q[E] \wedge p[B] = q[B] \wedge p[D] = q[D])\}$

Domain Calculus:

$\{< a, b, c, d, e > \mid < a, b, c, d > \in r \wedge < b, d, e > \in s\}$

DIVISION

$R = (A, B) \quad S = (B)$

Relational Algebra: $r \div s$

Tuple Calculus: $\{t \mid \exists\ p \in r\ \forall\ q \in s(p[B] = q[B] \Rightarrow t[A] = p[A])\}$

Domain Calculus: $\{< a > \mid < a > \in r \wedge \forall < b > (< b > \in s \Rightarrow < a, b > \in r)\}$

Source: *https://www2.cs.sfu.ca/CourseCentral/354/louie/Equiv_Notations.pdf*

Week 4 Lecture 3

Class	BSCCS2001
Created	@September 30, 2021 4:40 PM
Materials	
Module #	18
Type	Lecture
Week #	4

Entity-Relationship Model

Design Process

What is a Design?

A Design:

- Satisfies a given (perhaps informal) functional specification
- Conforms to the limitations of the target medium
- Meets implicit or explicit requirements on performance and resource usage
- Satisfies implicit or explicit design criteria on the form of the artifact
- Satisfies restrictions on the design itself, such as its length or cost, or the tools available for doing the design

Role of Abstraction

- Disorganized Complexity results from
    - Storage (STM) limitations of the human brain - an individual can simultaneously comprehend of the order of seven, plus or minus two chunks of information
    - Speed limitations of human brain - it takes the mind about five seconds to accept a new chunk of information
- **Abstraction** provides the major tool to handle Disorganized Complexity by chunking information
- Ignore in-essential details, deal only with the generalized, idealized model of the world

    Consider: A binary number `110010101001`

    Hard to remember

Try the octal form: `(110)(010)(101)(001)` $\Longrightarrow$ **6251**

Or the hex form: `(1100)(1010)(1001)` $\Longrightarrow$ **CA9**

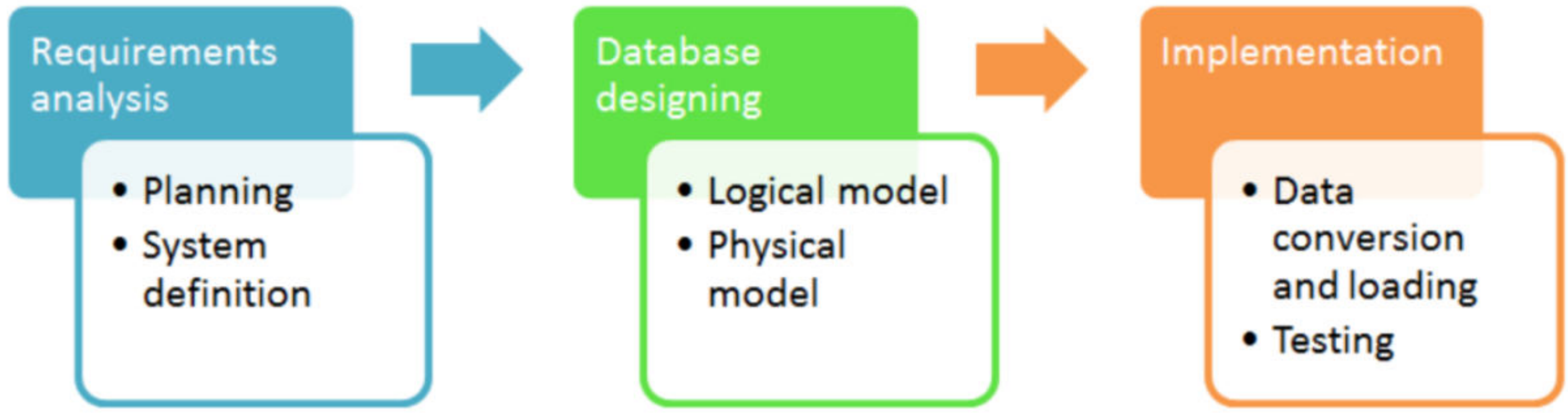
Model Building

- Physics
  - Time-Distance Equation
  - Quantum Mechanics
- Chemistry
  - Valency-bond Structures
- Geography
  - Maps
  - Projections
- Electrical Circuits
  - Kirchoff's Loop Equations
  - Time Series Signals and FFT
  - Transistor Models
  - Schematic Diagrams
  - Interconnect Routing
- Building & Bridges
  - Drawings - Plan, Elevation, Side view
  - Finite Element Models

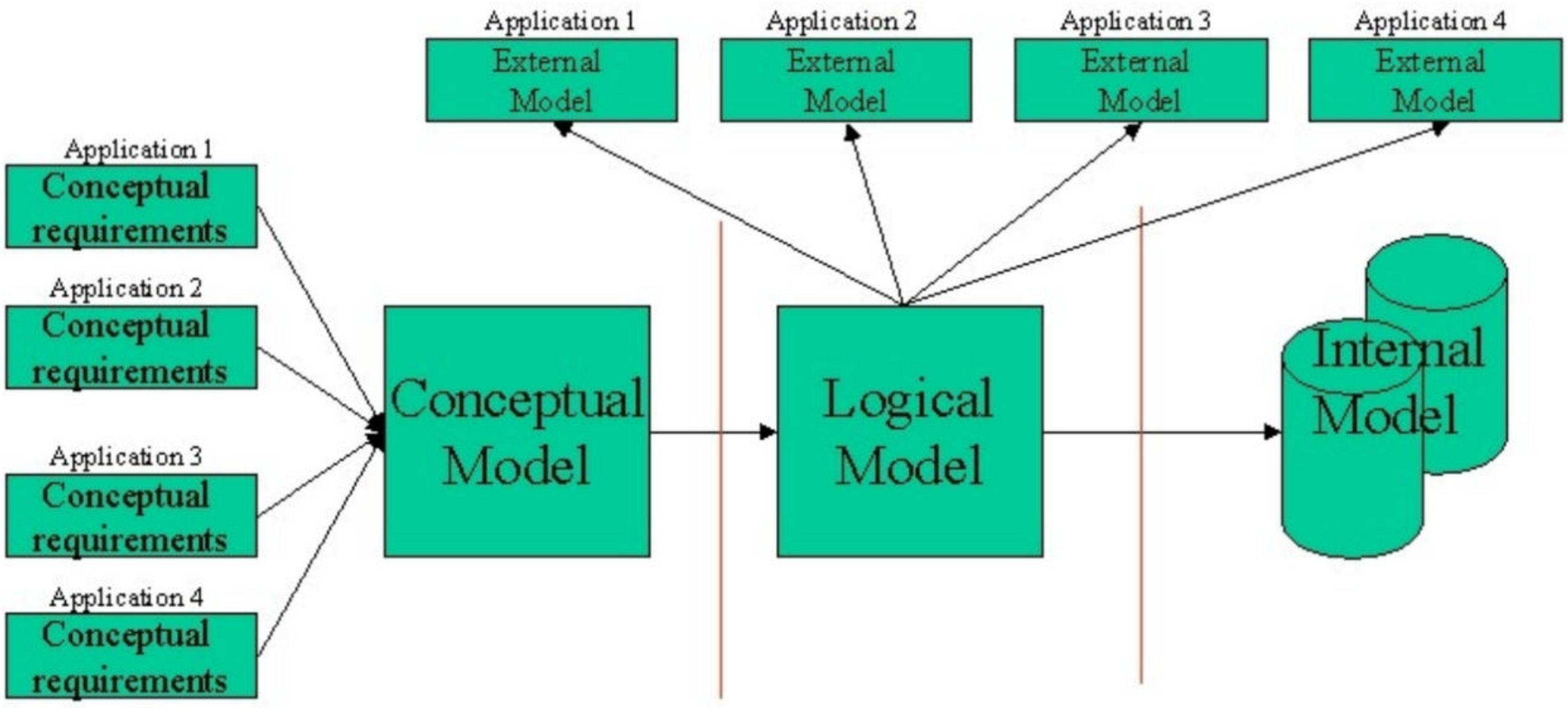
- Models are common in all engineering disciplines
- Model building follows principles of decomposition, abstraction and hierarchy
- Each model describes a specific aspect of the system
- Build new models upon old proven models

Design Approach

- **Requirement Analysis:** Analyse the data needs of the prospective DB users
  - Planning
  - System Defining
- **DB Designing:** Use a modeling framework to create abstraction of the real world
  - Logical Model
  - Physical Model
- **Implementation**
  - Data Conversion and Loading
  - Testing



- **Logical Model:** Deciding on a good DB schema
  - Business Decision: What attributes should we record in the DB?
  - Computer Science Decision: What relation schema should we have and how should the attributes be distributed among the various relation schema?
- **Physical Model:** Deciding on the physical layout of the DB



- **Entity Relationship Model**
  - Models an enterprise as a collection of entities and relationships
    - Entity → A distinguishable "thing" or "object" in the enterprise
      - Described by a set of attributes
    - Relationship → An association among multiple entities
  - Represented by an **Entity-Relationship** *or* **ER diagram**
- **Database Normalization**
  - Formalize what designs are bad and test for them

Entity Relationship (ER) Model

ER Model: Database Modeling

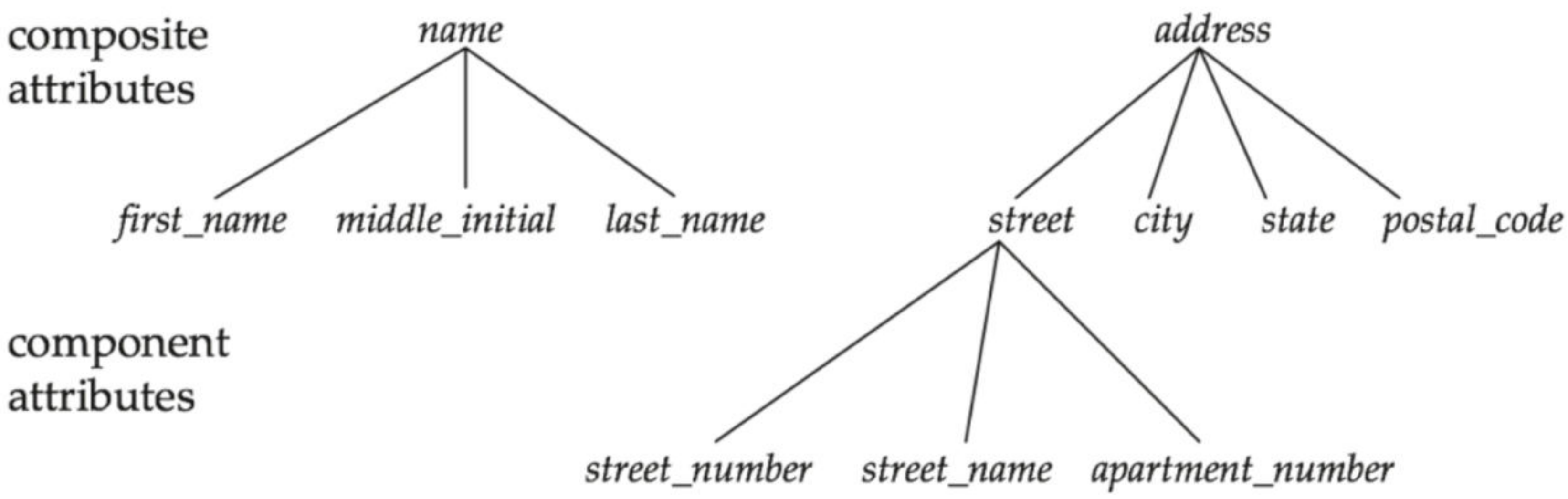
- The ER data model was developed to facilitate DB design by allowing specification of an enterprise schema that represents the overall logical structure of a DB
- The ER model is useful in mapping the meanings and interactions of the real world enterprises onto a conceptual schema
- The ER data model employs three basic concepts:
  - Attributes
  - Entity sets
  - Relationship sets
- The ER model also has an associated diagrammatic representation, the ER diagram, which can express the overall logical structure of a DB graphically

Attributes

- An attribute is a property associated with an entity / entity set
- Based on the values of certain attributes, an entity can be identified uniquely
- Attribute types:
  - Simple and Composite attributes
  - Single-valued and Multi-valued attributes
    - **Example:** Multi-valued 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

Attributes: Composite



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: ie, descriptive properties possessed by all members of an entity set
  - **Example:**

```
instructor = (ID, name, street, city, salary)
course = (course_id, title, credits)

-- Here ID and course_id are the primary keys, but
-- the tool I am using to make PDFs is not marking them underline
```

- A subset of the attributes form a primary key of the entity set; that is, uniquely identifying each member of the set
  - Primary key of an entity set is represented by underlining it

Entity sets - instructor and student

instructor

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

student

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

Relationship sets

- A relationship is an association among several entities

  Example:

  44553 (Peltier) *advisor* 22222 (Einstein)

  student entity relationship set instructor entity

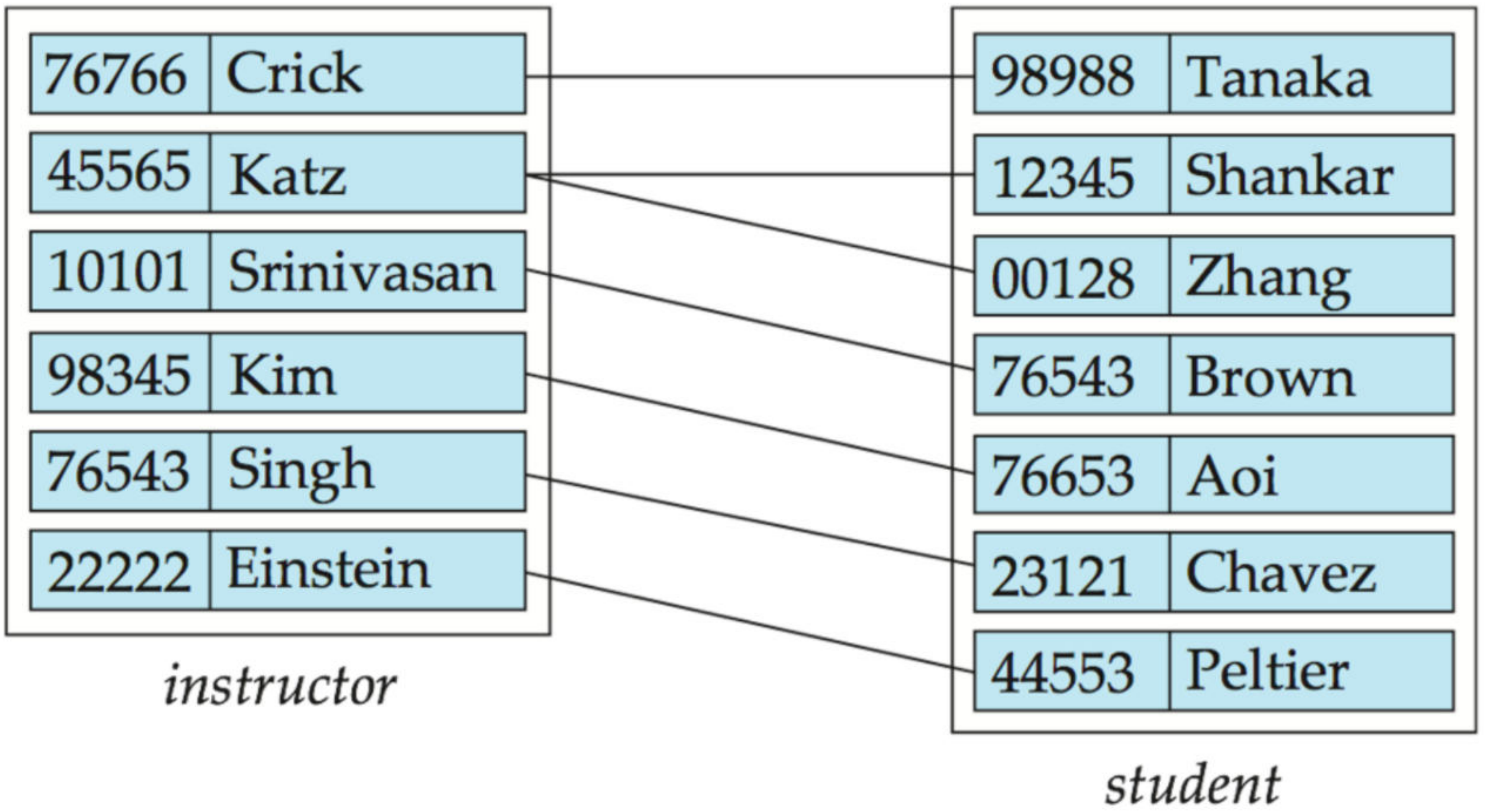
- A relationship set is a mathematical relation among $n \geq 2$ entities, each taken form entity sets

  $\{(e_1, e_2, ..., e_n) | e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$

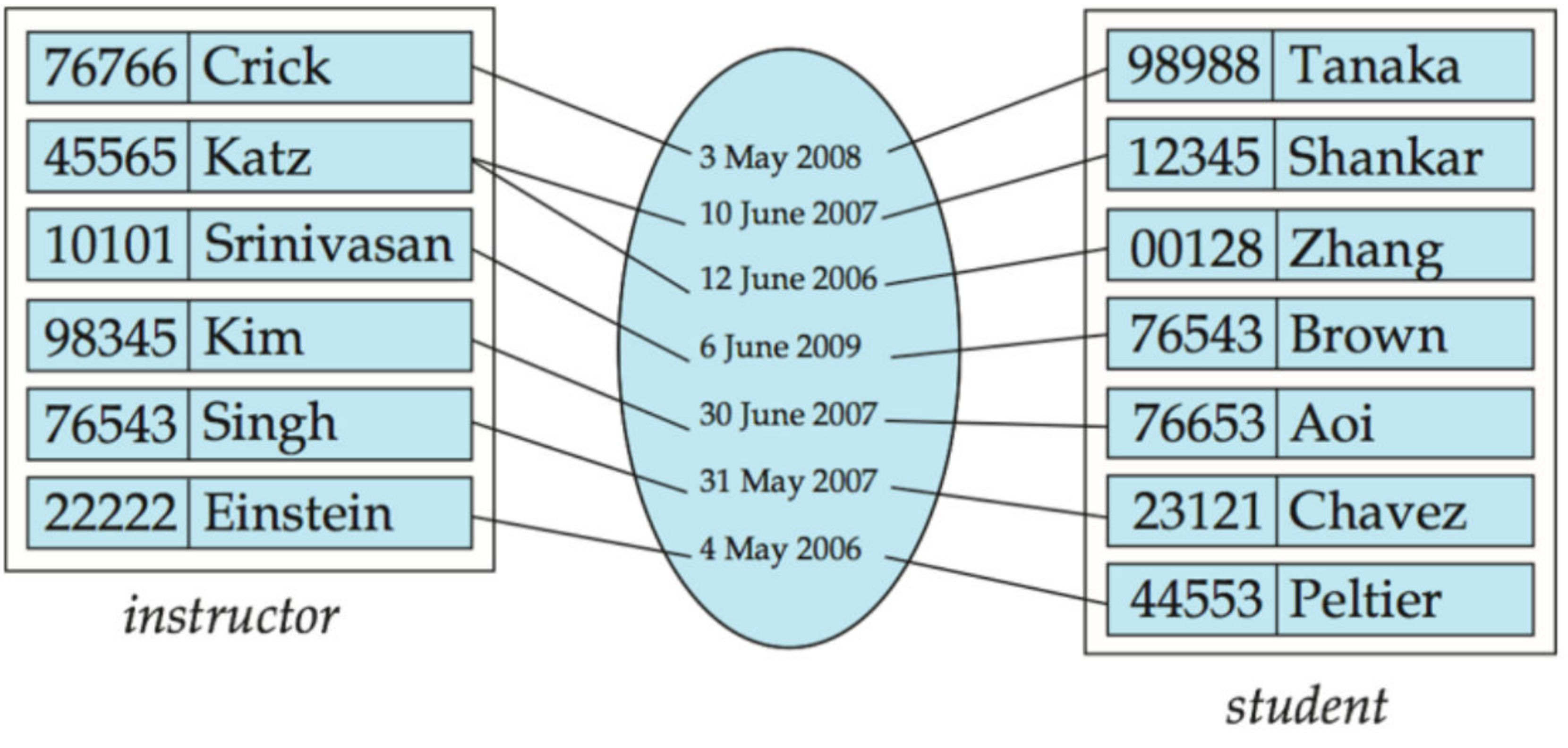
  where $(e_1, e_2, ..., e_n)$ is a relationship

- **Example:** $(44553, 22222) \in advisor$

Relationship set: advisor



- 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



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

Attributes: Redundant

- Suppose we have entity sets:

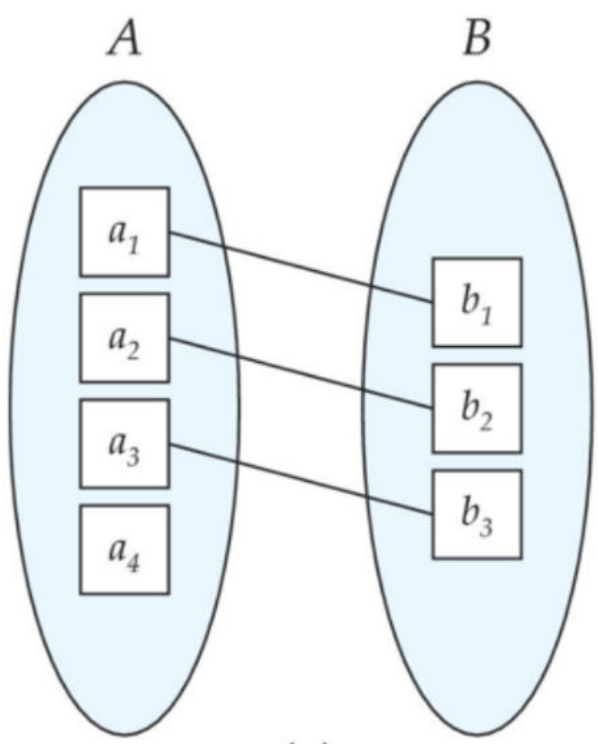
- *instructors*, with attributes: *ID, name, dept_name, salary*
- *department*, with attributes: *dept_name, building, budget*

- We model the fact that each instructor has an associated department using a relationship set *inst_dept*
- The attribute *dept_name* appears in both entity sets
    - Since it is the primary key for the entity set department, it replicates information present in the relationship and is therefore redundant in the entity set instructor and needs to be removed
- **BUT:** When converting back to tables, in some cases the attributes gets re-introduced, as we will see later

Mapping Cardinality: Constraints

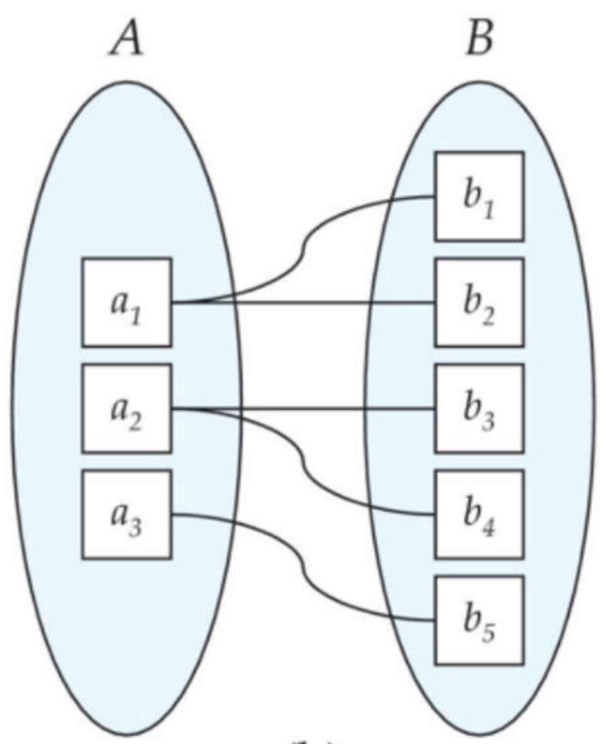
- Express the number of entities to which another entity can be associated via a relationship set
- Most useful in describing binary relationship sets
- For a binary relationship set the mapping cardinality must be one of the following types:
    - One to One
    - One to Many
    - Many to One
    -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

Mapping Cardinalities



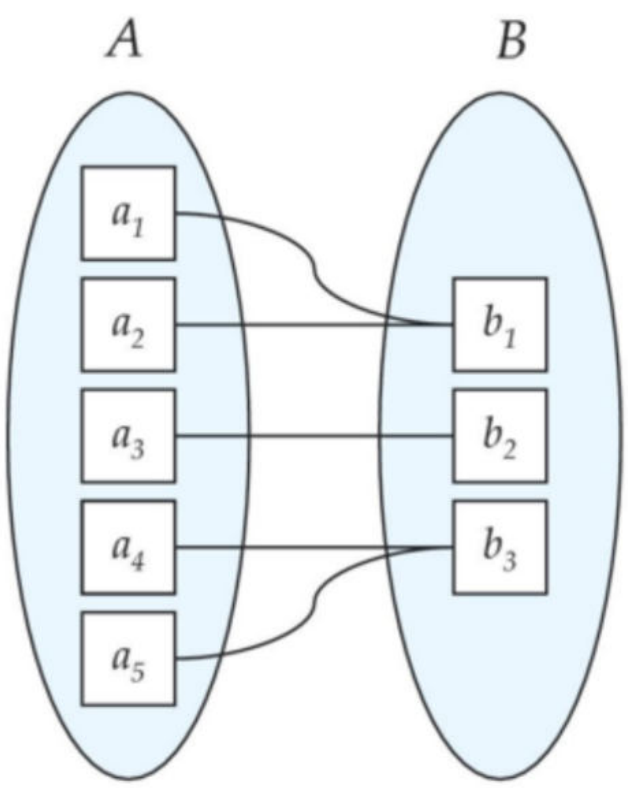
(a)

One to one



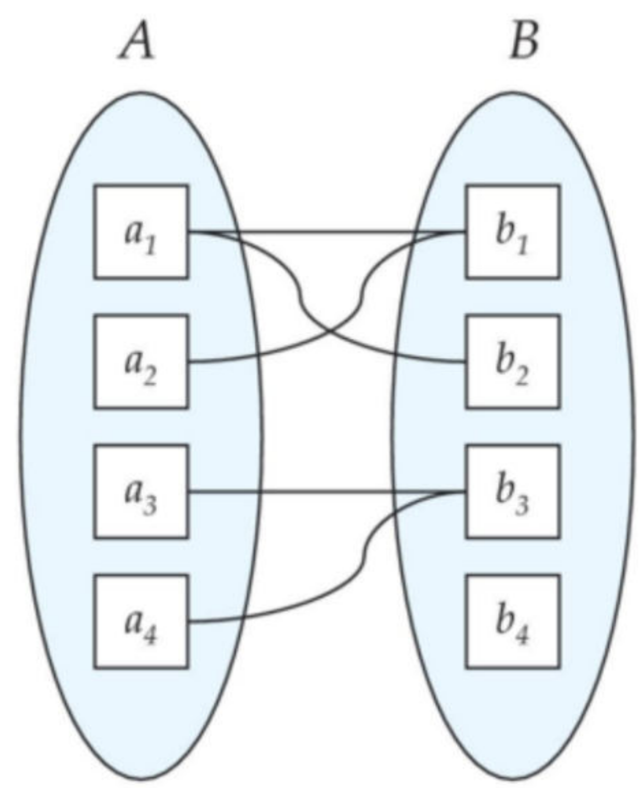
(b)

One to many



(a)

Many to one



(b)

Many to many

NOTE: Some elements in A and B may not be mapped to any elements in the other set

Weak Entity sets

An entity set may be one of the two types:

- Strong entity set
    - A strong entity set is an entity set that contains sufficient attributes to uniquely identify all its entities
    - In other words, ***a primary key exists for a strong entity set***
    - Primary key of a strong entity set is represented by underlining it
- Weak entity set
    - A weak entity set is an entity set that does not contain sufficient attributes to uniquely identify its entities
    - In other words, ***a primary key does not exist for a weak entity set***
    - However, it contains a partial key called as the **discriminator**
    - Discriminator can identify a group of entities from the entity set
    - Discriminator is represented by underlining with a dashed line
- Since a weak entity set does not have a primary key, it cannot independently exist in the ER model
- It features in the model in relationship with a strong entity set
    - This is called as **the identifying relationship**
- Primary Key of a Weak entity set
    - The combination of discriminator and primary key of the strong entity set makes it possible to uniquely identify all entities of the weak entity set
    - Thus, this combination serves as a primary key for the weak entity set
    - Clearly, this primary key is not formed by the weak entity set completely
    - **Primary Key of a Weak Entity Set = Its own discriminator + Primary Key of Strong Entity Set**
- Weak entity set must have **total participation** in the identifying relationship
    - That is, all the entities must feature in the relationship

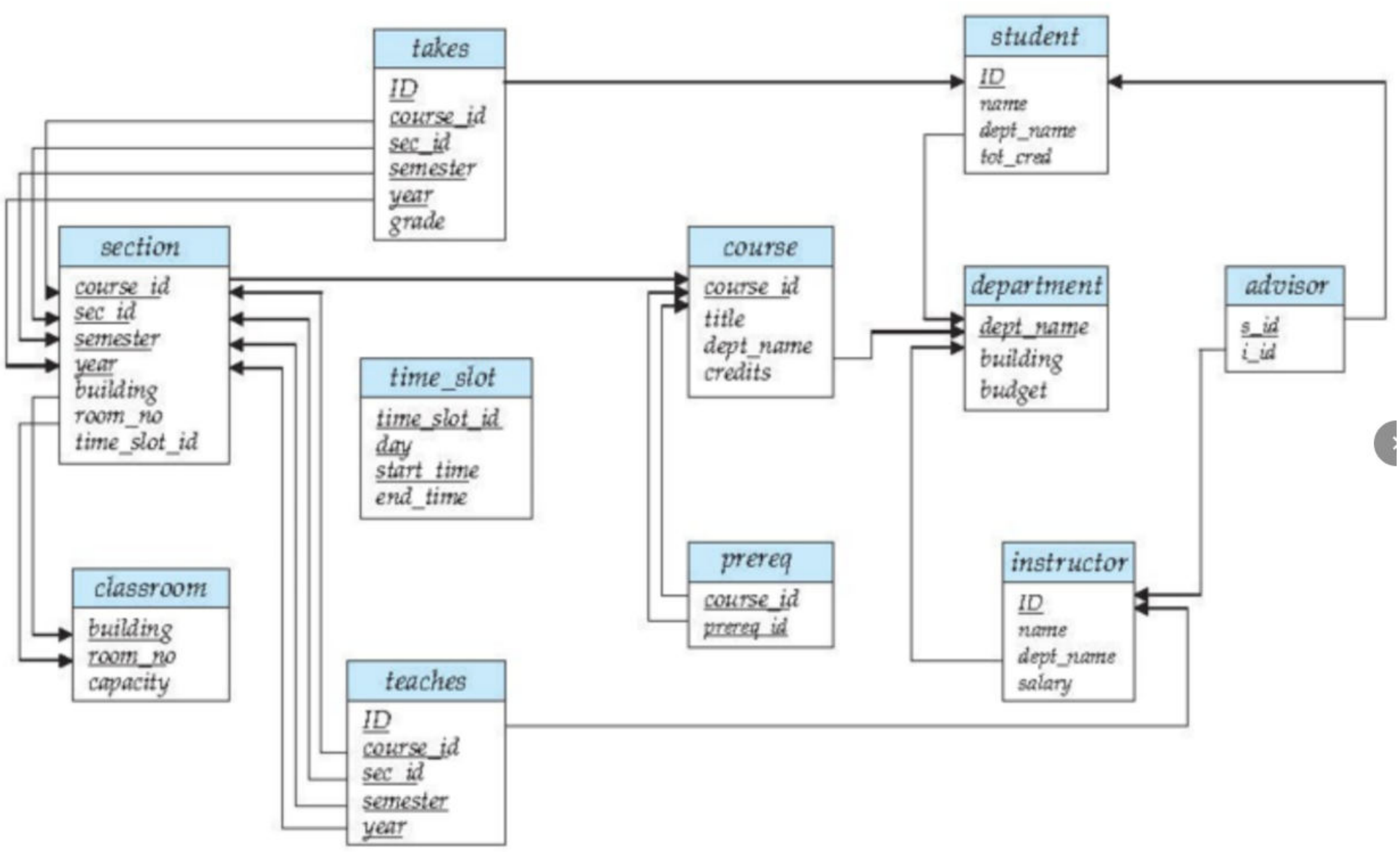
Weak Entity set: Example

- **Strong Entity Set:** *Building*(building_no, buildname, address)

- *building_no* is the primary key here
- **Weak Entity Set:** *Apartment*(door_no, floor)
  - *door_no* is its discriminator as *door_no* alone can not identify an apartment uniquely
  - There may be several other buildings having the same door number
- **Relationship:** *BA* between *Building* and *Apartment*
- By **total participation** in *BA*, each apartment must be present in at least one building
- In contrast, *Building* has **partial participation** in *BA* only as there might exist some buildings which has not apartment
- **Primary Key:** To uniquely identify an apartment
  - First, *building_no* is required to identify the particular building
  - Second, *door_no* of the apartment is required to uniquely identify the apartment
- Primary Key of Apartment = Primary Key of the Building + Its own discriminator = *building_no* + *door_no*

Weak Entity set: Example #2

- 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



Week 4 Lecture 4

Class	BSCCS2001
Created	@September 30, 2021 6:29 PM
Materials	
Module #	19
Type	Lecture
Week #	4

Entity-Relationship Model (part 2)

ER Diagram

Entity Sets

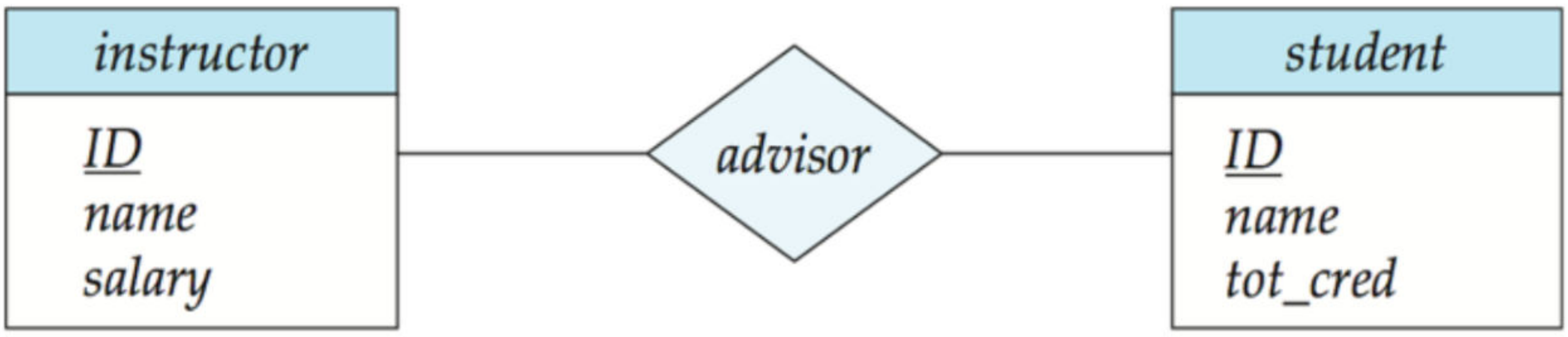
- Entities can be represented graphically as follows:
    - Rectangles represent entity set
    - Attributes are listed inside entity rectangle
    - Underline indicates primary key attributes

instructor
ID
name
salary

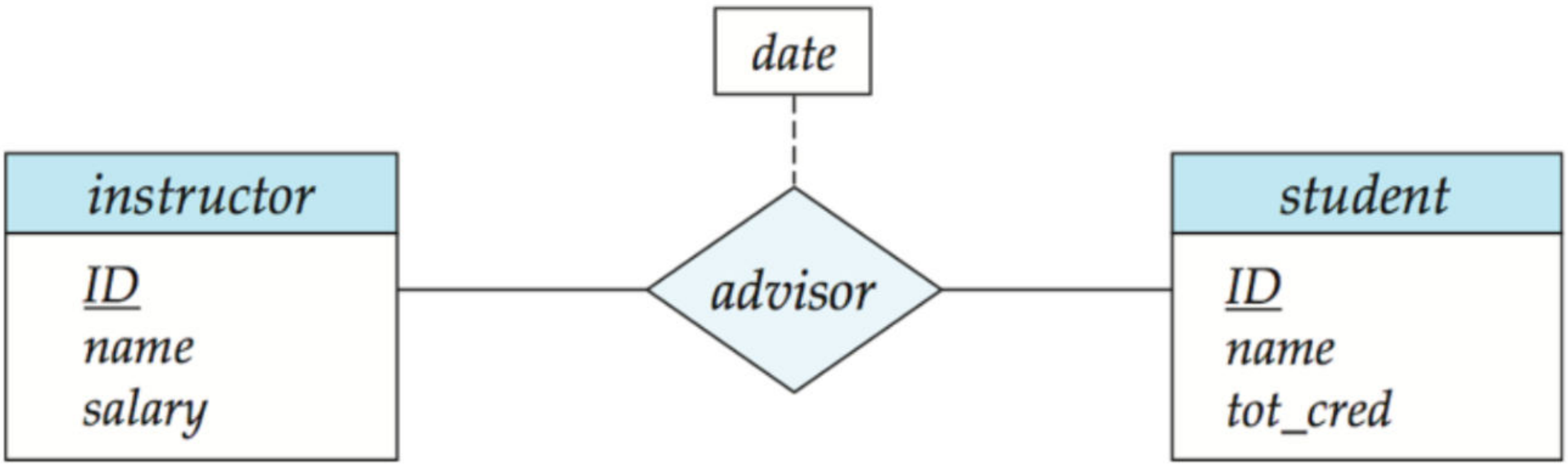
student
ID
name
tot_cred

Relationship sets

- Diamonds represent relationship sets

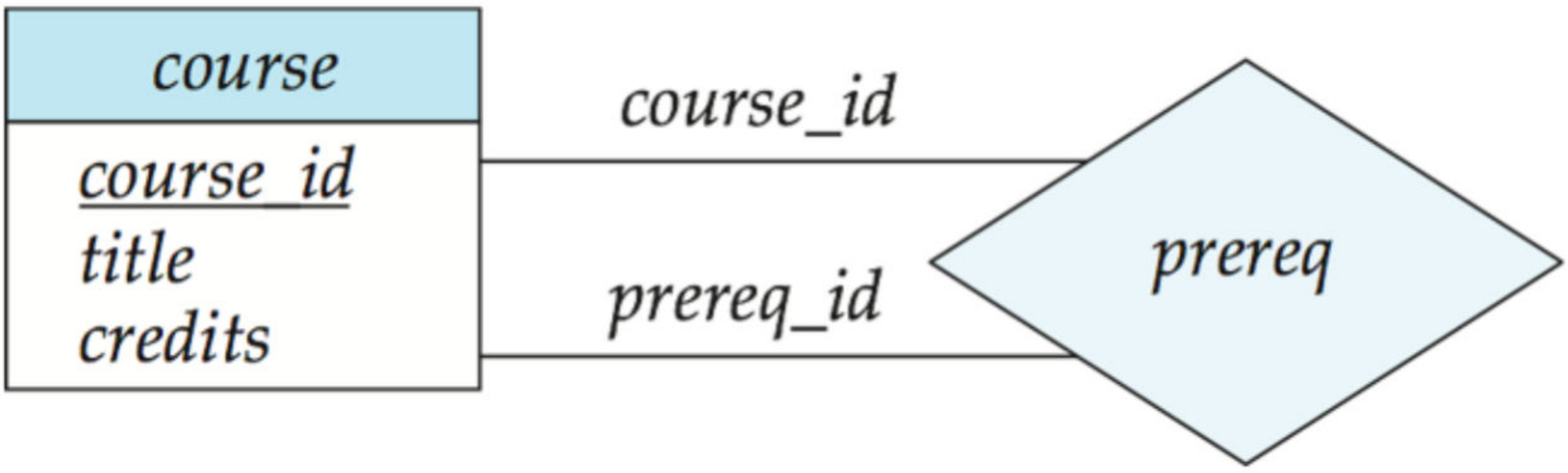


Relationship sets with attributes



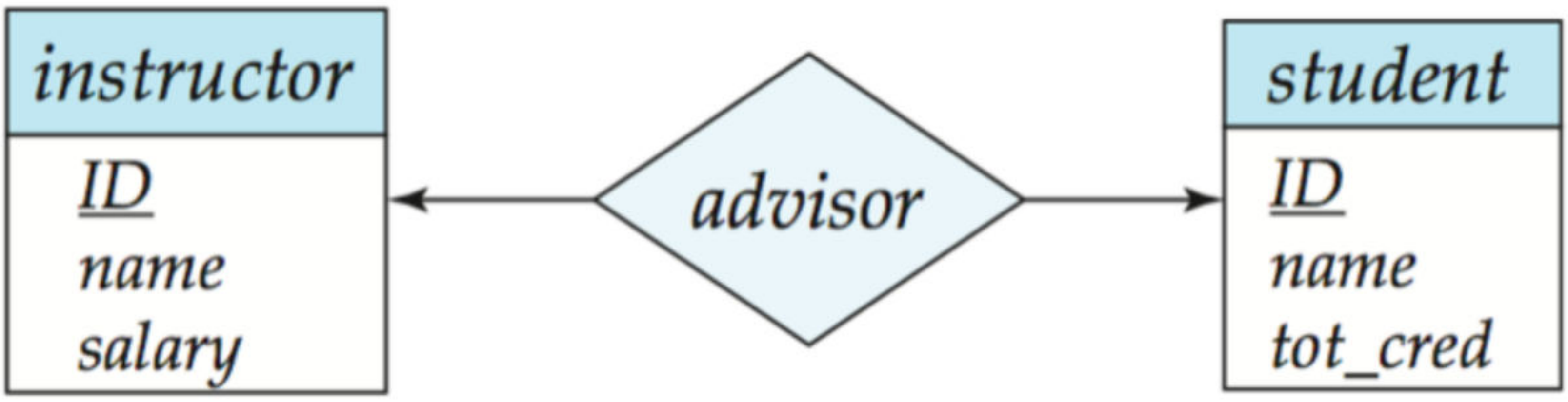
Roles

- Entity sets of 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**



Cardinality Constraints

- 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*
  - An instructor is associated with at most one student via the relationship *advisor*

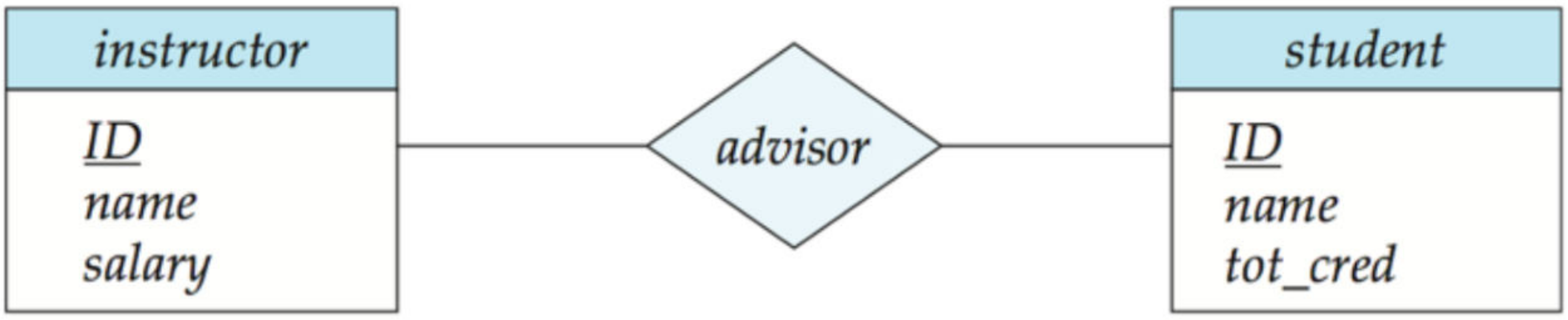


One-to-Many relationship

- 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*

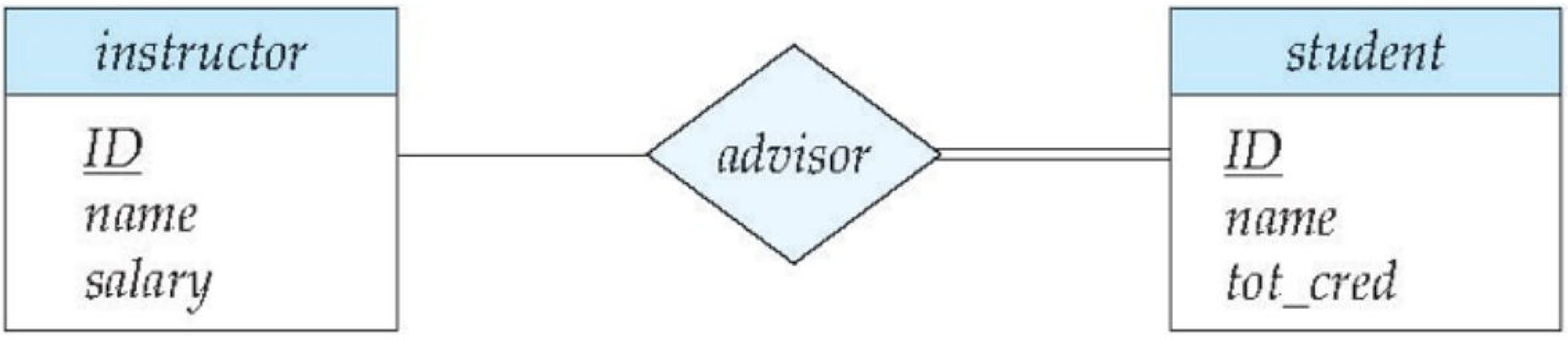
Many-to-Many relationship

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



Total and Partial participation

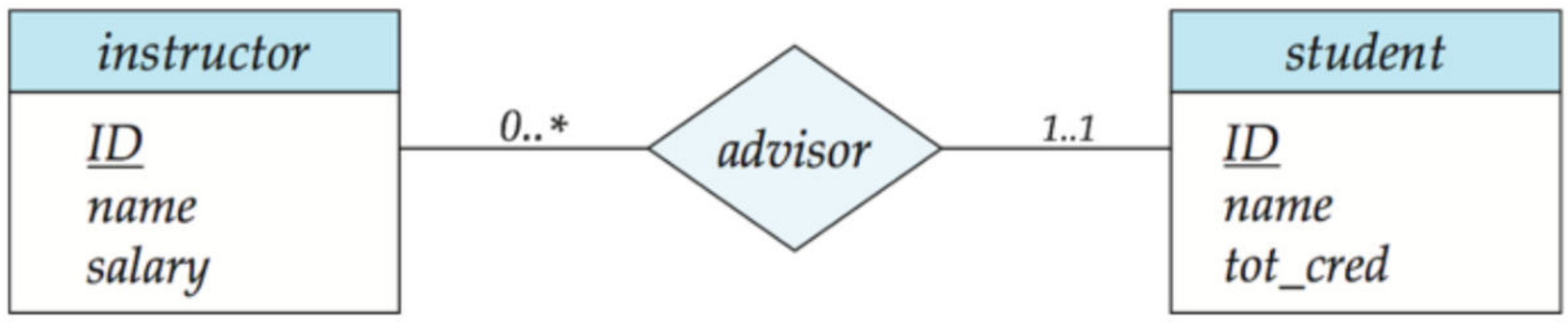
- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set



    - 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** is the maximum cardinality
    - A minimum value of 1 indicates total participation
    - A maximum value of 1 indicates that the entity participation in at most one relationship
    - A maximum value of ∗ indicates no limit



Instructor can advise 0 or more students

A student must have 1 advisor; cannot have multiple advisors

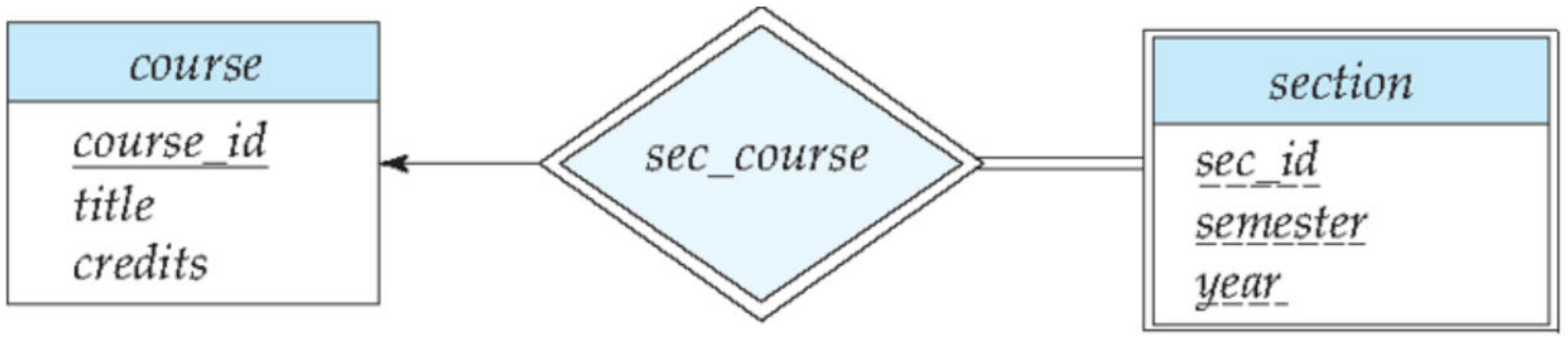
Notation to express entity with complex attributes

$instructor$

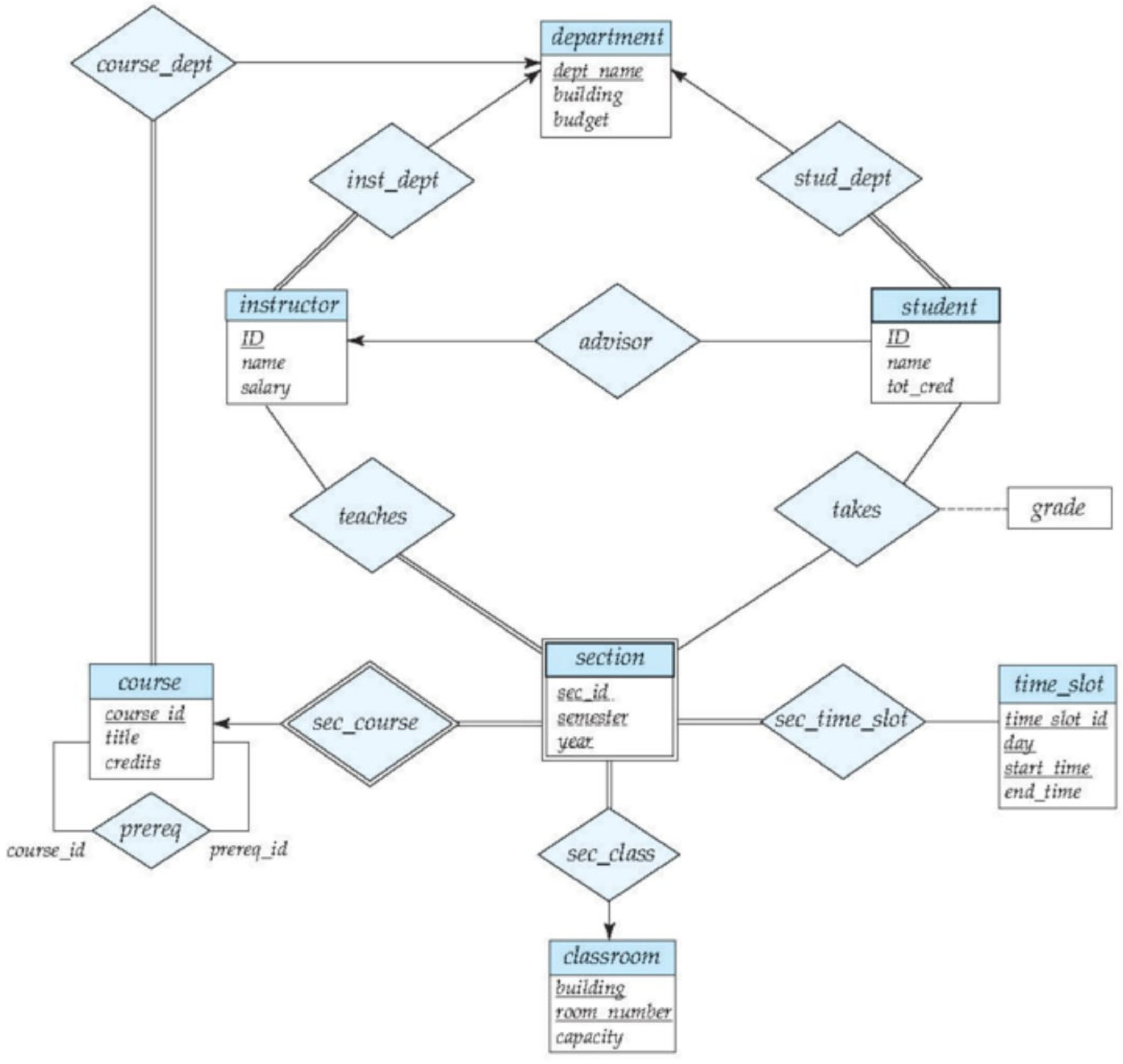
```
ID
name
  first_name
  middle_initial
  last_name
address
  street
    street_number
    street_name
    apt_number
  city
  state
  zip
{ phone_number }
date_of_birth
age()
```

Expressing Weak entity sets

- In ER 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*)



ER diagram for a University enterprise



ER Model to Relational Schema

Reduction to Relation Schema

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

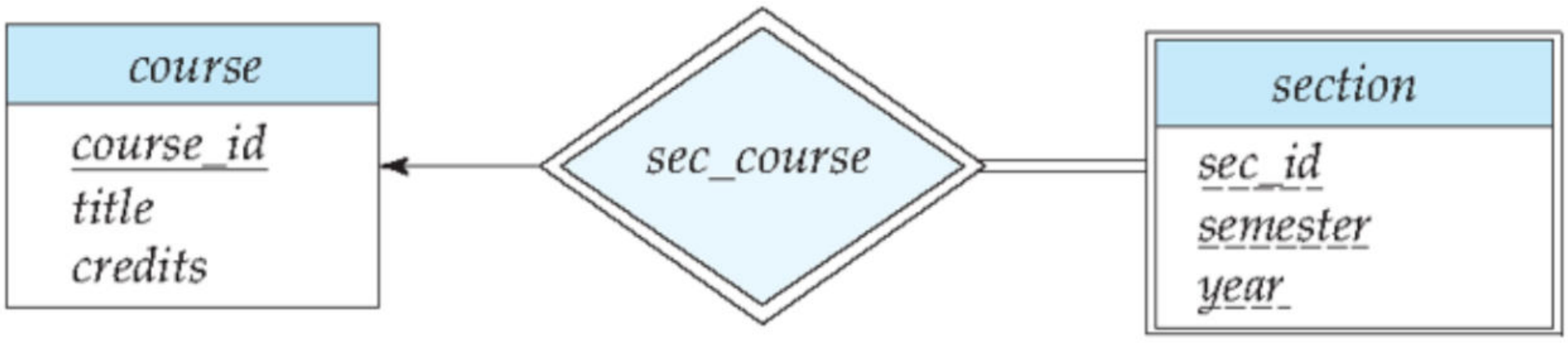
Representing entity sets

- A strong entity set reduces to a schema with the same attributes

  student (ID, name, tot_cred)

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

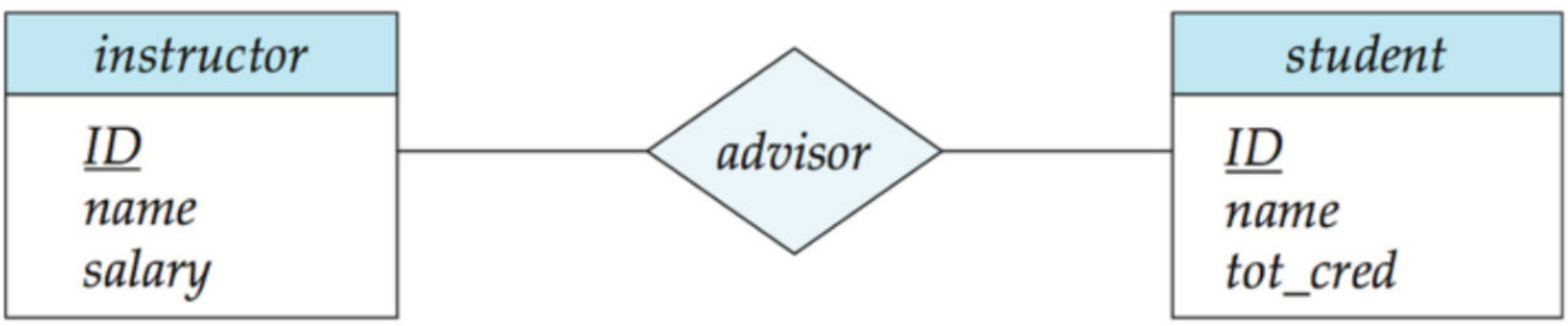
  section (course_id, sec_id, sem, year)



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)*



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 simply be **first_name**)
- Ignoring multi-valued 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)
```

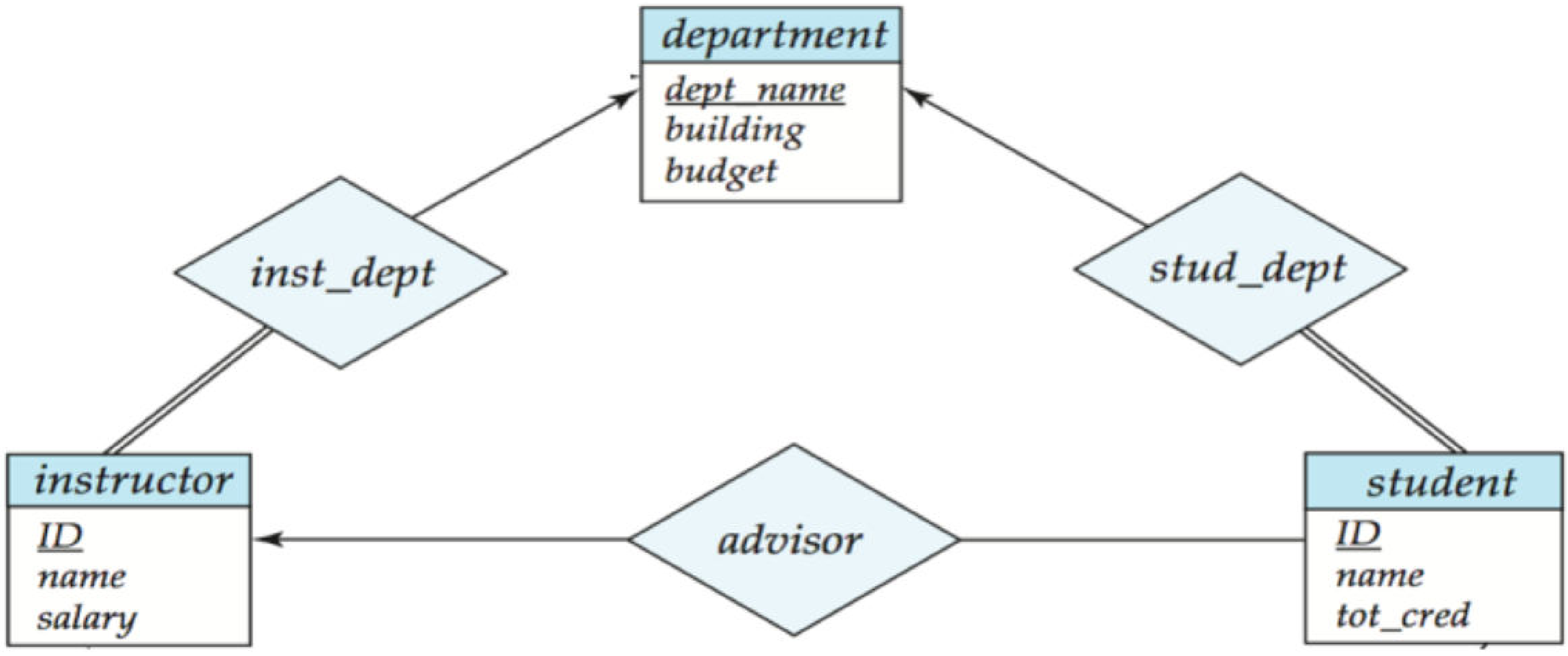
Representation of Entity sets with multi-valued attributes

- A multi-valued 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 multi-valued attribute M
- **Example:** Multi-valued attribute phone_number of instructor is represented by a schema:

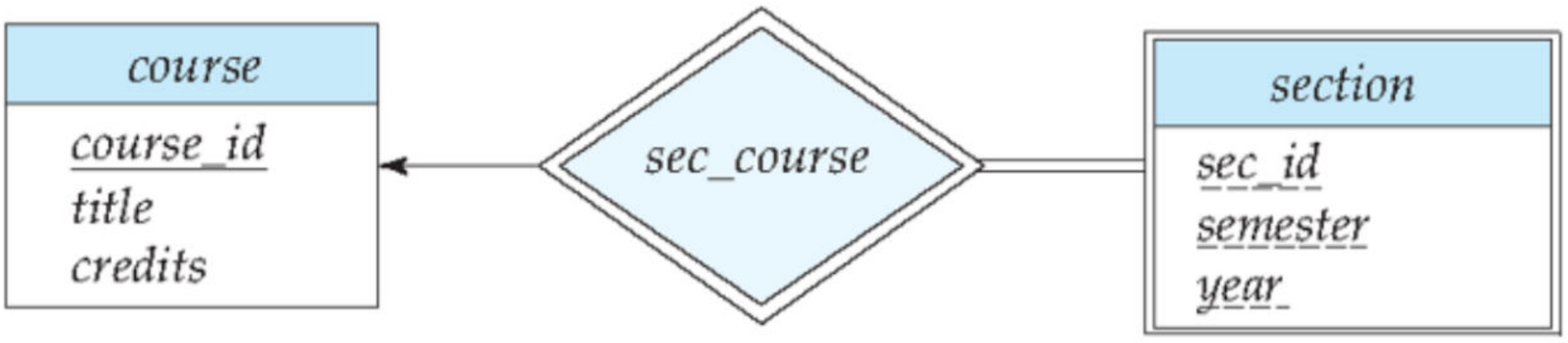
  inst_phone = (*ID*, *phone_number*)
- Each value of the multi-valued attribute maps to a separate tuple of the relation on schema EM
  - **For example:** an instructor entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples: (22222, 456-7890) and (22222, 123-4567)

Redundancy of the Schema

- 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*



- 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



Week 4 Lecture 5

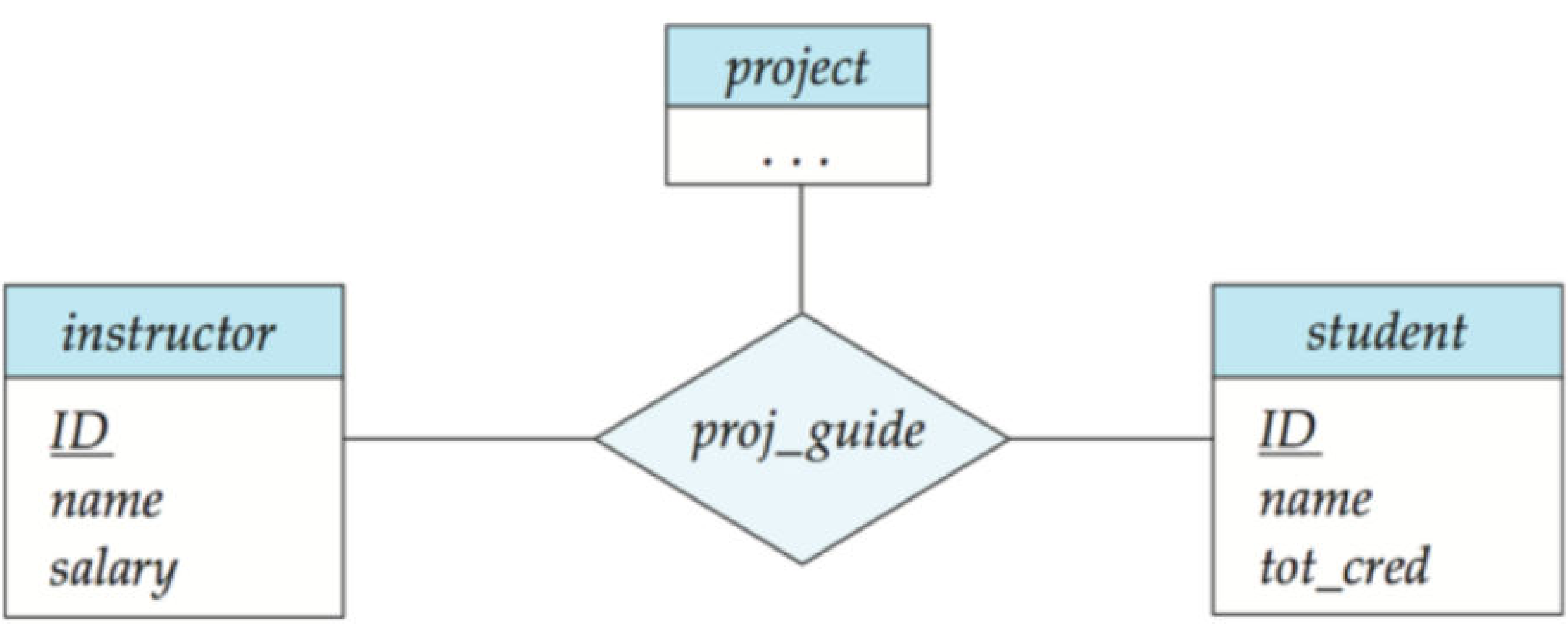
Class	BSCCS2001
Created	@September 30, 2021 8:37 PM
Materials	
Module #	20
Type	Lecture
Week #	4

Entity-Relationship Model (part 3)

Extended ER features

Non-binary Relationship sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary
- ER diagram with a Ternary Relationship



Cardinality constraints on Ternary Relationship

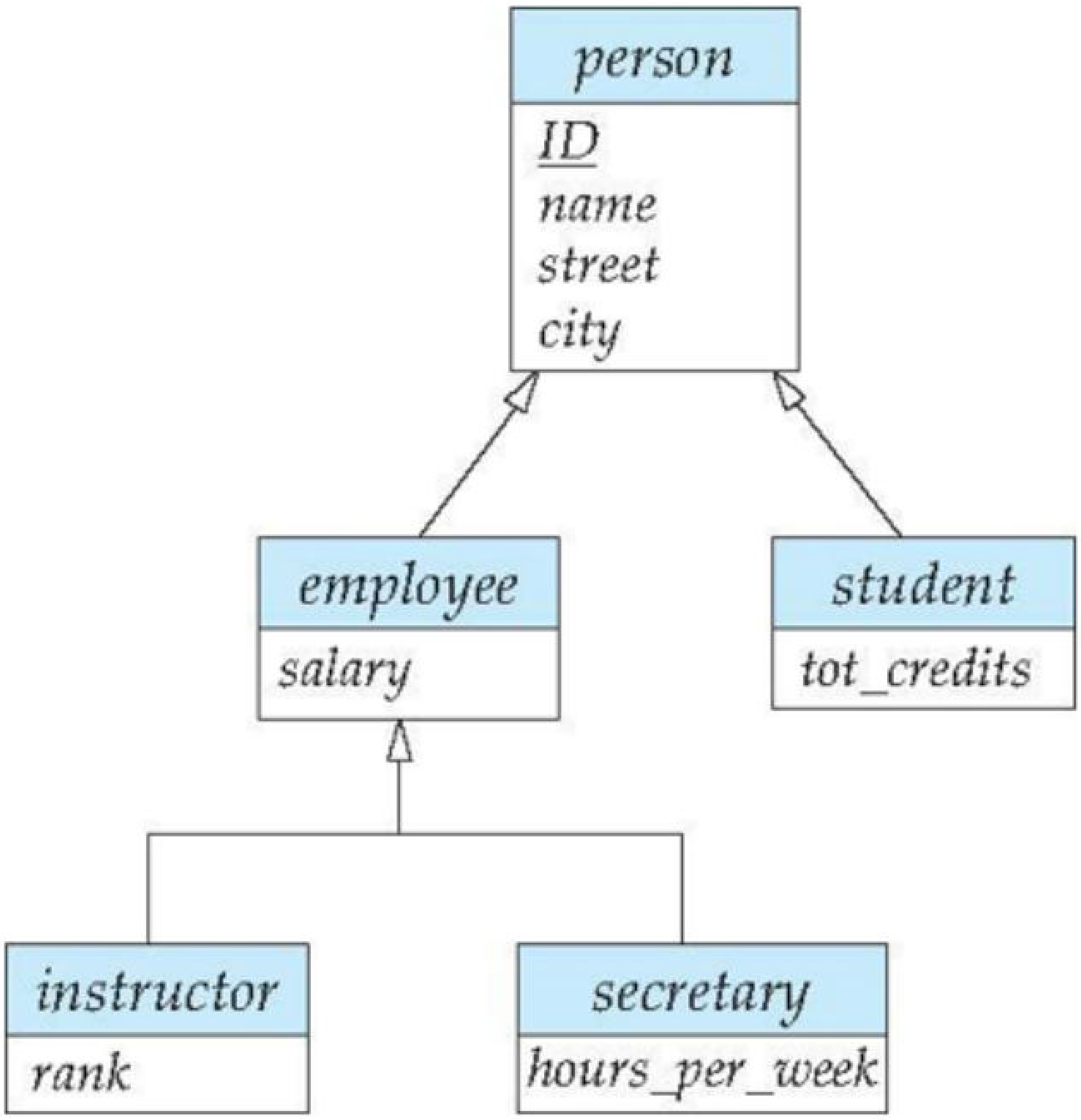
- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- For example, an arrow from *proj_guide* to *instructor* indicates each student has at most one guide for a project
- If there is more than one arrow, there are two ways of defining the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

Specialization: ISA

- **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 leveled ISA (eg: 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

---

- **Overlapping:** *employee* and *student*
- **Disjoint:** *instructor* and *secretary*
- Total and Partial



Representing Specialization via Schema

- 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

Name	Tags
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

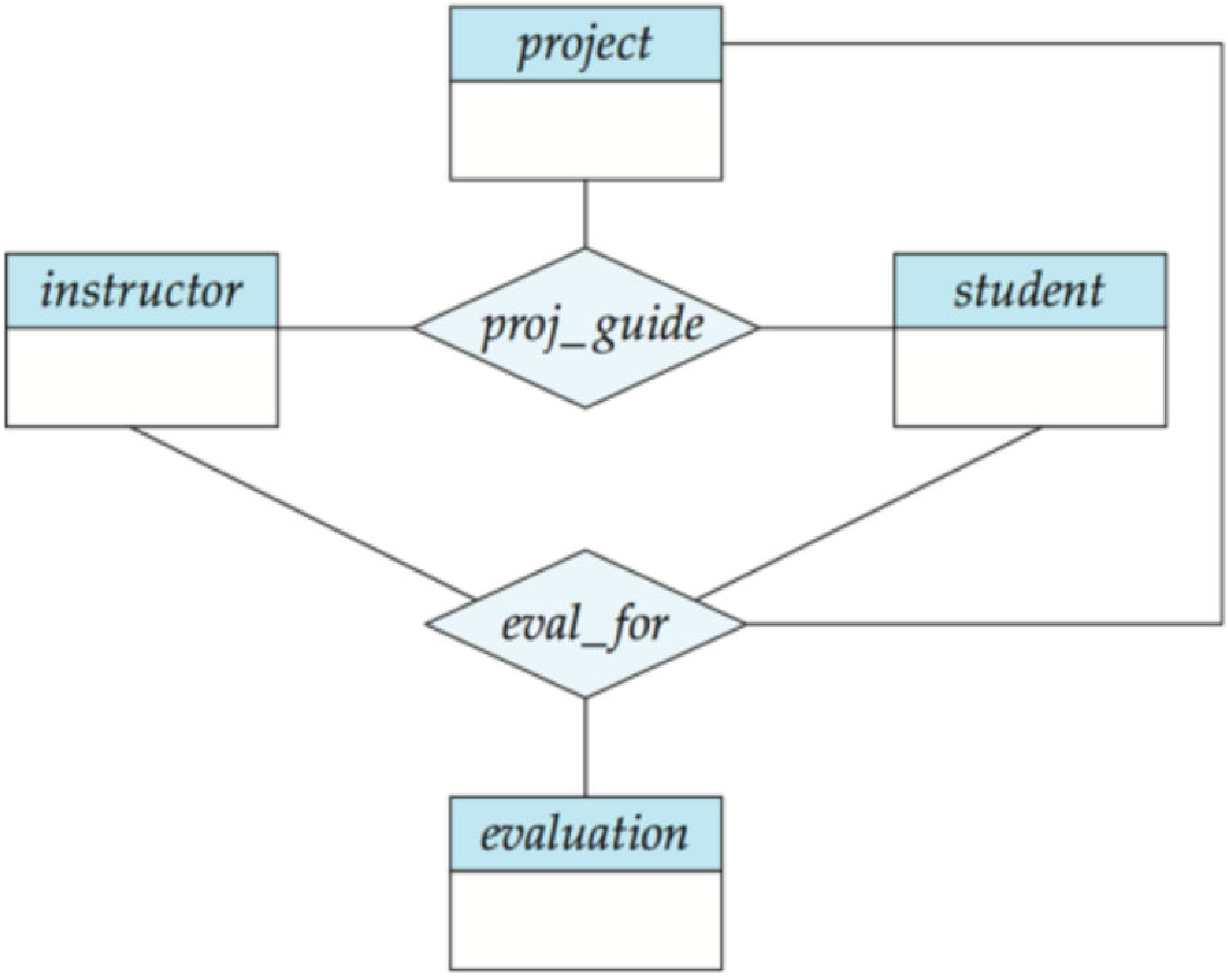
- **Bottom-up design process:** Combine a number of entity sets that share the same features into a higher-level entity set
- Specialization and generalization are simple inversions of each other; they are represented in an ER diagram in the same way
- The terms specialization and generalization are used interchangeably

Design constraints on a specialization / generalization

- **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
  - 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

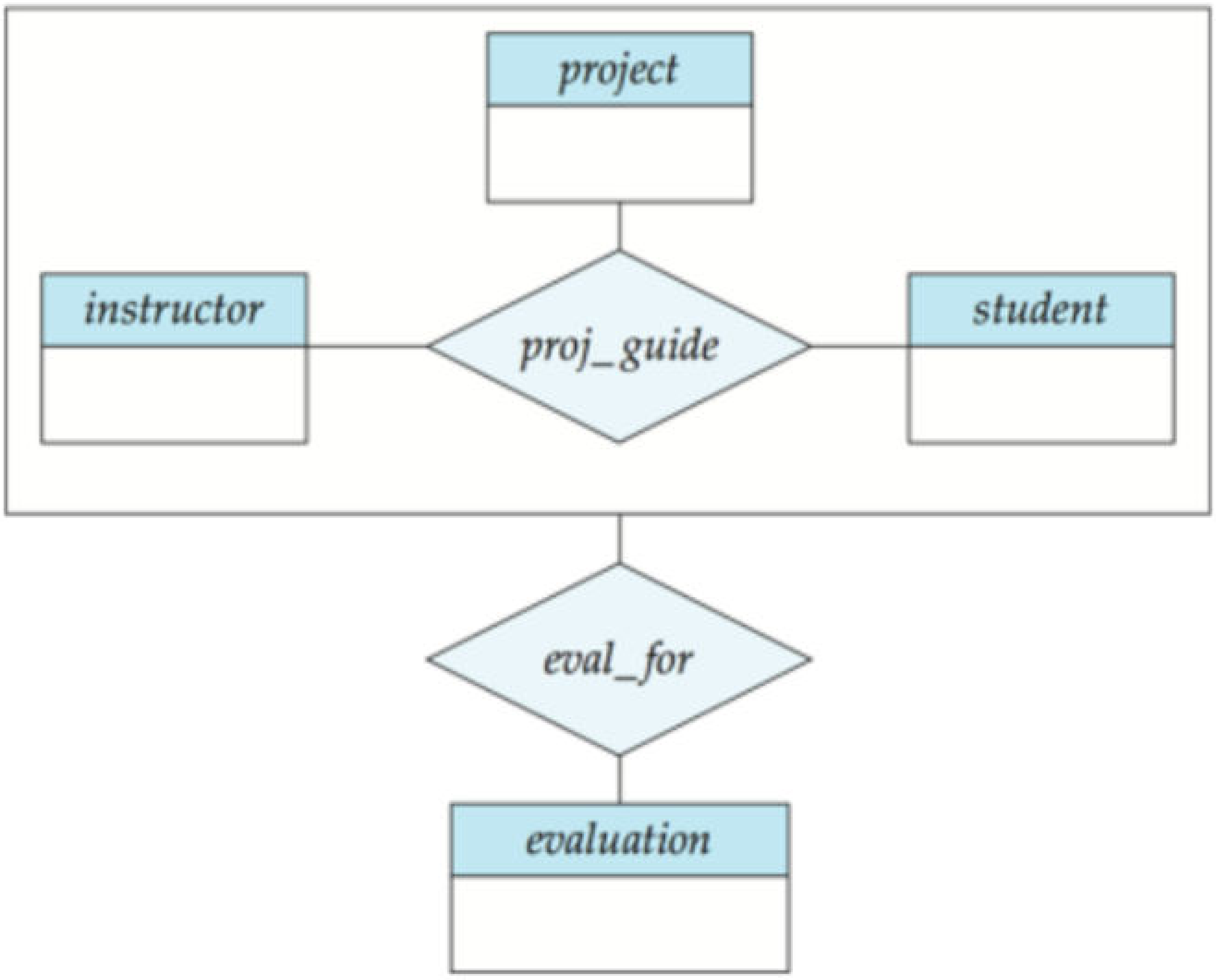
Aggregation

- 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 cannot 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



Representing aggregation via Schema

- 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

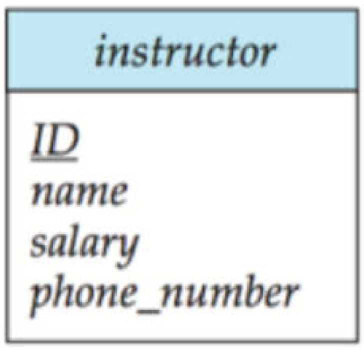
    textiteval_for is:

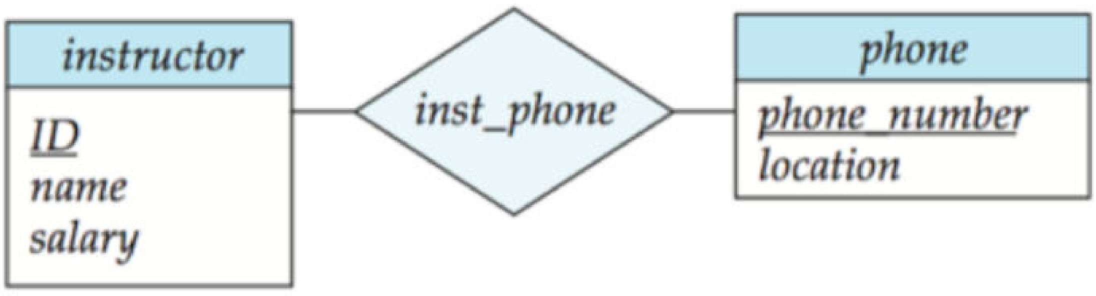
    eval_for (s_ID, project_id, i_ID, evaluation_id)
  - The schema *proj_guide* is redundant

Design Issues

Entities v/s Attributes

- Use of entity sets v/s attributes



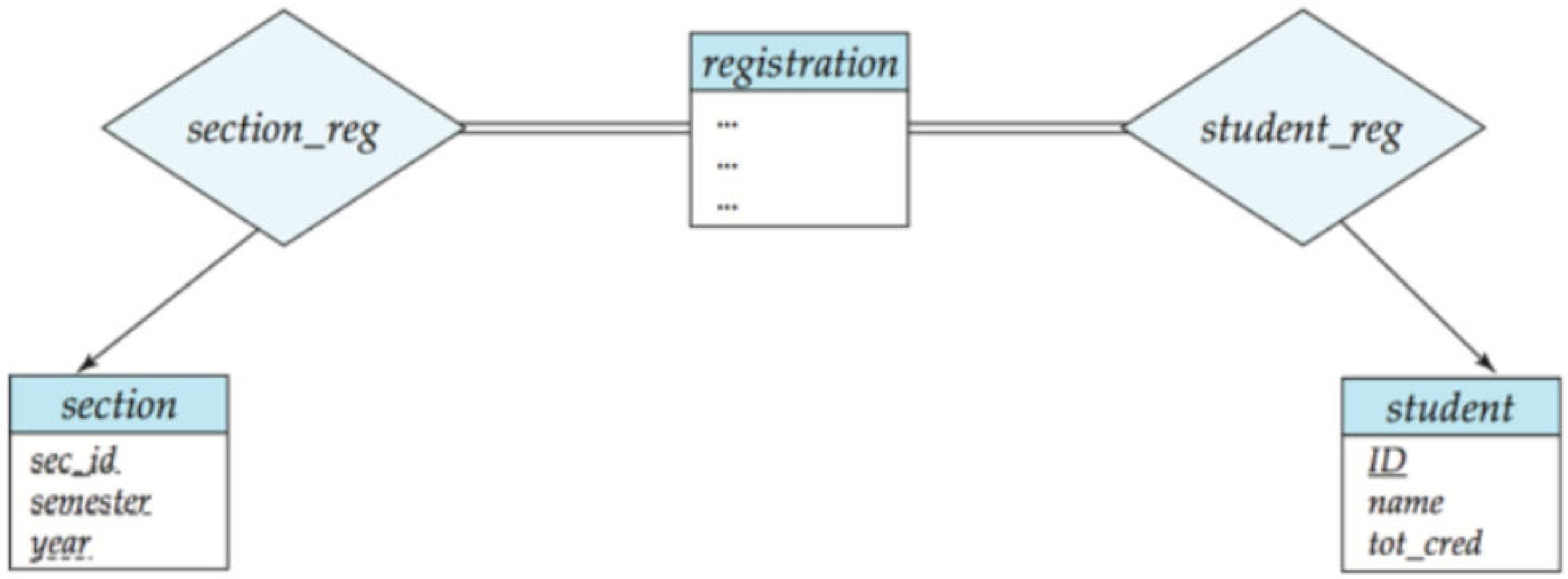


- Use of phone as an entity allows extra information about phone numbers (plus multiple phone numbers)

Entities v/s Relationship sets

- **Use of entity sets v/s relationship sets**

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



- **Placement of relationship attributes**

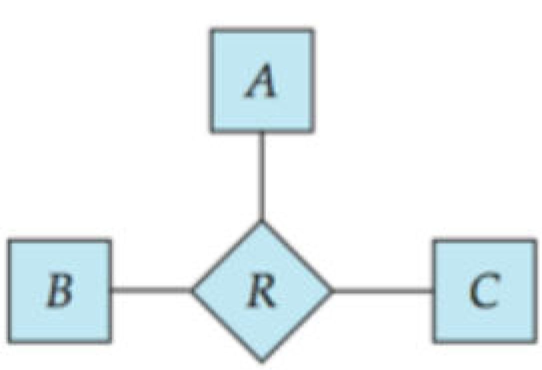
  For example, attribute date as attribute of advisor or as attribute of student

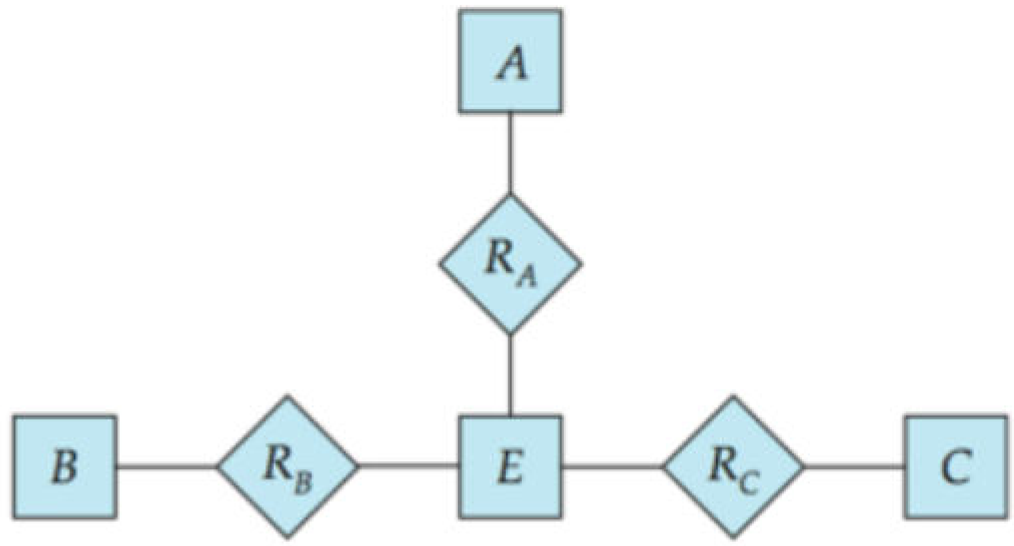
Binary v/s Non-binary Relationships

- Although, it is possible to replace any non-binary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, an 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 (eg: only mother being known)
  - But there are some relationships that are naturally non-binary
    - Example: proj_guide

Binary v/s Non-binary Relationships: Conversion

- 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:
    - R_A, relating E and A
    - R_B, relating E and B
    - 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
    - A new entity e_i in the entity set E
    - add (e_i, a_i) to R_A
    - add (e_i, b_i) to R_B
    - add (e_i, c_i) to R_C



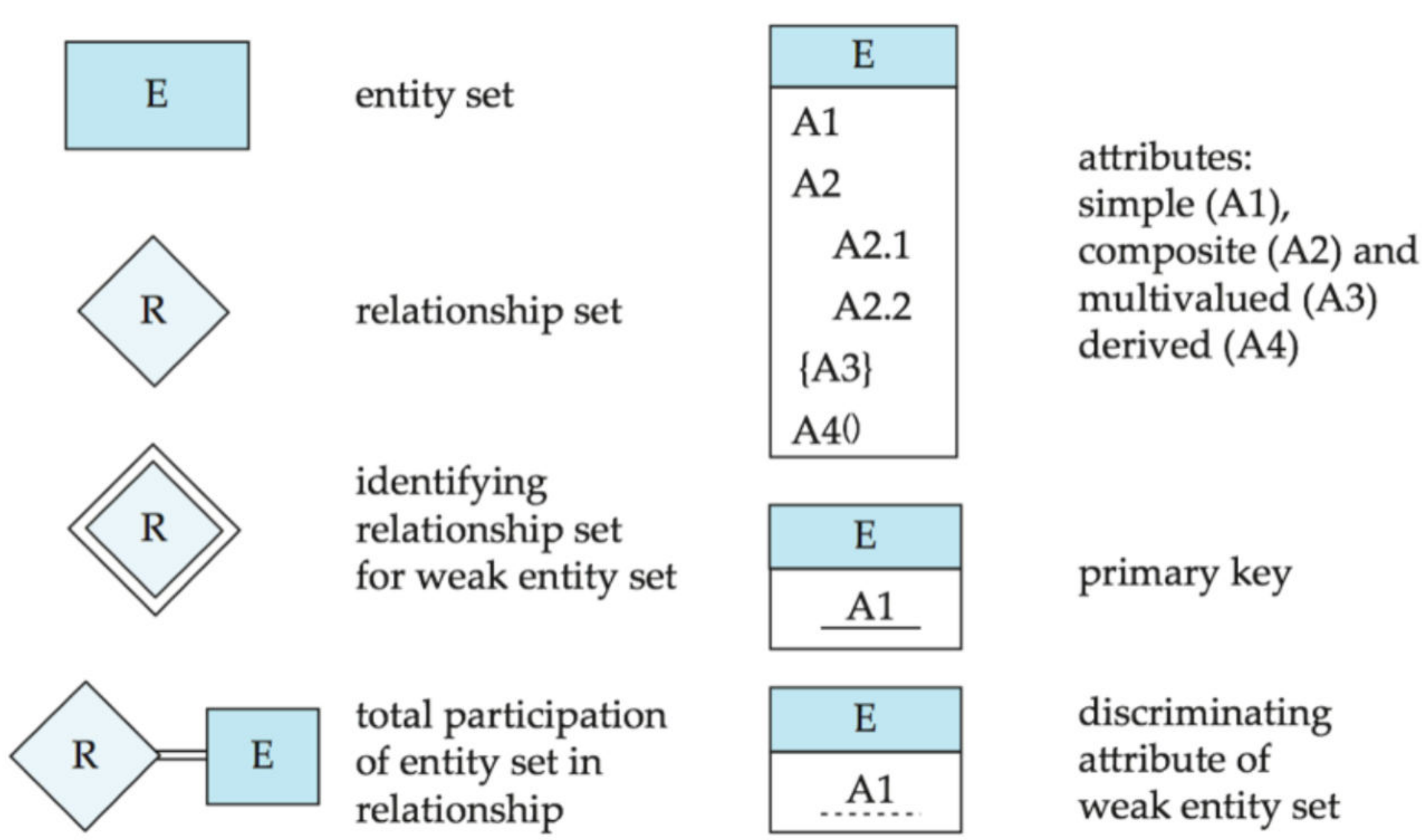


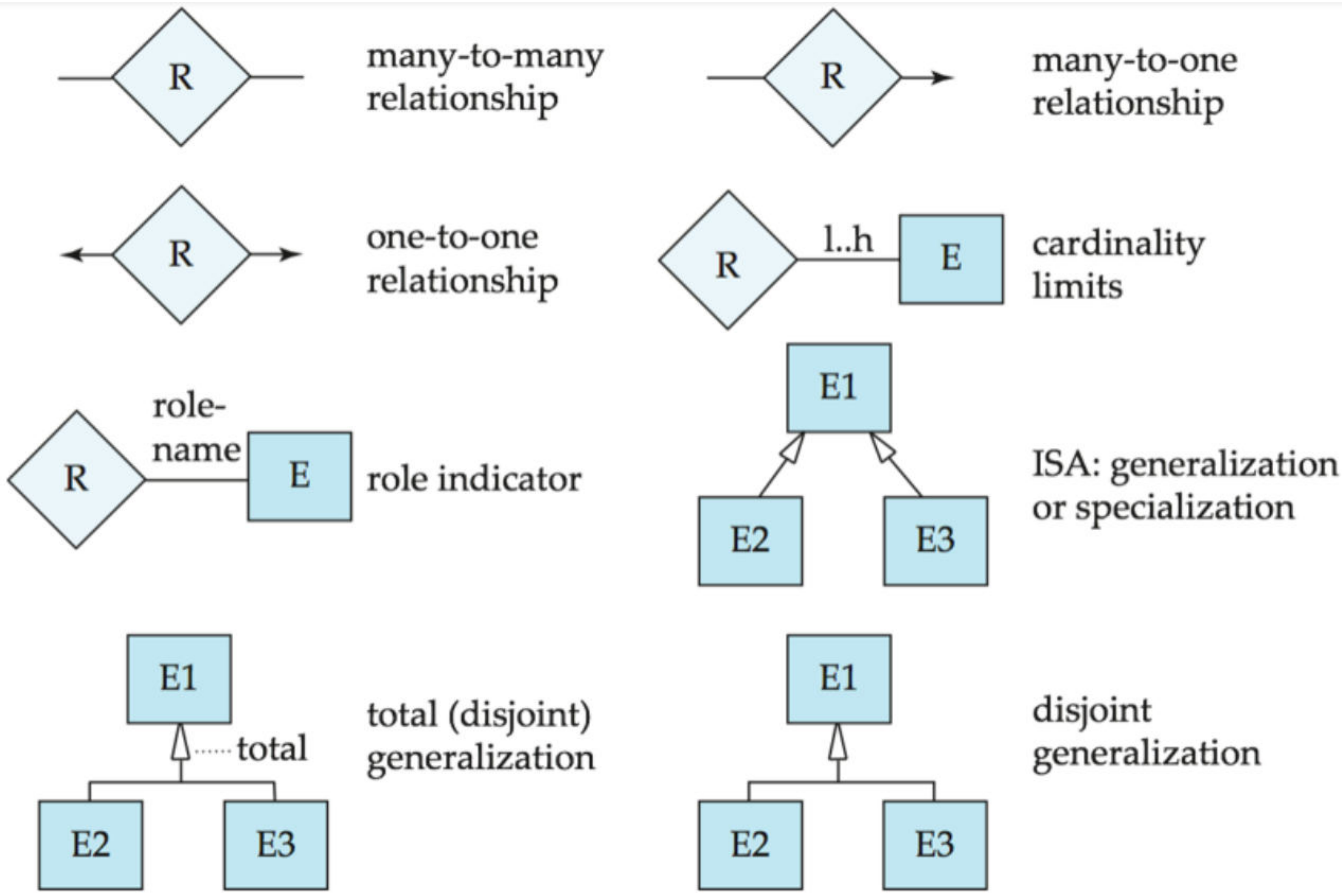
- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instance in the translated schema that cannot correspond to any instance of R
    - **Exercise:** add constraints to the relationships $R_A, R_B\ and\ R_C$ to ensure that a newly created entity corresponds 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 identified by the three relationship sets

ER Design Decisions

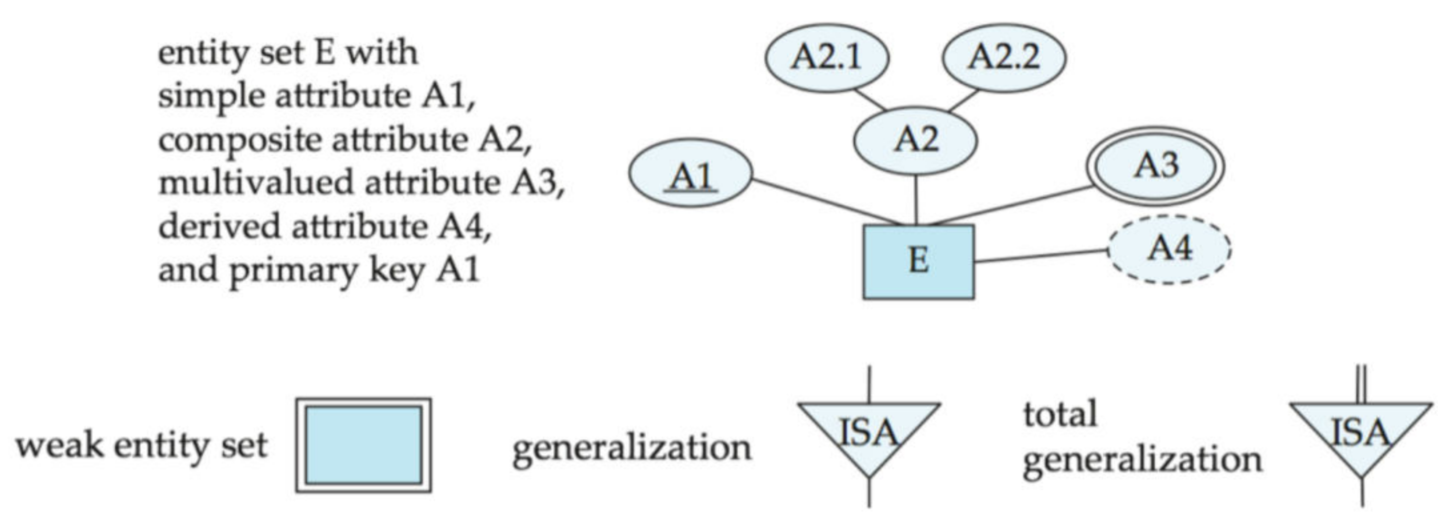
- The use of an attribute or entity set to represent an object
- Whether a real-world concept is best expressed by an entity or a relationship set
- The use of a ternary relationship versus a pair of binary relationships
- The use of strong or weak entity set
- The use of specialization/generalization — contributes to modularity in the design
- The use of aggregation — can treat the aggregate entity set as a single unit without concern for the details of its internal structure

Symbols used in the ER Notation

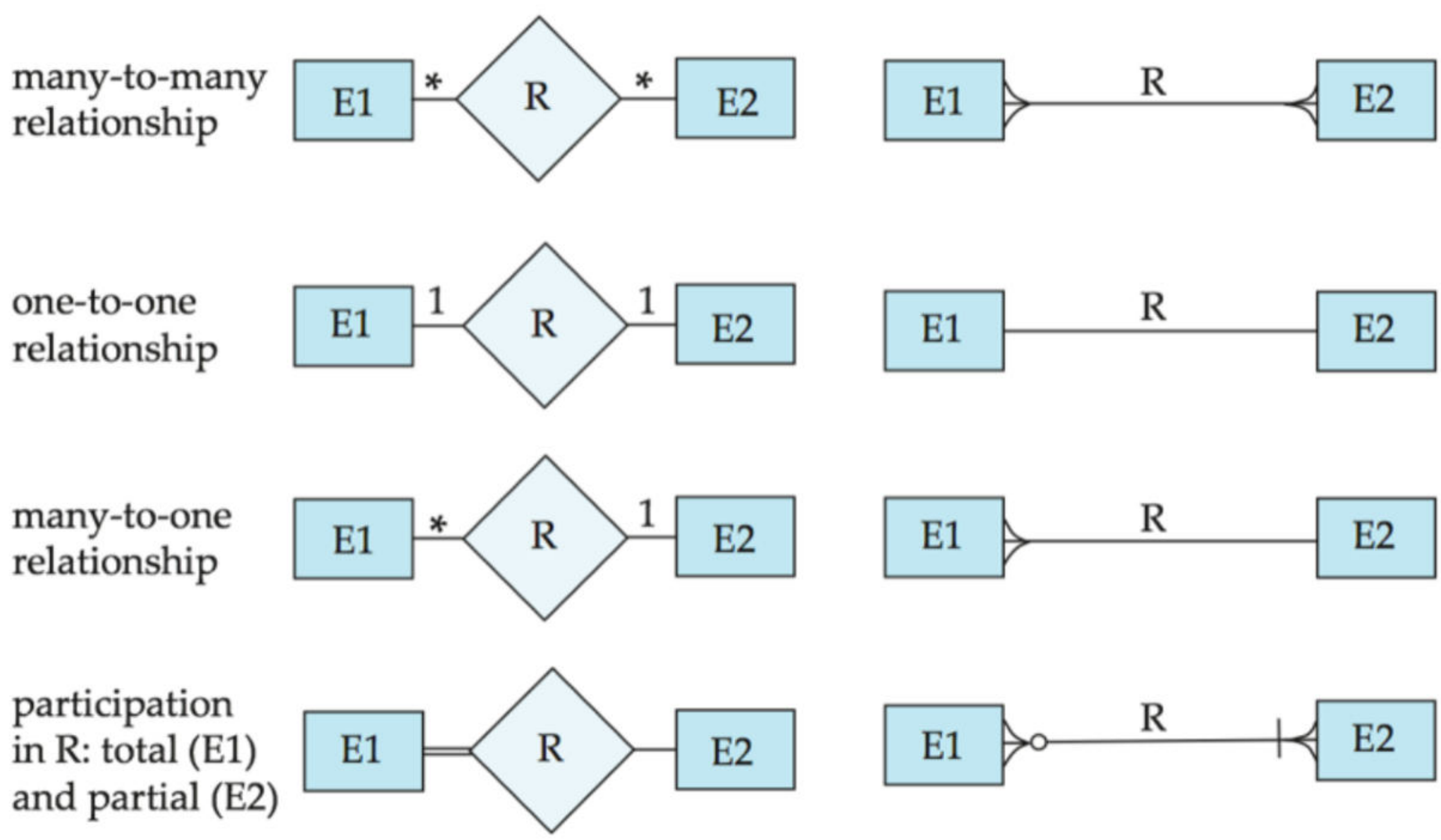



R
many-to-many relationship
R
many-to-one relationship
R
one-to-one relationship
R
l..h
E
cardinality limits
R
role-name
E
role indicator
E1
E2
E3
ISA: generalization or specialization
E1
total
E2
E3
total (disjoint) generalization
E1
E2
E3
disjoint generalization

- Chen, IDE1FX,...


entity set E with simple attribute A1, composite attribute A2, multivalued attribute A3, derived attribute A4, and primary key A1
A2.1
A2.2
A2
A1
A3
E
A4
weak entity set
generalization
ISA
total generalization
ISA

Chen **IDE1FX (Crows feet notation)**



Week 5 Lecture 1

Class	BSCCS2001
Created	@October 4, 2021 11:55 AM
Materials	
Module #	21
Type	Lecture
Week #	5

Relational Database Design

Features of Good Relational Design

Good Relational Design

- Reflects real-world structure of the problem
- Can represent all expected data over time
- Avoids redundant storage of data over time
- Provides efficient access to data
- Supports the maintenance of data integrity over time
- Clean, consistent and easy to understand
- **NOTE:** These objectives are sometimes contradictory ඞ

What is a good schema?

instructor_with_department

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

- *ID*: Key
- *building*, *budget*: Redundant Information
- *name*, *salary*, *dept_name*: No Redundant Information

instructor

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

department

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

Database Management Systems Partha Pratim Das

- Consider combining relations
  - *sec_class(<u>sec_id</u>, building, room_number)* and
  - *section(<u>course_id</u>, <u>sec_id</u>, semester, year)*
- No repetition in this case

Redundancy and Anomaly

- **Redundancy:** Having multiple copies of the same data in the DB
  - This problem arises when a DB is not normalized
  - It leads to anomalies
- **Anomaly:** Inconsistencies that can arise due to data changes in a database with insertion, deletion and update
  - These problems occur in poorly planned, un-normalized DBs where all the data is stored in one table (a flat-file DB)

  There can be 3 kinds of anomalies
  - Insertions anomaly
  - Deletion anomaly
  - Update anomaly
- **Insertions anomaly**
  - When the insertion of a data record is not possible without adding some additional unrelated data to the record
  - We cannot add an Instructor in *instructor_with_department* if the *department* does not have a *building* or *budget*
- **Deletion anomaly**
  - When deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table
  - We delete the last Instructor of a Department from *instructor_with_department,* we lose *building* and *budget* information
- **Update anomaly**
  - When a data is changed, which could involve many records having to be changed, leading to the possibility of some changes being made incorrectly
  - When the *budget* changes for a Department having a large number of Instructors in *instructor_with_department* application may miss some of them

- We have observed the following:
  - **Redundancy** $\Rightarrow$ **Anomaly**
  - Relations *instructor* and *department* is better than *instructor_with_department*
- What causes redundancy?
  - **Dependency** $\Rightarrow$ **Redundancy**
  - *dept_name* uniquely decides *building* and *budget*
  - A department cannot have two different budget or building
  - So, *building* and *budget* **depends on** *dept_name*
- How to remove, or at least minimize, redundancy?
  - Decompose (partition) the relation into smaller relations
  - *instructor_with_department* can be decomposed into *instructor* and *department*
  - **Good Decomposition** $\Rightarrow$ **Minimization of Dependency**
- Is every decomposition good?
  - No
  - It needs to preserve information, honor the dependencies, be efficient, etc
  - Various schemes of normalization ensure good decomposition
  - **Normalization** $\Rightarrow$ **Good decomposition**

Decomposition

- Suppose we had started with *inst_dept*
- How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**: *dept_name* $\rightarrow$ *building, budget*
- In *inst_dept*, because *dept_name* is not a candidate key, the *building* and *budget* of a *department* may have to repeated
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good
- Suppose we decompose

  employee(<u>ID</u>, name, street, city, salary) into

    employee1 (<u>ID</u>, name)

    employee2 (name, street, city, salary)
- Note that if *name* can be duplicate, then *employee2* is a weak entity set and cannot exist without an identifying relationship
- Consequently, this decomposition cannot preserve the information
- The next slide shows how we lose information — we cannot reconstruct the *original employee* relation — and so, this is a **lossy decomposition**

Decomposition: Lossy Decomposition

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

employee

Duplicate names cause the problem here.

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
⋮				

Decomposition: Lossless-join Decomposition

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

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

A	B	C
α	1	A
β	2	B

r

R_1

A	B
α	1
β	2

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

R_2

B	C
1	A
2	B

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

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

A	B	C
α	1	A
β	2	B

R(A, B, C, D, E, F)

$R_1(r_1)$ $R_2(r_2)$

① $R_1 \cup R_2 = R$

② $R_1 \cap R_2 \neq \phi$ (null set)

- **Lossless Join Decomposition** is a decomposition of a relation R into relations R_1, R_2 such that if we perform natural join of two smaller relations it will return the original relation

  $R_1 \cup R_2 = R, R_1 \cap R_2 \neq \phi$

  $\forall r \in R, r_1 = \sqcap_{R_1}(r), r_2 = \sqcap_{R_2}(r)$

  $r_1 \bowtie r_2 = r$

- This is effective in removing the redundancy from DBs while preserving the original data

③ 1 common constraint should be Prim/candidate key in either R_1/R_2
[in this case, B is PK of R_2]

- In other words, by lossless decomposition it becomes feasible to reconstruct the relation R from decomposed tables R_1 and R_2 by using Joins

Atomic Domains and First Normal Form

First Normal Form (1NF)

- A domain is atomic if its elements are considered to be indivisible units
    - Examples of non-atomic domains:
        - Set of names, composite attributes
        - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in **First Normal Form (1NF)** if
    - the domains of all attributes of R are **atomic**
    - the value of each attribute contains only a single value from that domain
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
    - **Example:** Set of accounts stored with each customer, and set of owners stored with each account
    - ***We assume all relations are in the first normal form***
- **Atomicity** is actually a property of how the elements of the domain are used
    - Strings would normally be considered indivisible
    - Suppose that students are given roll numbers which are strings of the form CS0012 or EE1127
    - If the first two characters are extracted to find the department, the domain of the roll numbers is not atomic
    - *Doing so is a bad idea ...*
        - It leads to encoding of the information in application program rather than in the database
- The following is not in 1NF

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

- A telephone number is composite
- Telephone number is multi-valued

- Consider:

Customer

Customer ID	First Name	Surname	Telephone Number1	Telephone Number2
123	Pooja	Singh	555-861-2025	192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53	182-929-2929
789	John	Doe	555-808-9633	

- is in 1NF if telephone number is not considered composite
- However, conceptually, we have two attributes for the same concept
    - Arbitrary and meaningless ordering of the attributes

        - How to search telephone numbers
        - Why only two numbers?

- Is the following in 1NF?

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025
123	Pooja	Singh	192-122-1111
456	San	Zhang	182-929-2929
456	San	Zhang	(555) 403-1659 Ext. 53
789	John	Doe	555-808-9633

    - Duplicated information
    - ID is no more the key
        - Key is (ID, Telephone Number)

- Better to have 2 relations:

Customer Name

Customer ID	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

Customer Telephone Number

Customer ID	Telephone Number
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

    - One-to-Many relationship between parent and child relations
    - Incidentally, satisfies 2NF and 3NF

- Decomposition helps to attain 1NF for the embedded one-to-many relationship

Week 5 Lecture 2

Class	BSCCS2001
Created	@October 4, 2021 2:41 PM
Materials	
Module #	22
Type	Lecture
Week #	5

Relational Database Design (part 2)

Functional Dependencies

Goal: Devise a theory for good relations

- Decide whether a particular relation R is in "good" form
- In the case that a relation R is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ such that
  - each relation is in good form
  - the decomposition is a lossless-join decomposition
- The theory is based on:
  - Functional dependencies
  - Multi-valued dependencies
  - Other dependencies

Functional Dependencies

- Constraints on the set of legal relations
- Require that the values 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

---

- Let R be a relation schema

  $\alpha \subseteq R$ and $\beta \subseteq R$

- The **functional dependency** or **FD**

  $\alpha \to \beta$ ⇒ α determine β

  holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 on r agree on the attributes α, they also agree on the attributes β

  Should be true for all such tuples having $t_1[\alpha] = t_2[\alpha]$

  That is:

  1 1 $\not\Rightarrow$ 4 $\neq$ 5

  $t_1[\alpha] = t_2[\alpha] \Rightarrow t_2[\beta] = t_2[\beta]$ $\alpha \not\to \beta$

- **Example:** Consider $r(A, B)$ with the following instance of r

  But if we add (2,4) then even $B \not\to A$ bcoz 2 conflicting vals. (4 → 1, 4 → 2)

  But if we add (1,4) to table & remove (2,4). $B \to A$ bcoz for equal B vals (4), corresponding A vals r also equal (1) (1,4), (1,4)

  Every unique A value (1), doesn't uniquely identify B (1→4, 1→5), so $A \not\to B$. But $B \to A$, as every unique B value can uniquely identify A vals (4→1, 5→1, 7→3)

  This should be true for all such tuples

	A	B
t_1	1	4
t_2	1	5
	3	7
	1	4
	2	4

- On this instance, $A \to B$ does **NOT** hold, but $B \to A$ does hold
- So, we cannot have tuples like $(2, 4)$ or $(3, 5)$ or $(4, 7)$ added to the current instance

---

- K is a superkey for relation schema R if and only if $K \to R$
- K is a candidate key for R if and only if
  - $K \to R$ and
  - for no $\alpha \subset K, \alpha \to R$

  There should be no α (belonging) to K which also determines R uniquely

  & If the entries r unique on the left hand side, the func^n dependency will always hold

- Functional dependencies allows us to express constraints that cannot be expressed using superkeys
- Consider the schema:

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

  R(A, B, C, D, E) K{A, B, C}

  ① ABC → ABCDE ⇒ K → R

  ② AB → R (if AB is candidate key)

  2.1) $A \not\to R$

  2.2) $B \not\to R$

- We expect these functional dependencies to hold:

  dept_name $\to$ *building*

  dept_name $\to$ *budget*

  ID $\to$ *budget*

  but would NOT expect the following to hold:

  dept_name $\to$ *salary*

- We use functional dependencies to:
  - test relations to see if they are legal under a given set of functional dependencies
    - If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F
  - 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 $\to$ *ID*
  - In such cases, we do not say that F holds on R

- A functional dependency is trivial if it is satisfied by all instance of the relation
  - Example:
    - *ID, name* $\to$ *ID*
    - *name* $\to$ *name*

- In general, $\alpha \to \beta$ is trivial if $\beta \subseteq \alpha$

- Functional dependencies are:

StudentID	Semester	Lecture	TA
1234	6	Numerical Methods	John
1221	4	Numerical Methods	Smith
1234	6	Visual Computing	Bob
1201	2	Numerical Methods	Peter
1201	2	Physics II	Simon

  - *StudentID* $\to$ *Semester*
    StudentID, Lecture $\to$ *TA*
    {StudentID, Lecture} $\to$ *{TA, Semester}*

- Functional dependencies are:

Employee ID	Employee Name	Department ID	Department Name
0001	John Doe	1	Human Resources
0002	Jane Doe	2	Marketing
0003	John Smith	1	Human Resources
0004	Jane Goodall	3	Sales

  - *EmployeeID* $\to$ *EmployeeName*
    EmployeeID $\to$ *DepartmentID*
    DepartmentID $\to$ *DepartmentName*

Functional Dependencies: Armstrong's Axioms

- Given a set of Functional Dependencies F, we can infer new dependencies by the **Armstrong's Axioms:**
  - **Reflexivity:** if $\beta \subseteq \alpha$, then $\alpha \to \beta$
  - **Augmentation:** if $\alpha \to \beta$, then $\gamma\alpha \to \gamma\beta$
  - **Transitivity:** if $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$
- These axioms can be repeatedly applied to generate new FDs and added to F
- A new FD obtained by applying the axioms is said to the **logically implied** by F
- The process of generations of FDs terminate after infinite number of steps and we call it the **Closure Set** F^+ for FDs F
  - This is the set of all FDs logically implied by F
- Clearly, $F \subseteq F^+$
- These axioms are:
  - **Sound** (generate only functional dependencies that actually hold) and

Q) $R(A,B,C,D,E)$, $F = \{A \to BCD, D \to E, C \to D\}$
Lossy / Lossless decomposition?

① $R_1(A,B,C)$, $R_2(B,C,D)$, $R_3(C,D,E)$ Lossless

② $R_1(A,B,C)$, $R_2(A,C,D)$, $R_3(A,D,E)$ Lossless

③ $R_1(A,B,C)$, $R_2(A,C)$, $R_3(A,D)$ Lossy

④ $R_1(A,B,C,D)$, $R_2(A,C,D,E)$ Lossless

① $R_1 \cup R_2 \cup R_3 = ABCDE$ ✓ ② $(R_1 \cap R_2) \cap R_3 \neq \emptyset$ ✓

③ $R_1 \cap R_2 = BC$, $(BC^+) = \{B, C, D\} \Rightarrow BC \to BCD$ (PK in R_2) ✓

$(R_1 \cap R_2) \cap R_3 = (C^+) = \{C, D, E\} \Rightarrow C \to CDE$ (PK in R_3) ✓

$C \to D \to E \Rightarrow C \to \{CDE\}$ $\Rightarrow$ Lossless

② $R_1 \cup R_2 \cup R_3 = A,B,C,D,E$ ✓ $R_1 \cap R_2 \neq \emptyset$, $R_1 \cap R_2 \cap R_3 \neq \emptyset$ ✓

$R_1 \cap R_2 = AC$, $(AC^+) = \{A,C,B,D,E\}$ so $AC \to ACD$ & ABC (so PK in atleast 1 relⁿ ✓)

$(R_1 \cap R_2) \cap R_3 = AD$, $(AD^+) = \{A,D,E,B,C\}$ so $AD \to ABC$ & ADE (—— " —— ✓)

} Lossless

③ $R_1 \cup R_2 \cup R_3 \neq ABCDE$ ✗
$\Rightarrow$ Lossy

④ $R_1 \cup R_2 \cup R_3 = ABCDE$ ✓ $R_1 \cap R_2 = ACD$, $(ACD^+) = \{A,C,D,E,B\}$
$\Rightarrow ACD \to ABCD$ ✓ $\Rightarrow$ Lossless

Q) $R(ABCD)$ $F = \{A \to B, B \to C, C \to D, D \to B\}$ Dependency preserved or not? Preserved

$R_1(A,B)$	$R_2(B,C)$	$R_3(B,D)$
$A \to B$	$B \to C$	$B \to C \to D \Rightarrow B \to D$
	$C \to B$	$B \to D$
	$C \to D \to B \Rightarrow C \to B$	$D \to B$

$A^+ = \{A,B,C,D\}$ $C^+ = \{C,D,B\}$
$B^+ = \{B,C,D\}$ $D^+ = \{D,B\}$

$C \to D$ isn't directly preserved but if u take child table (R_1, R_2, R_3)' dependencies, $C \to D$ can be inferred, so it's preserved
$C \to B$ (from R_2), $B \to D$ (from R_3) $\Rightarrow C \to D$

Q) $R = (A,B,C)$ $F = \{A \to B, B \to C\}$

$R_1(A,B)$	$R_2(A,C)$
$A \to B$	$A \to B \to C \Rightarrow A \to C$
$(B^+)R_1 = \{B\}$	$A \to C$
	$(C^+)R_2 = \{C\}$

$A^+ = \{A,B,C\}$

$B \to C$ isn't preserved directly. Even if u try to infer, u can't infer $B \to C$ cuz B^+ in R_1 only covers B & C^+ in R_2 only covers C, so can't infer $B \to C$ so not preserved

Dependency preserved? NO

  - **Complete** (eventually generate all functional dependencies that hold)
- Prove the axioms from definitions of FDs
- Prove the soundness and completeness of the axioms

Functional Dependencies: Closure of a Set of FDs

- $F = \{A \rightarrow B, B \rightarrow C\}$
- $F^+ = \{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

Week 5 Lecture 3

Class	BSCCS2001
Created	@October 4, 2021 3:28 PM
Materials	
Module #	23
Type	Lecture
Week #	5

Relational Database Design (part 3)

Functional Dependency Theory

Functional Dependencies: Closure of a Set FDs

- $R = (A, B, C, G, H, I)$
  $F = \{A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- 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 $CG \rightarrow I$ with CG to infer $CG \rightarrow CGI$ and augmenting $CG \rightarrow H$ with I to infer $CGI \rightarrow HI$ and then transitivity

Functional Dependencies: Closure of a Set FDs: Computing F^+

- To compute the closure of a set of functional dependencies F :

  $F^+ \leftarrow F$

  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 continued 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

Functional Dependencies: Armstrong's Axioms: Derived Rules

$A \rightarrow B$
$CB \rightarrow D$
$(A \rightarrow B)$
$\Rightarrow CA \rightarrow D$ $(AC \rightarrow D)$

Derived from Transitivity + Augment^n

- Additional Derived Rules:
  - **Union:** if $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds
  - **Decomposition:** if $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds
  - **Pseudotransitivity:** if $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds
- The above rules can be inferred from basic Armstrong's axioms (and hence are not included in the basic set)
  - They can be proven independently too
  - **Reflexivity:** if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ $\Rightarrow$ if β is a subset of α, then $\alpha \rightarrow \beta$
  - **Augmentation:** if $\alpha \rightarrow \beta$, then $\gamma\alpha \rightarrow \gamma\beta$ $\Rightarrow A=2, B=2 \Rightarrow A=B \Rightarrow 3A = 3B \Rightarrow \gamma A = \gamma B,$
  - **Transitivity:** if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ $\quad \gamma A \neq B$ (unless β is a subset of α)
- Prove the rules from:
  - Basic axioms
  - The definitions of FDs

Functional Dependencies: Closure of Attribute Sets

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

  result $\leftarrow \alpha$

  while (changes to result) **do**

      for each $\beta \rightarrow \gamma$ **in** F **do**

          begin

              if $\beta \subseteq$ *result* **then** *result* $\leftarrow$ *result* $\cup\ \gamma$

          end

Functional Dependencies: Closure of Attribute Sets: Example

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$
  - result = AG
  - result = ABCG ($A \rightarrow C$ and $A \rightarrow B$)
  - result = ABCGHI ($CG \rightarrow H$ and $CG \rightarrow AGBC$)
  - result = ABCGHI ($CG \rightarrow I$ and $CG \rightarrow AGBCH$)
- Is AG a candidate key?

- Is AG a super key?
  - Does $AG \to R$? == Is $(AG)^+ \supseteq R$
- Is any subset of AG a superkey?
  - Does $A \to R$? == Is $(A)^+ \supseteq R$
  - Does $G \to R$? == Is $(A)^+ \supseteq R$

Functional Dependencies: Closure of Attribute Sets: Use

There are several uses of the attributes closure algorithm:

- Testing for superkey:
  - To test is 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 $\alpha \to \beta$ 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 $\gamma \to S$

Decomposition using Functional Dependency

BCNF: Boyce-Codd Normal Form

- A relations schema R is in BCNF w.r.t a set F of FDs if for all FDs in F^+ of the form

  $\alpha \to \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$ at least one of the following holds:
  - $\alpha \to \beta$ is trivial (that is, $\beta \subseteq \alpha$)
  - α is a superkey for R
- Example schema not in BCNF:

  instr_dept (<u>ID</u>, name, salary, <u>dept_name</u>, building, budget)
- because the non-trivial dependency *dept_name* $\to$ *building, budget* holds on *instr_dept*, but *dept_name* is not a superkey

BCNF: Decomposition

- If in schema R and non-trivial dependency $\alpha \to \beta$ causes a violation of BCNF, we decompose R into:
  - $\alpha \cup \beta$
  - $(R - (\beta - \alpha))$
- In our example:
  - $\alpha = dept\_name$
  - $\beta = building,\ budget$
  - $dept\_name \to building,\ budget$

  inst_dept is replaced by
  - $(\alpha \cup \beta) = (dept\_name,\ building,\ budget)$
    - $dept\_name \to building,\ budget$
  - $(R - (\beta - \alpha)) = (ID,\ name,\ salary,\ dept\_name)$
    - $ID \to name,\ salary,\ dept\_name$

Lossless Join

- If we decompose a relation R into relations R_1 and R_2:
  - Decomposition is lossy if $R_1 \bowtie R_2 \supset R$

- Decomposition is lossless if $R_1 \bowtie R_2 = R$
- To check if lossless join decomposition using FD set, the following must hold:
  - Union of Attributes of R_1 and R_2 must be equal to attribute of R
    $R_1 \cup R_2 = R$
  - Intersection of Attributes of R_1 and R_2 must not be NULL
    - $R_1 \cap R_2 \neq \phi$
  - Common attribute must be a key for at least one relation (R_1 or R_2)
    $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$
- Prove that BCNF ensures Lossless Join

BCNF: Dependency Preservation

- Constraints, including FDs, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that all functional dependencies hold, then that decomposition is *dependency preserving*
- It is not always possible to achieve both BCNF and dependency preservation
- Consider:
  - R = CSZ, F = $\{CS \rightarrow Z, Z \rightarrow C\}$
  - Key = CS
  - $CS \rightarrow Z$ satisfies BCNF, but $Z \rightarrow C$ violates
  - Decompose as: $R_1 = ZC, R_2 = CSZ - (C - Z) = SZ$
  - $R_1 \cup R_2 = CSZ = R, R_1 \cap R_2 = Z = \phi$ and $R_1 \cap R_2 = Z \rightarrow ZC = R_1$
    - So, it has ***lossless join***
  - However, we cannot check $CS \rightarrow Z$ without doing a join
    - Hence, it is not ***dependency preserving***
- We consider a weaker normal form, known as **Third Normal Form (3NF)**

3NF: 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 (that is, $\beta \subseteq \alpha$)
  - 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 must hold)
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation

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, 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-join decomposition
  - Preferably, the decomposition should be dependency preserving

Problems with Decomposition

There are 3 potential problems to consider:

- May be impossible to re-construct the original relation (Lossiness)
- Dependency checking may require joins
- Some queries become more expensive
    - What is the building for an instructor?

Tradeoff: Must consider these issues vs redundancy

How good is BCNF?

- There are DB 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

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

inst_info

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies — that is, 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)

- Therefore, 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 the Fourth Normal Form (4NF)

Week 5 Lecture 4

Class	BSCCS2001
Created	@October 4, 2021 6:10 PM
Materials	
Module #	24
Type	Lecture
Week #	5

Relational Database Design (part 4)

Algorithms for Functional Dependencies

Attribute Set Closure

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
- $(AG)^+$
    - result = AG
    - result = $ABCG$ $(A \rightarrow C \text{ and } A \rightarrow B)$
    - result = $ABCGH$ $(CG \rightarrow H \text{ and } CG \subseteq AGBC)$
    - result = $ABCGHI$ $(CG \rightarrow I \text{ and } CG \subseteq AGBCH)$
- is AG a candidate key?
    - is AG a super key?
        - Does $AG \rightarrow R$? == is $(AG)^+ \supseteq R$
    - is any subset of AG a superkey?
        - Does $A \rightarrow R$? == is $(A)^+ \supseteq R$
        - Does $G \rightarrow R$? == is $(G)^+ \supseteq R$

Attribute Set Closure: Uses

There are several uses of the attribute closure algorithm

- Testing for a superkey:
  - To test if α is a superkey, we compute α^+ and check if α^+ contains all the attributes of R
- Testing functional dependencies
  - To check if a functional dependency $\alpha \to \beta$ 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
  - It 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 $\gamma \to S$

Extraneous Attributes

extra attribute

- Consider a set F of FDs and the FD $\alpha \to \beta$ in F
  - Attribute A is extraneous in α if $A \in \alpha$ and F logically implies
    $(F - \{\alpha \to \beta\}) \cup \{(\alpha - A) \to \beta\}$
  - Attribute A is extraneous in β if $A \in \beta$ and the set of FDs
    $(F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$ 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
- **Example:** Given $F = \{A \to C, AB \to C\}$
  - B is extraneous in $AB \to C$ because $\{A \to C, AB \to C\}$ logically implies $A \to C$
    (that is, the result of dropping B from $AB \to C$)
  - $A^+ = AC$ in $\{A \to C, AB \to C\}$

↳ basically even if u delete B from AB → C, A → C already exists, so the func^nt dependency will still hold

- **Example:** Given $F = \{A \to C, AB \to CD\}$
  - C is extraneous in $AB \to CD$ since $AB \to C$ can be inferred even after deleting C
  - $AB^+ = ABCD$ in $\{A \to C, AB \to D\}$

↳ $A \to C \Rightarrow AB \to C$

Extraneous Attributes: Tests

- Consider a set F of functional dependencies and the functional dependency $\alpha \to \beta$ in F
- To test if attribute $A \in \alpha$ is extraneous in α
  - Compute $(\{\alpha\} - A)^+$ using the dependencies in F
  - Check that $(\{\alpha\} - A)^+$ contains β; if it does, A is extraneous in α
- To test if attribute $A \in \beta$ is extraneous in β
  - Compute α^+ using only the dependencies in
    $F' = (F - \{\alpha \to \beta\}) \cup \{\alpha \to (\beta - A)\}$
  - Check that α^+ contains A; if it does, A is extraneous in β

Equivalence of Sets of Functional Dependencies

- Let F & G are two functional dependency sets
  - These two sets F & G are equivalent $F^+ = G^+$
  - That is: $(F^+ = G^+) \Leftrightarrow (F^+ \Rightarrow G \; and \; G^+ \Rightarrow F)$
  - Equivalence means that every functional dependency in F can be inferred from G and every functional dependency in G can be inferred from F
- F and G are equal only if
  - F covers G: Means that all functional dependency of G are logically numbers of functional dependency set $F \Rightarrow F^+ \supseteq G$

R(ABCDEF)   F{AB→C, C→D, D→E, F→B, E→F}

Candidate Key ⇒ ① Whichever attr. doesn't appear even once on any LHS, will always be a part of CK (A in this case)

{AB}, {AC}, {AD}, {AE}, {AF}

② But 'A' alone can't determine anything except itself, i.e. $A^+ = \{A\}$

③ Keep adding atts till all atts r uniquely identified

$AB^+ = \{\underbrace{A, B}_{\text{reflexivity}}, \underbrace{C}_{\text{given}}, D, E, F\}$ so AB is CK

$\left.\begin{array}{l} AB \to C \\ C \to D \end{array}\right\} AB \to D$ | $\left.\begin{array}{l} AB \to D \\ D \to E \end{array}\right\} AB \to E$ | $\left.\begin{array}{l} AB \to E \\ E \to F \end{array}\right\} AB \to F$

④ But there can be >1 CK $AC^+ = \{\underbrace{A, C}_{\text{reflexivity}}, D, E, F, B\}$

AC→C, C→D ⇒ AC→D | AC→D, D→E ⇒ AC→E | AC→E, E→F ⇒ AC→F

$AD^+ = \{\underbrace{A, D}_{\text{reflexivity}}, E, F, B, C\}$

$\left.\begin{array}{l} AD \to A \\ AD \to B \\ AB \to C \end{array}\right\} AD \to C$

$AE \to C \left\{\begin{array}{l} AE \to A \\ AE \to B \\ AB \to C \end{array}\right.$

$AE^+ = \{A, E, F, B, C, D\}$ $AF^+ = \{A, F, B, C, D, E\}$

⑤ Now, we won't check if {ABC} is a CK cuz it'll not be the minimal set ({AB})

* Calculate no of super keys when you've already found the candidate key/keys. If CK = 1, then no = 2^(no of remaining attributes in R)

Equivalence of func^n^l dependencies

$\alpha = \{A \to BC, B \to A, C \to A\}$, $\gamma = \{A \to B, B \to C, C \to A\}$

Check if $\alpha^+ = \gamma^+$ ⇒ α covers γ ⇒ find closure set of γ using func^nl depend in α

γA^+ (using α) = {A, B, C} $\gamma B^+ = \{A, B, C\}$ $\gamma C^+ = \{A, B, C\}$ $\left.\begin{array}{l} C \to A \\ A \to BC \end{array}\right\} C \to ABC$

α ⇒ A → A → BC ⇒ A → ABC $\left.\begin{array}{l} B \to A \\ A \to BC \end{array}\right\} B \to ABC$

γ covers α ⇒ find closure set of α using func^nl depend in γ

αA^+ (using γ) = {A, B, C} $\alpha B^+ = \{A, B, C\}$ $\alpha C^+ = \{A, B, C\}$

γ ⇒ A → B → C ⇒ A → ABC   B → C → A ⇒ B → ABC   C → A → B ⇒ C → ABC

⇒ $\alpha^+ = \gamma^+$ ⇒ Equivalent set of functional dependencies

Canonical Cover

① All single atts in RHS ⇒ A → BC ⇒ A → B, A → C

② If extraneous atts rnt, find & remove

③ Find out any redundant func^nl depend

Q) Find correct cannonical cover of R(ABCD)
F = {A→BC, AB→C, A→D, D→C}

① A→BC, AB→C   ② A→BC, AB→C, A→D
③ A→B, A→D, D→C   ④ A→B, B→C, A→D

Step 1 ⇒ A→B, A→C, AB→C, A→D, D→C

Step 2 ⇒ AB→C is B extraneous attr? Find $(AB-B)^+ = (A^+)$

$A^+ = \{A, B, C\}$, C is still getting determined by 'A' alone, so no need of 'B' in AB → C. B is extraneous ⇒ A→B, $\underbrace{A \to C, A \to C}_{\text{only 1 in final ans}}$, A→D, D→C

Step 3 ⇒ To check if redundant, find cover of attr without using that particular func^nl dependency

eg. ① A→B

A^+ (without A→B) = $\underbrace{\{A, C, D\}}_{\text{not covering 'B'}}$

So not redundant

② A→C

$A^+ = \{A, B, D, C\}$ ← A→B, A→D, D→C ⇒ A → ABCD
↳ covering 'C' so redundant

Final ans = $F^+ = \{A \to B, A \to D, D \to C\}$

P Atts = PQ   Non-Prim atts = RSTUVW

Q) R(PQRSTUVW)   F = {PQ → RSTU, P → R, Q → S, R → UV, V → W, W → U, V → U} CK?

PQ -nt in RHS of any FD ⇒ PQ will be a part of CK

$(PQ^+) = \{PQRSTUVW\}$ ⇒ PQ = CK & any combinⁿ involving PQ, PQR, PQRU etc will be a Superkey

Q) R(ABCDEF)   F = {AB → CDE, E → F, BF → A, C → B}   CK?

① There's nthng which isn't -nt in RHS of any FD, Now check closure of atts 1 by 1, then move onto combos

$(A^+) = \{A\}$, $B^+ = \{B\}$, $C^+ = \{CB\}$, $D^+ = \{D\}$, $E^+ = \{EF\}$, $F^+ = \{F\}$ ⇒ No attr determines all others, so move onto combos

$(AB^+) = \{ABCDEF\}$, $(AC^+) = \{ACBDEF\}$, $(AD^+) = \{AD\}$, $(AE^+) = \{AEF\}$, $(AF^+) = \{AF\}$

AC → AC, C → B; AC → ACB, AB → CDE, E → F ⇒ AC → DEF

$(BC^+) = \{BC\}$, $(BD^+) = \{BD\}$, $(BE^+) = \{ABCDEF\}$

E → F ⇒ BE → BEF; BF → A ⇒ BE → A; AB → CDE ⇒ BE → CDE

$(CD^+) = \{CDB\}$, $(CE^+) = \{CEFBAD\}$, $(BF^+) = \{ABCDEF\}$

$(CF^+) = \{CFBADE\}$, $(DE^+) = \{DEF\}$, $\{DF^+\} = \{DF\}$

CK = AB, AC, BF, BE, CF, CE ⇒ Any 3 combos inc any of these will be a superkey

Prime atts ⇒ Atts which're a part of CK   eg ABCEF (above q)

Non- —"— ⇒ —"— not —"—   eg D (—"—)

* Suppose we've a relⁿ R(ABCD) but no funcⁿˡ depend, i.e F = ∅ then CK = (ABCD)

* Any key out of the CK(s) can be a Prim key for a table

Q) R(ABCD), F = {A → B, B → C, C → D} No of superkey? ⑧

CK = $A^+ = \{R\}$ Superkey = A, AB, AC, AD, ABD, ABC, ACD, ABCD = ⑧

n = total no of atts
m = no of atts in CK
(only 1 CK -nt)

⇒ $\boxed{\text{no of superkey} = 2^{n-m}}$

Case 2 ⇒ CK = A & B ⇒ CK based on A = $2^{n-m} = 2^{4-1} = 8$
" " " B = $2^{n-m} = 2^{4-1} = 8$

$A \cap B = n(A) + n(B) - n(A \cup B)$
$= 8 + 8 - 2^{4-2} = 4$ ⇒ 8 + 8 - 4 = ⑫

• Every CK is also a Super-key

Q) R(ABCDEFGH)   F = {H → GD, E → D, HD → CE, BD → A}   Minimal cover of F?

① {H → GCE, BD → A}   ② {H → GCE, E → D, D → A}   ③ {H → GCE, E → D, BD → A} ✓   ④ {H → GCE, E → D, B → A}

Solⁿ) ① Not equivalent F covers F_1 but F_1 doesn't cover F so reject

$((H^+)_{F_1})$ using F = {HGCE}, doesn't cover D as in F so reject

② Not equivalent   ④ Not equivalent   ③ Only equivalent option

OR, F = {H → G, H → D, E → D, HD → CE, BD → A}

① Check extraneous atts   a) BD → A ⇒ check if (B^+) covers A
" " (D^+) " A
$(B^+) = \{B\}$ $(D^+) = \{D\}$
No extraneous ⇒ atts in BD → A

b) HD → CE ⇒ HD → C, HD → E

$(H^+) = \{HGDCE\}$ ⇒ D is extraneous in both ⇒ H → C, H → E or H → CE

LHS done, now RHS (or check redundant FD)

RHS ⇒ ⓐ H → GD ⇒ $(H^+) = \{HDCE\}$ ⇒ doesn't cover 'G' so not extraneous ⇒ H → D
but, H → GD ⇒ H → G ⇒ $(H^+) = \{HGCED\}$ ⇒ covers 'D' so extraneous ⇒ H → G

ⓑ H → CE ⇒ H → E ⇒ $(H^+) = \{HGED\}$ ⇒ no 'C' ⇒ H → CE
H → CE ⇒ H → C ⇒ $(H^+) = \{HGD\}$ ⇒ no 'E' ⇒ H → CE

② {H → G, E → D, H → CE, BD → A}
union
{H → GCE, E → D, BD → A} = Option 3rd

- G covers F: Means that all functional dependency of F are logically numbers of functional dependency set $G \Rightarrow G^+ \supseteq F$

Condition	CASES			
F Covers G	True	True	False	False
G Covers F	True	False	True	False
Result	F=G	F⊃G	G⊃F	No Comparison

Canonical Cover

• multiple cannonical cover possible

• If F_c^+ (given) is equivalent to (F^+), then (F_c^+) is a possible cannonical cover

- Sets of FDs may have redundant dependencies that can be inferred from the others
- Can we have some kind of "optimal" or "minimal" set of FDs to work with?
- A **Canonical Cover** for F is a set of dependencies F_c such that ALL the following properties are satisfied:
  - $F^+ = F_c^+$
    - F logically implies all dependencies in F_c
    - F_c logically implies all dependencies in F
  - No functional dependency in F_c contains an irrelevant attribute
  - Each left side of functional dependency in F_c is unique
  - That is, there are no two dependencies $\alpha_1 \to \beta_1$ and $\alpha_2 \to \beta_2$ in such that $\alpha_1 \to \alpha_2$
- Intuitively, a **Canonical cover** of F is a ***minimal set*** of FDs
  - Equivalent to F
  - Having no redundant FDs
  - No redundant parts of FDs
- **Minimal / Irreducible Set of Functional Dependencies**

Canonical Cover: Example

- For example: $A \to C$ is redundant in $\{A \to B, B \to C, A \to C\}$
- Parts of a functional dependency may be redundant
  - For example: on RHS: $\{A \to B, B \to C, A \to CD\}$ can be simplified to $\{A \to B, B \to C, A \to D\}$

– In the forward: (1) $A \to CD \Rightarrow A \to C$ and $A \to D$
(2) $A \to B, B \to C \Rightarrow A \to C$

– In the reverse: (1) $A \to B, B \to C \Rightarrow A \to C$
(2) $A \to C, A \to D \Rightarrow A \to CD$

∘ For example: on LHS: $\{\boldsymbol{A \to B, B \to C, AC \to D}\}$ can be simplified to $\{\boldsymbol{A \to B, B \to C, A \to D}\}$

– In the forward: (1) $A \to B, B \to C \Rightarrow A \to C \Rightarrow A \to AC$
(2) $A \to AC, AC \to D \Rightarrow A \to D$

– In the reverse: $A \to D \Rightarrow AC \to D$

Canonical Cover: RHS

- $\{A \to B, B \to C, A \to CD\} \Rightarrow \{A \to B, B \to C, A \to D\}$
  - (1) $A \to CD \Rightarrow A \to C$ and $A \to D$
    (2) $A \to B, B \to C \Rightarrow A \to C$
  - $A^+ = ABCD$
- $\{A \to B, B \to C, A \to D\} \Rightarrow \{A \to B, B \to C, A \to CD\}$
  - $A \to B, B \to C \Rightarrow A \to C$
  - $A \to C, A \to D \Rightarrow A \to CD$
  - $A^+ = ABCD$

Canonical Cover: LHS

- $\{A \to B, B \to C, AC \to D\} \Rightarrow \{A \to B, B \to C, A \to D\}$
  - $A \to B, B \to C \Rightarrow A \to C \Rightarrow A \to AC$
  - $A \to AC, AC \to D \Rightarrow A \to D$
  - $A^+ = ABCD$
- $\{A \to B, B \to C, A \to D\} \Rightarrow \{A \to B, B \to C, AC \to D\}$
  - $A \to D \Rightarrow AC \to D$
  - $AC^+ = ABCD$

Canonical Cover

- To compute a canonical cover for F :

  repeat

  Use the union rule to replace any dependencies in F

  $\alpha_1 \to \beta_1$ and $\alpha_1 \to \beta_2$ with $\alpha_1 \to \beta_1\beta_2$

  Find a functional dependency $\alpha \to \beta$ with an

  irrelevant attribute either in α or in β

  /* NOTE: test for irrelevant attributes done using F_c, not F */

  If an irrelevant attribute is found, delete it from $\alpha \to \beta$

  until F does not change
- **NOTE:** Union rule may become applicable after some irrelevant attributes have been deleted, so it has to be re-applied

Canonical Cover: Example

- $R = (A, B, C)$
  $F = \{A \rightarrow BC, B \rightarrow C, A \rightarrow B, AB \rightarrow C\}$
- 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: $A \rightarrow B$, $B \rightarrow C$

Practice Problems on Functional Dependencies

- **Find if a given functional dependency is implied from a set of Functional Dependencies:**
  a) For: $A \rightarrow BC, CD \rightarrow E, E \rightarrow C, D \rightarrow AEH, ABH \rightarrow BD, DH \rightarrow BC$
     i) Check: $BCD \rightarrow H$
     ii) Check: $AED \rightarrow C$
  b) For: $AB \rightarrow CD, AF \rightarrow D, DE \rightarrow F, C \rightarrow G, F \rightarrow E, G \rightarrow A$
     i) Check: $CF \rightarrow DF$
     ii) Check: $BG \rightarrow E$
     iii) Check: $AF \rightarrow G$
     iv) Check: $AB \rightarrow EF$
  c) For: $A \rightarrow BC, B \rightarrow E, CD \rightarrow EF$
     i) Check: $AD \rightarrow F$

- **Find Super Key using Functional Dependencies:**
  a) Relational Schema $R(ABCDE)$. Functional dependencies: $AB \rightarrow C, DE \rightarrow B, CD \rightarrow E$
  b) Relational Schema $R(ABCDE)$. Functional dependencies: $AB \rightarrow C, C \rightarrow D, B \rightarrow EA$

- **Find Candidate Key using Functional Dependencies:**
  a) Relational Schema $R(ABCDE)$. Functional dependencies: $AB \rightarrow C, DE \rightarrow B, CD \rightarrow E$
  b) Relational Schema $R(ABCDE)$. Functional dependencies: $AB \rightarrow C, C \rightarrow D, B \rightarrow EA$

- **Find Prime and Non Prime Attributes using Functional Dependencies:**
  a) $R(ABCDEF)$ having FDs $\{AB \to C, C \to D, D \to E, F \to B, E \to F\}$
  b) $R(ABCDEF)$ having FDs $\{AB \to C, C \to DE, E \to F, C \to B\}$
  c) $R(ABCDEFGHIJ)$ having FDs $\{AB \to C, A \to DE, B \to F, F \to GH, D \to IJ\}$
  d) $R(ABDLPT)$ having FDs $\{B \to PT, A \to D, T \to L\}$
  e) $R(ABCDEFGH)$ having FDs
  $\{E \to G, AB \to C, AC \to B, AD \to E, B \to D, BC \to A\}$
  f) $R(ABCDE)$ having FDs $\{A \to BC, CD \to E, B \to D, E \to A\}$
  g) $R(ABCDEH)$ having FDs $\{A \to B, BC \to D, E \to C, D \to A\}$

> - **Prime Attributes**: Attribute set that belongs to any candidate key are called Prime Attributes
>   - It is union of all the candidate key attribute: $\{CK_1 \cup CK_2 \cup CK_3 \cup \cdots\}$
>   - If Prime attribute determined by other attribute set, then more than one candidate key is possible.
>   - For example, If A is Candidate Key, and $X \to A$, then, X is also Candidate Key.
> - **Non Prime Attribute**: Attribute set does not belong to any candidate key are called Non Prime Attributes

- **Check the Equivalence of a Pair of Sets of Functional Dependencies:**
  a) Consider the two sets F and G with their FDs as below :
     i) $F : A \to C, AC \to D, E \to AD, E \to H$
     ii) $G : A \to CD, E \to AH$
  b) Consider the two sets P and Q with their FDs as below :
     i) $P : A \to B, AB \to C, D \to ACE$
     ii) $Q : A \to BC, D \to AE$

- **Find the Minimal Cover or Irreducible Sets or Canonical Cover of a Set of Functional Dependencies:**
  a) $AB \to CD, BC \to D$
  b) $ABCD \to E, E \to D, AC \to D, A \to B$

Week 5 Lecture 5

Class	BSCCS2001
Created	@October 4, 2021 8:05 PM
Materials	
Module #	25
Type	Lecture
Week #	5

Relational Database Design (part 5)

Lossless Join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R
  $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r)$
- A decomposition of R into R_1 and R_2 is lossless join 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

To Identify whether a decomposition is lossy or lossless, it must satisfy the following conditions:

- $R_1 \cup R_2 = R$
- $R_1 \cap R_2 \neq \phi$ and
- $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

Lossless Join Decomposition: Example

- Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies: **S#** $\rightarrow$ **Sname, S#** $\rightarrow$ **City, (S#, P#)** $\rightarrow$ **Qty**
- Decompose as: **Supplier (<u>S#</u>, Sname, City, Qty): Parts (<u>P#, Qty</u>)**

- Take natural join to reconstruct: **Supplier** $\bowtie$ **Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City	Qty
3	Smith	London	20
5	Nick	NY	50
2	Steve	Boston	10
5	Nick	NY	40
5	Nick	NY	10

P#	Qty
301	20
500	50
20	10
400	40
301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
5	**Nick**	**NY**	**20**	**10**
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10
2	**Steve**	**Boston**	**301**	**10**

- We get extra tuples! ***Join is lossy***
- Common attribute Qty is not a superkey in **Supplier** or in **Parts**
- Does not preserve **(S#, P#)** $\rightarrow$ **Qty**

- Consider **Supplier_Parts** schema: **Supplier_Parts(S#, Sname, City, P#, Qty)**
- Having dependencies: **S#** $\rightarrow$ **Sname, S#** $\rightarrow$ **City, (S#, P#)** $\rightarrow$ **Qty**
- Decompose as: **Supplier (S#, Sname, City, Qty): Parts (P#, Qty)**
- Take natural join to reconstruct: **Supplier** $\bowtie$ **Parts**

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

S#	Sname	City
3	Smith	London
5	Nick	NY
2	Steve	Boston
5	Nick	NY
5	Nick	NY

S#	P#	Qty
3	301	20
5	500	50
2	20	10
5	400	40
5	301	10

S#	Sname	City	P#	Qty
3	Smith	London	301	20
5	Nick	NY	500	50
2	Steve	Boston	20	10
5	Nick	NY	400	40
5	Nick	NY	301	10

- We get the original relation. **Join is lossless.**
- Common attribute **S#** is a superkey in **Supplier**
- Preserve all the dependencies

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C\}$
  - Can be decomposed in two different ways
- $R_1 = (A, B), R_2 = (B, C)$
  - 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 $R_1 \bowtie R_2$)

Practice Problem on Lossless join

- **Check if the decomposition of R into D is lossless:**
  a) $R(ABC) : F = \{A \to B, A \to C\}$. $D = R_1(AB), R_2(BC)$
  b) $R(ABCDEF) : F = \{A \to B, B \to C, C \to D, E \to F\}$.
     $D = R_1(AB), R_2(BCD), R_3(DEF)$
  c) $R(ABCDEF) : F = \{A \to B, C \to DE, AC \to F\}$. $D = R_1(BE), R_2(ACDEF)$
  d) $R(ABCDEG) : F = \{AB \to C, AC \to B, AD \to E, B \to D, BC \to A, E \to G\}$
     i) $D1 = R_1(AB), R_2(BC), R_3(ABDE), R_4(EG)$
     ii) $D2 = R_1(ABC), R_2(ACDE), R_3(ADG)$
  e) $R(ABCDEFGHIJ) : F = \{AB \to C, B \to F, D \to IJ, A \to DE, F \to GH\}$
     i) $D1 = R_1(ABC), R_2(ADE), R_3(BF), R_4(FGH), R_5(DIJ)$
     ii) $D2 = R_1(ABCDE), R_2(BFGH), R_3(DIJ)$
     iii) $D3 = R_1(ABCD), R_2(DE), R_3(BF), R_4(FGH), R_5(DIJ)$

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i
  - A decomposition is **dependency preserving** if

    $(F_1 \cup F_2 \cup ... \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing join, which is expensive

Let R be the original relational schema having FD set F. Let R_1 and R_2 having FD set F_1 and F_2 respectively, are the decomposed sub-relations of R. The decomposition of R is said to be preserving if

- $F_1 \cup F_2 \equiv F$ *{Decomposition Preserving Dependency}*
- *If* $F_1 \cup F_2 \subset F$ *{Decomposition NOT Preserving Dependency} and*
- $F_1 \cup F_2 \supset F$ *{this is not possible}*

Dependency Preservation: Testing

- To check if a dependency $\alpha \to \beta$ is preserved in a decomposition of R into $D = \{R_1, R_2, ..., R_n\}$ we apply the following test (with attribute closure done with respect to F)
- The **restriction** of F^+ to R_i is the set of all functional dependencies in F^+ that include only attributes of R_i

  compute F^+;

  for each schema R_i in D **do**

      begin

          F_i = the restriction of F^+ to R_i;

      end

  $F' = \phi$

  for each restriction F_i **do**

      begin

          $F' = F' \cup F_i$

      end

  compute F'^+;

  if $(F'^+ = F^+)$ **then** return (true)

      else return (false);
- The procedure for checking dependency preservation takes exponential time to compute F^+ and $(F_1 \cup F_2 \cup ... \cup F_n)^+$

Dependency Preservation: Example

- $R\ (A, B, C, D, E, F)$

$F = \{A \to BCD, A \to EF, BC \to AD, BC \to E, BC \to F, B \to F, D \to E\}$

- Decomposition: $R1(A, B, C, D)$ $R2(B, F)$ $R3(D, E)$
  - $A \to BCD$, $BC \to AD$ are preserved on table R1
  - $B \to F$ is preserved on table R2
  - $D \to E$ is preserved on table R3
  - We have to check whether the remaining FDs: $A \to E$, $A \to F$, $BC \to E$, $BC \to F$ are preserved or not

R1	R2	R3
$F_1=\{$**A** $\to ABCD$, **B** $\to B$, **C** $\to C$, **D** $\to D$, **AB** $\to ABCD$, **BC** $\to ABCD$, **CD** $\to CD$, **AD** $\to ABCD$ **ABC** $\to ABCD$, **ABD** $\to ABCD$, **ACD** $\to ABCD$ **BCD** $\to ABCD\}$	$F_2=\{$**B** $\to BF$, **F** $\to F\}$	$F_3=\{$**D** $\to DE$, **E** $\to E\}$

- $F' = F_1 \cup F_2 \cup F_3$
- Checking for: $A \to E$, $A \to F$ in F'^{+}
  - $A \to D$ (from R1), $D \to E$ (from R3) : $A \to E$ (By transitivity)
  - $A \to B$ (from R1), $B \to F$ (from R2) : $A \to F$ (By transitivity)
- Checking for: $BC \to E$, $BC \to F$ in F'^{+}
  - $BC \to D$ (from R1), $D \to E$ (from R3) : $BC \to E$ (by transitivity)
  - $B \to F$ (from R2) : $BC \to F$ (by augmentation)

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

  $F = \{A \to B, B \to C, C \to D, D \to A\}$
- Decomposition: $R1(A, B)$ $R2(B, C)$ $R3(C, D)$
  - $A \to B$ is preserved on table R1
  - $B \to C$ is preserved on table R2
  - $C \to D$ is preserved on table R3
  - We have to check whether the one remaining FD: $D \to A$ is preserved or not

R1	R2	R3
$F_1=\{$**A** $\to AB$, **B** $\to BA\}$	$F_2=\{$**B** $\to BC$, **C** $\to CB\}$	$F_3=\{$**C** $\to CD$, **D** $\to DC\}$

- $F' = F_1 \cup F_2 \cup F_3$
- Checking for: $D \to A$ in F'^{+}
  - $D \to C$ (from R3), $C \to B$ (from R2), $B \to A$ (from R1) : $D \to A$ (by transitivity)

    Hence, all dependencies are preserved

Dependency Preservation: Testing

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

    while (changes to result) do

    for each R_i in the decomposition

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

    result = result ∪ t
  - If result contains all attributes in β, then the functional dependency $\alpha \to \beta$ is preserved
- We apply the test of 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 $(F_1 \cup F_2 \cup ... \cup F_n)^+$

Dependency Preservation: Example

- $R(ABCDEF)$:. $F = \{A \to BCD, A \to EF, BC \to AD, BC \to E, BC \to F, B \to F, D \to E\}$
- $Decomp = \{ABCD, BF, DE\}$
- On projections:

ABCD (R1)	BF (R2)	DE (R3)
A → BCD BC → AD	B → F	D → E

- Need to check for: $\cancel{A \to BCD}$, $\boldsymbol{A \to EF}$, $\cancel{BC \to AD}$, $\boldsymbol{BC \to E}$, $\boldsymbol{BC \to F}$, $\cancel{B \to F}$, $\cancel{D \to E}$
- $(BC)+/F1 = ABCD$. $(ABCD)+/F2 = ABCDF$. $(\boldsymbol{ABCDF})+/\boldsymbol{F}3 = \boldsymbol{ABCDEF}$. Preserves $\boldsymbol{BC \to E}$, $\boldsymbol{BC \to F}$
  $BC \to AD$ $(R1)$, $AD \to E$ $(R3)$ implies $BC \to E$
  $B \to F$ $(R2)$ implies $BC \to F$
- $(A)+/F1 = ABCD$. $(ABCD)+/F2 = ABCDF$. $(\boldsymbol{ABCDF})+/\boldsymbol{F}3 = \boldsymbol{ABCDEF}$. Preserves $\boldsymbol{A \to EF}$
  $A \to B$ $(R1)$, $B \to F$ $(R2)$ implies $A \to F$
  $A \to D$ $(R1)$, $D \to E$ $(R3)$ implies $A \to E$

- $R(ABCDEF)$: $F = \{A \to BCD, A \to EF, BC \to AD, BC \to E, BC \to F, B \to F, D \to E\}$. $Decomp = \{ABCD, BF, DE\}$
- On projections:

ABCD (R1)	BF (R2)	DE (R3)
A → B, A → C, A → D, BC → A, BC → D	B → F	D → E

- Infer reverse FD's:
  - $B+/F = BF$: $B \to A$ cannot be inferred
  - $C+/F = C$: $C \to A$ cannot be inferred
  - $D+/F = DE$: $D \to A$ and $D \to BC$ cannot be inferred
  - $A+/F = ABCDEF$: $A \to BC$ can be inferred, but it is equal to $A \to B$ and $A \to C$
  - $F+/F = F$: $F \to B$ cannot be inferred
  - $E+/F = E$: $E \to D$ cannot be inferred
- Need to check for: $\cancel{A \to BCD}$, $\boldsymbol{A \to EF}$, $\cancel{BC \to AD}$, $\boldsymbol{BC \to E}$, $\boldsymbol{BC \to F}$, $\cancel{B \to F}$, $\cancel{D \to E}$
  - $(\boldsymbol{BC})+/\boldsymbol{F} = \boldsymbol{ABCDEF}$. Preserves $\boldsymbol{BC \to E}$, $\boldsymbol{BC \to F}$
  - $(\boldsymbol{A})+/\boldsymbol{F} = \boldsymbol{ABCDEF}$. Preserves $\boldsymbol{A \to EF}$

Week 6 Lecture 1

Class	BSCCS2001
Created	@October 12, 2021 12:52 PM
Materials	
Module #	26
Type	Lecture
Week #	6

Relational Database Design (part 6)

Normal Forms

Normalization or Schema Refinement

- Normalization or Schema Refinement is a technique of organizing the data in the DB
- A systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics
    - Insertion Anomaly
    - Update Anomaly
    - Deletion Anomaly
- Most common technique for the Schema refinement is decomposition
    - Goal of Normalization: Eliminate redundancy
- Redundancy refers to the repetition of same data or duplicate copies of the same data stored in different locations
- Normalization is used for mainly 2 purposes:
    - Eliminating redundant (useless) data
    - Ensuring the data dependencies make sense, that is, data is logically stored

Anomalies

- **Update Anomaly:** Employee 519 is shown as having different addresses on different records

Employees' Skills

Employee ID	Employee Address	Skill
426	87 Sycamore Grove	Typing
426	87 Sycamore Grove	Shorthand
519	94 Chestnut Street	Public Speaking
519	96 Walnut Avenue	Carpentry

Resolution: Decompose the Schema

a) *Update:* (ID, Address), (ID, Skill)

b) *Insert:* (ID, Name, Hire Date), (ID, Code)

c) *Delete:* (ID, Name, Hire Date), (ID, Code)

- **Insertion Anomaly:** Until the new faculty member, Dr. Newsome, is assigned to teach at least one course, his details cannot be recorded

Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201
424	Dr. Newsome	29-Mar-2007	?

- **Deletion Anomaly:** All information about Dr. Giddens is lost if he temporarily ceases to be assigned to any courses

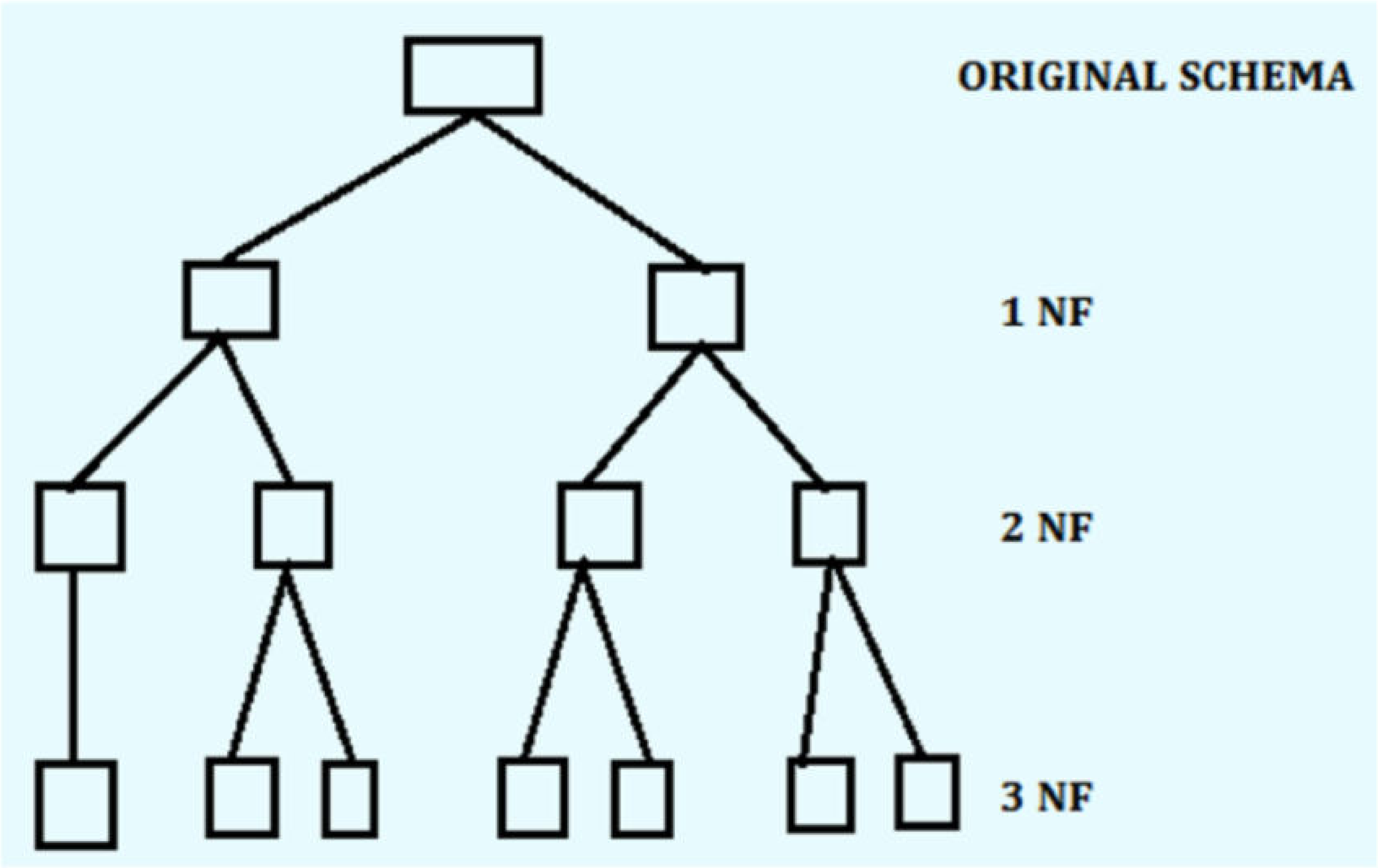
Faculty and Their Courses

Faculty ID	Faculty Name	Faculty Hire Date	Course Code
389	Dr. Giddens	10-Feb-1985	ENG-206
407	Dr. Saperstein	19-Apr-1999	CMP-101
407	Dr. Saperstein	19-Apr-1999	CMP-201

DELETE

Desirable Properties of Decomposition

- Lossless Join Decomposition Property
  - It should be possible to reconstruct the original table
- Dependency Preserving Property
  - No functional dependency (or other constraints should get violated)



Normalization and Normal Forms

- A normal form specifies a set of of conditions that the relational schema must satisfy in terms of its constraints — they offer varied levels of guarantee for the design
- Normalization rules are divided into various normal forms
- Most common normal forms are:
    - First Normal Form (1NF)
    - Second Normal Form (2NF)
    - Third Normal Form (3NF)
- Informally, a relational DB relation is often described as "normalized" if it meets the 3NF (Third Normal Form)
- Most 3NF are free from insertion, update and deletion anomalies

---

- Additional Normal Forms:
    - Elementary Key Normal Form (EKNF)
    - Boyce-codd Normal Form (BCNF)
    - Multi-valued Dependencies and Fourth Normal Form (4NF)
    - Essential Tuple Normal Form (ETNF)
    - Join Dependencies and Fifth Normal Form (5NF)
    - Sixth Normal Form (6NF)
    - Domain/Key Normal Form (DKNF)

1NF: First Normal Form

- A relation is in First Normal Form if and only if all underlying domains contain atomic values only (doesn't have multi-valued attributes (MVA))
- **STUDENT (Sid, Sname, Cname)**

Students

SID	Sname	Cname
S1	A	C,C++
S2	B	C++, DB
S3	A	DB

SID : Primary Key

MVA exists ⇒ Not in 1NF

Students

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB

SID, Cname : Primary Key

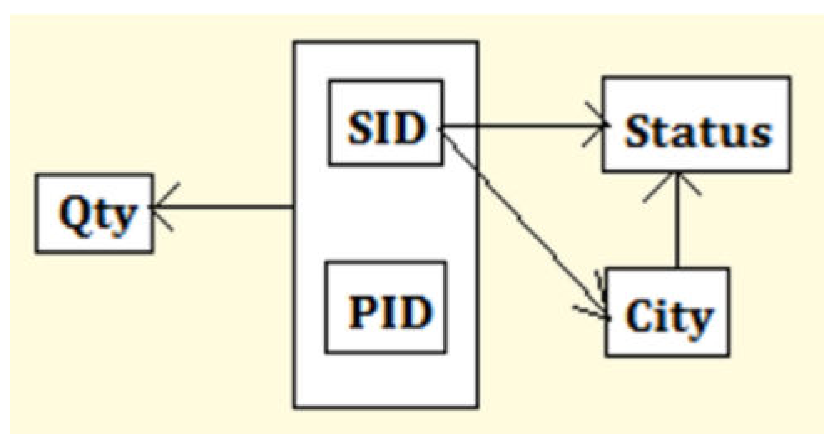
No MVA ⇒ In 1NF

1NF: Possible Redundancy

- **Example: Supplier (SID, Status, City, PID, Qty)**

Supplier

SID	Status	City	PID	Qty
S1	30	Delhi	P1	100
S1	30	Delhi	P2	125
S1	30	Delhi	P3	200
S1	30	Delhi	P4	130
S2	10	Karnal	P1	115
S2	10	Karnal	P2	250
S3	40	Rohtak	P1	245
S4	30	Delhi	P4	300
S4	30	Delhi	P5	315



Drawbacks:

- **Deletion Anomaly:** If we delete <S3, 40, Rohtak, P1, 245>, then we lose the information that S3 lives in Rohtak
- **Insertion Anomaly:** We cannot insert a Supplier S5 located in Karnal, until S5 supplies at least one part
- **Update Anomaly:** If Supplier S1 moves from Delhi to Kanpur, then it is difficult to update all the tuples having SID as S1 and City as Delhi

Normalization is a method to reduce redundancy

However, sometimes 1NF increases redundancy

1NF: Possible Redundancy

- **When LHS is not a Superkey:**
  - Let $X \rightarrow Y$ be a non-trivial FD over R with X is not a superkey of R, then redundancy exist between X and Y attribute set
  - Hence, in order to identify the redundancy, we need not to look at the actual data, it can be identified by given functional dependency
  - Example: $X \rightarrow Y$ and X is not a Candidate Key
    - X can duplicate
- **When LHS is a Superkey:**
  - If $X \rightarrow Y$ is a non-trivial FD over R with X is a superkey of R, then redundancy does not exist between X and Y attribute set
  - Example: $X \rightarrow Y$ and X is a Candidate Key
    - X cannot duplicate
    - Corresponding Y value may or may not duplicate

- Corresponding Y value would duplicate also

X	Y
1	3
1	3
2	3
2	3
4	6

X	Y
1	4
2	6
3	4

2NF: Second Normal Form

- Relation R is in Second Normal Form (2NF) only iff:
  - R is in 1NF and
  - R contains no Partial Dependency

any part of CK doesn't determine non prime atts
$\Rightarrow$ subset of CK $\not\to$ non prime atts

Partial Dependency:

Let R be a relational schema and X, Y, A be the attribute sets over R where X: Any Candidate Key, Y: Proper subset of Candidate Key and A: Non-prime attribute

If $Y \to A$ exists in R, then R is not in 2NF

$(Y \to A)$ is a Partial dependency only if

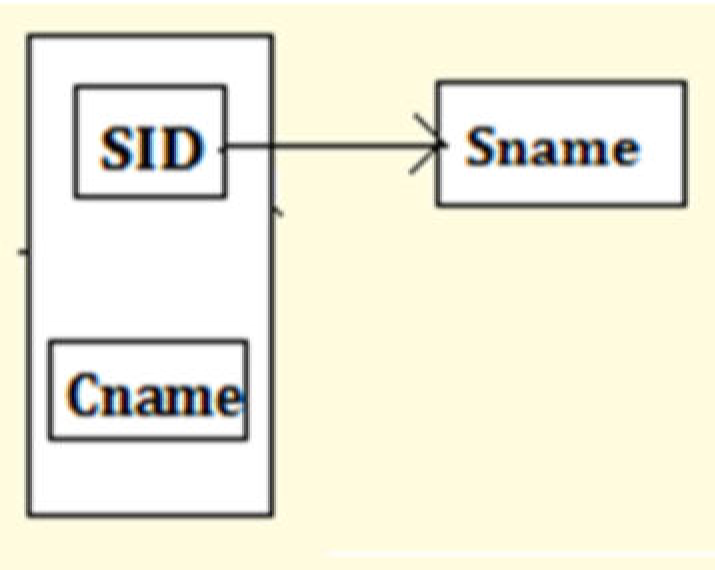
- Y: Proper subset of Candidate Key
- A: Non-Prime Attribute

A **prime attribute** of a relation is an attribute that is a part of a candidate key of the relation

- STUDENT (Sid, Sname, Cname) (already in 1NF)

Students

SID	Sname	Cname
S1	A	C
S1	A	C++
S2	B	C++
S2	B	DB
S3	A	DB



- Redundancy?
  - Sname
- Anomaly?
  - Yes
- ~~Hotel?~~
  - ~~*Trivago*~~

Functional Dependencies:

{SID, Cname} $\to$ *Sname*

SID $\to$ *Sname*

Partial Dependencies:

$SID \rightarrow Sname$ (as SID is a Proper Subset of Candidate Key $\{SID, Cname\}$)

Key Normalization

R1

SID	Sname
S1	A
S2	B
S3	A

{SID}: Primary Key

R2

SID	Cname
S1	C
S1	C++
S2	C++
S2	DB
S3	DB

{SID, Cname}: Primary Key

The above two relations R1 and R2 are

1. Lossless Join
2. 2NF
3. Dependency Preserving

2NF: Possible Redundancy

- **Supplier (SID, Status, City, PID, Qty)**

  Supplier

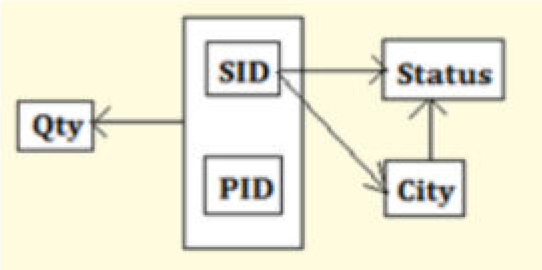
SID	Status	City	PID	Qty
S1	30	Delhi	P1	100
S1	30	Delhi	P2	125
S1	30	Delhi	P3	200
S1	30	Delhi	P4	130
S2	10	Karnal	P1	115
S2	10	Karnal	P2	250
S3	40	Rohtak	P1	245
S4	30	Delhi	P4	300
S4	30	Delhi	P5	315

  Key: (SID, PID)

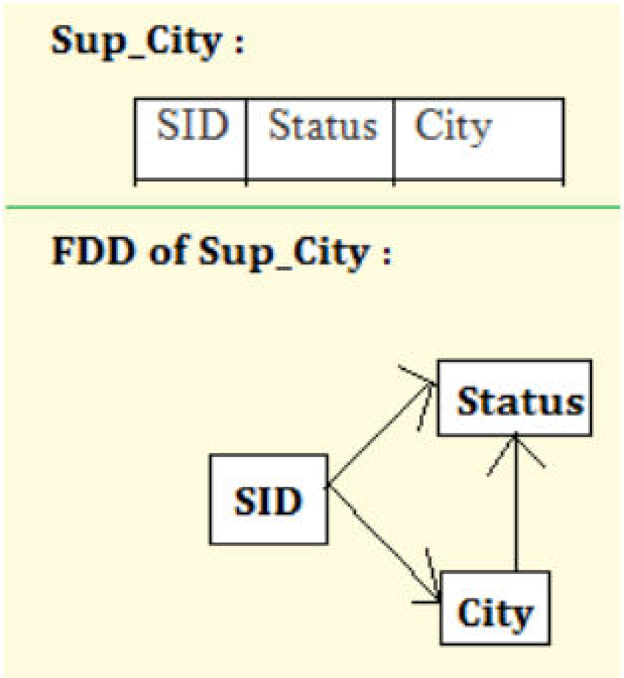
  Partial Dependencies:

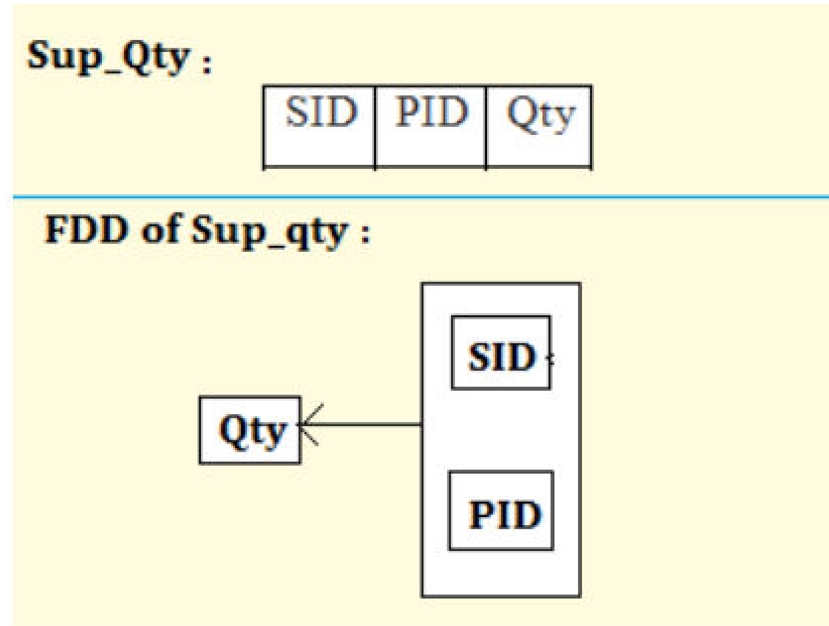
  $SID \rightarrow Status$

  $SID \rightarrow City$



Post Normalization





Drawbacks:

- **Deletion Anomaly:** If we delete a tuple in *Sup_City*, then we not only lose the information about a supplier, but also lose the status value of a particular city
- **Insertion Anomaly:** We cannot insert a City and its status until a supplier supplies at least one part
- **Update Anomaly:** If the status value for a city is unchanged, then we will face the problem of searching every tuple for that city

3NF: Third Normal Form

Let R be the relational schema

- [E.F. Codd, 1971] R is in 3NF only if:
  - R should be in 2NF
  - R should not contain transitive dependencies (OR, Every non-prime attribute of R is non-transitively dependent on every day of R)
- [Carlo Zaniolo, 1982] Alternately, R is in 3NF iff for each of its functional dependency X → A, at least one of the following conditions holds:
  - X contains A (that is, A is a subset of X, meaning X → A is trivial functional dependency) or
  - X is a superkey or
  - Every element of A — X, the set difference between A and X, is a prime attribute (ie. each attribute of A — X is contained in some candidate key)
- [Simple Statement] A relational schema R is in 3NF if for every FD X → A associated with R either
  - $A \subseteq X$ (that is, the FD is trivial) or
  - X is a superkey of R or
  - A is part of some candidate key (not just superkey)
- A relation is 3NF is naturally in 2NF

No $\underbrace{\text{transitive dependency}}$
⇓
NPA → NPA.

eg W5 1st q of CK
R → UV, V → W,
W → U, V → U

3NF: Transitive Dependency

- A transitive dependency is a functional dependency which holds by virtue of transitivity
- A transitive dependency can occur only in a relation that has 3 or more attributes
- Let A, B and C designate 3 distinct attributes (or distinct collections of attributes) in the relation
- Suppose all 3 of the following conditions hold:
  - A → B
  - It is not the case that B → A
  - B → C
- Then the functional dependency A → C (which follows from1 and 3 by the axiom of transitivity) is a transitive dependency

- Example of transitive dependency
- The functional dependency {Book} → {Author Nationality} applies; that is, if we know the book, we know the author's nationality
- Furthermore:
    - {Book} → {Author}
    - {Author} does not → {Book}
    - {Author} → {Author Nationality}
- Therefore, {Book} → {Author Nationality} is a transitive dependency
- Transitive dependency occurred because a non-key attribute (Author) was determining another non-key attribute (Author Nationality)

Book	Genre	Author	Author Nationality
Twenty Thousand Leagues Under the Sea	Science Fiction	Jules Verne	French
Journey to the Center of the Earth	Science Fiction	Jules Verne	French
Leaves of Grass	Poetry	Walt Whitman	American
Anna Karenina	Literary Fiction	Leo Tolstoy	Russian
A Confession	Religious Autobiography	Leo Tolstoy	Russian

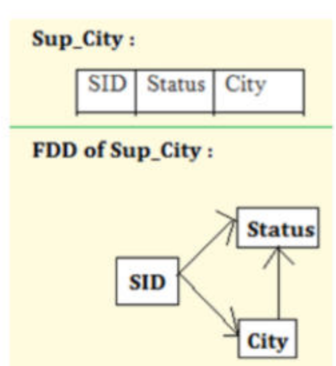
3NF: Example

- Example:
  Sup_City(SID, Status, City) (already in 2NF)

Sup_City:

SID	Status	City
S1	30	Delhi
S2	10	Karnal
S3	40	Rohtak
S4	30	Delhi

SID: Primary Key



- Redundancy?
    - Status
- Anomaly?
    - Yes

Functional Dependencies:
SID → Status,
SID → City,
City→ Status
Transitive Dependency :
SID → Status
{As SID → City and City → Status}

Post Normalization

SC:

SID	City
S1	Delhi
S2	Karnal
S3	Rohtak
S4	Delhi

SID: Primary Key

CS:

City	Status
Delhi	30
Karnal	10
Rohtak	40

City: Primary Key

The above two relations SC and CS are
- Lossless Join
- 3NF
- Dependency Preserving

3NF: Example #2

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

A relational schema R is in 3NF if for every FD X → A associated with R either

- $A \subseteq X$ (ie. the FD is trivial) or

- X is a superkey of R or
- A is part of some key (not just superkey)

3NF: Redundancy

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF (*J : s_ID, L: i_ID, K : dept_name*)
  - R = (J, L, K)
  - F = {JK → L, L → K}

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

- Repetition of information (for example, the relationship l_1, k_1)
  - (*i_ID, dept_name*)
- Need to use null values (for example, to represent the relationship l_2, k_2 where there is no corresponding value for *J*)
  - (*i_ID, dept_name*) if there is no separate relation mapping instructors to departments

Week 6 Lecture 2

Class	BSCCS2001
Created	@October 12, 2021 6:11 PM
Materials	
Module #	27
Type	Lecture
Week #	6

Relational Database Design (part 7)

3NF Decomposition: Motivation

- 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, call Third Normal Form (3NF)
  - Allows some redundancy (with resultant problems, as seen above)
  - But functional dependencies can be checked on individual relations without computing a join
  - There is always lossless-join, dependency-preserving decomposition into 3NF

3NF Decomposition: 3NF Definition

- A relational schema ***R*** is in 3NF if for every FD ***X*** → ***A*** associated with R either
  - $A \subseteq X$ (that is, the FD is trivial) or
  - X is a superkey of R or
  - A is part of some candidate key (not just superkey)
- A relation is 3NF is naturally in 2NF

3NF Decomposition: Testing for 3NF

- **Optimization:** Need to check only FDs in F, need not check all FDs in F^+
- Use attribute closure to check for each dependency $\alpha \to \beta$, if α is the 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
  - Decomposition into 3NF can be done in polynomial time

3NF Decomposition: Algorithm

- **Given:** relation R, set F of functional dependencies
- **Find:** decomposition of R into a set of 3NF relation R_i
- **Algorithm:**
  - Eliminate redundant FDs, resulting in a canonical cover F_c of F
  - Create a relation $R_i = XY$ for each FD X → Y in F_c
  - If the key K of R does not occur in any relation R_i, create one more relation $R_i = K$

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 - 1

return $(R_1, R_2, ..., R_i)$

3NF Decomposition: Algorithm

- Upon decomposition:
  - Each relation schema R_i is in 3NF
  - Decomposition is ...
    - Dependency Preserving
    - Lossless Join
- Prove these properties

3NF Decomposition: Example

- Relation schema:

  cust_banker_branch = *(<u>customer_id</u>, <u>employee_id</u>, branch_name, type)*
- The functional dependencies for this relation schema are:
  - *customer_id, employee_id* → *branch_name, type*

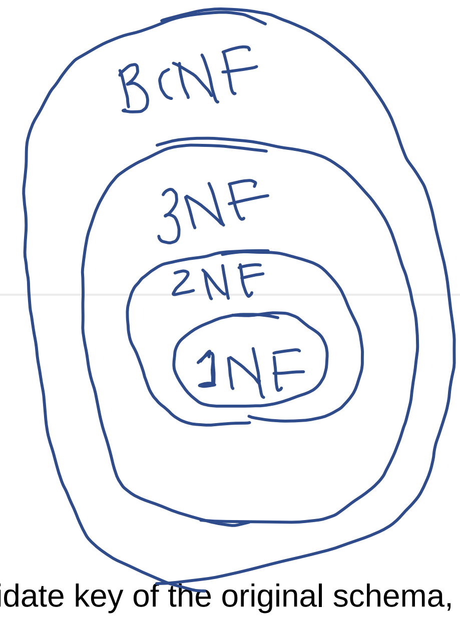
- *employee_id* → *branch_name*
- *customer_id, branch_name* → *employee_id*

- We first compute a canonical cover
  - *branch_name* is irrelevant in the RHS of the 1^{st} dependency
  - No other attribute is irrelevant, so we get $F_c =$

    customer_id, employee_id → *type*

    employee_id → *branch_name*

    customer_id, branch_name → *employee_id*



- The **for** loop generates the following 3NF schema:

  (customer_id, employee_id, type)

  (employee_id, branch_name)

  (customer_id, branch_name, employee_id)
  - Observing that *(customer_id, employee_id, type)* contains a candidate key of the original schema, so no further relation schema needs be added
- At the end of for loop, detect and delete schemas, such as *(employee_name, 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)

BCNF Decomposition: BCNF Definition

LHS of every func^n! depend mst be a key

- A relation schema R is in BCNF with respect to a set of F of FDs if for all FDs 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 (that is, $\beta \subseteq \alpha$)
  - α is a superkey for R

4NF ⇒ No multivalued dependency

BCNF Decomposition: Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
  - Compute α^+ (the attribute closure of α), and
  - 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 cause a violation in BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either
- However, simplified test using only F is incorrect when testing a relation in a decomposition of R
  - Consider R = (A, B, C, D, E) with F = {A → B, BC → D}
    - Decompose R into 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 → D in F^+ shows R_2 is not in BCNF

BCNF Decomposition: Testing for BCNF Decomposition

- To check if a relation R_i in a decomposition of R is in BCNF
  - Either test R_i for BCNF w.r.t. 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 — α 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 R_i and R_i violates BCNF
- We use above dependency to decompose R_i

BCNF Decomposition: Testing Dependency Preservation: Using Closure Set of FD (Exp. Algo.):

Consider the example given below, we will apply both the algorithms to check preservation and will discuss the results

- **R** (*A, B, C, D*)

  F = {A → B, B → C, C → D, D → A}
- Decomposition: **R1**(A, B) **R2**(B, C) **R3**(C, D)
  - A → B is preserved on table R1
  - B → C is preserved on table R2
  - C → D is preserved on table R3
  - We have to check whether the one remaining FD: D → A is preserved or not

R1	R2	R3
F_1={**A** → *AB*, **B** → *BA*}	F_2={**B** → *BC*, **C** → *CB*}	F_3={**C** → *CD*, **D** → *DC*}

- $F' = F_1 \cup F_2 \cup F_3$
- Checking for: D → A in F'^+
  - D → C (from R3), C → B (from R2), B → A (from R1) : D → A (by transitivity)

    Hence, all the dependencies are preserved

BCNF Decomposition: Testing Dependency Preservation: Using Closure of Attributes (Poly. Algo.)

- R(ABCD) :. F = {A → B, B → C,C → D, D → A}
- *Decomp* = {*AB, BC, CD*}
- On projections:

R1	R2	R3
F1 **A → B**	F2 **B → C**	F3 **C → D**

  In this algo F1, F2, F3 are not the closure sets, rather the sets of dependencies directly applicable on R1, R2, R3 respectively

- Need to check for: A → B, B → C, C → D, **D** → **A**
- (D) + /F1 = D. (D) + /F2 = D. (D) + /F3 = D. So, **D** → **A** could not be preserved
- In the previous method we saw the dependency was preserved
- In reality also it is preserved
- Therefore, the polynomial time algorithm may not work in case of all examples
- To prove preservation, Algo 2 is sufficient but not necessary whereas Algo 1 is both sufficient as well as necessary

  NOTE: This difference in result can occur in any example where a functional dependency of one decomposed table uses another functional dependency in its closure which is not applicable on any of the decomposed table because of the absence of all attributes in the table

BCNF Decomposition: Algorithm

- For all dependencies A → B in F^+, check if A is a superkey
  - By using attribute closure
- If not, then ...
  - Choose a dependency in F^+ that breaks the BCNF rules, say A → B
  - Create R1 = AB
  - Create R2 = (R — (B — A))
  - **NOTE:** $R1 \cap R2 = A$ and A → AB (=R1), so this is lossless decomposition
- Repeat for R1 and R2
  - By defining $F1^+$ to be all the dependencies in F that contain only attributes in R1
  - Similarly $F2^+$

result := {R};

done := false;

compute F^+;

while (**not** done) **do**

    if (there is 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 $\alpha \rightarrow \beta$ is not in F^+

                    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 BCNF and decomposition is lossless-join

BCNF Decomposition: Example

- R = (A, B, C)

  F = {A → B

  B → C}

  Key = {A}
- R is not in BCNF (B → C but B is not superkey)
- Decomposition
  - $R_1 = (B, C)$
  - $R_2 = (A, B)$

BCNF Decomposition: Example #2

- *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 (*course_id, sec_id, semester, year, building, room_number, capacity, time_slot_id*)
    - 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

BCNF Decomposition: Dependency Preservation

- It is not always possible to get a BCNF Decomposition that is dependency preserving
- R = (J, K, L)

  F = {JK → L

  L → K}

  Two candidate keys = JK and JL
- R is not in BCNF
- Any decomposition of R will fail to preserve

  JK → L

  This implies that testing for JK → L requires a join

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

S#	3NF	BCNF
1.	It concentrates on Primary Key	It concentrates on Candidate Key
2.	Redundancy is high as compared to BCNF	0% redundancy
3.	It preserves all the dependencies	It may not preserve the dependencies
4.	A dependency $X \to Y$ is allowed in 3NF if X is a super key or Y is a part of some key	A dependency $X \to Y$ is allowed if X is a super key

Q) Z(PQRS) F = {P → QRS, Q → R, RS → P} Normalizⁿ form? 3NF, 2NF, 1NF but not BCNF

CK ⇒ everything not in RHS ⇒ Take one by one $P^+ = \{QRS\}$, $Q^+ = \{R\}$, $R^+ = \{R\}$, $(QR^+) = \{QR\}$, $(QS^+) = \{QSRP\}$, $(RS^+) = \{RSPQ\}$
= P, (QS), (RS)

⇒ P Attr = P, Q, R, S   Non-prime atts = ∅ ⇝ since null set, no transitive dependency (NPA ⟶ NPA) so in 3NF (3NF already contains 1 & 2NF)

F = {P → QRS, Q → R, RS → P} for BCNF, LHS of every FD mst be a key (P, RS r keys but Q isn't, so BCNF fails

Convert the question so it passes BCNF? ① Find out which FD(s) violate the BCNF condⁿ Take it, find the LHS closure, name it Z_1, derive Z_2 from it without violating 3 laws of lossless decomposition

Q → R ⇒ (Q^+) = QR — Z_1 ⇒ F_1 = {Q → R}

Z_2 = PSQ ⇒ F_2 = {P → QS}

to pass union law of lossless decomposⁿ

to add common elem (not R, cuz it isn't determining anything)

Multi-valued dependency ⇒ $\alpha \to\to \beta$ ① atleast 3 atts in table ② 2 mst be independent of each other

↳ B & C r ind if B ↛ C & C ↛ B

R(αβγδ)

Keep Same

③ $t_1[\alpha] = t_2[\alpha]$

(45 27 eg)

exchange vals b/w any 2 tuples having same 'α' val & if those 2 exchanged valued rows are already existing in table, MVD holds & $\alpha \to\to \beta$ (take sch vals having $t_1[\beta] \neq t_2[\beta]$)

this should hold treue for all such tuples having $t_1[\alpha] = t_2[\alpha]$

* If $\alpha \to\to \beta$, then $\alpha \to\to (R-\beta)$, eg R(αβγδ), if $\alpha \to\to \beta$, then $\alpha \to\to (R-\beta)$, i.e. $\alpha \to\to \gamma, \delta$

* If there're only 2 att in a relⁿ R(α, β) then it's in 1NF, 2NF, 3NF, BCNF & 4NF

Q) A(TUVW) F = {W → UT, UV → W, V → T, W → U} A decomposed into B(TV) & C(UVW) ① Lossless? ✓ ② Dependency preserved? ✗ ③ B & C in BCNF ✗ ④ B in BCNF? ✓

① B ∪ C = (TVUW) = A ✓
② B ∩ C = V ≠ ∅ ✓
③ B ∩ C = V → T ⇒ V is CK in 'B' ✓
⇒ Lossless

$(U^+_{parent}) = U$ $(V^+_{parent}) = V$

F_B = {V → T}, F_C = {W → U, UV → W} ⇒ $F^+ = F_B \cup F_C$ = {V → T, W → U, VU → W}

(W^+_{par}) = {WUT}
≠ W^+(child) = {WU}

⇒ The 1st FD isn't preserved directly, find cover of atts from parent FD & union FD & compare
(Just check if cover of LHS in parent FD = cover of that LHS att from child (union) FD)

B is in BCNF cuz only 2 atts. tnt
C isn't in BCNF cuz W isn't a key ⇒ C(UVW), F_C = {W → U, UV → W}
⇒ (UV^+_C) = {UVW} = C, (W^+_C) = {WU} ≠ C (no V) ⇒ not a CK

Week 6 Lecture 3

Class	BSCCS2001
Created	@October 13, 2021 11:36 AM
Materials	
Module #	28
Type	Lecture
Week #	6

Relational Database Design (part 8)

Case Study

Library Information System (LIS)

We are asked to design a relational DB schema for a Library Information System (LIS) of an Institute

- The specification document of the LIS has already been shared with you
- We include key points from the Specs
- We carry out the following tasks in the module:
  - Identify the Entity sets with attributes
  - Identify the relationships
  - Build the initial set of relational schema
  - Refine the set of schema with FDs that hold on them
  - Finalize the design of the schema
- The coding of various queries in SQL, based on these schema are left as exercise

LIS Specs Excerpts

- An institute library has 200,000+ books and 1,000+ members
- Books are regularly issued by members on loan and returned after a period
- The library needs an LIs to manage the books, the members and the issue-term process

- Every book has
  - title
  - author (in case of multiple authors, only the first author is mentioned)
  - publisher
  - year of publication
  - ISBN number (which is unique for the publication)
  - accession number (which is the unique number of the copy of the book in the library)

  There may be multiple copies of the same book in the library

---

There are 4 categories of members of the library:

- Undergraduate students
- Post-graduate students
- Research scholars
- Faculty members

Every student has ...

- Name
- Roll number
- Department
- Gender
- Mobile number
- Date of Birth
- Degree
  - Undergrad
  - Grad
  - Doctoral

Every faculty has ...

- Name
- Employee ID
- Department
- Gender
- Mobile number
- Date of Joining

Library also issues a unique membership number to every member

Every member has a max quota for the number of books he/she can issue for the maximum duration allowed to her/him

Currently, these are set as:

- Each undergraduate student can issue up to 2 books for 1 month duration
- Each postgraduate student can issue up to 4 books for 1 month duration
- Each research scholar can issue up to 6 books for 3 months duration
- Each faculty member can issue up to 10 books for 6 months duration

---

The library has the following rules for issue:

- A book may be issued to a member if it is not already issued to someone else (trivial)
- A book may not be issued to a member if another copy of the same book is already issued to the same member

- No issue will be done to a member if at the time of issue one or more of the books issued by the member has already exceeded its duration of issue
- No issue will be allowed also if the quota is exceeded for the member
- It is assumed that the name of every author or member has two parts
  - First name
  - Last name

LIS Specs Excerpts: Queries

LIS should support the following operations / queries:

- Add / Remove members, categories of members, books
- Add / Remove / Edit quota for a category of member, duration for a category of member
- Check if the library has a book given its title (part of title should match)
  - If yes, title, author, publisher, year and ISBN should be listed
- Check if the library has a book given its author
  - If yes, title, author, publisher, year and ISBN should be listed
- Check if a copy of a book (given its ISBN) is available with the library for issue
  - All accession numbers should be listed with issued or available information
- Check the available (free) quota of a member
- Issue a book to a member
  - This should check for the rules of the library
- Return a book from a member
- and so on ...

LIS Entity Sets: books

- Every book has title, author (in case of multiple authors, only the first author is maintained), published, year of publication, ISBN number (which is unique for the publication) and accession number (which is the unique number of the copy of the book in the library)
  - There may be multiple copies of the same book in the library
- Entity set:
  - **books**
- Attributes:
  - title
  - author_name (composite);
  - publisher
  - year
  - ISBN_no
  - accession_no

LIS Entity Sets: students

- Every student has name, roll number, department, gender, mobile number, date of birth and degree (undergrad, grad, doctoral)
- Entity Set:
  - **students**
- Attributes
  - member_no - is unique
  - name (composite)

- roll_no - is unique
- department
- gender
- mobile_no - may be null
- dob
- degree

LIS Entity Sets: faculty

- Every faculty has name, employee id, department, gender, mobile number and date of joining
- Entity Set:
  - **faculty**
- Attributes:
  - member_no - is unique
  - name (composite)
  - id - is unique
  - department
  - gender
  - mobile_no - may be null
  - doj

LIS Entity Sets: members

- Library also issues a unique membership number to every member
- There are 4 categories of members of the library:
  - undergraduate students
  - post graduate students
  - research scholars
  - faculty members
- Entity Set:
  - **members**
- Attributes:
  - member_no
  - member_type (takes a value in ug, pg, rs or fc)

LIS Entity Sets: quota

- Every member has a max quota for the number of books she / he can issue for the max duration allowed to her / him
- Currently, these are set as:
  - Each undergraduate student can issue up to 2 books for 1 month duration
  - Each postgraduate student can issue up to 4 books for 1 month duration
  - Each research scholar can issue up to 6 books for 3 months duration
  - Each faculty member can issue up to 10 books for 6 months duration
- Entity Set:
  - **quota**
- Attributes:
  - member_type
  - max_books

  - max_duration

LIS Entity Sets: staff

- Thought not explicitly stated, library would have staffs to manage the LIS
- Entity Set:
  - **staff**
- Attributes: (speculated — to ratify from customer)
  - name (composite)
  - id - is unique
  - gender
  - mobile_no
  - doj

LIS Relationships

- Books are regularly issued by members on loan and returned after a period
- The library needs an LIS to manage the books, the members and the issue-return process
- Relationship
  - **book_issue**
- Involved Entity Sets
  - **students / faculty / members**
    - member_no
  - **books**
    - accession_no
- Relationship Attribute
  - doi — date of issue
- Type of relationship
  - Many-to-one from **books**

LIS Relational Schema

- **books** (title, author_fname, author_lname, publisher, year, ISBN_no, accession_no)
- **book_issue** (members, accession_no, doi)
- **members** (member_no, member_type)
- **quota** (member_type, max_books, max_duration)
- **students** (member_no, student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
- **faculty** (member_no, faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)
- **staff** (staff_fname, staff_lname, id, gender, mobile_no, doj)

LIS Schema Refinement: books

- **books** (title, author_fname, author_lname, publisher, year, ISBN_no, accession_no)
  - ISBN_no → title, author_fname, author_lname, publisher, year
  - accession_no → ISBN_no
  - Key: accession_no
- Redundancy of book information across copies
- Good to normalize:
  - **book_catalogue** (title, author_fname, author_lname, publisher, year, ISBN_no)

        - ISBN_no → title, author_fname, author_lname, publisher, year
        - Key: ISBN_no
    - **book_copies** (ISBN_no, accession_no)
        - accession_no → ISBN_no
        - Key: accession_no
- Both in BCNF
- Decomposition is lossless join and dependency preserving

LIS Schema Refinement: book_issue

- book_issue (member_no, accession_no, doi)
    - member_no, accession_no → doi
    - Key: members, accession_no
- In BCNF

LIS Schema Refinement: quota

- **quota** (member_type, max_books, max_duration)
    - member_type → max_books, max_duration
    - Key: member_type
- In BCNF

LIS Schema Refinement: members

- **members** (member_no, member_type)
    - member_no → member_type
    - Key: menber_no
    - Value constraint on member_type
        - ug, pg or rs: if the member is a student
        - fc: if the member is a faculty
    - In BCNF
    - How to determine the member_type?

LIS Schema Refinement: students

- **students** (member_no, student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
    - roll_no → student_fname, student_lname, department, gender, mobile_no, dob, degree
    - member_no → roll_no
    - roll_no → member_no
    - 2 Keys: roll_no | member_no
- In BCNF
- Issues:
    - member_no is needed for issue / return queries
        - It is unnecessary to have student's details with that
    - member_no may also come from faculty relation
    - member_type is needed for issue / return queries
        - This is implicit in degree — not explicitly given

LIS Schema Refinement: faculty

- **faculty** (member_no, faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)

- id → faculty_fname, faculty_lname, department, gender, mobile_no, doj
    - id → member_no
    - member_no → id
    - 2 Keys: id | member_no
- In BCNF
- Issues:
    - member_no is needed for the issue / return queries
        - It is unnecessary to have faculty details with that
    - member_no may also come from **student** relation
    - member_type is needed for issue / return queries
        - This is implicit by the fact that we are in faculty relation

LIS Schema Refinement: Query

- Consider a query:
    - Get the name of the member who has issued the book having accession number = 162715
        - If the member is a student

        ```
        SELECT student_fname as First_Name, student_lname as Last_Name
        FROM students, book_issue
        WHERE accession_no = 162715 AND book_issue.member_no = students.member_no;
        ```

        - If the member is a faculty

        ```
        SELECT faculty_fname as First_Name, faculty_lname as Last_Name
        FROM faculty, book_issue
        WHERE accession_no = 162715 AND book_issue.member_no = faculty.member_no;
        ```

    - Which query to fire!?

LIS Schema Refinement: members

There are 4 categories of members: ug students, grad students, research scholars and faculty members

This leads to the following specialization relationships

- Consider the entity set **members** of a library and refine:
    - Attributes:
        - member_no
        - member_class — 'student' or 'faculty', used to choose table
        - member_type — ug, pg, rs, fc, ...
        - roll_no (if member_class — 'student', else null)
        - if (if member_class — 'faculty', else null)
- We can the exploit some hidden relationship:
    - student IS A members
    - faculty IS A members
- Types of relationship
    - One-to-one

LIS Schema Refinement: Query

- Consider the access query again:
    - Get the name of the member who has issued the book having accession number = 162715

```
SELECT
((SELECT faculty_fname as First_Name, faculty_lname as Last_Name
FROM faculty
WHERE member_class = 'faculty' AND members.id = faculty.id)
UNION
(SELECT student_fname as First_Name, student_lname as Last_Name
FROM students
WHERE member_class = 'student' AND members.roll_no = students.roll_no))
FROM members, book_issue
WHERE accession_no = 162715 AND book_issue.member_no = members.member_no;
```

LIS Schema Refinement: members

- **members** (member_no, member_class, member_type, roll_no, id)
  - member_no → member_type, member_class, roll_no, id
  - member_type → member_class
  - Key: member_no

LIS Schema Refinement: students

- **students** (student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
  - roll_no → student_fname, student_lname, department, gender, mobile_no, dob, degree
  - Keys: roll_no
  - Note:
    - member_no is no longer used
    - member_type and member_class are set in **members** from degree at the time of creation of a new record

LIS Schema Refinement: faculty

- **faculty** (faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)
  - id → faculty_fname, faculty_lname, department, gender, mobile_no, doj
  - Keys: id
  - Note:
    - member_no is no longer used
    - member_type and member_class are set in **members** at the time of creation of a new record

LIS Scheme Refinement: Final

- **book_catalogue** (title, author_fname, author_lname, publisher, year, ISBN_no)
- **book_copies** (ISBN_no, accession_no)
- book_issue (member_no, accession_no, doi)
- **quota** (member_type, max_books, max_duration)
- **members** (member_no, member_class, member_type, roll_no, id)
- **students** (student_fname, student_lname, roll_no, department, gender, mobile_no, dob, degree)
- **faculty** (faculty_fname, faculty_lname, id, department, gender, mobile_no, doj)
- **staff** (staff_fname, staff_lname, id, gender, mobile_no, doj)

Week 6 Lecture 4

Class	BSCCS2001
Created	@October 13, 2021 6:02 PM
Materials	
Module #	29
Type	Lecture
Week #	6

Relational Database Design (part 9)

MVD: Multi-valued Dependency

- Persons (Man, Phones, Dog_Like)

Person :			**Meaning of the tuples**
Man(M)	**Phones(P)**	**Dogs_Like(D)**	Man M have phones P, and likes the dogs D.
M1	P1/P2	D1/D2	M1 have phones P1 and P2, and likes the dogs D1 and D2.
M2	P3	D2	M2 have phones P3, and likes the dog D2.
Key : MPD			

There are no non-trivial FDs because all attributes are combined forming Candidate Key, that is, MDP

In the above relation, 2 multi-valued dependencies exist:

- Man $\twoheadrightarrow$ Phones
- Man $\twoheadrightarrow$ Dog_Like

A man's phone is independent of the phone they like

But, after converting the above relation in Single Valued Attribute, each of a man's phone appears with each of the dogs they like in all combinations

Post 1NF Normalization

Man(M)	Phones(P)	Dogs_Likes(D)
M1	P1	D1
M1	P2	D2
M2	P3	D2
M1	P1	D2
M1	P2	D1

MVD

- If two or more independent relations are kept in a single relation, then Multi-valued Dependency is possible
- For example, let there be 2 relations:
  - **Student (SID, Sname)** *where* **(SID → Sname)**
  - **Course (CID, Cname)** *where* **(CID → Cname)**
- There is no relation defined between Student and Course
- If we kept them in a single relation named **Student_Course**, then MVD will exist because of m:n Cardinality
- If two or more MVDs exist in a relation, then while converting into SVAs, MVD exists

Student:

SID	Sname
S1	A
S2	B

Course:

CID	Cname
C1	C
C2	B

SID	Sname	CID	Cname
S1	A	C1	C
S1	A	C2	B
S2	B	C1	C
S2	B	C2	B

2 MVDs exist:
1. SID →→ CID
2. SID →→ Cname

- Suppose we record names of the children, and phone numbers for the instructors
  - *inst_child (ID, child_name)*
  - *inst_phone (ID, phone_number)*
- If we were to combine these schema 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

MVD: Definition

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$

- The multi-valued dependency $\alpha \twoheadrightarrow \beta$ holds on R if in any legal relation $r(R)$, for all pairs of 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:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$

$$t_3[\beta] = t_1[\beta]$$

$$t_3[R - \beta] = t_2[R - \beta]$$

$$t_4[\beta] = t_2[\beta]$$

$$t_4[R - \beta] = t_1[R - \beta]$$

Example: A relation of university courses, the books recommended for the course, and the lecturers who will be teaching the course:

- **course $\twoheadrightarrow$ book**
- **course $\twoheadrightarrow$ lecturer**

Test: course $\twoheadrightarrow$ book

Course	Book	Lecturer	Tuples
AHA	Silberschatz	John D	t1
AHA	Nederpelt	William M	t2
AHA	Silberschatz	William M	t3
AHA	Nederpelt	John D	t4
AHA	Silberschatz	Christian G	
AHA	Nederpelt	Christian G	
OSO	Silberschatz	John D	
OSO	Silberschatz	William M	

will hold cuz 'β' same

- Let R be a relation schema with a set of attributes that are partitioned into 3 non-empty subsets

  Y, Z, W

- We say that Y $\twoheadrightarrow$ Z (Y multidetermines Z) if and only if for all possible relations $r(R)$ $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$

  $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$

- Note that since the behaviour of Z and W are identical it follows that

  Y $\twoheadrightarrow$ Z if Y $\twoheadrightarrow$ W

In our example:

- ID $\twoheadrightarrow$ child_name
- ID $\twoheadrightarrow$ 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 $\twoheadrightarrow$ Z
- Indeed we have (in above notation) $Z_1 = Z_2$

  The claim follows

MVD: Use

- We use multi-valued dependencies in 2 ways:
  - To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
  - To specify the constraints on the set of legal relations
  - We shall thus concern ourselves only with the relations that satisfy a given set of functional and multivalued dependencies
- If a relation r fails satisfy a given multivalued dependency, we can construct a relation r' that does satisfy the multivalued dependency by adding tuples to r

MVD: Theory

	Name	Rule
C-	Complementation	If $X \twoheadrightarrow Y$, then $X \twoheadrightarrow (R - (X \cup Y))$.
A-	Augmentation	If $X \twoheadrightarrow Y$ and $W \supseteq Z$, then $WX \twoheadrightarrow YZ$.
T-	Transitivity	If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow (Z - Y)$.
	Replication	If $X \rightarrow Y$, then $X \twoheadrightarrow Y$ but the reverse is not true.
	Coalescence	If $X \twoheadrightarrow Y$ and there is a W such that $W \cap Y$ is empty, $W \rightarrow Z$ and $Y \supseteq Z$, then $X \rightarrow Z$.

- A MVD **X** ↠ **Y** in **R** is called a trivial MVD is
  - **Y** is a subset of X (X ⊇ Y) or
  - X ∪ Y = R
    - Otherwise, it is a non-trivial MVD and we have to repeat values redundantly in the tuples

---

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

  That is, every functional dependency is also a multi-valued dependency
- The closure D^+ of D is the set of all functional and multi-valued dependencies logically implied by D
  - We can compute D^+ from D, using the formal definitions of functional dependencies and multi-valued dependencies
  - We can manage with such reasoning for very simple multi-valued 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

Decomposition of 4NF

Fourth Normal Form (4NF)

- A relation schema R is in 4NF w.r.t. a set D of functional and multi-valued dependencies if for all multi-valued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \twoheadrightarrow \beta$ is trivial (that is, $\beta \subseteq \alpha$ 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 is R_i is the set of D_i consisting of
  - All functional dependencies in D^+ that include only attributes of R_i
  - All multivalued dependencies of the form

    $\alpha \twoheadrightarrow (\beta \cap R_i)$

    where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+

4NF Decomposition Algorithm

- For all dependencies $A \twoheadrightarrow B$ in D^+, check if A is a superkey
    - By using attribute closure
- if not, then
    - Choose a dependency in F^+ that breaks the 4NF rules, say $A \twoheadrightarrow B$
    - Create R1 = A B
    - Create R2 = (R - (B - A))
    - Note: $R1 \cap R2 = A$ and $A \twoheadrightarrow AB(= R1)$, so this is lossless decomposition
- Repeat for R1 and R2
    - By defining $D1^+$ to be all dependencies in F that contain only attributes in R1
    - Similarly $D2^+$

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 R_i in result that is not in 4NF) then

    begin

      let $\alpha \twoheadrightarrow \beta$ be a non-trivial multi-valued 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

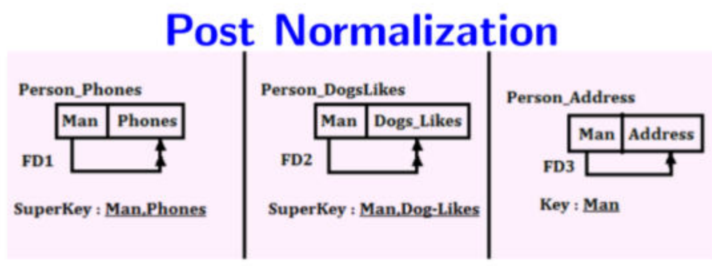
  else done := true;

NOTE: each R_i is in 4NF and decomposition is lossless-join

4NF Decomposition: Example

- Example:
    - **Person_Modify(Man(M), Phones(P), Dog_Likes(D), Address(A))**
    - FDs:
        - ▷ FD1 : Man $\twoheadrightarrow$ Phones
        - ▷ FD2 : Man $\twoheadrightarrow$ Dogs_Like
        - ▷ FD3 : Man $\rightarrow$ Address
    - Key = MPD
    - All dependencies violate 4NF

Man(M)	Phones(P)	Dogs_Likes(D)	Address(A)
M1	P1	D1	49-ABC,Bhiwani(HR.)
M1	P2	D2	49-ABC,Bhiwani(HR.)
M2	P3	D2	36-XYZ,Rohtak(HR.)
M1	P1	D2	49-ABC,Bhiwani(HR.)
M1	P2	D1	49-ABC,Bhiwani(HR.)



In the above relations for both the MVD's – **'X' is Man**, which is again not the super key, but as ***X ∪ Y = R*** i.e. (Man & Phones) together make the relation.
So, the above MVD's are trivial and in FD 3, Address is functionally dependent on Man, where **Man** is the key in **Person_Address**, hence all the three relations are in 4NF.

- R =(A, B, C, G, H, I)
  F = A $\twoheadrightarrow$ B
  B $\twoheadrightarrow$ HI
  CG $\twoheadrightarrow$ H
- R is not in 4NF since $A \twoheadrightarrow B$ 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)
    - $A \twoheadrightarrow B$ and $B \twoheadrightarrow$ HI $\rightarrow$ A $\twoheadrightarrow$ HI, (MVD transitivity), and
    - and hence A $\twoheadrightarrow$ 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)

Week 6 Lecture 5

Class	BSCCS2001
Created	@October 14, 2021 12:08 AM
Materials	
Module #	30
Type	Lecture
Week #	6

Relational Database Design (part 10)

Database Design Process

Design Goals

- Goal for a relational DB design:
  - BCNF / 4NF
  - 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 DB)
- Even if we had a dependency preserving decomposition, using SQL we could not be able to efficient test a functional dependency whose left hand side is not a key

Further Normal Forms

- Further NFs:
  - Elementary Key Normal Form (EKNF)

- Essential Tuple Normal Form (ETNF)
- Join Dependencies and Fifth Normal Form (5NF)
- Sixth Normal Form (6NF)
- Domain/Key Normal Form (DKNF)

- Join dependencies generalize multi-valued 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 exist
- Hence rarely used

Overall DB 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 (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 optimization
- 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

    and a 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
  - **Course (course_id, title, ...)**
  - **Prerequisite (course_id, prereq)**
- **Alternative #1:** Use denormalized relation containing attributes of course as well as prereq with all above attributes: **Course (course_id, title, prereq, ...)**
  - faster lookup
  - extra space and extra execution time for updates
  - extra coding work for programmers and possibility of error in extra code
- **Alternative #2:** Use a materialized view defined as **Course** ⋈ **Prerequisite**
  - Benefits and drawbacks same as above, except no extra coding work for programmers and avoids possible errors

Other Design Issues

- Some aspects of DB design are not caught by normalization
- Examples of bad DB design, to be avoided:

Instead of earnings (company_id, year, amount), use

- earnings_2004, earnings_2005, earnings_2005, 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 **crosstab**, where values for one attribute become column names
    - Used in spreadsheets, and in data analysis tools

LIS Example for 4NF

- Consider a different version of relation ***book_catalogue*** having the following attributes:
    - *book_title*
    - *book_catalogue, author_lname*: A *book_title* may be associated with more than one author
- **book_title** {*book_title, author_fname, author_lname, edition*}

book_catalogue

book_title	author_fname	author_lname	edition
DBMS CONCEPTS	BRINDA	RAY	1
DBMS CONCEPTS	AJAY	SHARMA	1
DBMS CONCEPTS	BRINDA	RAY	2
DBMS CONCEPTS	AJAY	SHARMA	2
JAVA PROGRAMMING	ANITHA	RAJ	5
JAVA PROGRAMMING	RIYA	MISRA	5
JAVA PROGRAMMING	ADITI	PANDEY	5
JAVA PROGRAMMING	ANITHA	RAJ	6
JAVA PROGRAMMING	RIYA	MISRA	6
JAVA PROGRAMMING	ADITI	PANDEY	6

- Since, the relation has no FDs, it is already in BCNF
- However, the relation has 2 non-trivial MVDs

  book_title $\rightarrow$ *{author_fname, author_lname}* and *book_title* $\rightarrow$ *edition*

  Thus, it is not in 4NF
- Non-trivial MVDs must be decomposed to convert it into a set of relationsin 4NF

- We decompose **book_catalogue** into **book_author** and **book_edition** because:
    - **book_author** has trivial MVD

      book_title $\twoheadrightarrow$ {author_fname, author_lname}
    - **book_edition** has trivial MVD

      book_title $\twoheadrightarrow$ edition

book_title	author_fname	author_lname
DBMS CONCEPTS	BRINDA	RAY
DBMS CONCEPTS	AJAY	SHARMA
JAVA PROGRAMMING	ANITHA	RAJ
JAVA PROGRAMMING	RIYA	MISRA
JAVA PROGRAMMING	ADITI	PANDEY

Figure: **book_author**

book_title	edition
DBMS CONCEPTS	1
DBMS CONCEPTS	2
JAVA PROGRAMMING	5
JAVA PROGRAMMING	6

Figure: **book_edition**

Temporal Databases

- Some data may be inherently historical because they include time-dependent / time-varying data, such as:
    - Medical Records
    - Judicial Records
    - Share prices
    - Exchange rates
    - Interest rates
    - Company profits
    - etc.
- The desire to model such data means that we need to store not only the respective value but also an associated data or a time period for which the value is valid
- Typical queries expressed informally might include:
    - Give me last month's history of the Dollar-Pound Sterling exchange rate
    - Give me the share prices of the NYSE on October 17, 1996
- Temporal DB provides a uniform and systematic way of dealing with historical data

Temporal Data

- Temporal data have an association time interval during which the data is valid
- A snapshot is the value of the data at a particular point in time
- In practice, DB engineers may add start and end time attributes to relations
- For example, course (course_id, course_title) is replaced by

  course (course_id, course_title, start, end)

- Constraint: no 2 tuples can have overlapping valid times and are hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
  - For example: student transcript should refer to the course information at the time the course was taken

Temporal Database Theory

- **Model of Temporal Domain:** Single-dimensional linearly ordered which may be ...
  - Discrete or dense
  - Bounded or unbounded
  - Single dimensional or multi-dimensional
  - Linear or non-linear
- **Timestamp Model**
- **Temporal ER model** by adding valid time to
  - Attributes: address of an instructor at different points in time
  - Entities: time duration when a student entity exists
  - Relationships: time during which a student attended a course
  - But no accepted standard
- **Temporal Functional Dependency Theory**
- **Temporal Logic**
- **Temporal Query Language:**
  - TQuel [1987]
  - TSQL2 [1995]
  - SQL/Temporal [1996]
  - SQL/TP [1997]

Modeling Temporal Data: Uni / Bi Temporal

- There are 2 different aspects of time in temporal DBs
  - **Valid Time:** Time period during which a fact is true in the real world, provided to the system
  - **Transaction Time:** Time period during which a fact is stored in the DB, based on transaction serialization order and is the timestamp generated automatically by the system
- Temporal Relation is one where each tuple has associated time; either valid time or transaction time or both associated with it
  - **Uni-Temporal Relations:** Has one axis of time, either Valid Time or Transaction Time
  - Bi-Temporal Relations: Has both axis of time — Valid time and Transaction time
    - It includes Valid Start Time, Valid End Time, Transaction Start Time, Transaction End Time

Modeling Temporal Data: Example

- **Example**
  - Let's see an example of a person, John:
    - John was born on April 3, 1992 in Chennai
    - His father registered his birth after 3 days on April 6, 1992
    - John did his entire schooling and college in Chennai
    - He got a job in Mumbai and shifted to Mumbai on June 21, 2015
    - He registered his change of address only on Jan 10, 2016

John's Data in Non-Temporal DB

Date	Real world event	Address
April 3, 1992	John is born	
April 6, 1992	John's father registered his birth	Chennai
June 21, 2015	John gets a job	Chennai
Jan 10, 2016	John registers his new address	Mumbai

- In a non-temporal DB, John's address is entered as Chennai from 1992
- When he registers his new address in 2016, the DB gets updated and the address field now shows his Mumbai address
- The previous Chennai address details will not be available
- So, it will be difficult to find out exactly when he was living in Chennai and when he moved to Mumbai

---

Uni-Temporal Relation (Adding Valid Time to John's Data)

Name	City	Valid From	Valid Till
John	Chennai	April 3, 1992	June 20, 2015
John	Mumbai	June 21, 2015	∞

- The valid time temporal DB contents look like this:
  `Name, City, Valid From, Valid Till`
- Johns father registers his birth on 6th April 1992, a new DB entry is made:
  `Person (John, Chennai, 3-Apr-1992, ∞ )`
- On January 10, 2016 John reports his new address in Mumbai:
  `Person (John, Mumbai, 21-June-2015, ∞ )`
  - The original entry is updated:
    `Person (John, Chennai, 3-Apr-1992, 20-June-2015)`

Bi-Temporal Relation (John's Data Using Both Valid And Transaction Time)

Name	City	Valid From	Valid Till	Entered	Superseded
John	Chennai	April 3, 1992	June 20, 2015	April 6, 1992	Jan 10, 2016
John	Mumbai	June 21, 2015	∞	Jan 10, 2016	∞

- The database contents look like this:
  `Name, City, Valid From, Valid Till, Entered, Superseded`
- Johns father registers his birth on 6th April 1992:
  `Person(John, Chennai, 3-Apr-1992, ∞ , 6-Apr-1992, ∞ )`

- On January 10, 2016 John reports his new address in Mumbai:

  `Person(John, Mumbai, 21-June-2015, ∞ , 10-Jan-2016, ∞ )`

  - The original entry is updated as:

    `Person(John, Chennai, 3-Apr-1992, 20-June-2015, 6-Apr-1992 , 10-Jan-2016)`

Modeling Temporal Data: Summary

- **Advantages**
  - The main advantages of this bi-temporal relations is that it provides historical and roll back information
    - **Historical information** — Valid time
    - **Rollback information** — Transaction time
  - For example, you can get the result of a query on John's history, like: Where did John live in the year 2001?
    - The result for this query can be got with the valid time entry
    - The transaction time entry is important to get the rollback information
- **Disadvantages**
  - More storage
  - Complex query processing
  - Complex maintenance including backup and recovery

Week 8 Lecture 1

Class	BSCCS2001
Created	@October 24, 2021 6:21 PM
Materials	
Module #	36
Type	Lecture
Week #	8

Algorithms and Data Structures: Algorithms and Complexity Analysis

Algorithms and Programs

- **Algorithms**
    - An algorithm is a finite sequence of well-defined, computer-implementable (optional) instructions, typically solves a class of specific problems or to perform a computation
    - Algorithms are always un-ambiguous and are used as specifications for performing calculations, data processing, automated reasoning and other tasks
    - An algorithm must terminate
- **Program**
    - A computer program is a collection of instructions that can be executed by a computer to perform a specific task
    - A computer program is usually written by a computer programmer in a programming language
    - A program implements an algorithm
    - A program may or may not terminate
        - For example → An Operating System

Analysis of Algorithms

- **Why?**
    - Set the motivation for algorithm analysis

    - Why analyze?
- **What?**
    - Identify what all needs to be analyzed
    - What to analyze?
- **How?**
    - Learn the techniques for analysis
    - How to analyze?
- **Where?**
    - Understand the scenarios for application
    - Where to analyze?
- **When?**
    - Realize your position for seeking the analysis
    - When to analyze?

Why analyze?

Practical reasons:

- Resources are scarce
- Greed to do more with less
- Avoid performance bugs

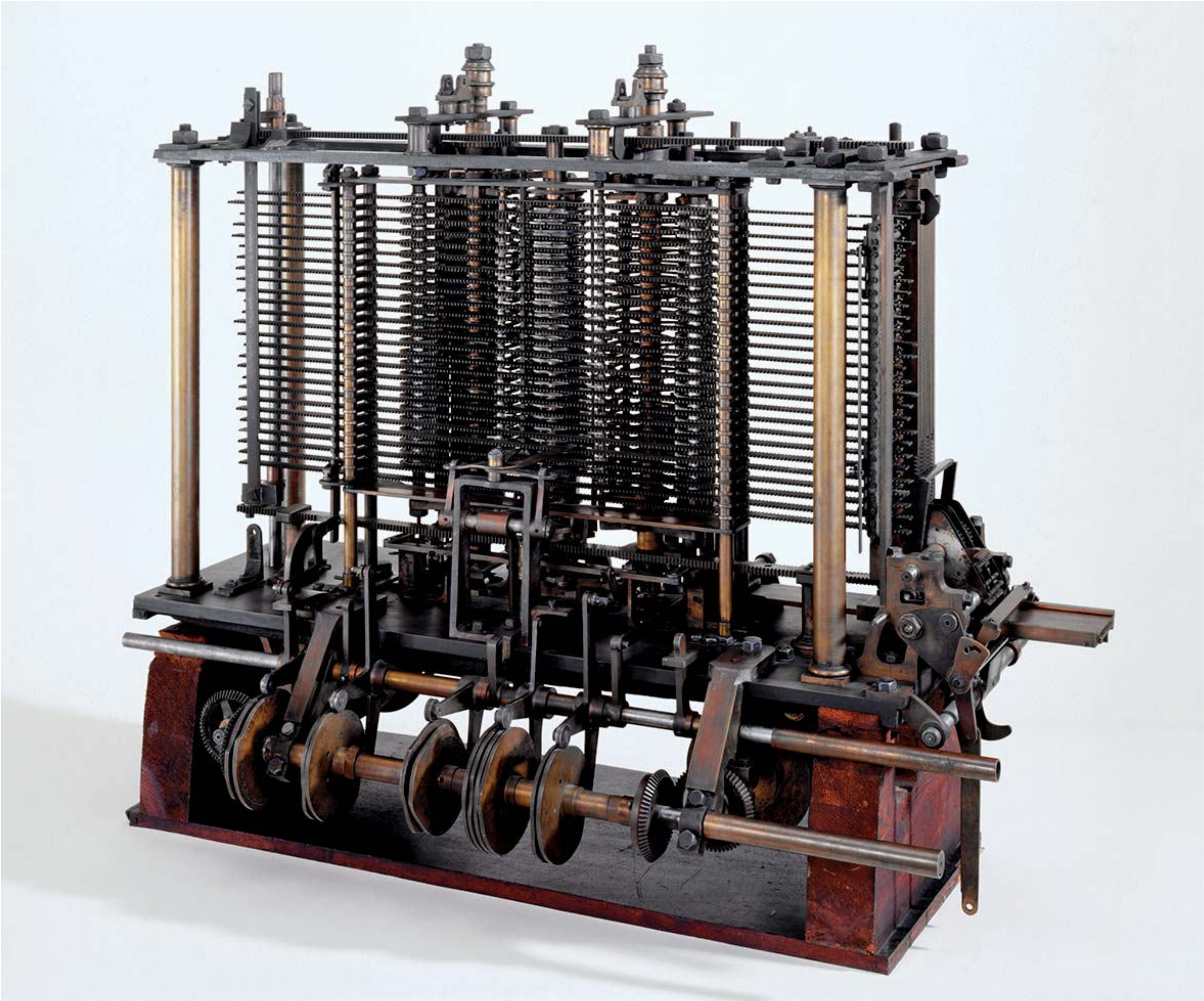
Core Issues:

- Predict performance
    - How much time does binary search take?
- Compare algorithms
    - How quick is Quicksort? (heh)
- Provide guarantees
    - Size notwithstanding, Red-Black tree inserts in $O(log\ n)$
- Understand theoretical basis
    - Sorting by comparison cannot do better than $\Omega(n\ log\ n)$

What to analyze?

Core Issues → Cannot control what we cannot measure

- Time
    - Story starts with the Analytical Engine



  - Most common analysis factor
  - Representative of various related analysis factors like Power, Bandwidth, Processors
  - Supported by Complexity Classes
- Space
  - Widely exported
  - Important for hand-held devices
  - Supported by Complexity Classes

What to analyze?

- Sum of natural numbers

```
int sum(int n)
{
  int s = 0;
  for(; n > 0; --n)
    s = s + n;
  return s;
}
```

- Time $T(n) = n$ (additions)
- Space $S(n) = 2$ (n, s)

What to analyze?

- Find a character in a string

will calc everytime the loop runs

```
int find(char *str, char c)
{
  for(int i = 0; i < strlen(str); ++i)
    if(str[i] == c)
      return i;
```

n $n = n^2$

$\Rightarrow n^2 + n = O(n^2)$

n

```
  return 0;
}
n = strlen(str)
```

- Time $T(n) = n$ (compare) + $n * T(strlen(str)) \approx n + n^2 \approx n^2$
- Space $S(n) = 3$ (str, c, i)

What to analyze?

- Minimum of a sequence of numbers

```
int min(int a[], int n)
{
  for(int i = 0; i < n; ++i)
    cin >> a[i];

  int t = a[--n];
  for(; n > 0; --n)
    if(t < a[--n])
      t = a[n];
  return t;
}
```

- Time $T(n) = n - 1$ (comparison of value)
- Space $S(n) = n + 3$ (`a[]`'s, n, i, t)

How to analyze?

- Counting model
- Asymptotic model
- Generating functions
- Master Theorem

How to analyze? Counting Models

- **Core Idea** → Total running time = Sum of cost $\times$ frequency for all operations
  - Need to analyze program to determine set of operations
  - Cost depends on machine, compiler
  - Frequency depends on the algorithm, input data
- **Machine Model** → Random Access Machine (RAM) Computing Model
  - Input data & size
  - Operations
  - Intermediate Stages
  - Output data & size

How to analyze? Counting Models

- **Factorial (Recursive)**

```
int fact(int n)
{
  if (n != 0)
    return n * fact(n - 1);
  return 1;
}
```

  - Time $T(n) = n - 1$ (multiplication)
  - Space $S(n) = n + 1$ (n's in recursive calls)
- **Factorial (Iterative)**

```
int fact(int n)
{
  int t = 1;
  for(; n > 0; --n)
    t = t * n;
  return t
}
```

- Time $T(n) = n$ (multiplication)
- Space $S(n) = 2$ (`n, t`)

How to analyze? Asymptotic Analysis

Asymptotic Analysis

- **Core Idea** → Cannot compare actual times; hence, compare Growth or how the time increases with input size
  - Function Approximation (tilde ($\tilde{}$) notation)
  - Common growth functions
    - Big-Oh → $O(.)$
    - Big-Omega → $\Omega(.)$
    - Big-Theta $\Theta(.)$
  - Solve recurrence with Growth functions

How to analyze? Asymptotic Analysis

```
int count = 0;
for(int i = 0; i < N; ++i)
  for(int j = i + 1; i < N; ++j)
    if (a[i] + a[j] == 0)
      count++;
```

Function Approximation (tilde (˜) notation)

Operation	Frequency	Approximation
variable declaration	$N+2$	$\sim N$
assignment statement	$N+2$	$\sim N$
less than compare	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2}N^2$
equal to compare	$\frac{1}{2}N(N-1)$	$\sim \frac{1}{2}N^2$
array access	$N(N-1)$	$\sim N^2$
increment	$\frac{1}{2}N(N-1)$ to $N(N-1)$	$\sim \frac{1}{2}N^2$ to $\sim N^2$

- Estimate running time (or memory) as a function of input size N
- Ignore lower order terms
  - When N is large, terms are negligible
  - When N is small, we don't care

$f(n) \sim g(n)$ **means**

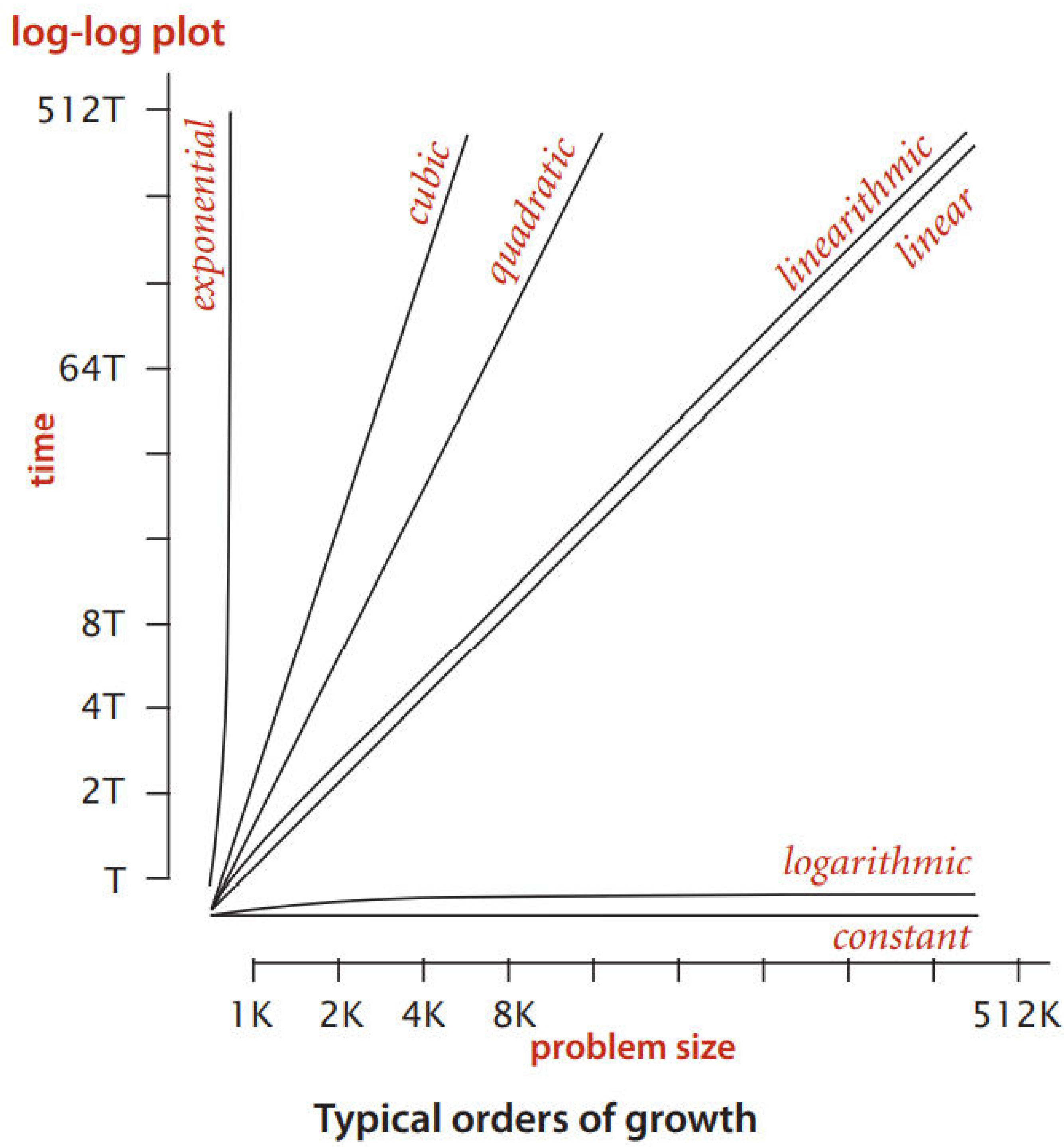
$$\lim_{N\to\infty} \frac{f(n)}{g(n)} = 1$$

How to analyze? Asymptotic Analysis

Common order-of-growth classifications

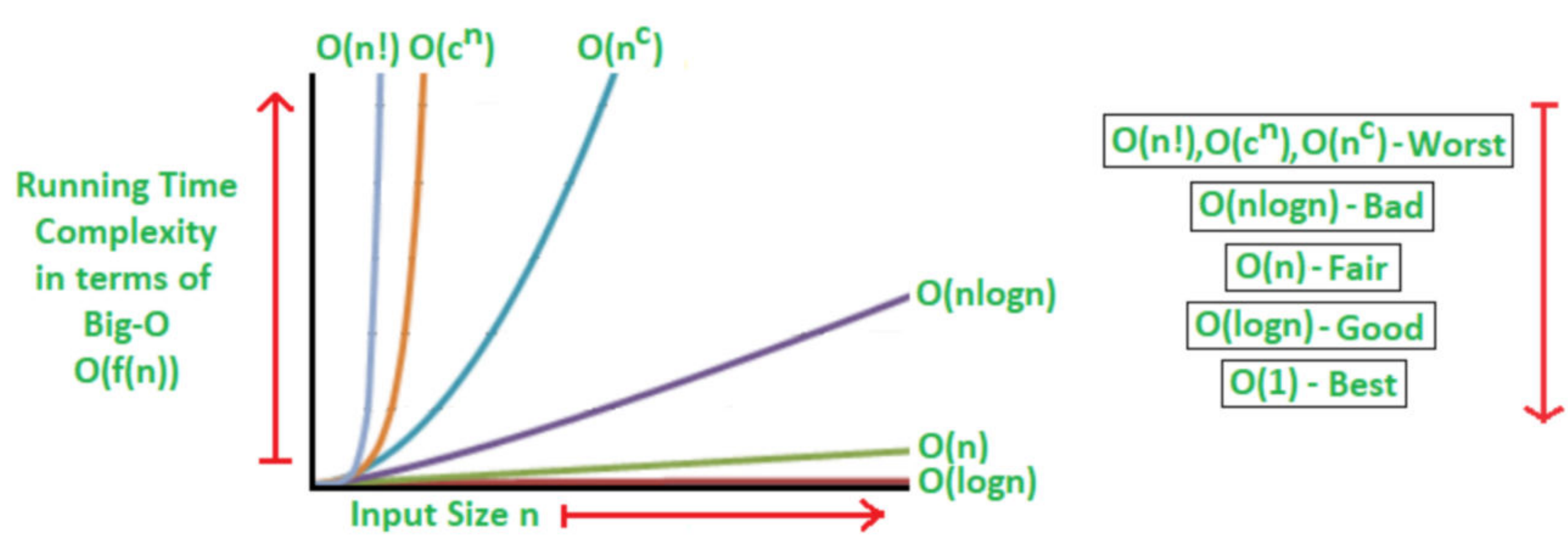
Good news. The set of functions

$1, logN, N, NlogN, N^2, N^3, and, 2^N$ suffices to describe to describe the order of growth of most common algorithms



Source: Page #188, Chapter 1: Fundamentals, Section 1.4, Algorithms (4th Edition) by Robert Sedgewick & Kevin Wayne

How to analyze? Asymptotic Analysis



How to analyze? Asymptotic Analysis

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`   if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      if (a[i] + a[j] == 0)`
		`         cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      for (int k = j+1; k < N; k++)`
		`         if (a[i] + a[j] + a[k] == 0)`
		`            cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhaustive search*	*check all subsets*

Summary of common order-of-growth hypotheses

Source: Page #187, Chapter 1: Fundamentals, Section 1.4, Algorithms (4th Edition) by Robert Sedgewick & Kevin Wayne

Asymptotic Notation

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions:

$$O(g(n)) = \{f(n) \text{: there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n), \text{ for all } n > n_0\}$$

- We use O-notation to give an upper bound on a function, to within a constant factor
- When we say that the running time of A is $O(n^2)$, we mean that there is a function $f(n)$ that is $O(n^2)$ such that for any value of n, no matter what particular input of size n is chosen, the running time on that input is bounded from above by the value $f(n)$
- Equivalently, we mean that the worst-case running time is $O(n^2)$

Where to analyze?

Algorithmic situation

- **Core Idea** → Identify data configurations or scenarios for analysis
  - Best case
    - Minimum running time on an input
  - Worst case
    - Running time guarantees for any input of size n
  - Average case
    - Expected running time for a random input of size n

- Probabilistic case
  - Expected running time of a randomized algorithm
- Amortized case
  - Worst case running time for any sequence of n operations

Big-O Algorithm Complexity Chart

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Source: https://www.bigocheatsheet.com

Week 8 Lecture 2

Class	BSCCS2001
Created	@October 25, 2021 12:10 AM
Materials	
Module #	37
Type	Lecture
Week #	8

Algorithms and Data Structures: Data Structures

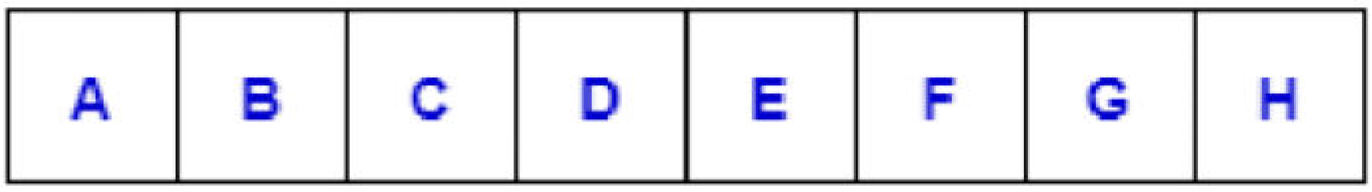
Data Structure

- A data structure specifies the way of organizing and storing in-memory data that enables efficient access and modification of the data
    - Linear Data Structures
    - Non-linear Data Structures
- Most data structure has a container for the data and typical operations that it needs to perform
- For applications relating to data management, the key operations are:
    - Create
    - Insert
    - Delete
    - Find/Search
    - Close
- Efficiency is measured in terms of time and space taken for these operations

Linear Data Structures

- A linear data structure has data elements arranged in linear of sequential manner such that each member element is connected to its previous and next element
- Since data elements are sequentially connected, each element is traversable through a single run

- Examples of linear data structures are Array, Linked List, Queue, Stack, etc

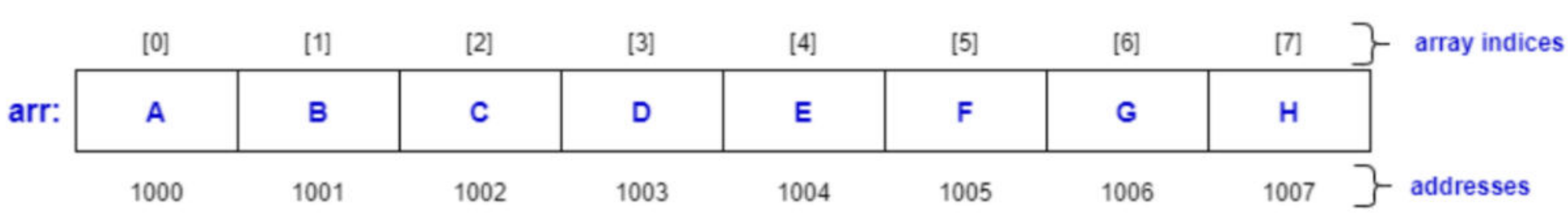


Different examples of linear data structures:

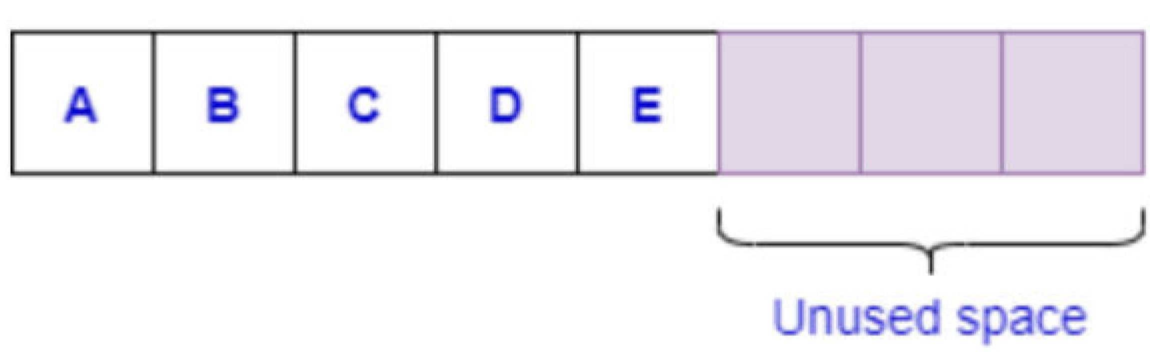
- **Array** → The data elements are stored at contiguous locations in the memory
- **Linked List** → The data elements are not required to be stored in contiguous locations in memory
    - Rather, each element stores a link (a pointer to a reference) to the location of the next element
- **Queue** → It is a FIFO (First In, First Out) data structure
    - The element that has been inserted first in the queue would be removed first
    - Thus, insert and removal of the elements in this take place in the same order
- **Stack** → It is a LIFO (Last In, First Out) data structure
    - The element that has been inserted last in the stack would be removed first
    - Thus, insert and removal of the elements in this take place in the reverse order

Linear Data Structure: Array

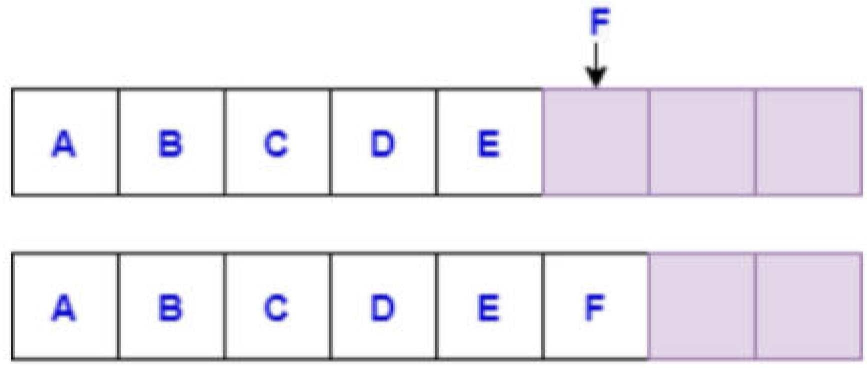
- The elements are stored in contiguous memory locations



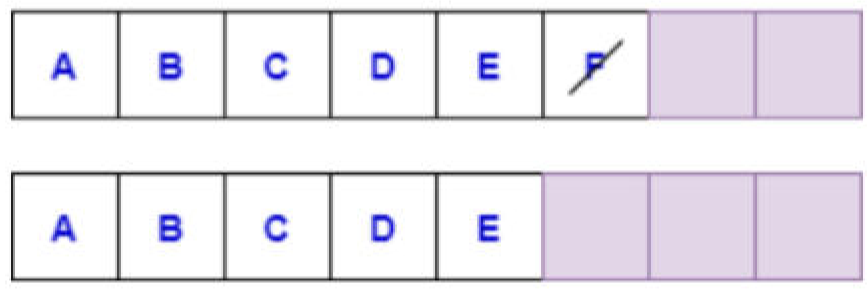
- Simple access using indices
    - For example → let the array name be `arr`, we can access the element at index 5 as `arr[5]`
- Arrays allow random access using its index which is fast (cost of $O(1)$)
    - Useful for operations like sorting, searching
- **<u>Have fixed size, not flexible →</u>** Since we do not know the number of elements to be stored in runtime, If we create it too large then it can be a waste of memory, if we create it too small then some elements may not be accommodated in the array
    - *For example* → Suppose we create an array to store 8 elements
        - However, during the execution of the program only 5 elements are available, which results in the wastage of the memory space



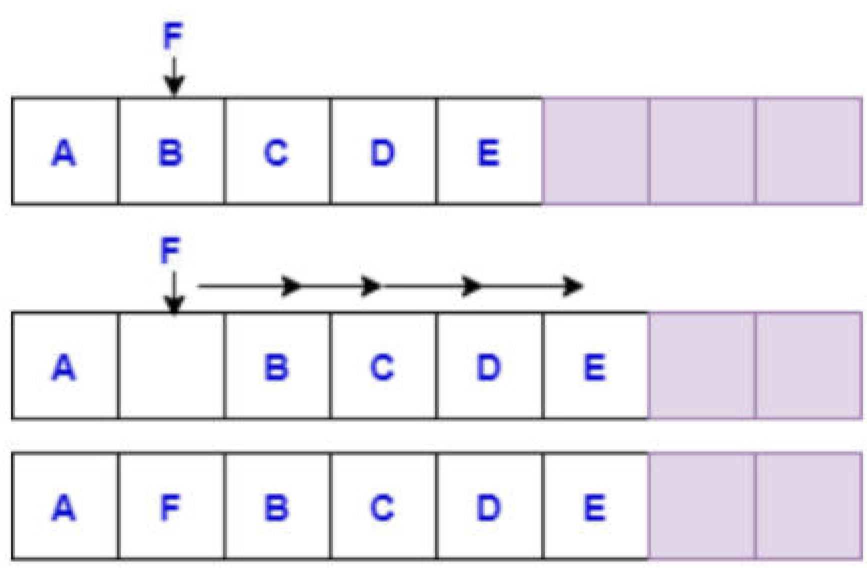
- Insertion and removal of elements from an array are costlier since the memory locations have to be consecutive
    - Insertion or removal of an element from the end of an array is easy
        - Insert at the end:



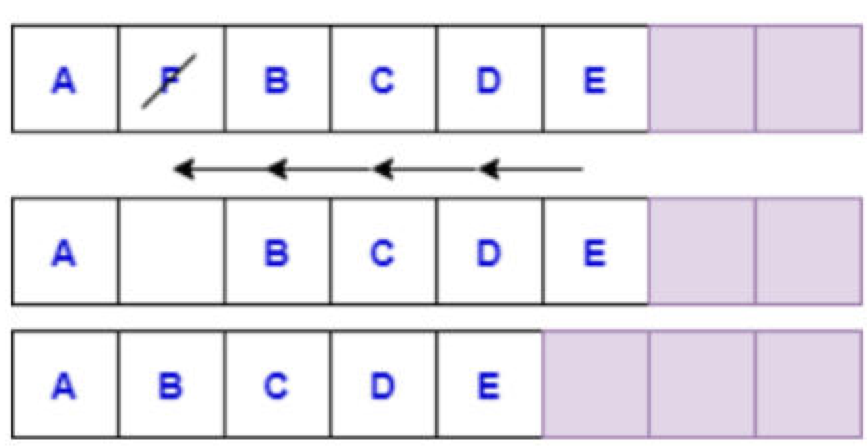
- Remove from the end:



- Insert and remove elements at any arbitrary position is costly (cost of $O(n)$)
  - Insert at any arbitrary position

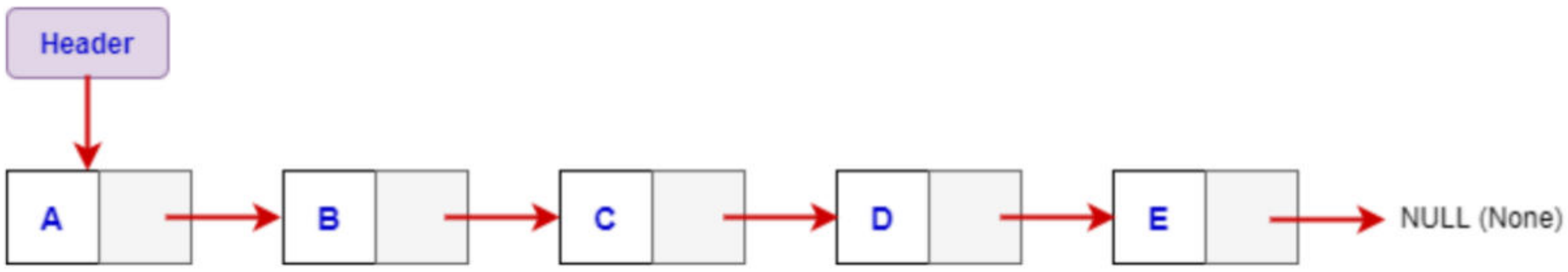


  - Remove from any arbitrary position



Linear Data Structure: Linked List

- Elements are not required to be stored at contiguous memory locations
  - A new element can be stored anywhere in the memory where free space is available
  - Thus, it provides better memory usage than arrays
- For each new element allocated, a link (a pointer or a reference) is created for the new element using which the element can be added to the linked list



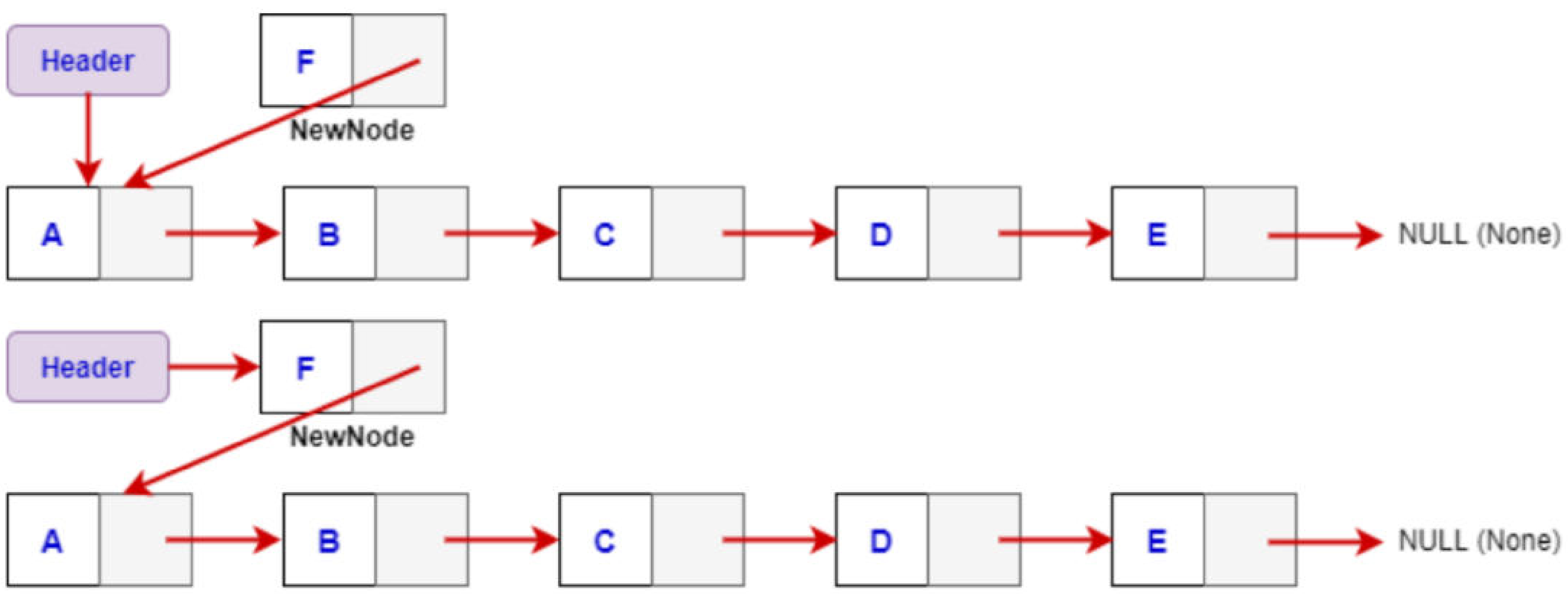
Each element is stored in a node

A node has 2 parts

- **Info** → stores the element
- **Link** → stores the location of the next node

- Header is a link to the first node of the linked list
- **Flexible in size**
  - Size of a linked list grows or shrinks as and when new elements are inserted or deleted
- Random access is not possible in linked lists
  - The elements will have to be access sequentially
- Insertion or Removal of an element at/from any arbitrary position is efficient as none of the elements are required to be moved to new locations
  - Insertion at front

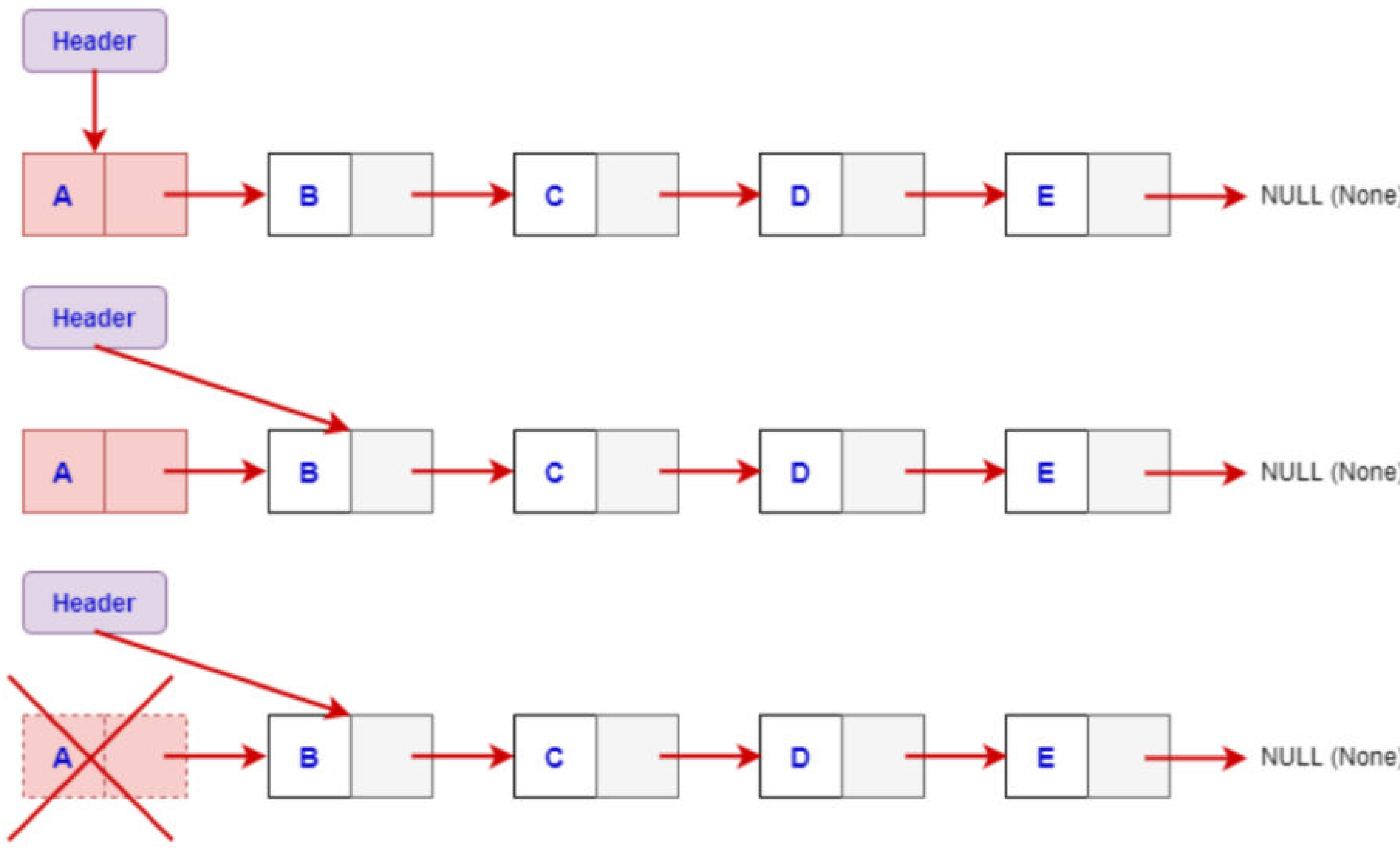
```
1. NewNode.Link = Header
2. Header = NewNode
```



  - Remove from the front

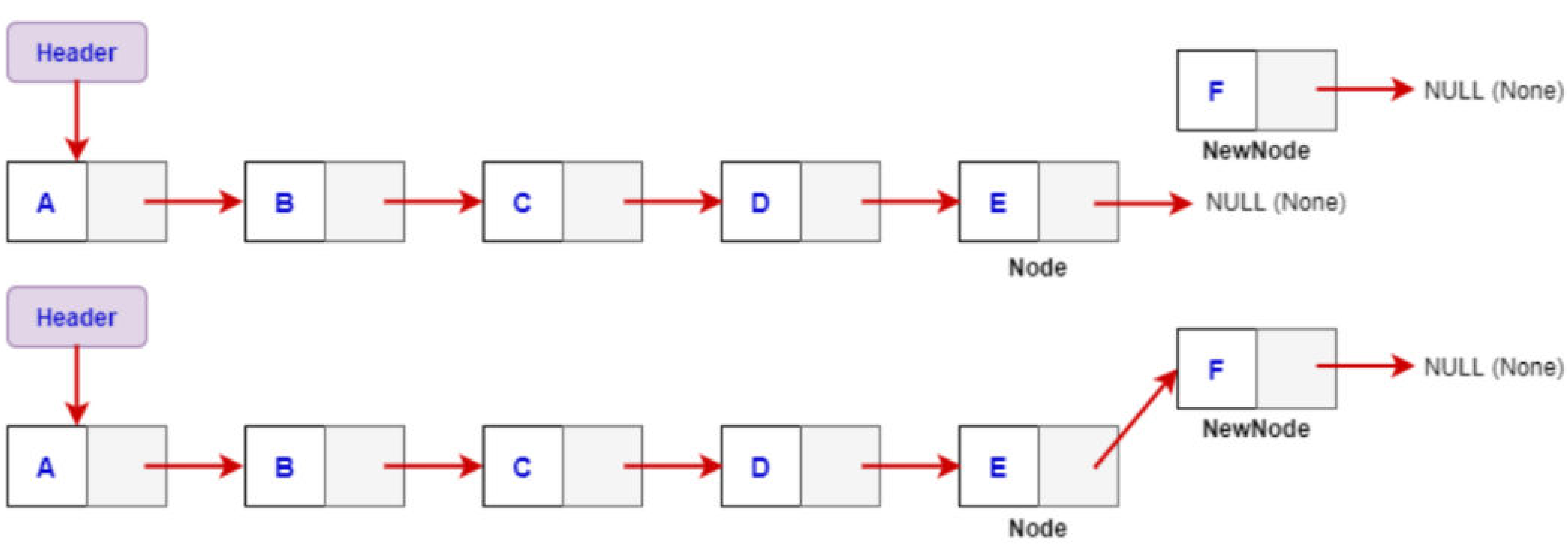
```
1. Temp = Header
2. Header = Header.Link
3. Delete(Temp)
```

Reader.next.next



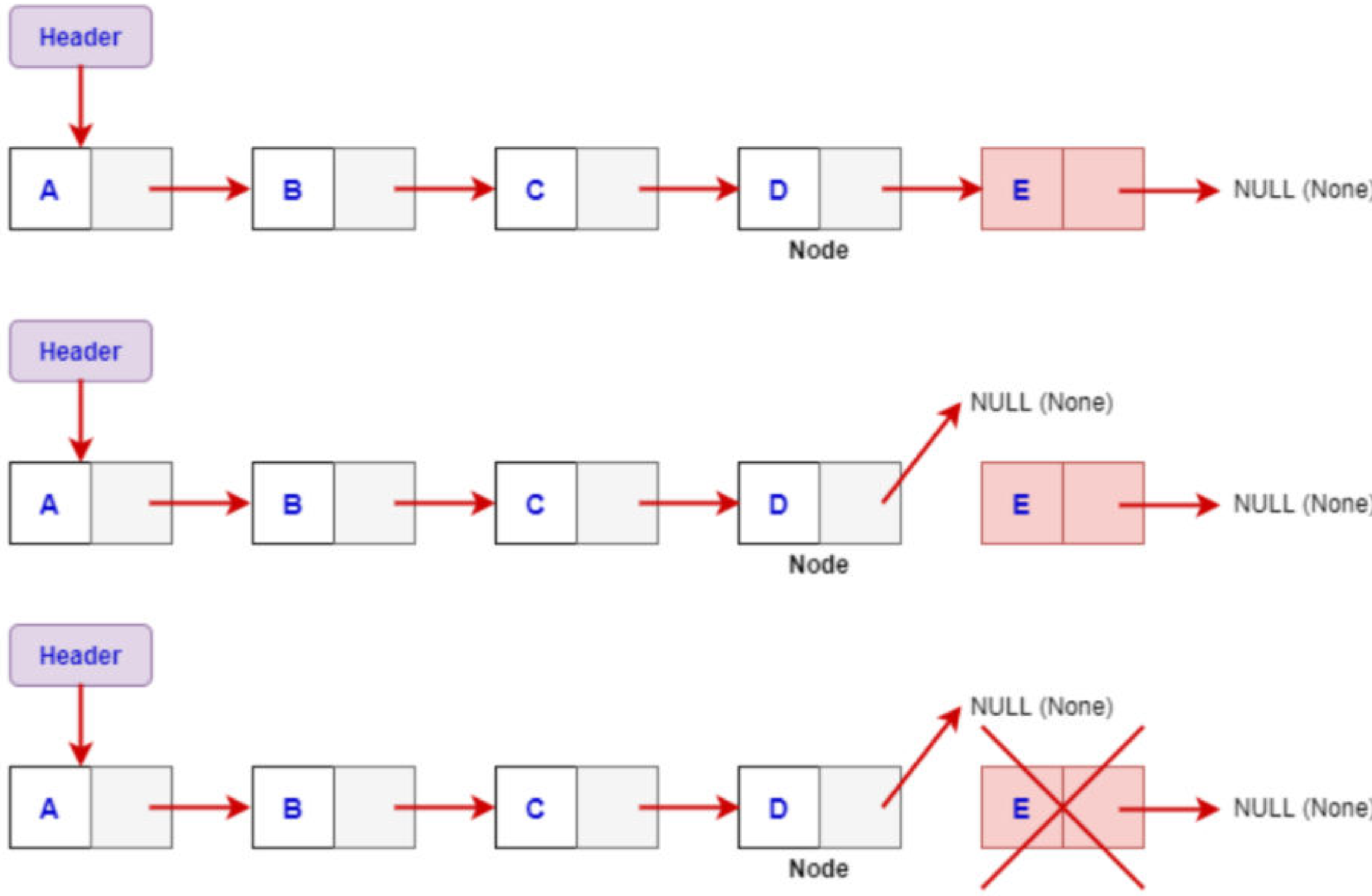
- Insertion at end

```
1. Node.Link = NewNode
```



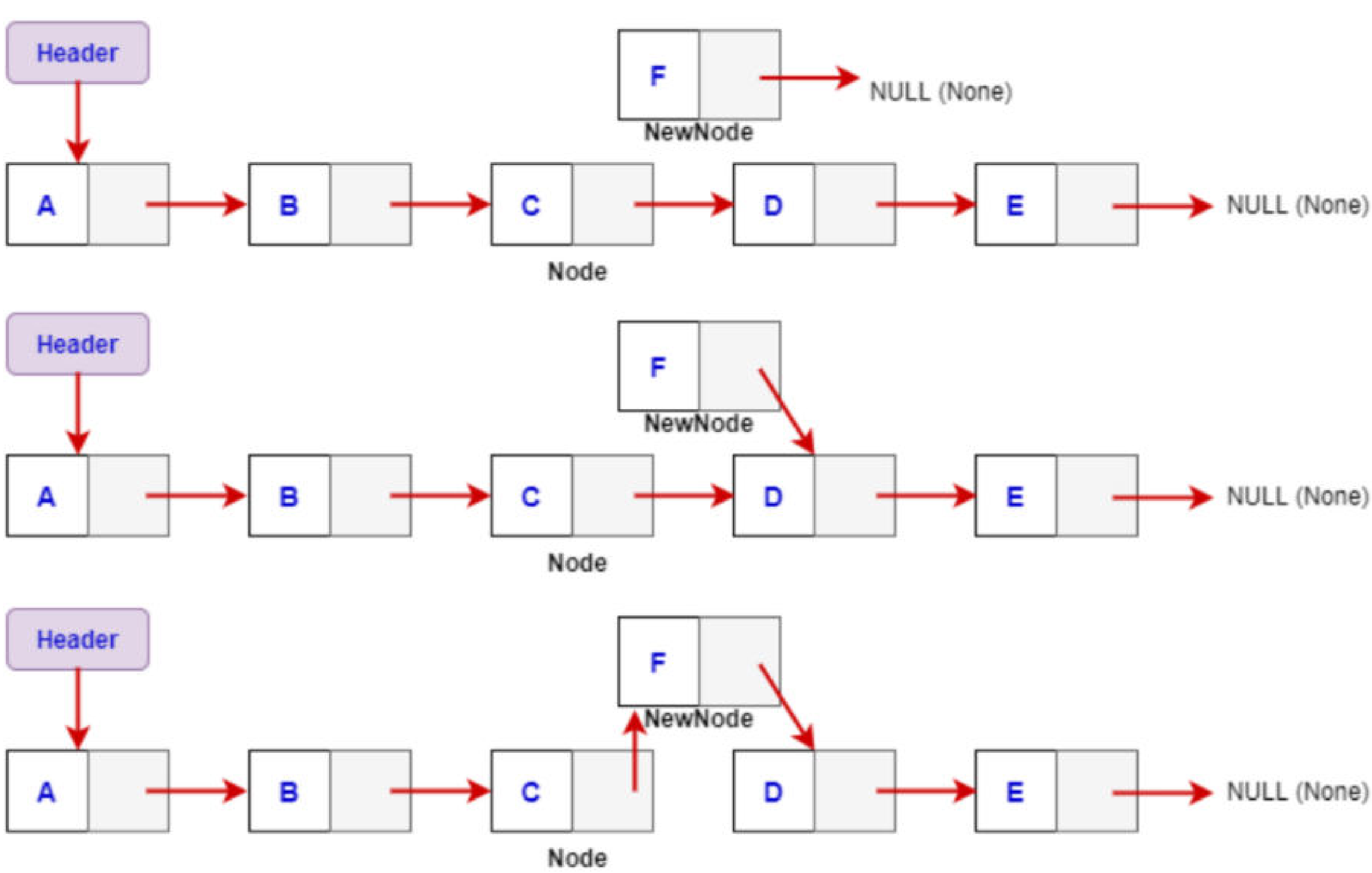
- Remove from end

```
1. Temp = Node.Link
2. Node.Link = NULL
3. Delete(Temp)
```



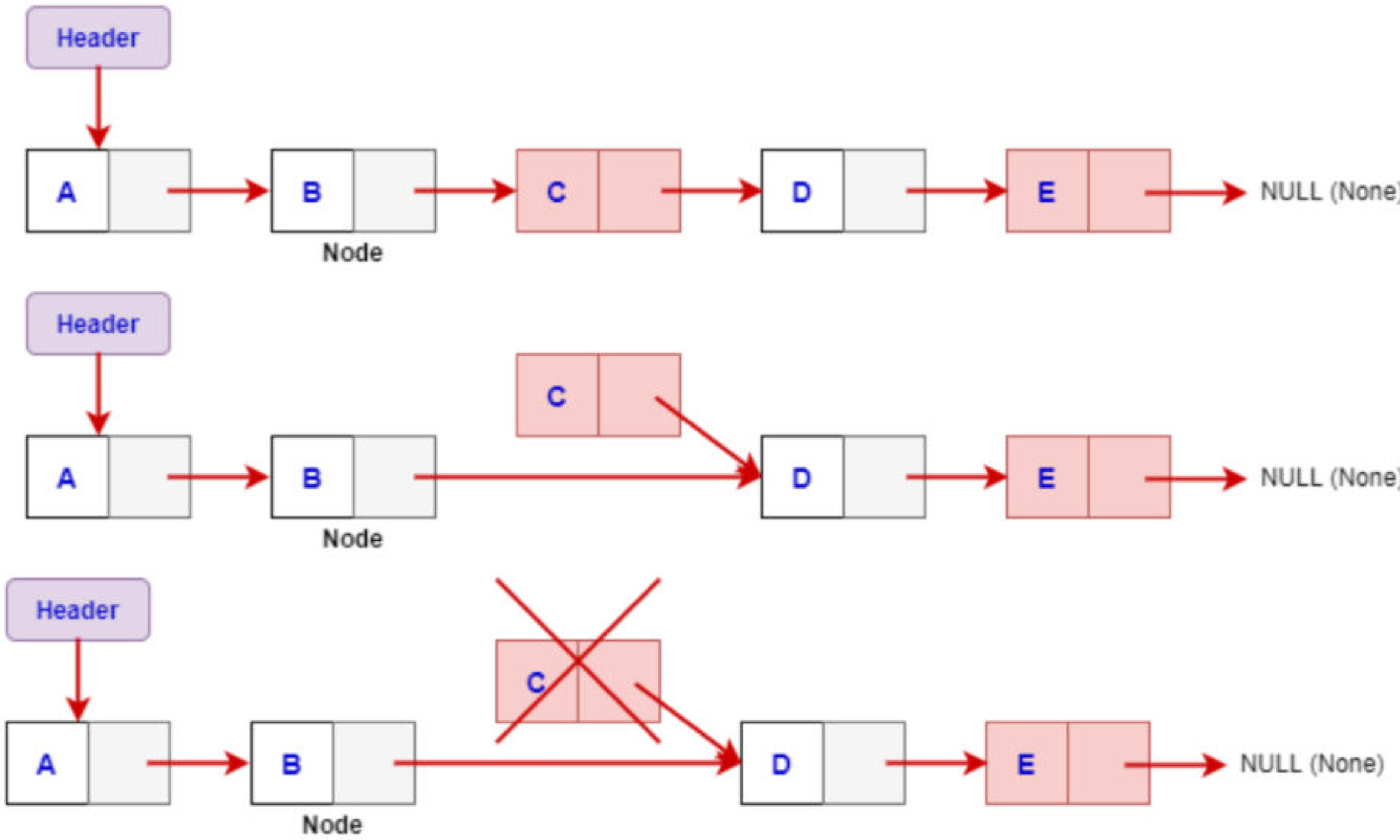
- Insertion at any intermediate position

```
1. NewNode.Link = Node.Link
2. Node.Link = NewNode
```



- Remove from any intermediate position

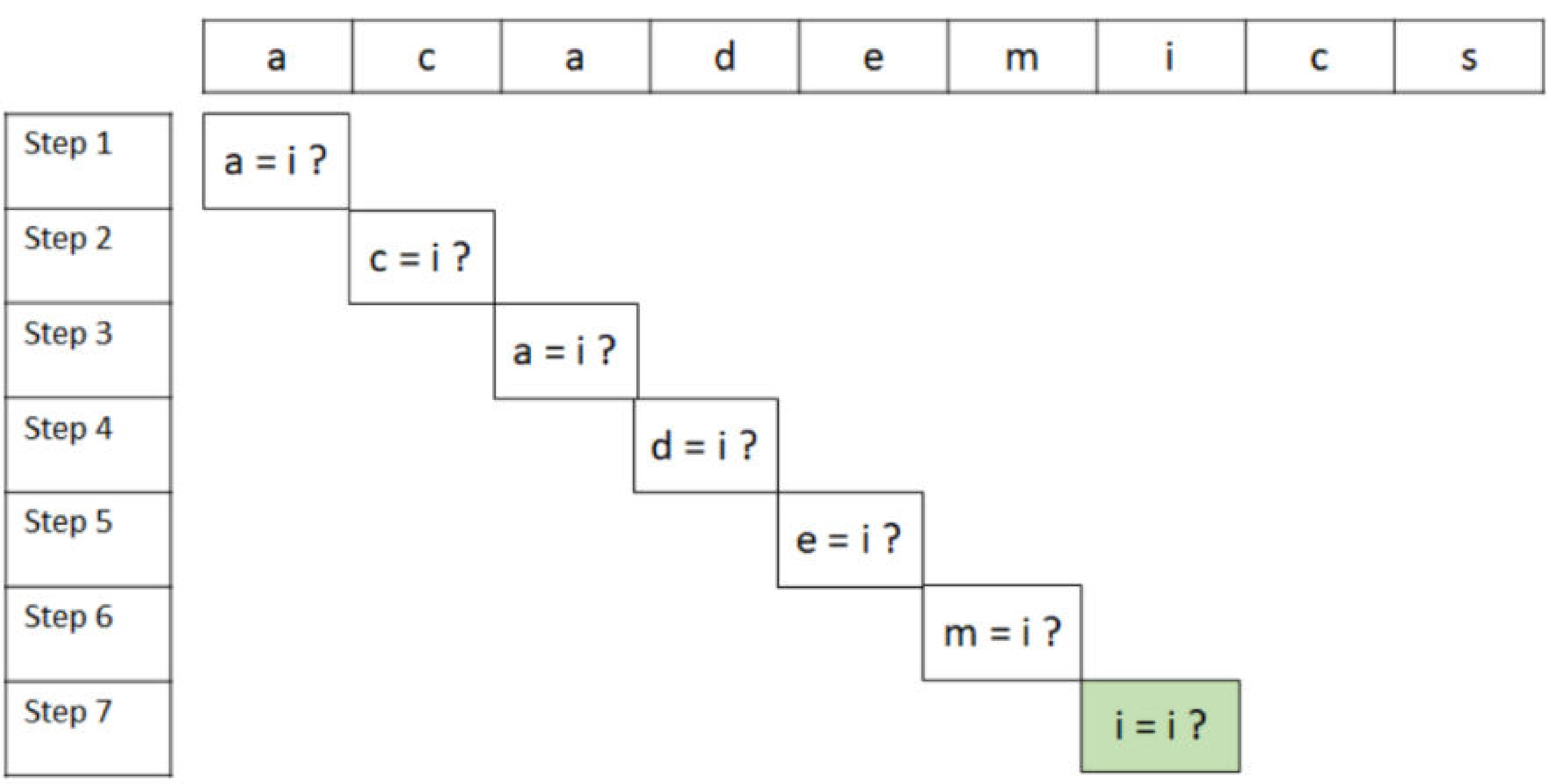
```
1. Temp = Node.Link
2. Node.Link = Node.Link.Link
3. Delete(Temp)
```



Linear Search

- The algorithm starts with the first element, compares with the given key value and returns yes if a match is found
- If it does not match, then it proceeds sequentially comparing each element of the list with the given key until a match has been found or the full list is traversed

Let the given input list be `inputArr = ['a', 'c', 'a', 'd', 'e', 'm', 'i', 'c', 's']` and the search key be `'i'`



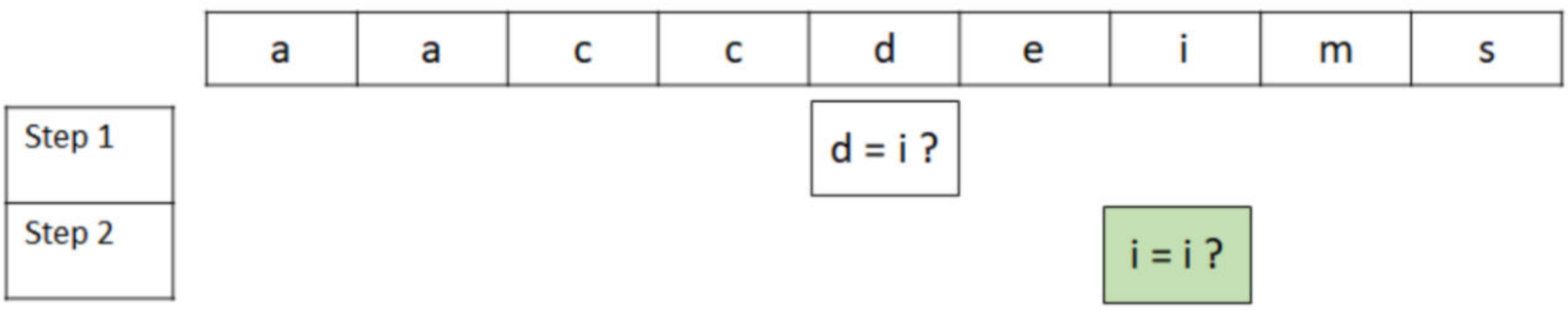

```
def linear_search(input_array, k):
  for i in range(len(input_array)):
    if input_array[i] == k:
      return i
  return -1

inputArr = ['a', 'c', 'a', 'd', 'e', 'm', 'i', 'c', 's']
k = 'i'
```

```
index = linear_search(inputArr, k)
if index != -1:
  print("Element found at " + index)
```

Binary Search

- The input for the algorithm is always a sorted list
- The algorithm compares the key k with the middle element in the list
- If the key matches, then it returns the index
- If the key does not match and is greater than the middle element, then the new list is the list to the right of the middle element
- If the key does not match and is less than the middle elements, then the new list is the list to the left of the middle element

Let the given input list be `inputArr = ['a', 'a', 'c', 'c', 'd', 'e', 'i', 'm', 's']` and the search key be `'i'`




```
def binary_search(arr, k):
  low = 0
  high = len(arr) - 1
  mid = 0

  while low <= high:
    mid = (low + high) // 2

    if arr[mid] < k:
      low = mid + 1
    elif arr[mid] > k:
      high = mid - 1
    else:
      return mid

  return -1

inputArr = ['a', 'a', 'c', 'c', 'd', 'e', 'i', 'm', 's']
k = 'i'
index = binary_search(inputArr, k)

if index != -1:
  print("Element found at " + index)
```

Common Data Structure operations

	Data Structure	Time Complexity								Space Complexity
		Average				Worst				Worst
		Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Linear Data Structures	Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
	Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
	Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Non-Linear Data Structures	Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
	Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
	Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
	Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
	B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
	AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
	KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Source: https://www.bigocheatsheet.com

Week 8 Lecture 3

Class	BSCCS2001
Created	@October 25, 2021 11:26 AM
Materials	
Module #	38
Type	Lecture
Week #	8

Algorithms and Data Structures: Data Structures

Non-linear data structures

Non-linear data structures: Why?

- From the study of Linear Data Structures in the previous module, we can make the following summary observations:
  - All of them have the space complexity $O(n)$, which is optimal
    - However, the actual used space may be lower in array while linked list has an overhead of 100% (double)
  - All of them have complexities that are identical for Worst as well as Average case
  - All of them offer satisfactory complexity for some operations while being unsatisfactory on the others

	Array		Linked List	
	Unordered	Ordered	Unordered	Ordered
Access	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Search	$O(n)$	$O(\lg n)$	$O(n)$	$O(n)$

- Non-linear data structures can be used to trade-off between extremes and achieve a balanced good performance for all

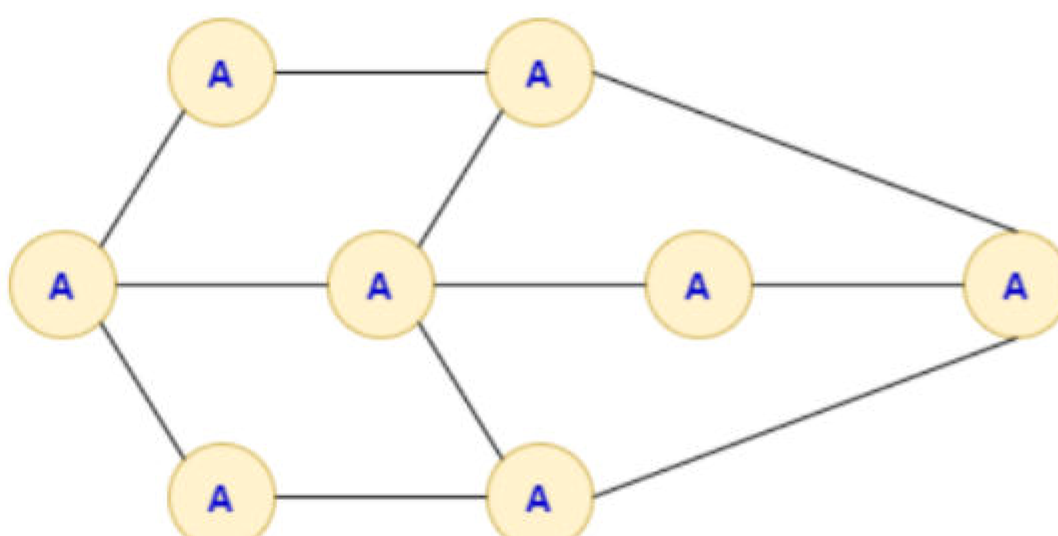
---

- Non-linear data structures are those data structures in which data items are not arranged in a sequence and each element may have multiple paths to connect to other elements
- Unlike linear data structures, in which each element is directly connected with utmost 2 neighbouring elements (previous and next elements), non-linear data structures may be connected with more than 2 elements
- The elements don't have a single path to connect to the other elements but have multiple parts
    - Traversing through the elements is not possible in one run as the data is non-linearly arranged
- Common Non-linear data structures include:
    - **Graph** → Undirected or Directed, Unweighted or Weighted and variants
    - **Tree** → Rooted or Unrooted, Binary or n-ary, Balance or Unbalanced and variants
    - **Hash Table** → Array with lists (coalesced chains) and one or more hash functions
    - **Skip List** → Multi-layered interconnected linked lists
    - and so on ...

Graph

- **Graph** → Graph G is a collection of vertices V (store the elements) and connecting edges (links) E between the vertices:

  $G =< V, E > \ \ where\ E \subseteq V \times V$

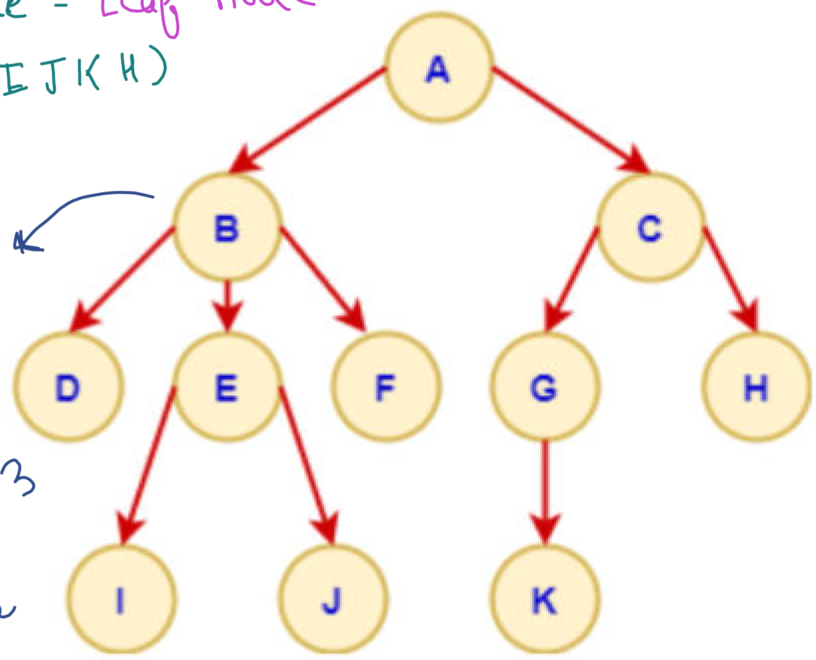


- A graph may be:
    - Undirected or Directed
    - Unweighted or Weighted
    - Cyclic or Acyclic
    - Disconnected or Connected
    - and so on ...
- Examples of a graph include
    - ER Diagram
    - Network: Electrical, Water
    - Friendships in Facebook
    - Knowledge Graph

Tree

- **Tree** → Is a connected acyclic graph representing hierarchical relationship

* n = node ⇒ $n-1$ = max edges
* Non-linear, acyclic & connected
* Node with no child node = Leaf node (eg D I J K H)



arity = 3 (max) ⇓ arity of whole tree = 3

- A tree may be:
  - Rooted or Unrooted
  - Binary or n-ary
  - Balance or Unbalanced
  - Disconnected (forest) or Connected
  - and so on ...
- Examples of tree include:
  - Composite Attributes
  - Family Genealogy
  - Search Trees
  - and so on ...

- **Root** → The node at the top of the tree is called the root
  - There is only one root per tree and one path from the root node to any node
  - **A** is the root node in the given tree
- **Parent** → The node which is a predecessor of any node is called the parent node
  - In the given tree, **B** is the parent of **E**
  - Every node, except the root node, has a unique parent
- **Child** → A node which is the descendant of a node
  - **D**, **E** and **F** are the child nodes of **B**
- **Leaf** → A node which does not have any child node
  - **I**, **J** and **K** are leaf nodes

- **Internal Nodes** → The node which has at least one child is called the Internal Node
- **Sub-tree** → Sub-tree represents the tree rooted at that node
- **Path** → Path refers to the sequence of nodes along the edges of a tree
- **Siblings** → Nodes having the same parent
  - **D, E** and **F** are siblings
- **Arity** → Number of children of a node
  - **B** has arity 3, **E** has arity 2, **G** has arity 1 and **D** has arity 0 (Leaf)

  Maximum arity of a node is defined as the arity of the tree

---

- **Levels** → The root node is said to be at level 0 and the children of the root node are at level 1 and the children of the nodes which are at level 1 will be at level 2 and so on ...

Level is the length of the path (number of links) or distance of a node from the root node

So, level of **A** is 0, level of **C** is 1, level of **G** is 2 and the level of **J** is 3

- **Height** → Max level in a tree
- **Binary Tree** → It is a tree, where each node can have at most 2 children
  - It has arity 2

* LC < RC , Parent < RC

* Max no of node at level 'l' of a binary tree = 2^l

* —— " —— height 'h' "

—— " —— = $2^{h+1} - 1$

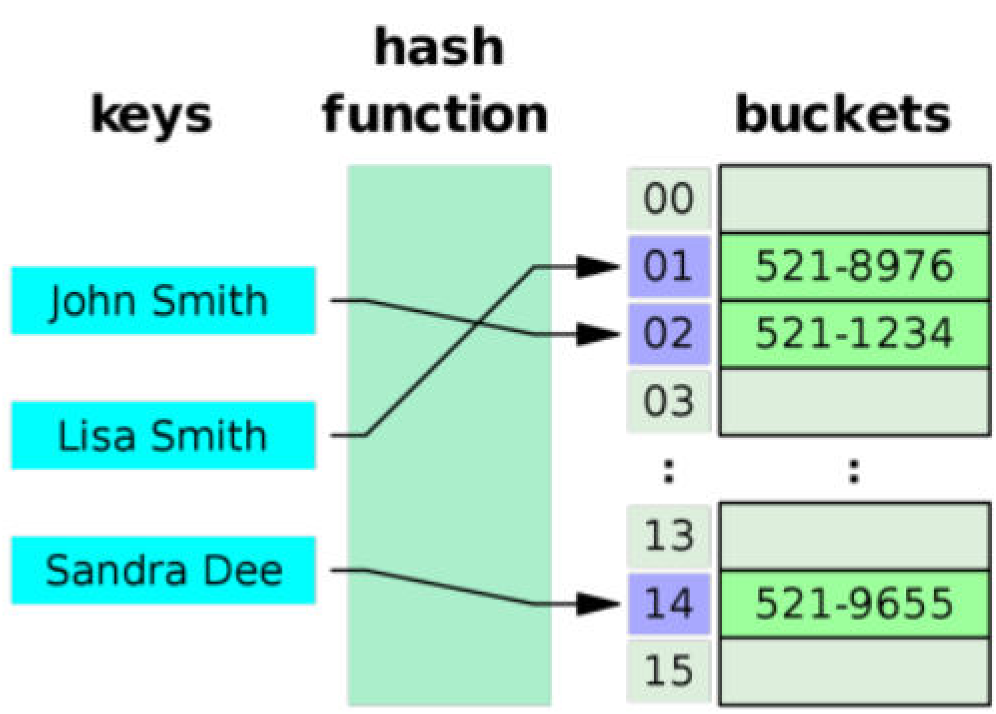
- **Fact 1** → A tree with n nodes has $n-1$ edges
- **Fact 2** → The maximum number of nodes at level l of a binary tree is 2^l
- **Fact 3** → If h is the height of a binary tree of n nodes, then:
  - $h + 1 \leq n \leq 2^{h+1} - 1$
  - $\lceil lg(n+1) \rceil - 1 \leq h \leq n - 1$
  - $O(lg\ n) \leq h \leq O(n)$
  - For a k-ary tree ,$O(lg_k\ n) \leq h \leq O(n)$

Q) Max level of complete BST = 5
Max & min no of nodes ?

Max = min nodes = $2^{5+1} - 1$
$= 63$

Hash Table

- **Hash Table (or Hash Map)** → Implements an associative array abstract data type, a structure that can map keys to values by using a hash function to compute an index (hash code), into an array of buckets or slots, from which the desired value can be found



- A hash table may be using:
  - Static or Dynamic Schemes
  - Open Addressing
  - 2-Choice Hashing
  - and so on ...
- Examples of Hash Tables include:
  - Associative arrays
  - Database indexing
  - Caches
  - and so on ...

Binary Search Tree

- During the study of linear data structure, we observed that
  - Binary search is efficient in the search of a key: $O(log\ n)$
    - It needs to be performed on a sorted array

Max ht = $(n-1)$

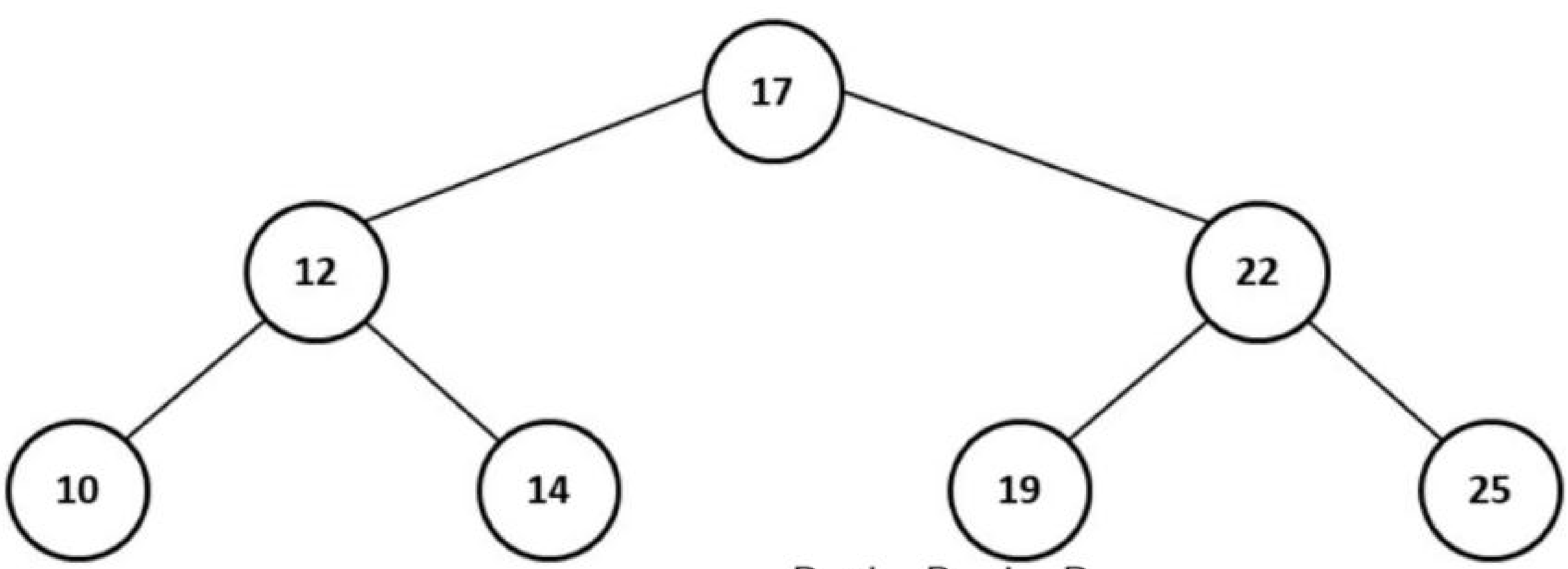
Min " = $\lceil \log_2(n) - 1 \rceil$

        - The array makes insertion and deletion expensive at $O(n)$
    - The linked list, on the other hand, is efficient in insertion and deletion at $O(1)$, while it makes the search expensive at $O(n)$
        - $O(1)$ insert/delete is possible because we just need to manipulate the pointers and not physically move the data
- Using the non-linearity, specifically (binary) trees, we can combine the benefits of both
- Note that once an array is sorted, we know the order in which its elements may be checked (for any key) during a search
- As the binary search splits the array, we can conceptually consider the ***Middle Element*** to be the ***Root*** of a tree and ***left (right) sub-array*** to be its ***left (right) sub-tree***
- Progressing recursively, we have a **Binary Search Tree (BST)**

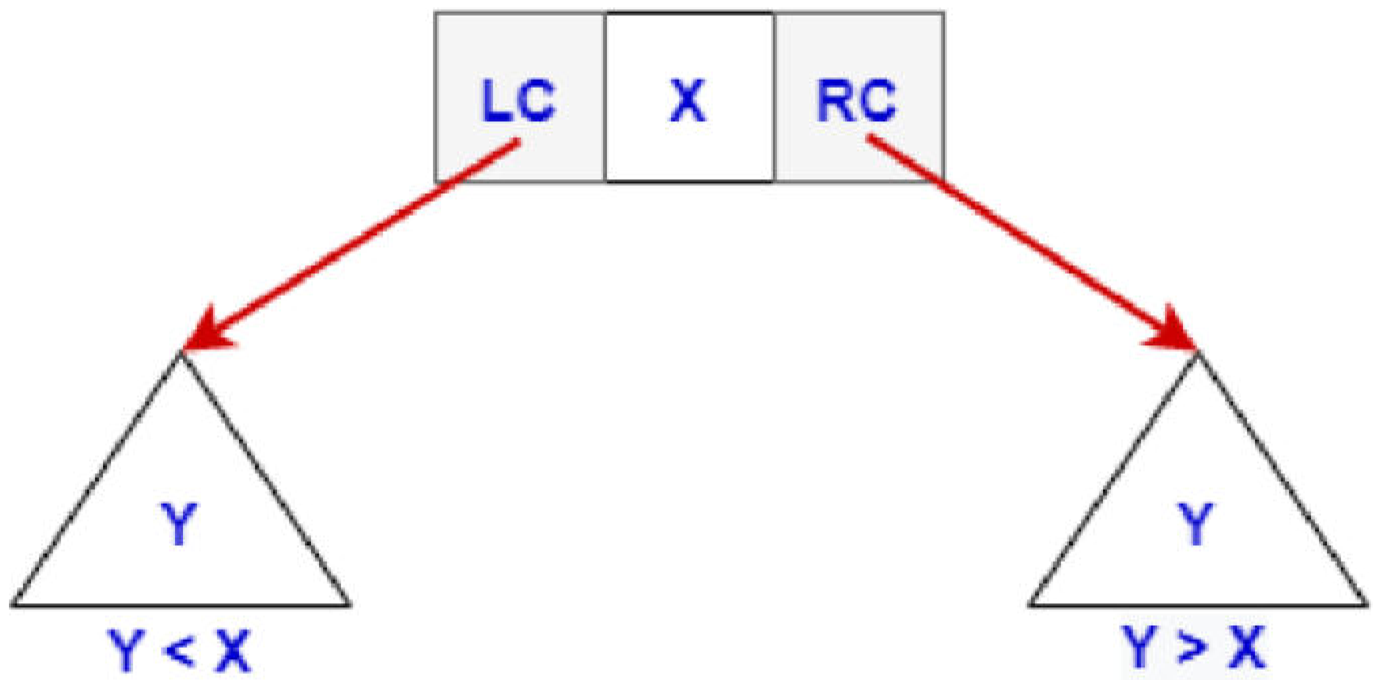
- Consider the data set:

10	12	14	17	19	22	25
LL	L	LR	M	RL	R	RR

- Search order is:
    - First → M
    - Second → L or R
    - Third:
        - For L → LL or LR
        - For R → RL or RR
    - Recursive ...
- Put as a tree



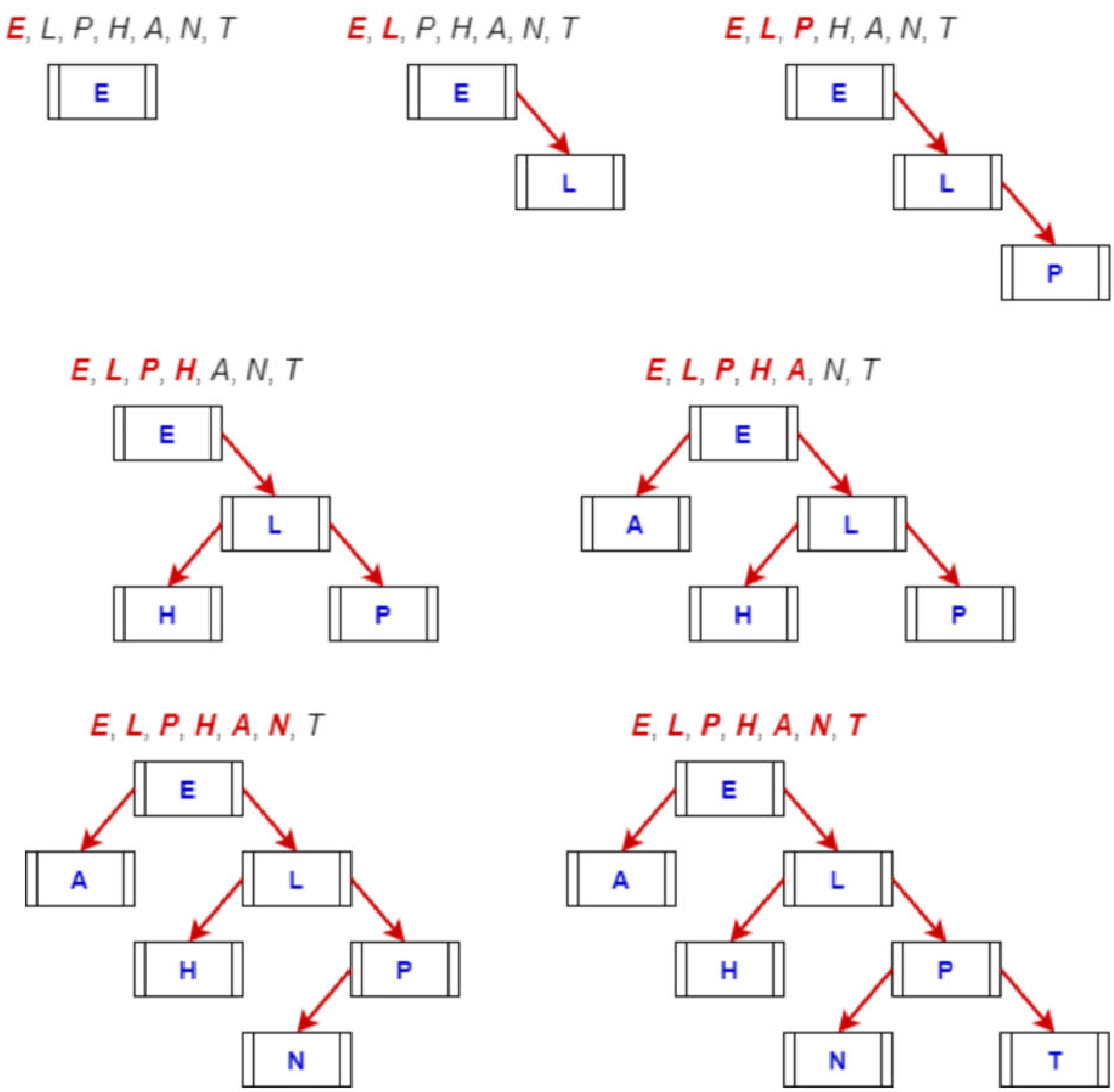
- **Binary Search Tree (BST)** → It is a tree in which all the nodes hold the following:
    - The value of each node in the left subtree is less than the value of its root
    - The value of each node in the right subtree is greater than the value of its root



- **Structure of BST node** → Each node consists of an element (**X**), and a link to the left child or the left subtree (**LC**), and a link to the right child or the right subtree (**RC**)

- **Example** → Obtain the BST by inserting the following values:

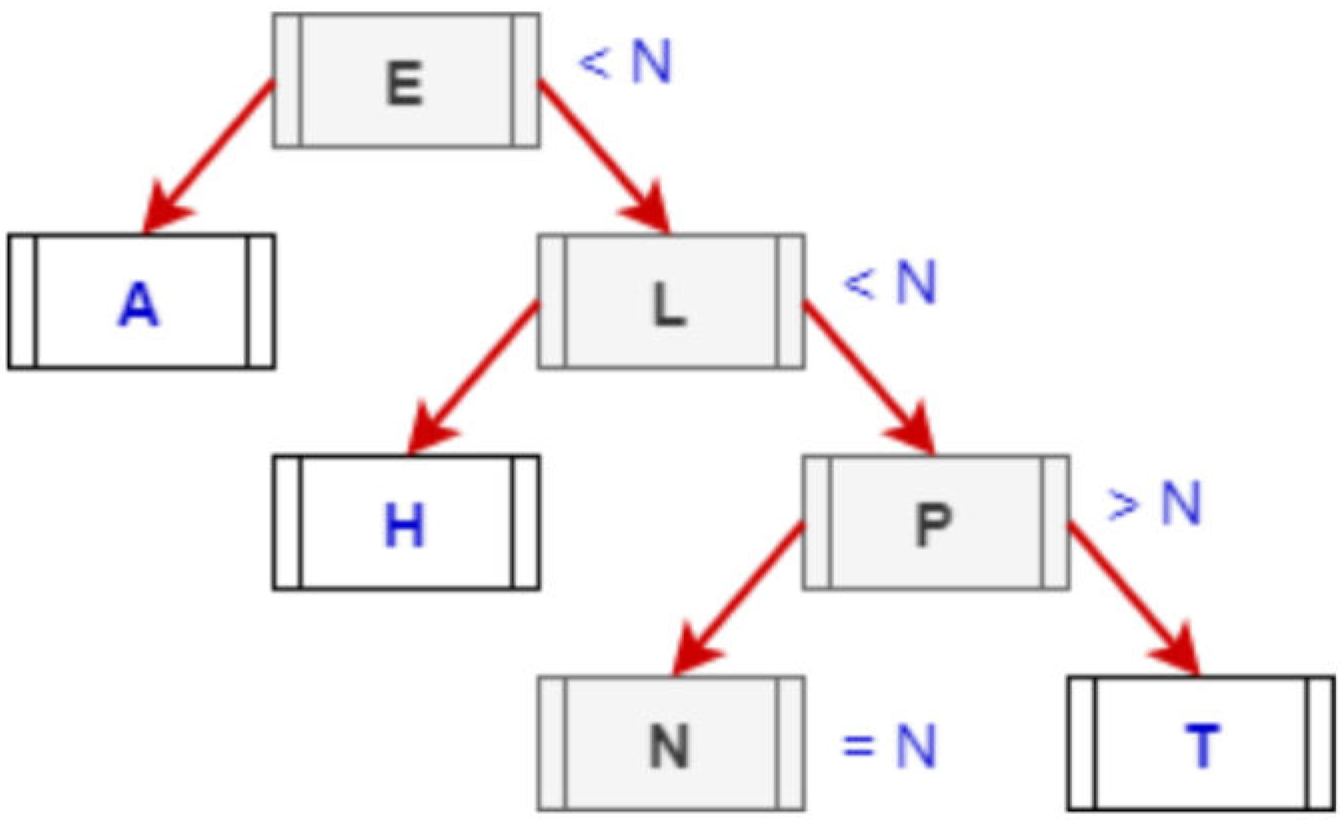
  E, L, P, H, A, N, T



Searching a key in the BST

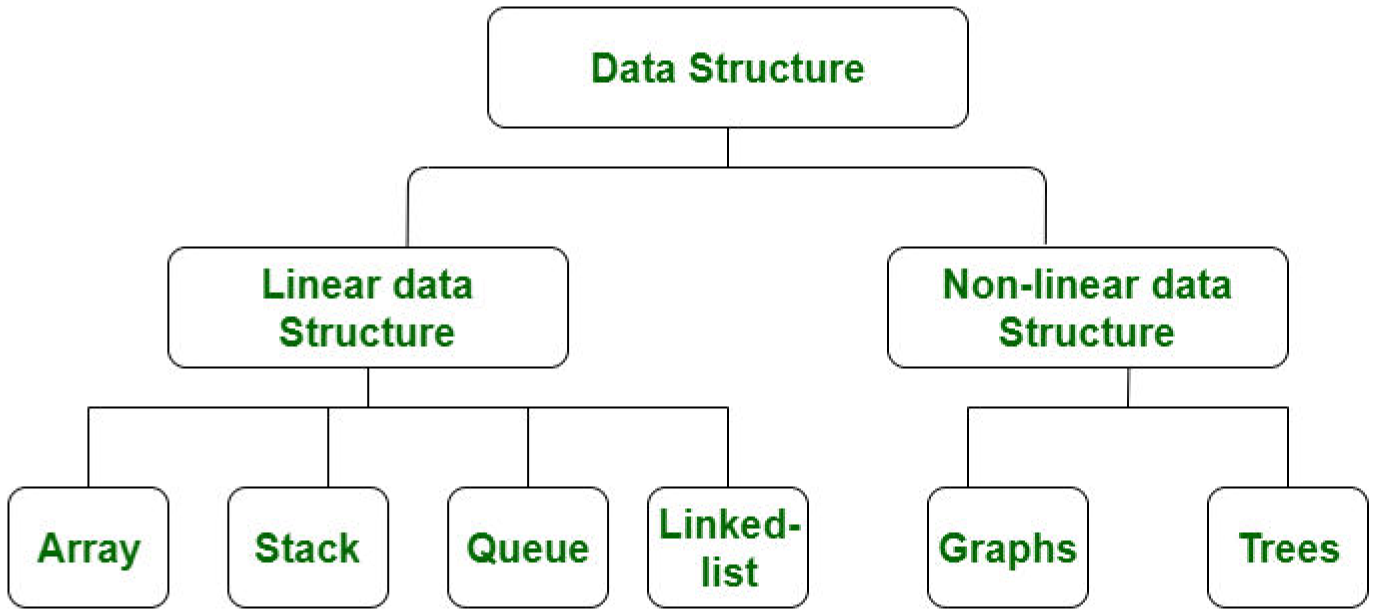
`search(root, key)`

1. Compare the key with the element at root
    a. If the key is equal to root's element then
        i. Element found and return
    b. else if they key is lesser than the root's element
        i. `search(root.lc)` # Search on the left subtree
    c. else: `(the key is greater than the root's element)`
        i. `search(root.rc)` # Search on the right subtree



- Searching a key in the BST is $O(h)$, where h is the height of the key
- **Worst Case**
    - The BST is skewed binary search tree (all the nodes except the leaf would have one child)
    - This can happen if they keys are inserted in sorted order
    - Height (h) of the BST having n elements becomes $n-1$
    - Time complexity of search in BST becomes $O(n)$
- **Best Case**
    - The BST is balanced binary search tree
    - This is possible if
        - If the keys are inserted in purely randomized order
        - If the tree is explicitly balanced after every insertion
    - Height (h) of the binary search tree becomes $log\ n$
    - Time complexity of search in BST becomes $O(log\ n)$

Linear and non-Linear Data Structures



Source: https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/

Linear Data Structure	Non-linear Data Structure
In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

Week 8 Lecture 4

Class	BSCCS2001
Created	@October 25, 2021 1:02 PM
Materials	
Module #	39
Type	Lecture
Week #	8

Storage and File Structure: Physical Storage

Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
    - Data loss on power failure or system crash
    - Physical failure of the storage device
- Can differentiate storage into:
    - **Volatile storage** → Loses content when power is switched off
    - **Non-volatile storage** →
        - Content persists even when the power is off
        - Includes secondary and tertiary storage, as well as battery backed-up main memory

Physical Storage Media

- **Cache**
    - Fastest and most costly form of storage
    - Volatile
    - Managed by the computer system hardware
- **Main memory**

- Fast access (10's to 100's of nanoseconds (ns))
    - 1 ns $= 10^{-9}$ seconds
- Generally too small (or too expensive) to store the entire DB
    - Capacities of up to a few gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2-3 years)
- **Volatile**
  - Contents of the main memory are usually lost if a power failure or system crash occurs

Physical Storage Media: Flash Memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
  - Can support only a limited number $(10K - 1M)$ of write/erase cycles
  - Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones and USB keys

Physical Storage Media: Magnetic Disk

- Data is stored on spinning disk, and read/write magnetically
- Primary medium for the long-term storage of data
  - Typically stores the entire DB
- Data must be moved from the disk to main memory for access, and written back for storage — much slower access than the main memory
- Direct-access
  - Possible to read data on the disk in any order, unlike magnetic tape
- Capacities range up to roughly 16-32TB
  - Much larger capacity and much lower cost/byte than the main memory/flash memory
  - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
  - Disk failure can destroy data, but is rare

Physical Storage Media: Optical Storage

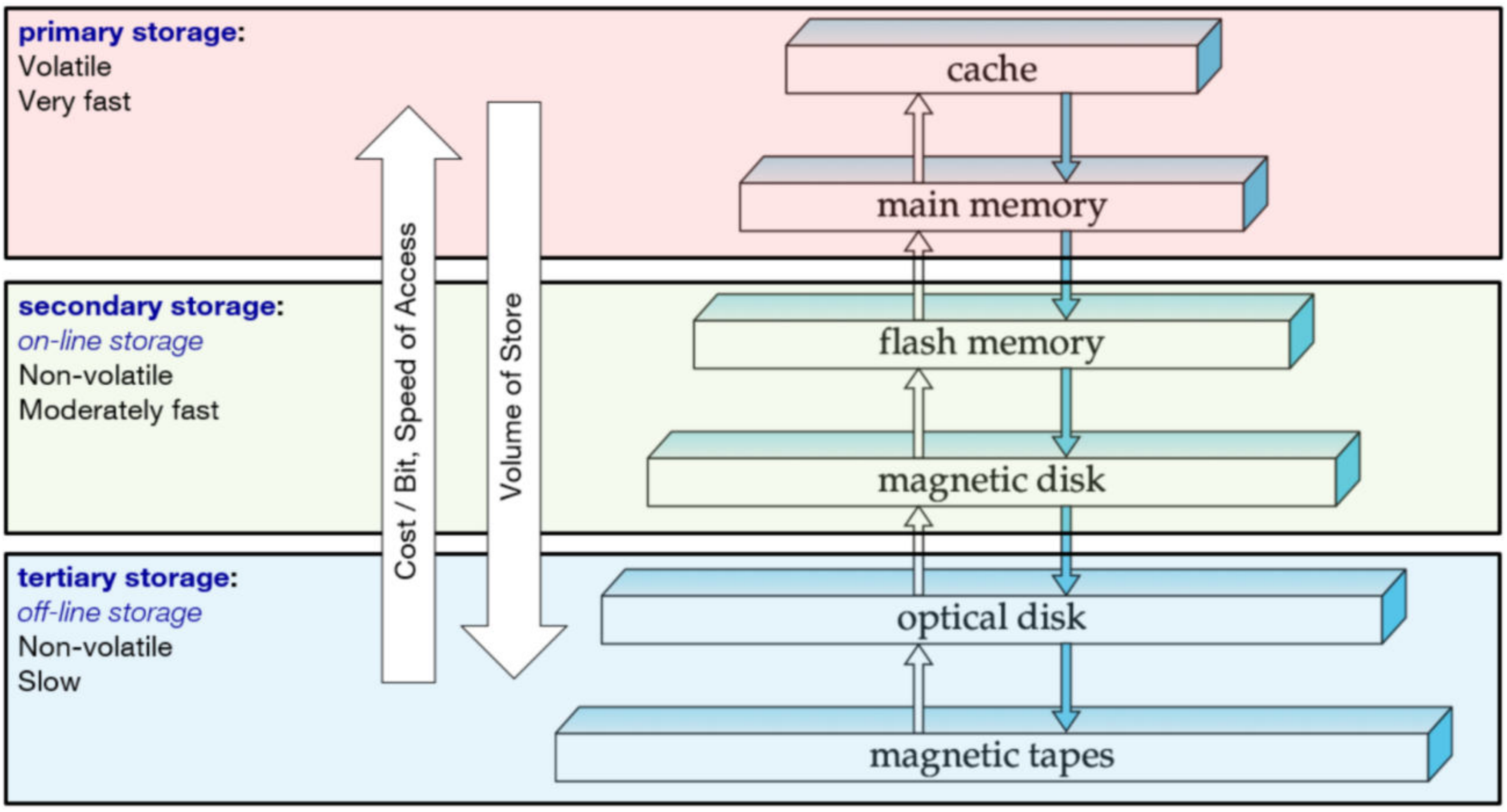
- Non-volatile, data is read optically from a spinning disk using a LASER
- CD-ROM (640MB) and DVD(4.7 to 17GB) most popular forms
- Blu-ray disks: 27GB to 54GB
- Write-one, Read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW and DVD-RAM)
- Reads and Writes are slower than with magnetic disks
- ***Juke-box*** systems, with large number of removable disks, a few drives and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Physical Storage Media: Tape Storage

- Non-volatile, used primarily for backup (to recover from a disk failure) and for archival data
- **Sequential-access**
  - Much slower than a disk

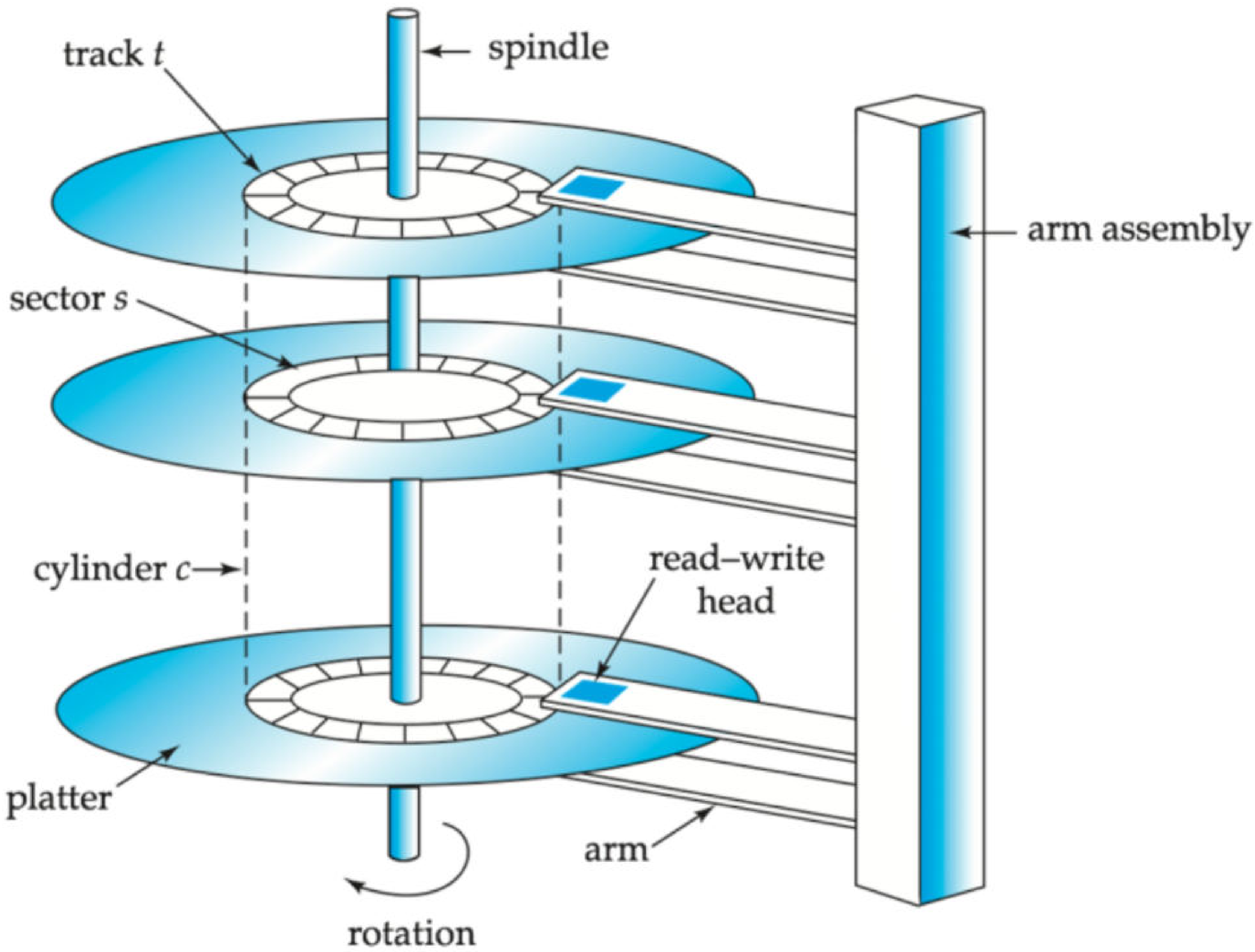
- Very high capacity (40-300TB tapes available)
- Tape can be removed from drive storage costs much cheaper than the disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
    - Hundreds of terabytes (TB) $(1TB = 10^{12}\ bytes)$ to even multiple petabytes (PB) $(1PB = 10^{15}\ bytes)$

Storage hierarchy



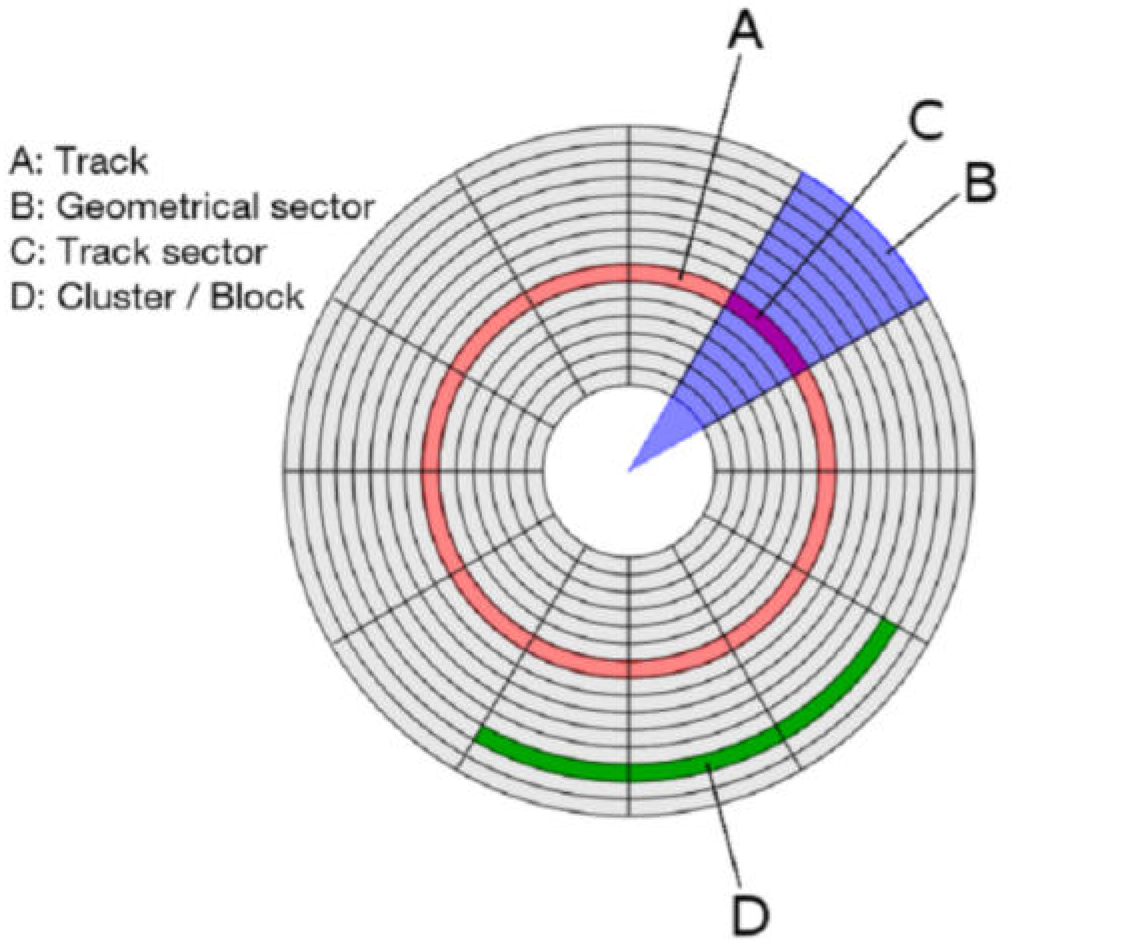
Magnetic Disk

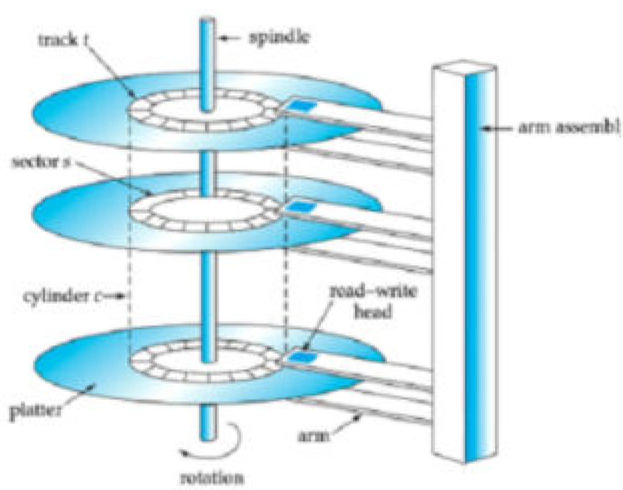
Magnetic Disk: Mechanism

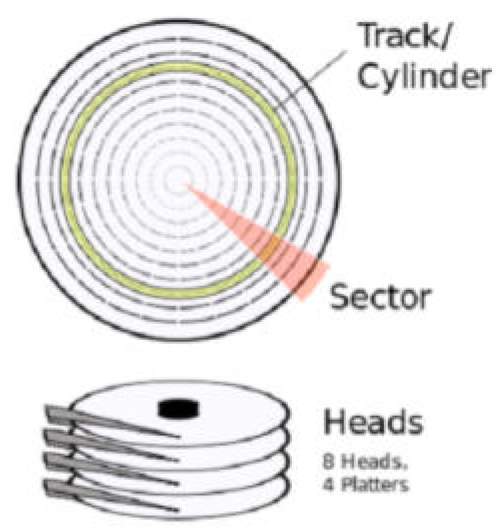


NOTE: Diagram is schematic, and simplifies the structure of actual disk drives

- Read-write head
  - Positioned very close to the platter surface
- Surface of the platter is divided into circular ***tracks***
  - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into ***sectors***
  - A sector is the smallest unit of data read or written
  - Sector size typically 512 bytes
  - Sectors / track → 500 to 1K (inner) to 1K to 2K (outer)
- To read/write a sector
  - Disk arm swings to position head on right track
  - Platter spins → Read/Write as sector passes under the head
- Head-disk assemblies
  - Multiple disk platters on a single spindle (1 to 5 usually)
  - One head per platter, mounted on a common arm
- Cylinder *i* consists of i^{th} track of all the platters

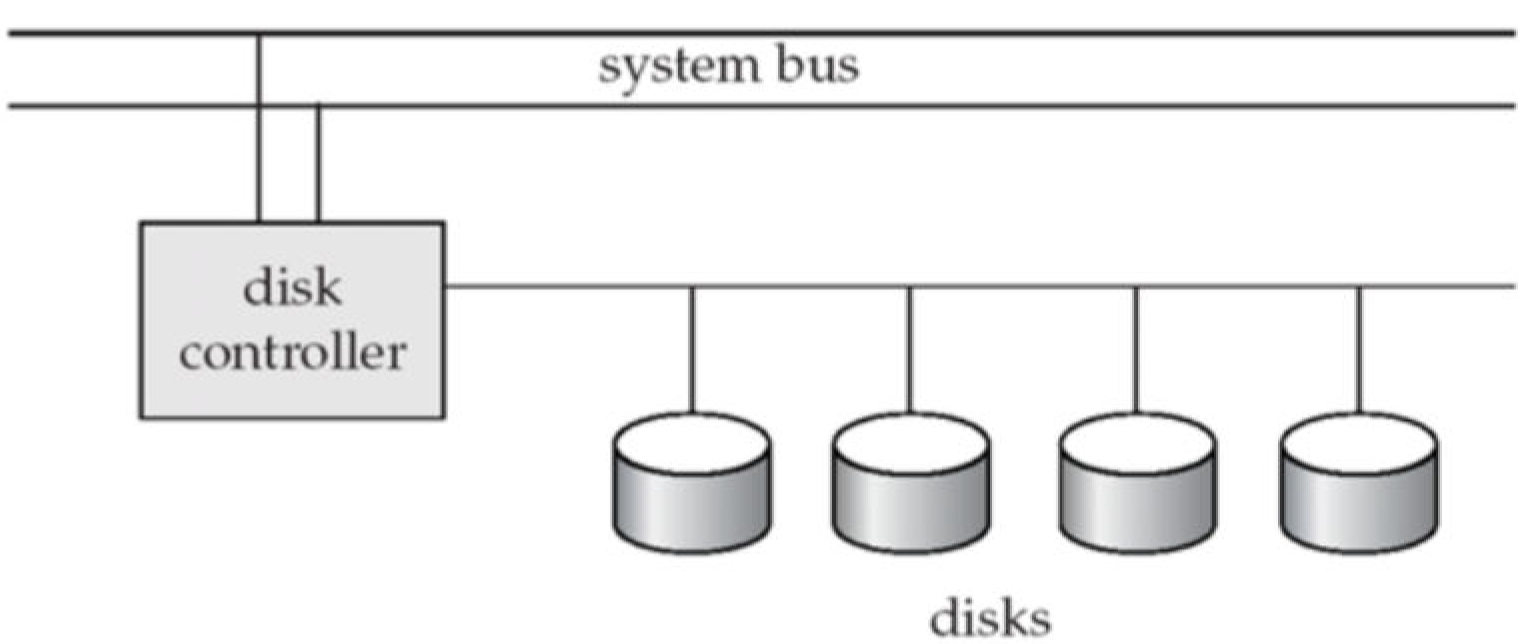






Magnetic Disk: Disk Controller, Subsystems and Interfaces

- **Disk Controller** → Interfaces between the computer system and the disk drive hardware
    - Accepts high-level commands to read or write a sector
    - Initiates actions moving the disk arm to the right track, reading or writing the data
    - Computes and attaches checksums to each sector to verify that correct read back
    - Ensures successful writing by reading back sector after writing it
    - Performs remapping of bad sectors
- **Disk Subsystem**



2^{10} bytes = 1 KB

2^{20} " = 1 MB

2^{30} " = 1 GB

1 min = 60 K msec

- **Disk Interface Standards Families** → ATA, SATA, SCSI, SAS and several variants
- **Storage Area Networks (SAN)** → Connects disks by a high-speed network to a number of servers
- **Network Attached Storage (NAS)** → Provides a file system interface using networked file system protocol

Magnetic Disk: Performance Measures

* Avg rot^n latency = $1/2$ × one comp disk rot^n

- **Access Time** → Time from a read or write request issue to start of data transfer
  - **Seek Time** → Time to reposition the arm over the correct track
    - Avg. seek time is 1/2 the worst case seek time; 1/3 if all the tracks have same number of sectors
    - 4 to 10 milliseconds on typical disks
  - **Rotational Latency** → Time for the sector to be accessed to appear under the head
    - Average latency is 1/2 of the worst case latency
    - 4 to 11 milliseconds on typical disks (5400 to 15000 RPM)

* No of Sector = 2 × no of platter × no of track per surface × no of sector per track

* Disk capacity = no of sector on disk × capacity of 1 sector

- **Data-transfer rate** → The rate at which data can be retrieved from or stored to the disk
  - 25 to 100MB per second max rate, lower for inner tracks
  - Multiple disks may share a controller, so rate that controller can handle is also important
- **Mean Time to Failure (MTTF)** → Avg. time the disk is expected to run continuously without any failure
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a theoretical MTTF of 500,000 to 1,200,000 hours for a new disk
    - For example → an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as the disk ages

* Transfer rate = $\frac{\text{Byte on 1 track}}{\text{Time req to complete 1 rot}^n}$

Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Tape formats
    - Few GB for DAT (Digital Audio Tape) format
    - 10 - 40GB with DLT (Digital Linear Tape) format
    - 100GB+ with Ultrium format
    - 330GB with Ampex helical scan format
  - Transfer rates from a few to 10's of MB/s

* Transfer time = $\frac{\text{File size}}{\text{Transfer rate}}$

* No of bits req to address all sectors = $\log_2$ (no of sectors)

- Tapes are cheap, but the cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
  - Limited to sequential access
  - Some formats (Accelis) provide faster seek (10's of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information and as an offline medium for transferring information from one system to another
- Tape jukeboxes used for very large capacity storage
  - Multiple petabytes $(10^{15}\ bytes)$

Cloud Storage

- Cloud storage is purchased from a 3rd party cloud vendor who owns and operates the data storage capacity and delivers it over the internet in a pay-as-you-go model
- The cloud storage vendors manage capacity, security and durability to make data accessible to applications all around the world
- Applications access cloud storage through traditional storage protocols or directly via an API

- Many vendors offer complementary services designed to help collect, manage, secure and analyze data at a massive scale
  - Various available options for cloud storage are:
    - Google Drive
    - Amazon Drive
    - Microsoft Onedrive
    - Evernote
    - Dropbox
    - and so on ...

Cloud storage v/s Traditional Storage

Parameters	**Cloud Storage**	**Traditional Storage**
Cost	Cloud storage is cheaper per GB than using external drives.	The hardware and infrastructure costs are high and adding on more space and upgrading only adds extra costs.
Reliability	Cloud storage is highly reliable as it takes less time to get under functioning	Traditional storage requires high initial effort and is less reliable.
File Sharing	Cloud storage supports file sharing dynamically as it can be shared anywhere with network access	Traditional storage requires physical drives to share data and a network is to be established between both
Accessibility	Cloud storage gives you access to your files from anywhere	Restricted to local access
Backup/ Recovery	Very safe from on site disaster. In case of a hard drive failure or other hardware malfunction, you can access your files on the cloud, which acts as a backup solution for your local storage on physical drives	Data that is stored locally is much more susceptible to unexpected events and local storage and local backups could be easily lost

Other storage

Optical Disks

- Compact disk-read only memory (CD-ROM)
  - Removable disks, 640MB per disk
  - Seek time about 100 msec (optical read is heavier and slower)
  - Higher latency (3000 RPM) and lower data-transfer rates (3 - 6MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
  - DVD-5 holds 4.7GB and DVD-9 holds 8.5GB
  - DVD-10 and DVD-18 are double sided formats with capacities of 9.4GB and 17GB
  - Blu-ray DVD → 27GB (54GB for double sided disk)
  - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R) are popular
  - Data can only be written once and cannot be erased
  - High capacity and long lifetime; used for archival storage
  - Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available

Flash drives

- Flash drives are often referred to as pen drives, thumb drives or jump drives
  - They have completely replaced floppy drives for portable storage

- Considering how large and inexpensive they have become, they have also replaced CDs and DVDs for data storage purposes

- USB flash drives are removable and re-writable storage devices that, as the name suggests, require a USB port for connection and utilizes non-volatile flash memory technology
- The storage space in USB is quite large with sizes ranging from 128MB to 2TB
- The USB standard a flash drive is built around will determine the number of things about its potential performance, including maximum transfer rate

Secure Digital Cards (SD Cards)

- A Secure Digital (SD) card is a type of removable memory card used to read and write large quantities of data
- Due to their relatively small size, SD cards are widely used in mobile electronics, cameras, smart devices, video game consoles and more
- There are several types of SD cards sold and used today:

Card Type	Year of Debut	Capacity	Supported Devices
SD	1996	128MB to 2GB	All host devices that support SD, SDHC, SDXC
SDHC	2006	4GB to 32GB	All host devices that support SDHC, SDXC
SDXC	2009	64GB to 2TB	All host devices that support SDXC

Card Type	Capacity	File System	Remarks
SD	128MB to 2GB	FAT16	FAT16 supports 16 MB to 2 GB
SDHC	4GB to 32GB	FAT32	FAT32 can be support up to 16 TB
SDXC	64GB to 2TB	exFAT	exFAT is non-standard, supports file up to 4 GB

Source: https://integralmemory.com/faq/what-are-differences-between-fat16-fat32-and-exfat-file-systems

Flash storage

- NOR Flash vs NAND Flash
- NAND Flash
    - Used widely for storage, since it is much cheaper than NOR Flash
    - Requires page-at-a-time read (page: 515 bytes to 4KB)
    - Transfer rate around 20MB/s
    - **Solid State Disks** → Use multiple flash storage devices to provide higher transfer rate of 200MB/s or higher
    - Erase is very slow (1 to 2ms)
        - Erase block contains multiple pages
        - Remapping of logical page addresses to physical page addresses avoids waiting for erase
            - Translation table tracks mapping
            - Also stored in a label field of flash page
            - Remapping carried out by flash translation layer
        - After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
            - Wear leveling

Solid State Drives (SSDs)

- SSDs replace traditional mechanical hard disks by using flash-based memory, which is significantly faster
- SSDs speed up computers significantly due to their low read-access time and fast throughput
- The idea of SSDs was introduced in 1978
    - It was implemented using semiconductors

- It stores the data in the persistent state even when no power is supplied

- The speed of SSD much larger than that of HDD as it reads/writes data at a higher input-output per second
- Unlike HDDs, SSDs do not include any moving parts
    - SSDs resists vibrations and high temperatures

SSD v/s HDD

Parameters	SSD	HDD
Technology	Integrated circuit using Flash memory	Mechanical Parts, including spinning disks or platters
Access Time	0.1 ms	5.5-8.0 ms
Average Seek Time	0.08-0.16 ms	$<$ 10 ms
Speed (SATA II)	80-250 MB/sec	65-85 MB/sec
Random I/O Performance	6000 io/s	400 io/s
Backup rates	6 hours	20- 24 hours
Reliability	The failure rate of less than 0.5%	Failure rate fluctuates between 2-5%
Energy Consumption	2 to 5 watts	6 to 15 watts

Future of Storage

DNA Digital Storage

Oooooooh, we going Cyberpunk

- DNA digital data storage is the process of encoding and decoding binary data to and from synthesized strands of DNA
- While DNA as a storage medium has enormous potential because of its high storage density, its practical use is currently severely limited because of its high cost and very slow read and write times
- Digital storage systems encode the text, photos, videos or any kind of information as a series of 0s and 1s
    - This same information can be encoded in DNA using the 4 nucleotides that make up the genetic code: A, T, G and C
    - For example → G and C could be used to represent 0 while A and T represent 1
- DNA has several other features that makes it desirable as a storage medium; it is extremely stable and is fairly easy (but expensive) to synthesize and sequence
- Also, because of its high density — each nucleotide, equivalent to up to 2 bits, is about 1 cubic nanometer - an exabyte $(10^{18}\ bytes)$ of data stored as DNA could fit in the palm of our hands
- DNA synthesis → A DNA synthesizer machine builds synthetic DNA strands matching the sequence of digital code

Quantum Memory

- Quantum Memory is the quantum-mechanical version of ordinary computer memory
- Whereas ordinary memory stores information as binary states (represented by 1's and 0's)
    - Quantum memory stores a quantum state for later retrieval
- These states hold useful computation information knows as ***qubits***
- Quantum memory is essential for the development of many devices in quantum information processing applications such as quantum network, quantum repeater, linear optical quantum computation or long-distance quantum communication

- Unlike the classical memory of everyday computers, the states stored in quantum memory can be in a quantum superposition, giving much more practical flexibility in quantum algorithms than classical information storage

Week 8 Lecture 5

Class	BSCCS2001
Created	@October 25, 2021 2:53 PM
Materials	
Module #	40
Type	Lecture
Week #	8

Storage and File Structure: File Structure

File Organization

- A Database is
    - A collection of files
        - A file is
            - A sequence of records
                - A record is
                    - A sequence of fields
- One approach:
    - Assume record size is fixed
    - Each file has records of one particular type only
    - Different files are used for different relations
    - This case is easiest to implement; will consider variable length records later
- A Database file is partitioned into fixed-length storage units called blocks
    - Blocks are units of both storage allocation and data transfer

Fixed-Length Records

- Simple approach
    - Store record i starting from byte $n * (i - 1)$, where n is the size of each record

- Record access is simple but records may cross blocks
  - Modification → Do not allow records to cross block boundaries
- Deletion of record i → Alternatives
  - Move records $i+1, ..., n$ to $i, ..., n-1$
  - Move record n to i
  - Do not move records, but link all the free records on a free list

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 with Comparison

Before deletion

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

After deletion & Compaction

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Deleting Record 3 with Moving last record

Before deletion

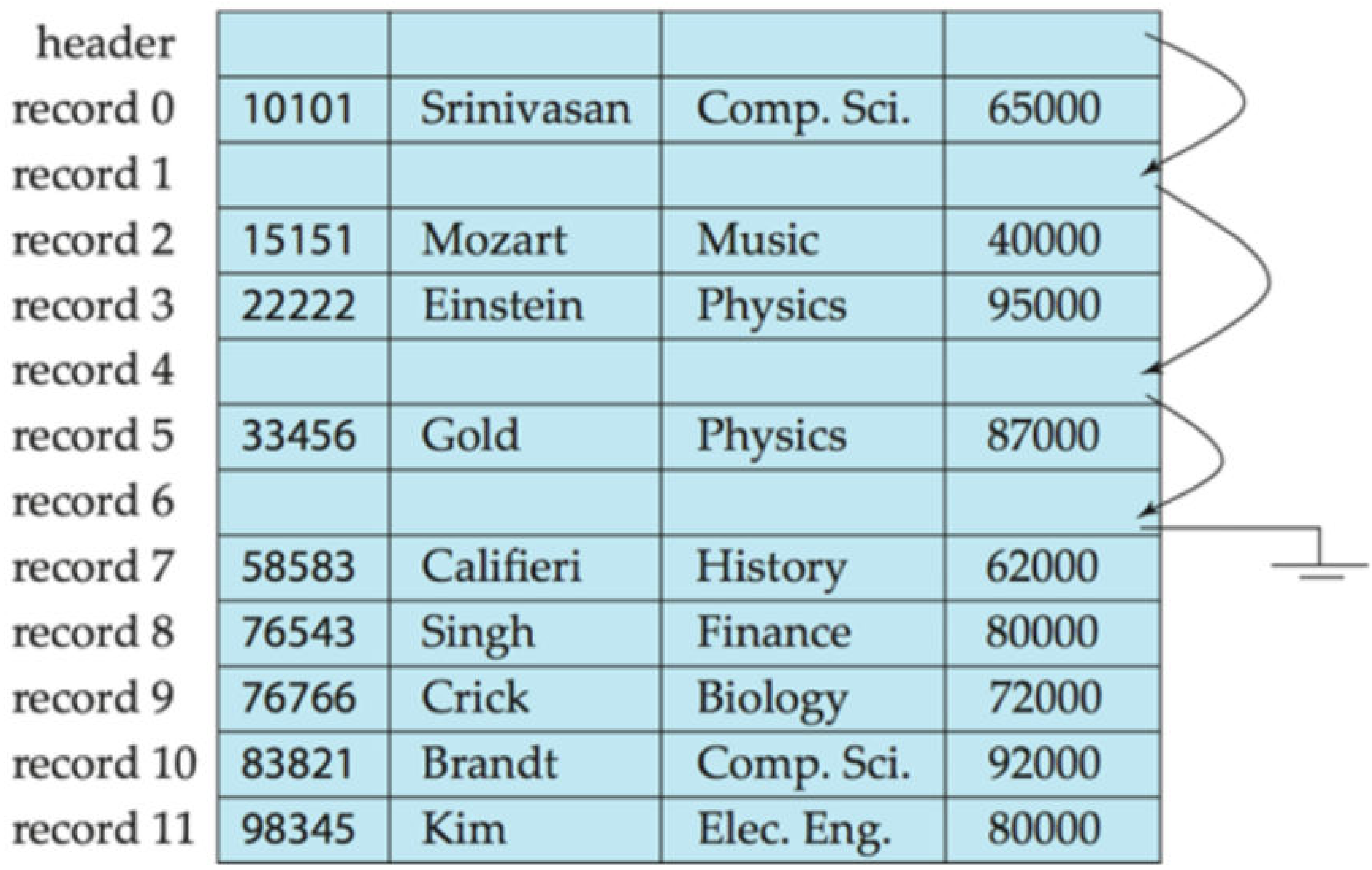
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

After deletion & Movement

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

Free Lists

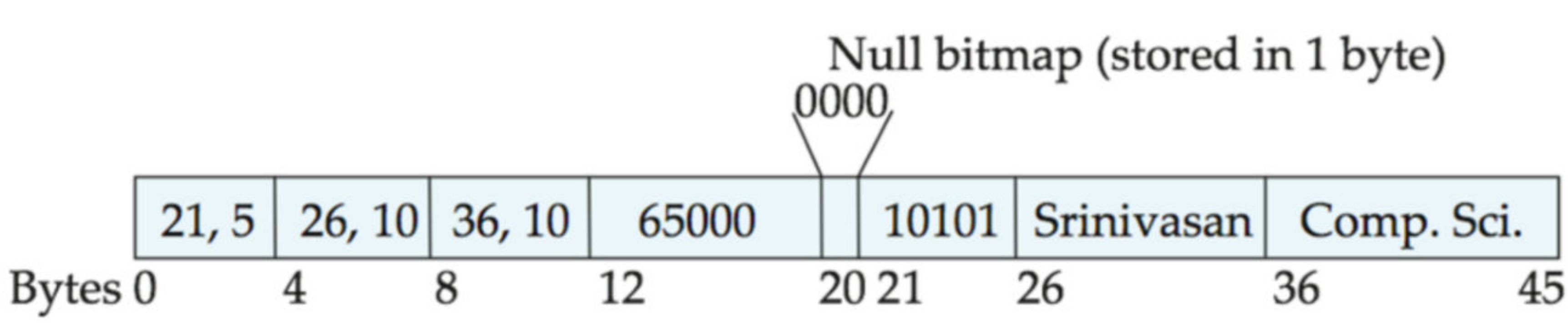
- Store the address of the first deleted record in the file header
- Use this first record to store the address of the second deleted record, and so on ...
- Consider these stored addresses as pointers since they point to the location of the record
- More space efficient representation → Re-use space for normal attributes of free records to store pointers (No pointers stored in in-use records)

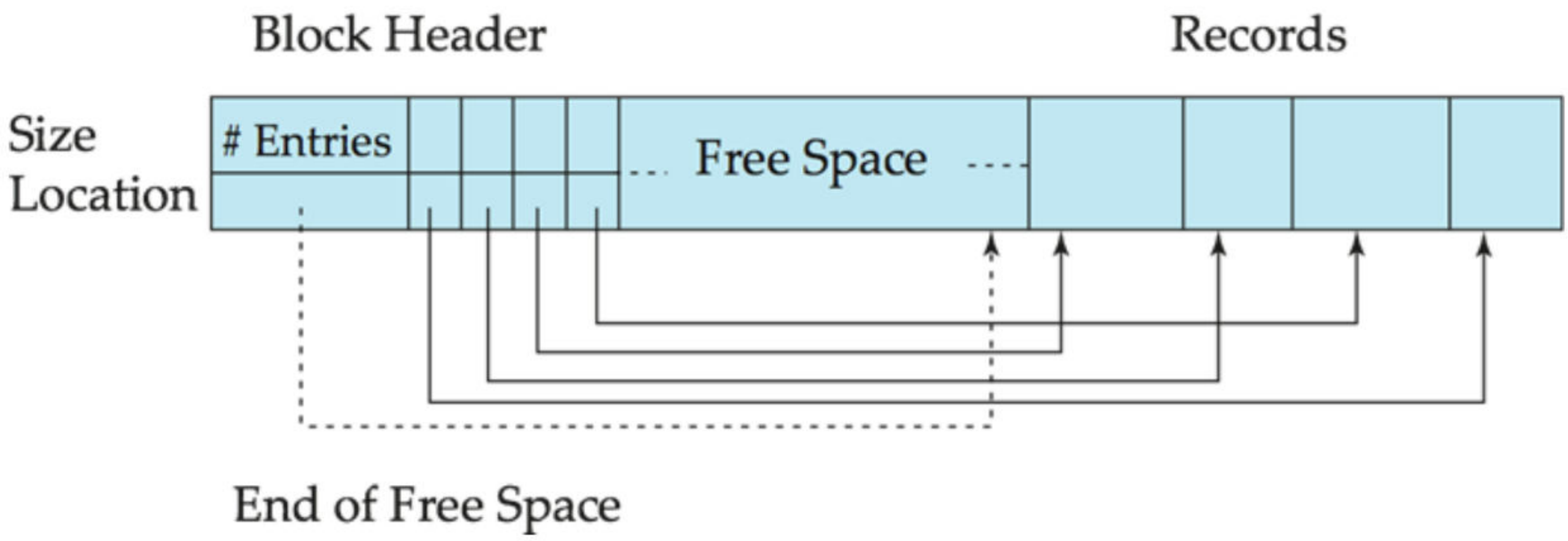


header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records

- Variable-length records arise in DB systems in several ways:
  - Storage of multiple record types in a file
  - Record types that allow variable lengths for one or more fields such as strings (varchar)
  - Record types that allow repeating fields (used in some older data models)
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length) with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





- Slotted page header contains:
  - Number of record entries
  - End of free space in the block
  - Location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated
- Pointers should not point directly to record — instead they should point to the entry for the record in the header

Organization of records in Files

- **Heap** → A record can be placed anywhere in the file where there is space
- **Sequential** → Store records in sequential order, based on the value of the search key of each record
- **Hashing** → A hash function computed on some attributes of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file
  - In a ***multitable clustering file organization*** records of several different relations can be stored in the same file
    - Motivation → Store related records on the same block to minimize I/O

Sequential File Organization

- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a ***search-key***

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

- Deletion → use pointer chains
- Insertion → Locate the pointer where the record is to be inserted
  - if there is free space, insert there
  - if no free space, insert the record in an ***overflow block***
  - In either case, pointer must chain must be updated
- Need to re-organize the file from time to time to restore sequential order

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

32222	Verdi	Music	48000	

Multitable Clustering File Organization

Store several relations in one file using a ***multitable clustering file organization***

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

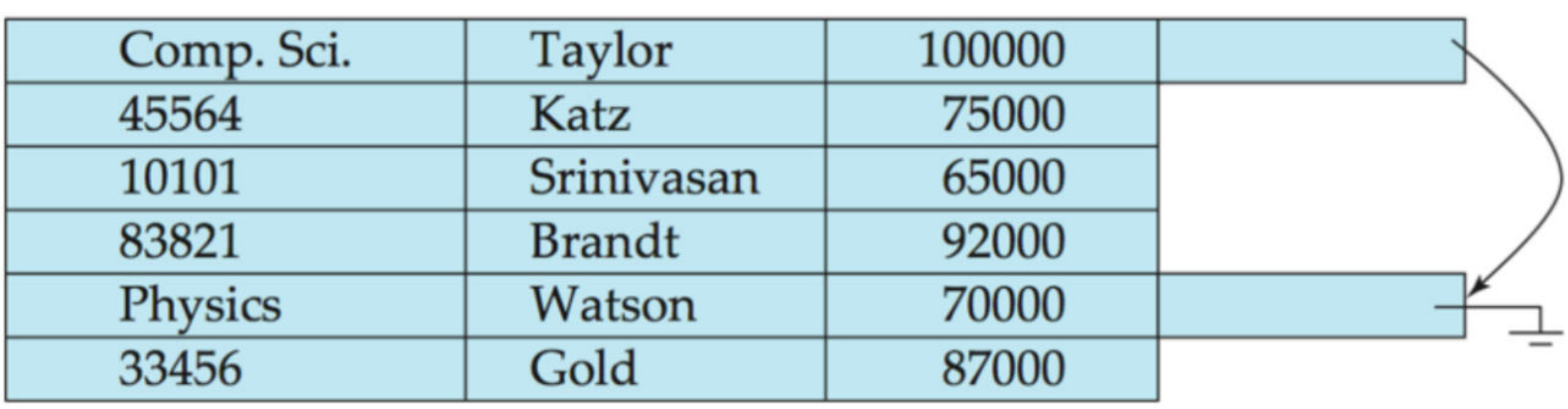
instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering of department and instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000

- Good for queries involving $department \bowtie instructor$ and for queries involving one single department and its instructors
- Bad for queries involving only $department$
- Results in variable size records
- Can add pointers chains to link records of a particular relation



Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

Data Dictionary Storage

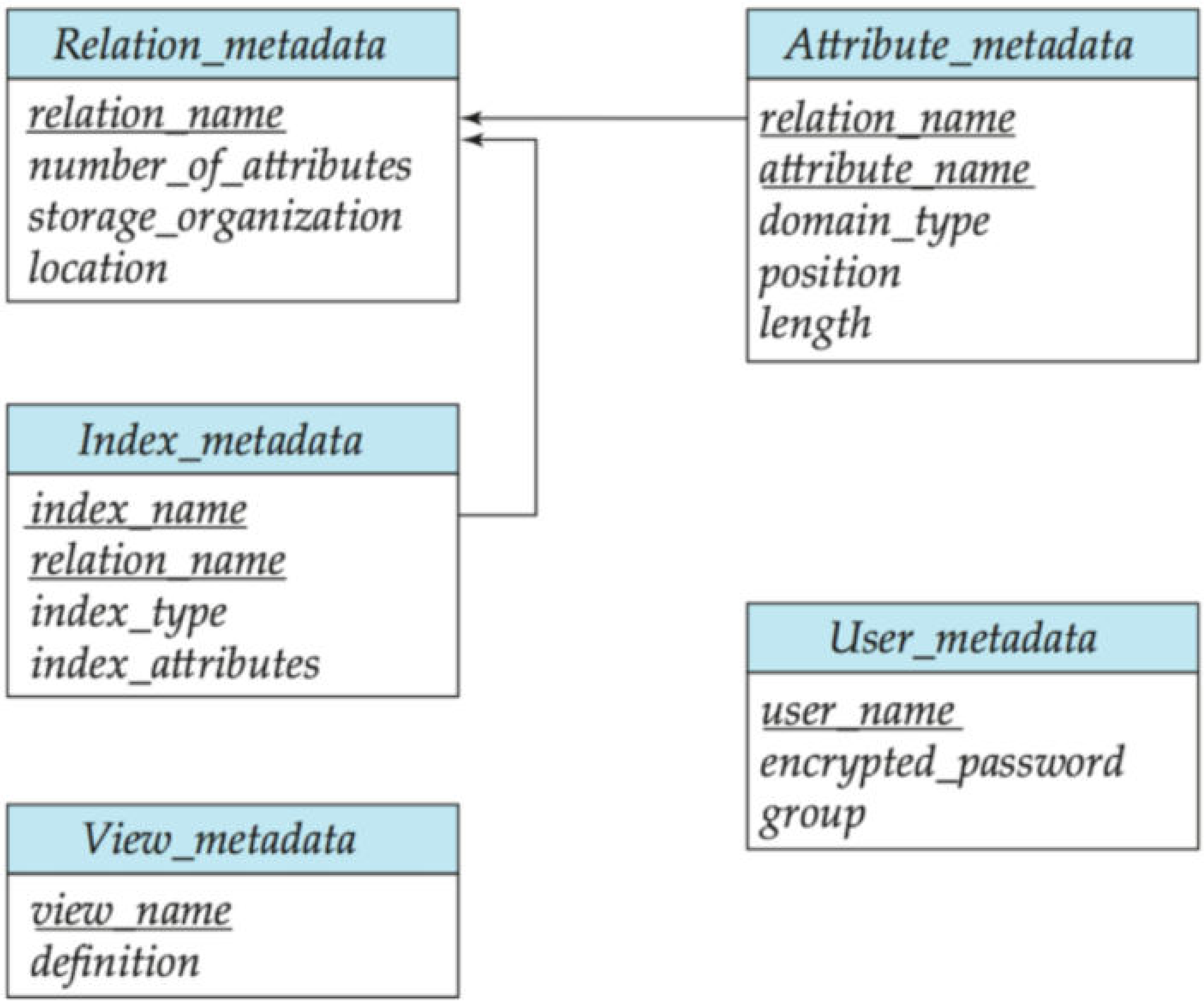
Data Dictionary (also, **System Catalog**) stores ***metadata*** (data about data) such as:

- Information about relations
  - names of relations
  - names, types and lengths of attributes of each relation
  - names and definition of views
  - integrity constraints
- User and accounting information, including password
- Statistical and descriptive data
  - Number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/...)
  - Physical location of relation

- Information about indices

Relational Representation of System metadata

- Relational representation on disk
- Specialized data structures designed for efficient access, in memory



Storage Access

- A database file is partitioned into fixed-length storage units called ***blocks***
    - Blocks are units of both storage allocation and data transfer
- Database system seeks to minimize the number of block transfers between the disk and the memory
    - We can reduce the number of disk accesses by keeping as many blocks as possible in the main memory
- **Buffer** → Portion of the main memory available to store copies of disk blocks
- **Buffer Manager** → Subsystem responsible for allocating buffer space in the main memory

Buffer Manager

- Programs call on the buffer manager when they need a block from the disk
    - If the block is already in the buffer, buffer manager returns the address of the block in the main memory
    - If the block is not in the buffer, the buffer manager
        - Allocates space in the buffer of the block
            - Replacing (throwing out) some other block, if required, to make space for the new block
            - Replaced block written back to disk only if it was modified since the most recent time that it was written to / fetched from the disk
        - Reads the block from the disk to the buffer, and returns the address of the block in the main memory to the requester

Buffer Replacement Policies

- Most Operating Systems replace the block ***least recently used (LRU strategy)***
- Idea behind LRU — Use past pattern of block references as a predictor of future references

- Queries have well-defined access patterns (such as sequential scans) and a database system can use the information in a user's query to predict future references
    - LRU may be a bad strategy for certain access patterns involving repeated scans of data
        - For example → When computing the join of 2 relations r and s by a nested loop

          for each tuple tr of r do

            for each tuple ts of s do

              if the tuples tr and ts match ...
    - Mixed strategy with hints on replacement strategy provided by the query optimized is preferable

---

- **Pinned block** → Memory block that is not allowed to be written back to the disk
- **Toss-immediate strategy** → Frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** → System must pin the block currently being processed
    - After the final tuple of that block has been processed, the block is unpinned and it becomes the most recently used block
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
    - For example → the data dictionary is frequently accessed
        - Heuristic → keep data-dictionary blocks in the main memory buffer
- Buffer managers also support forced output of blocks for the purpose of recovery

3, 4, 1, 4, 2, 3, 1, 4, 2, 3 ⇒ 3 buffer slots ⇒ How many misses? 9

bcoz '3' was least recently used till this pt.

" '1' ——— "

3	4	1	4	2	3	1	4	2	3
3	3	3	3	2	2	2	4	4	4
	4	4	4	4	4	1	1	1	3
		1	1	1	3	3	3	2	2
Miss	Miss	M	Hit	M	M	M	M	M	M

Week 9

- **Fact 1**: A tree with n nodes has $n-1$ edges
- **Fact 2**: The maximum number of nodes at level l of a binary tree is 2^l.
- **Fact 3**: If h is the height of a binary tree of n nodes, then:
  - $h+1 \le n \le 2^{h+1}-1$
  - $\lceil \lg(n+1) \rceil - 1 \le h \le n-1$
  - $O(\lg n) \le h \le O(n)$
  - For a k-ary tree, $O(\lg_k n) \le h \le O(n)$

Types of Indices

- **Ordered Indices:** Search keys are stored in sorted order
- **Hash indices:** Search keys are distributed uniformly across buckets using a hash function.

Ordered Indices

Primary Index

- In a sequentially ordered fine, the index whose search key specifies a sequential order of the file.
- also known as **clustering index**

Secondary Index

- an index whose search key specifies an order different from the sequential order of the file
- also known as **non-clustering index**

Index-Sequential File

ordered sequential file with a primary index

Dense Index

Index record appears for every search-key value in the file

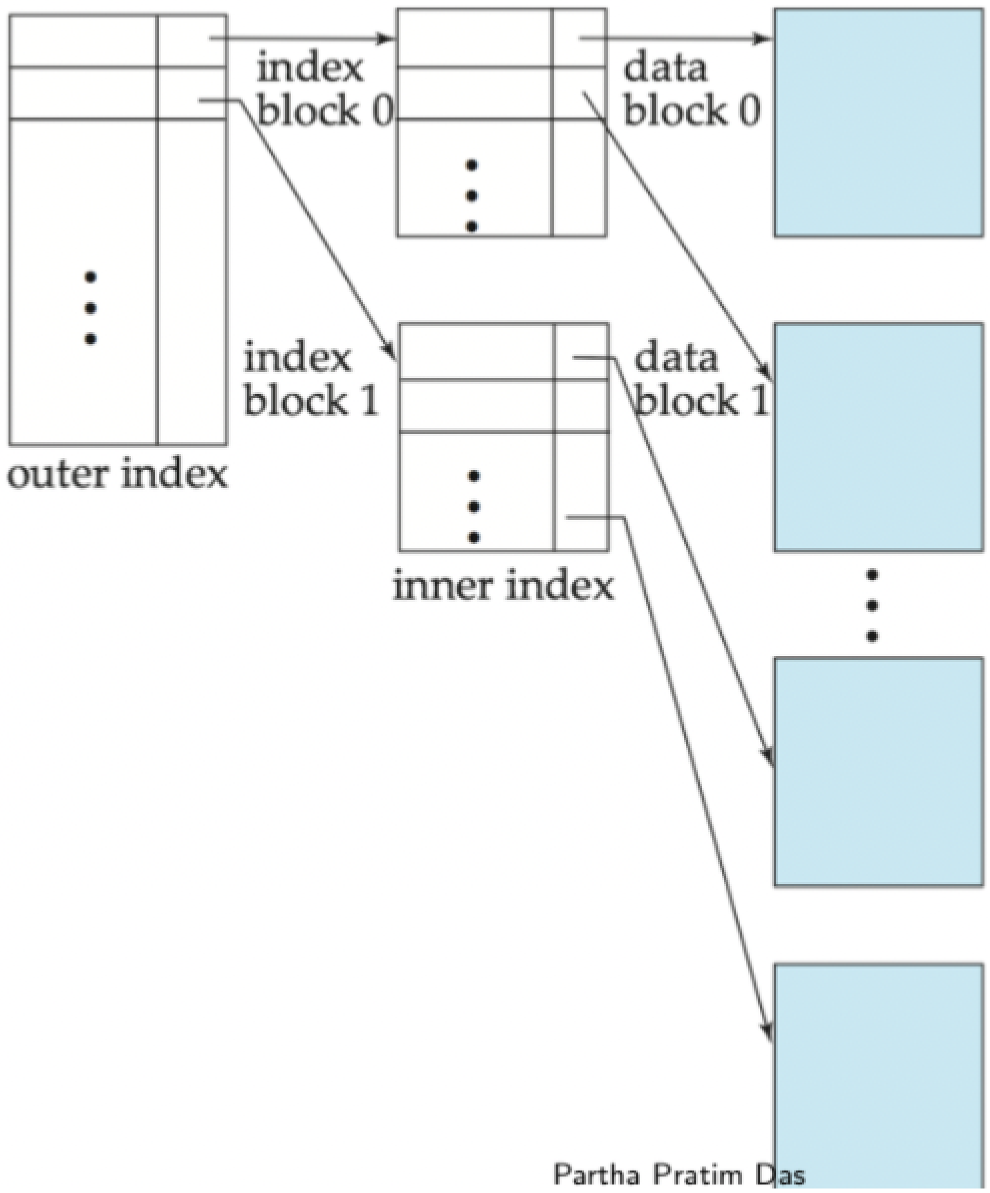
Sparse Index

- Contains index records for only some search key values. (applicable Only when records are sequentially ordered on search key)
- less space and maintenance overhead
- slower than dense index for locating records

Multilevel Index

If primary index does not fit in memory, access becomes expensive

- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it
  - outer index – a sparse index of primary index
  - inner index – the primary index file



B^+ Tree

- balanced tree
- link nodes are linked using linked lists
- all leaf nodes remain at the same height
- max no of nodes to be accessed in a lookup operation → $log_{\frac{n}{2}}(k)$
- max no of pointers root node can have → n
- min no of pointers root node can have → 2
- A non-empty B+ tree must have min of one key and max of p-1 keys where p is order of tree.

B^+ vs B Trees

- The size of non leaf nodes in B-Tree are larger compared to B^+ tree. Thus, B-trees typically have greater depth than corresponding B^+ trees.
- In B trees, leaf nodes are not linked together. In B^+ trees, leaf nodes are linked using a linked list.
- A non-root node of B tree of order p has minimum p/2 child node pointers and (p-1)/2 keys.

Hash function

- A hash function maps a data of arbitrary size to fixed size.
- From the perspective of hashing, a **bucket** is a unit of Storage containing one or more records.
- An ideal hash function maps the same number of search key values to each bucket from the site of all possible values.
- An ideal hash function maps the same number of search key values to each packet irrespective of the actual distribution of search-key values in the file.

Bucket Overflow

- insufficient buckets
- multiple records have same search-key values
- chosen hash function produces non-uniform distribution of key values.

Bitmap Index

Let **n** be the number of records of a relation **R**, and **A** be an attribute of a relation **R** having **m** number of distinct values. If we build a bitmap index on **A** of **R** the **size of bitmap index** will be **m x n** bits.