Database Management Systems

Dr. Dhananjoy Bhakta
Assistant Professor, CSE Department
Indian Institute of InformationTechnology Ranchi

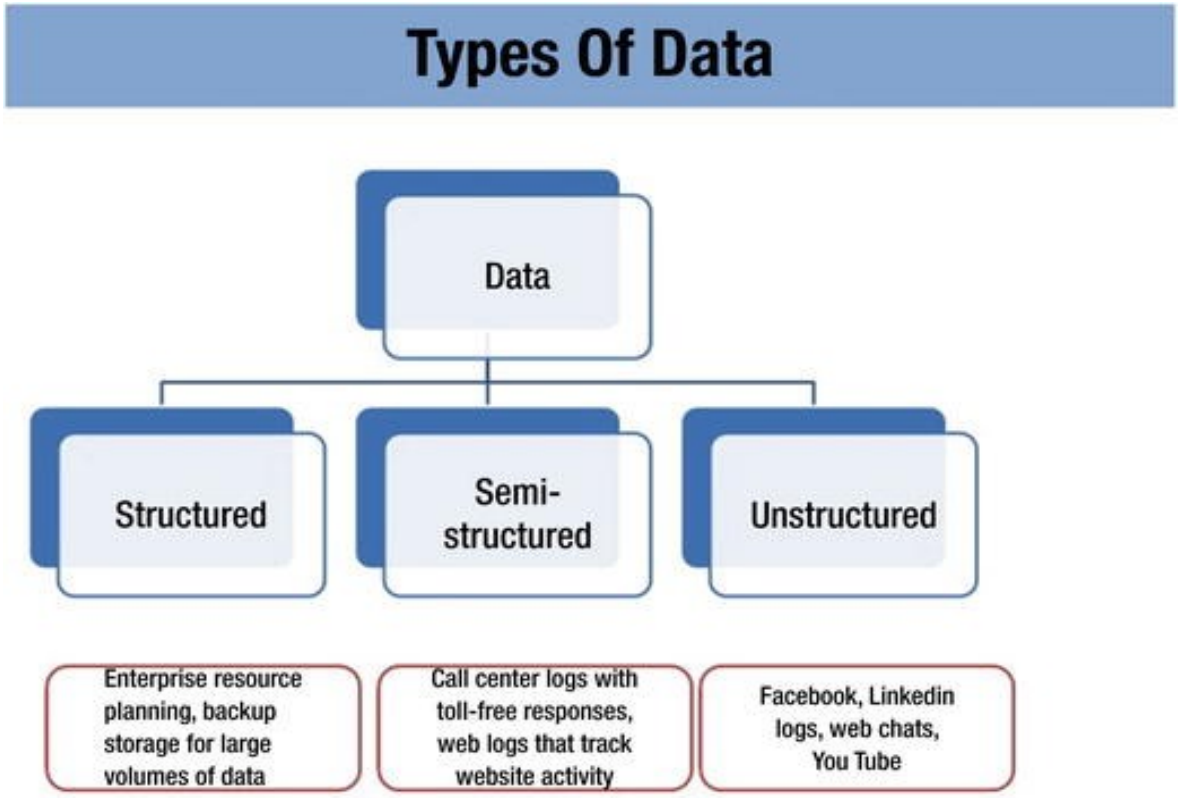
Topics Cover

- Introduction
  - Basic concepts-Data, Database, DBMS
  - Advantages of a DBMS over file-processing systems
  - Database architecture
  - Data abstraction
  - Data Models and data independence
  - Components of DBMS and overall structure of DBMS

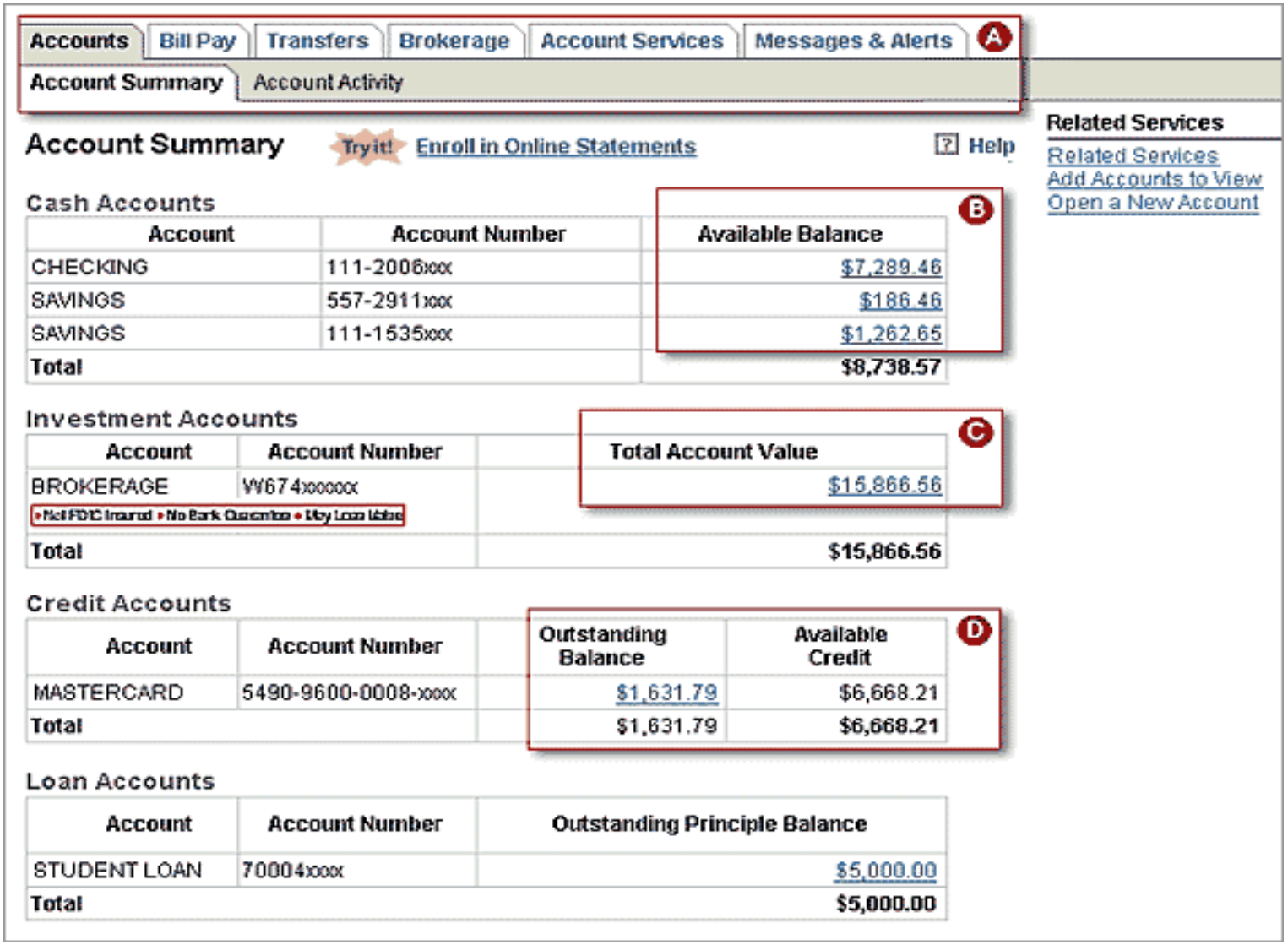
What is Database Systems

- Database: where application related data is stored
- Collection of files
- Data: known facts that can be recorded & can have implicit meaning. Simply it is information
- Software: to manage info. i.e. storing, processing & extracting data
- Data persistence: how long user wants the data would be available
- Structured, Semi-structured and Unstructured data

What is Database Systems



What: Database Systems Today



Accounts | Bill Pay | Transfers | Brokerage | Account Services | Messages & Alerts (A)

Account Summary | Account Activity

Account Summary Try it! Enroll in Online Statements Help

Related Services
Related Services
Add Accounts to View
Open a New Account

Cash Accounts (B)

Account	Account Number	Available Balance
CHECKING	111-2006xxx	$7,289.46
SAVINGS	557-2911xxx	$186.46
SAVINGS	111-1535xxx	$1,262.65
Total		**$8,738.57**

Investment Accounts (C)

Account	Account Number	Total Account Value
BROKERAGE	W674xxxxxx	$15,866.56
• Not FDIC Insured • No Bank Guarantee • May Lose Value		
Total		**$15,866.56**

Credit Accounts (D)

Account	Account Number	Outstanding Balance	Available Credit
MASTERCARD	5490-9600-0008-xxxx	$1,631.79	$6,668.21
Total		$1,631.79	**$6,668.21**

Loan Accounts

Account	Account Number	Outstanding Principle Balance
STUDENT LOAN	70004xxxx	$5,000.00
Total		**$5,000.00**

So… What is a Database?

- A Database:
  - collection of interrelated data + description of data
- A Conceptual Model to Describe Data
  - Entities (e.g., teams, games)
  - Relationships (e.g. The A's are playing in the World Series)
- Might surprise you how flexible this is
  - Web search:
    - Entities: words, documents
    - Relationships: word in document, document links to document.
  - P2P filesharing:
    - Entities: words, filenames, hosts
    - Relationships: word in filename, file available at host

Types of Databases

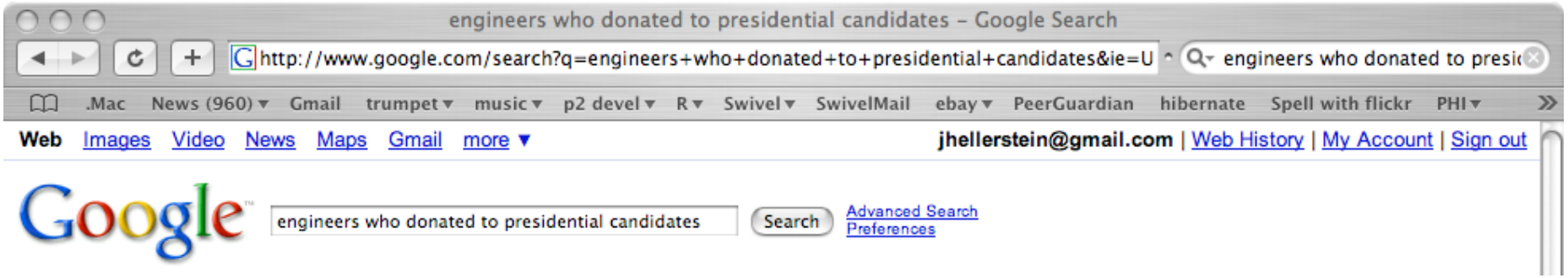
- Relational databases
  - Microsoft SQL Server, Oracle Database, MySQL, PostgreSQL and IBM Db2
- NoSQL databases
  - Apache Cassandra, MongoDB, CouchDB, and CouchBase
- Cloud databases
  - Microsoft Azure SQL Database, Amazon Relational Database Service, Oracle Autonomous Database
- Object oriented databases
  - Wakanda, ObjectStore
- Hierarchical databases
  - IBM Information Management System (IMS), Windows Registry
- Graph databases
  - Datastax Enterprise Graph, Neo4J

What is a Database Management System?

- A Database Management System (DBMS) is:
  - A software system designed to store, manage, and facilitate access to databases.
- Typically this term used narrowly
  - Relational databases with transactions
    - E.g. Oracle, DB2, SQL Server

What: “Search” vs. Query

- Web query is specifically a search engine query, which is a user's request to a search engine to generate results based on a specific user-provided topic or keyword phrase.
- Queries are used to find specific data in a database by providing or filtering explicit criteria.



- If it isn’t “published”, it can’t be searched!

Advantages of a Traditional DBMS

- Efficiency, easeness, faster-No need to write a new program to carry out each new task
- Data redundancy and inconsistency-Multiple file formats, duplication of information in different files
- Data isolation-Multiple files and formats
- Atomicity of updates

– Failures may leave database in an inconsistent state with partial updates carried out

– Example: Transfer of funds from one account to another should either complete or not happen at all

Advantages of a Traditional DBMS

- 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: Data security/ restricting unauthorized access
  - Hard to provide user access to some, but not all, data
- Integrity constraints
- Relationship among data
- So why not use them always?
  - Expensive/complicated to set up & maintain
  - This cost & complexity must be offset by need
  - General-purpose, not suited for special-purpose tasks (e.g. text search!)

Summary

- Relational DBMS: maintain/query structured data
  - broadly applicable
  - can manipulate data and exploit *semantics*
  - recovery from system crashes
  - concurrent access
  - robust application development and *evolution*
  - data integrity and security

What: Is a File System a DBMS?

- Thought Experiment 1:
  - You and your project partner are editing the same file.
  - You both save it at the same time.
  - Whose changes survive?

A) Yours B) Partner's C) Both D) Neither E) ???

- Thought Experiment 2:
  - You're updating a file.
  - The power goes out.
  - Which changes survive?

A) All B) None C) All Since Last Save D) ???

OS Support for Data Management

- Data can be stored in RAM
  - this is what every programming language offers!
  - RAM is fast, and random access
  - Isn't this heaven?
- Every OS includes a File System
  - manages *files* on a magnetic disk
  - allows *open, read, seek, close* on a file
  - allows protections to be set on a file
  - drawbacks relative to RAM?

Database Management Systems

- 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
  - Human resources: employee records, salaries, tax deductions

Syllabus

- Unit 1: Introduction: Basic concepts, Advantages of a DBMS over file-processing systems, Data abstraction, Data Models and data independence, Components of DBMS and overall structure of DBMS, Data Modeling, entity, attributes, relationships, constraints, keys, E-R diagrams, Components of E-R Model.
- Unit 2: Relational Model: Relational Model: Basic concepts. Attributes and domains, concept of integrity and referential constraints, schema diagram. Relational Query Languages: Relational Algebra and Relational Calculus: Tuple relational and domain relational calculus.
- Unit 3: SQL: Introduction to SQL, Characteristics and advantages of SQL, SQL Data Types and Literals, DDL, Tables: Creating, modifying, deleting, Views: Creating, dropping, Updating using Views, DML, SQL Operators, SQL DML queries, SELECT query and clauses, Set Operations, Predicates and Joins, Set membership, Tuple variables, set comparison, ordering of tuples, aggregate functions, nested queries, Database modification using SQL Insert, Update and Delete queries, Concept of stored procedures, Query-by-example.
- Unit 4: Relational Database Design: Notion of normalized relations, functional dependency, decomposition and properties of decomposition, Normalization using functional dependency, Multi-valued dependency and Join dependency.
- Unit 5: Query Management and Transaction Processing: Selection operation, sorting and join operation, Transaction Concept, Components of transaction management, Concurrency and recovery system, Different concurrency control protocols such as timestamps and locking, validation, Multiple granularity, Deadlock handling.

References

Text Book:

- **Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database system concepts", 5th Edition, McGraw Hill International Edition.**
- **Raghu Ramkrishnan, Johannes Gehrke, "Database Management Systems", Second Edition, McGraw Hill International Editions.**

Reference Book:

- **Rob Coronel, "Database systems: Design implementation and management", 4th Edition, Thomson Learning Press.**
- **Ramez Elmasri and Shamkant B. Navathe, "Fundamental Database Systems", Third Edition, Pearson Education, 2003.**

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 averages (GPA) and generate transcripts
- In the early days, database applications were built directly on top of file systems

Example: University Database

- Conceptual schema:
  - Students(sid text, name text,
    login text, age integer,
    gpa float)
  - Courses(cid text, cname text,
    credits integer)
  - Enrolled(sid text, cid text,
    grade text)
- Physical schema:
  - Relations stored as unordered files.
  - Index on first column of Students.
- External Schema (View):
  - Course_info(cid text,
    enrollment integer)

Data Independence

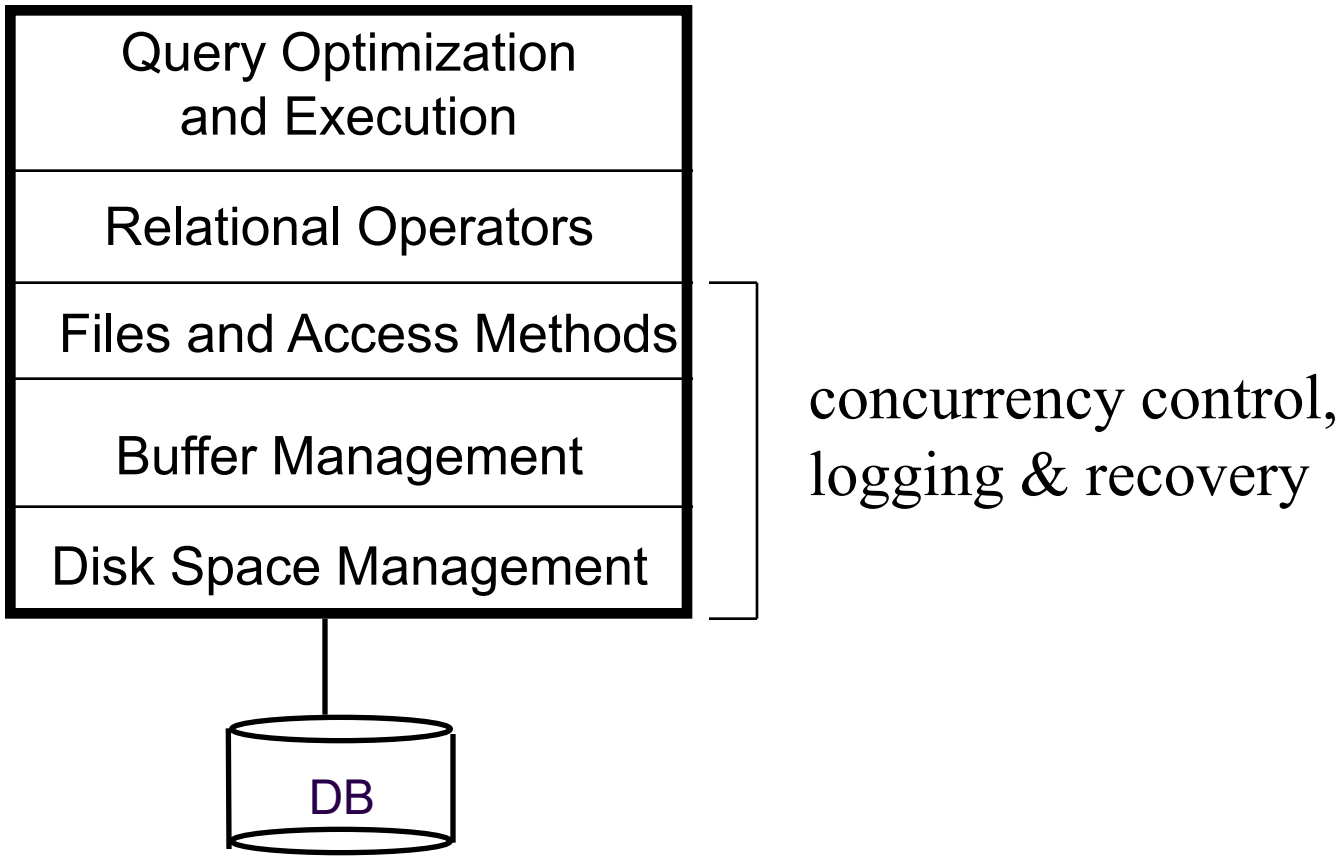
- Applications (should be) insulated from how data is structured and stored.
- Logical Data Independence: data at conceptual level & external level schema must be independent.
- Protection from changes in *logical* structure of data
  - Difficult to achieve.
  - Changes in conceptual schema are reflected in user's view

Data Independence

- Physical Data Independence: any change in physical location of tables & indexes should not affect conceptual level/ external view of data.
- Protection from changes in *physical* structure of data
  - Easy to achieve & implemented by most of the DBMS
- Q: Why is this particularly important for DBMS?

Because databases and their associated applications persist.

Typical DBMS Structure



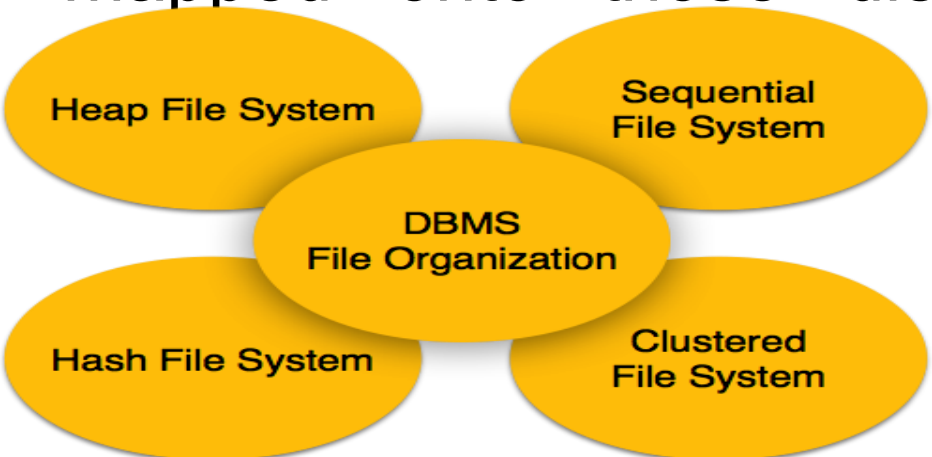
Query Optimization and Execution

- A database system generates an efficient query evaluation plan, which minimizes its cost. This type of task performed by the database system and is known as Query Optimization. For optimizing a query, the query optimizer should **have an estimated cost analysis of each operation**
- **Query Processing** includes translations on high level Queries into low level expressions that can be used at physical level of file system, query optimization and actual execution of query to get the actual result.

Relational Operators

- Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output.
- Uses operators to perform queries. An operator can be either unary or binary.
- They accept relations as their input and yield relations as their output.
- Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

- Relative data and information is stored collectively in file formats.
- A file is a sequence of records stored in binary format.
- A disk drive is formatted into several blocks that can store records.
- File records are mapped onto those disk blocks.



Disk Space Management

- DBMS stores information on disks.
  - In an electronic world, disks are a mechanical anachronism!
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Disk Space Management

- Lowest layer of DBMS software manages space on disk (using OS file system or not?).
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Best if a request for a *sequence* of pages is satisfied by pages stored sequentially on disk!
  - Responsibility of disk space manager.
  - Higher levels don't know how this is done, or how free space is managed.
  - Though they may assume sequential access for files!
    - Hence disk space manager should do a decent job.

Buffer Management

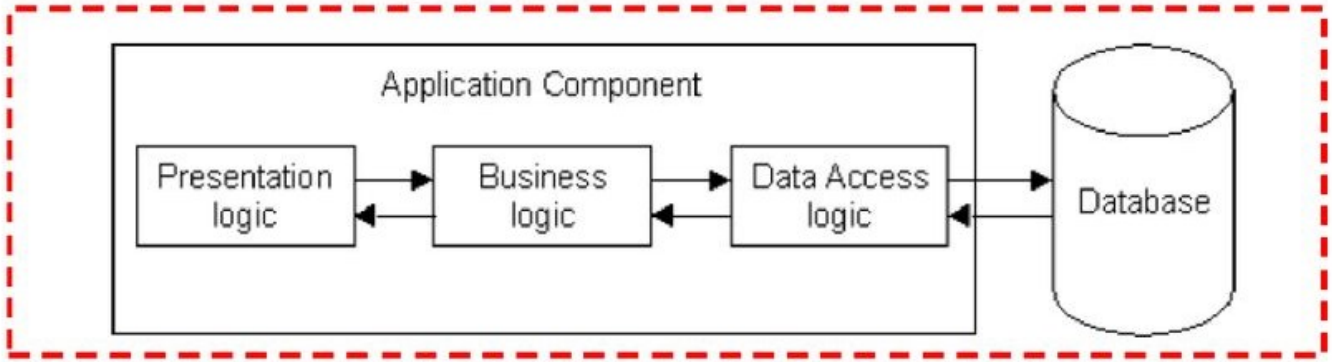
- *Data must be in RAM for DBMS to operate on it!*
- *Buffer Mgr hides the fact that not all data is in RAM*
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, etc.

Database Architecture

- Types
  - One -Tier: database is readily available on the client m/c. Any request made by client doesn't require n/w connection
  - Two -Tier: database system is present at server & DB application is present at client m/c
  - Three –Tier: another layer is present between the client machine and server machine.
    - the client application doesn’t communicate directly with the database systems present at the server machine,
    - rather the client application communicates with server application and
    - the server application internally communicates with the database system present at the server.
  - N –Tier: presentation, application, and database layers. These three layers can be further subdivided into different sub-layers depending on the requirements.

Database Architecture

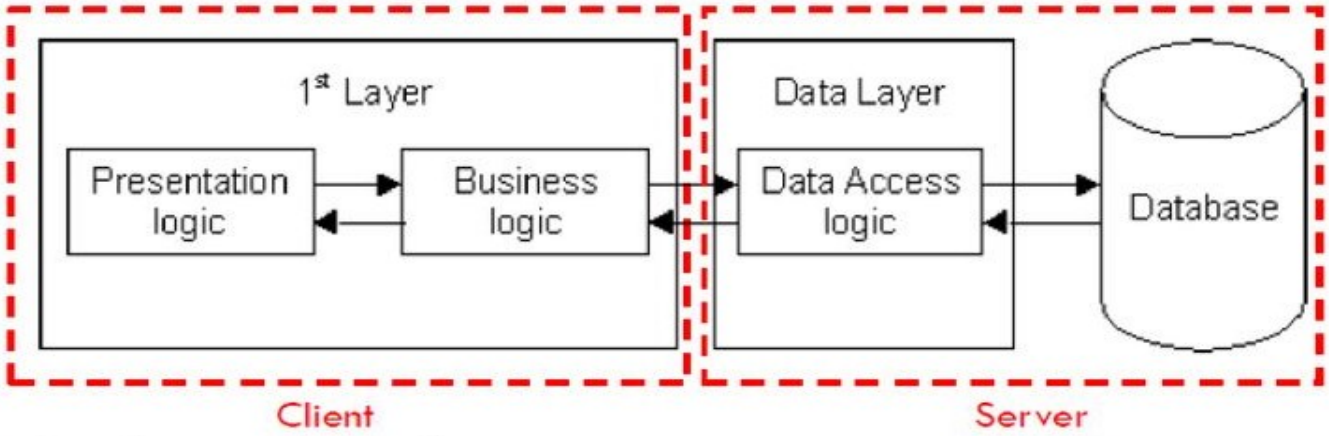
1-Tier Architecture



- All 3 layers are on the same machine
  - All code and processing kept on a single machine
- Presentation, Logic, Data layers are tightly connected
  - Scalability: Single processor means hard to increase volume of processing
  - Portability: Moving to a new machine may mean rewriting everything
  - Maintenance: Changing the layer requires changing other layers

Database Architecture

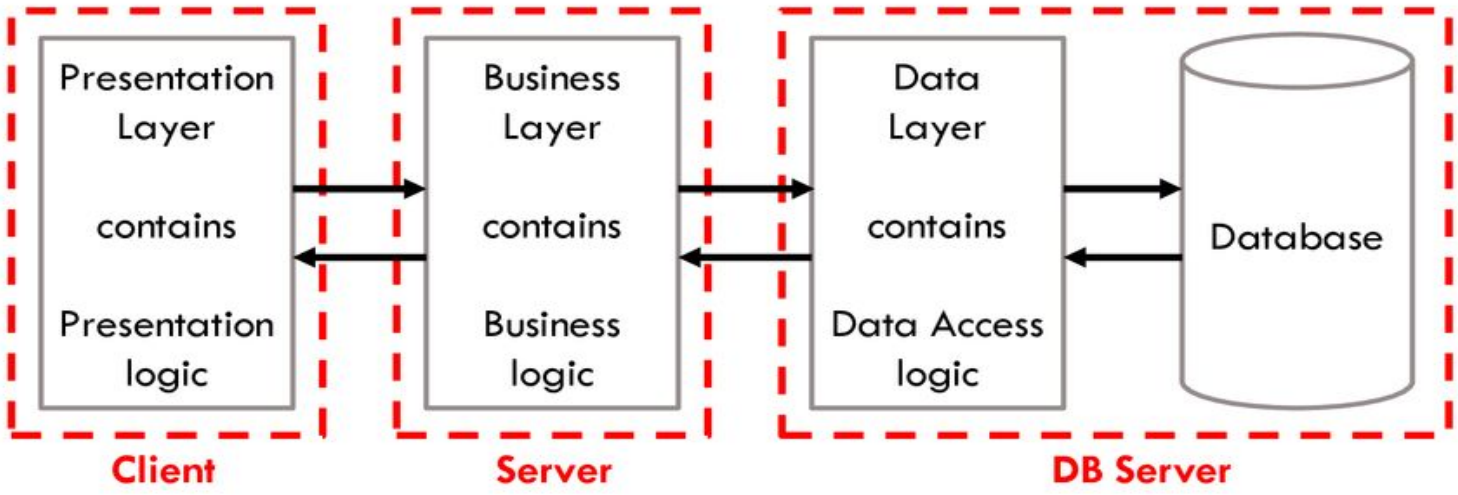
2-Tier Architecture



- Database runs on Server
  - Separated from client
  - Easy to switch to a different database
- Presentation and logic layers still tightly connected
  - Heavy load on server
  - Potential congestion on network
  - Presentation still tied to business logic

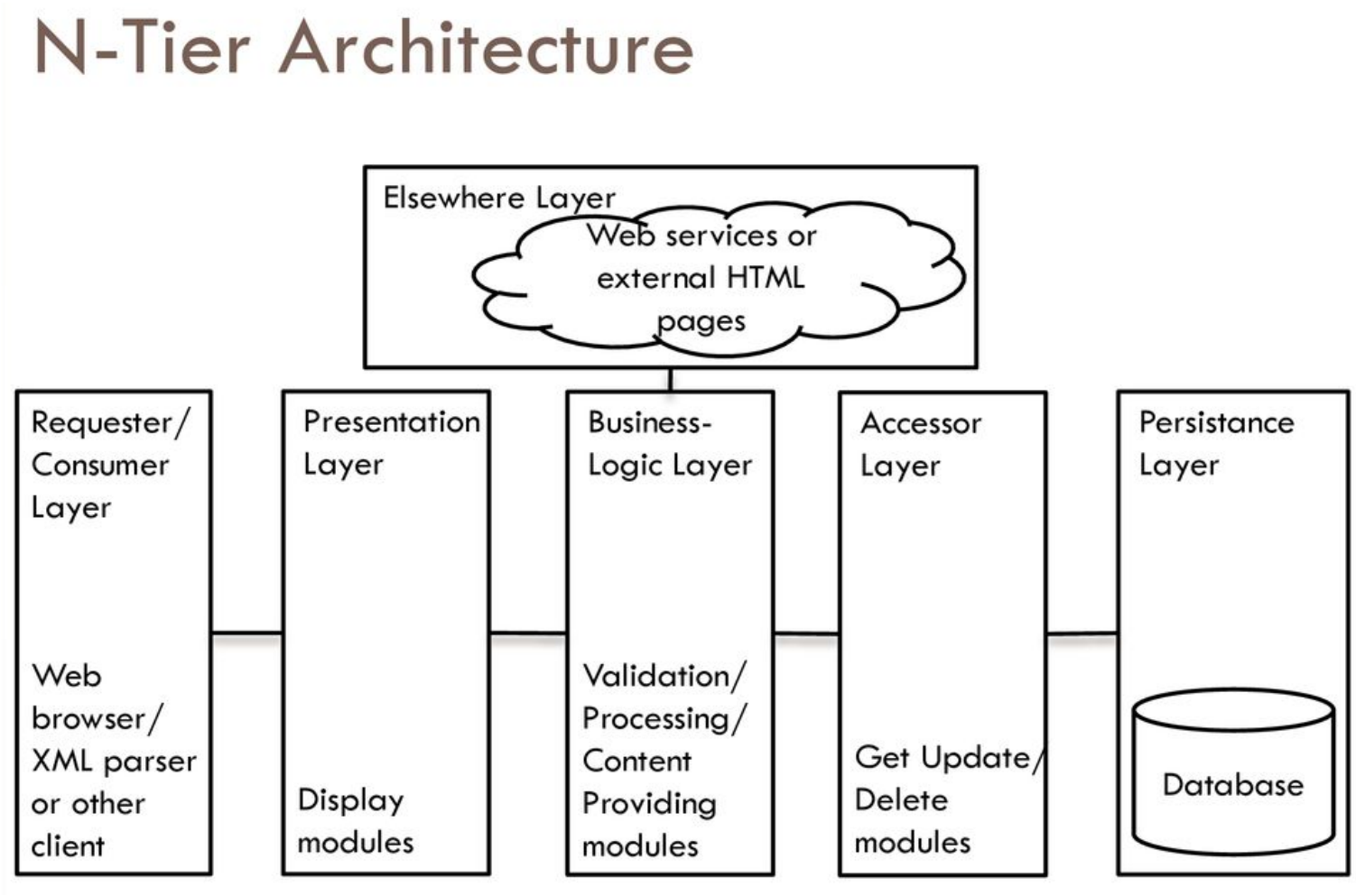
Database Architecture

3-Tier Architecture

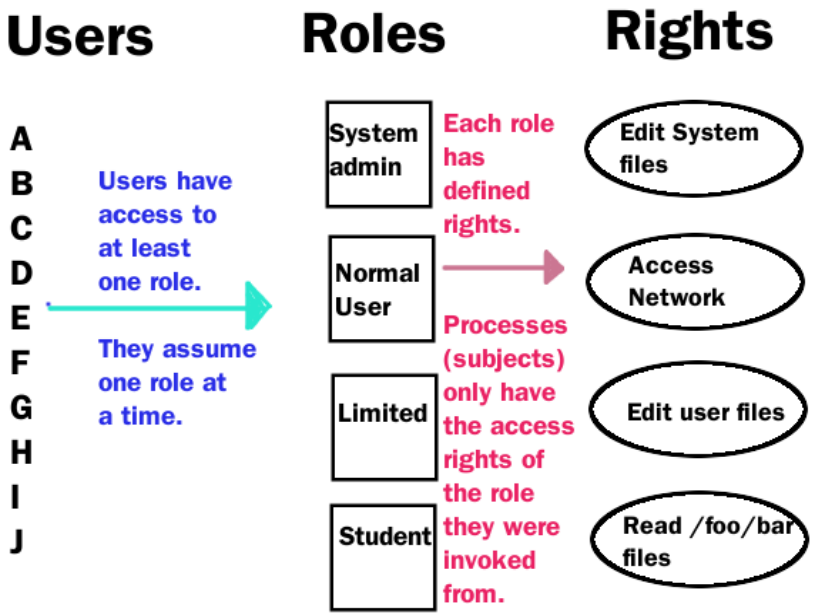


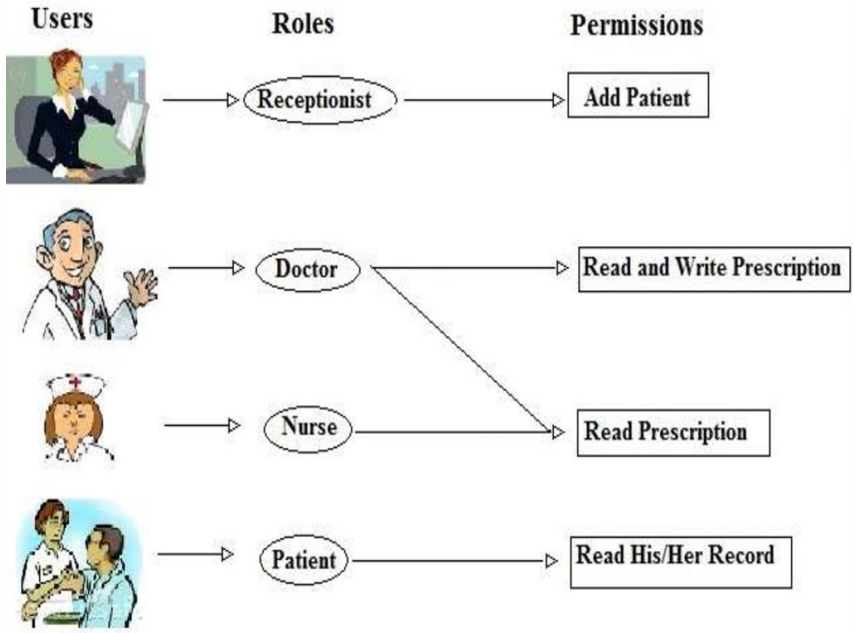
- Each layer can potentially run on a different machine
- Presentation, logic, data layers disconnected

Database Architecture



Role based model



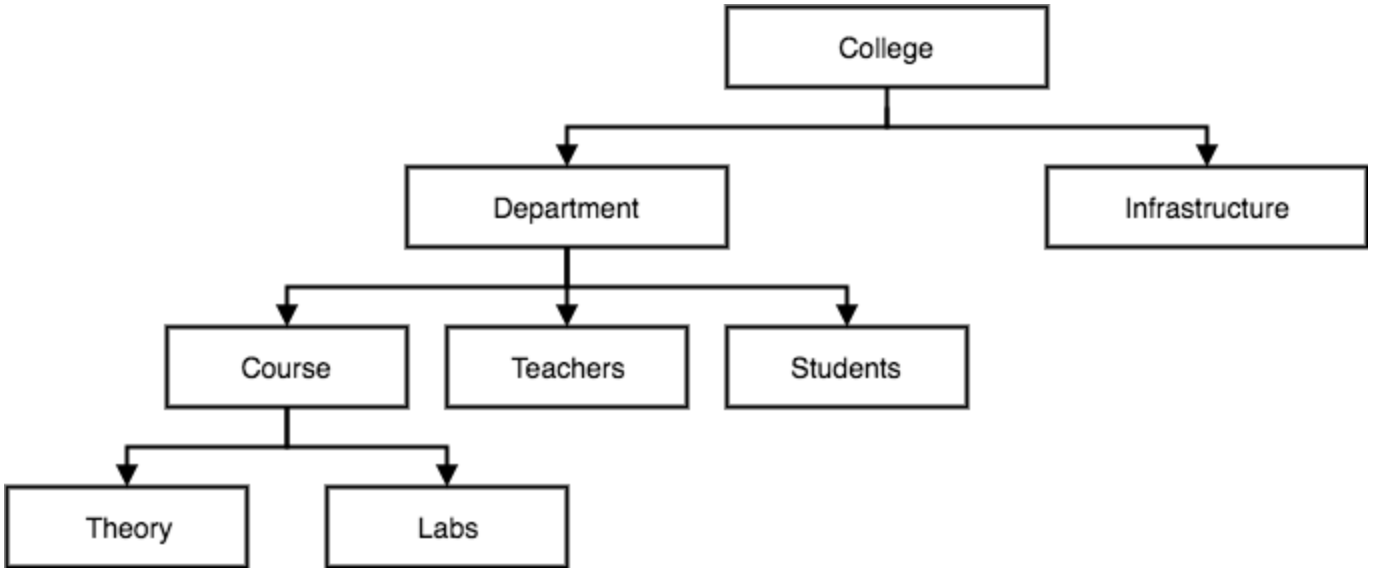


Database Models

- Defines the
  - logical design
  - structure of database
  - how data will be stored, accessed & updated in DBMS
- Types
  - Hierarchical
  - Network
  - E-R Model
  - Relational model

Hierarchical Database Model

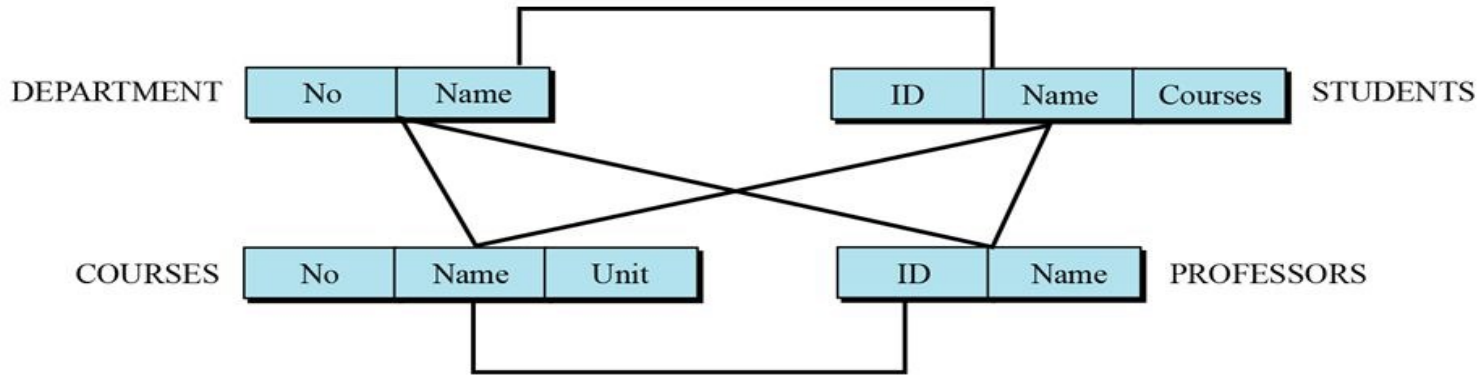
- Organizes data into a tree like structure, with a single root to which all the other data is linked



Network Database Model

Network database model

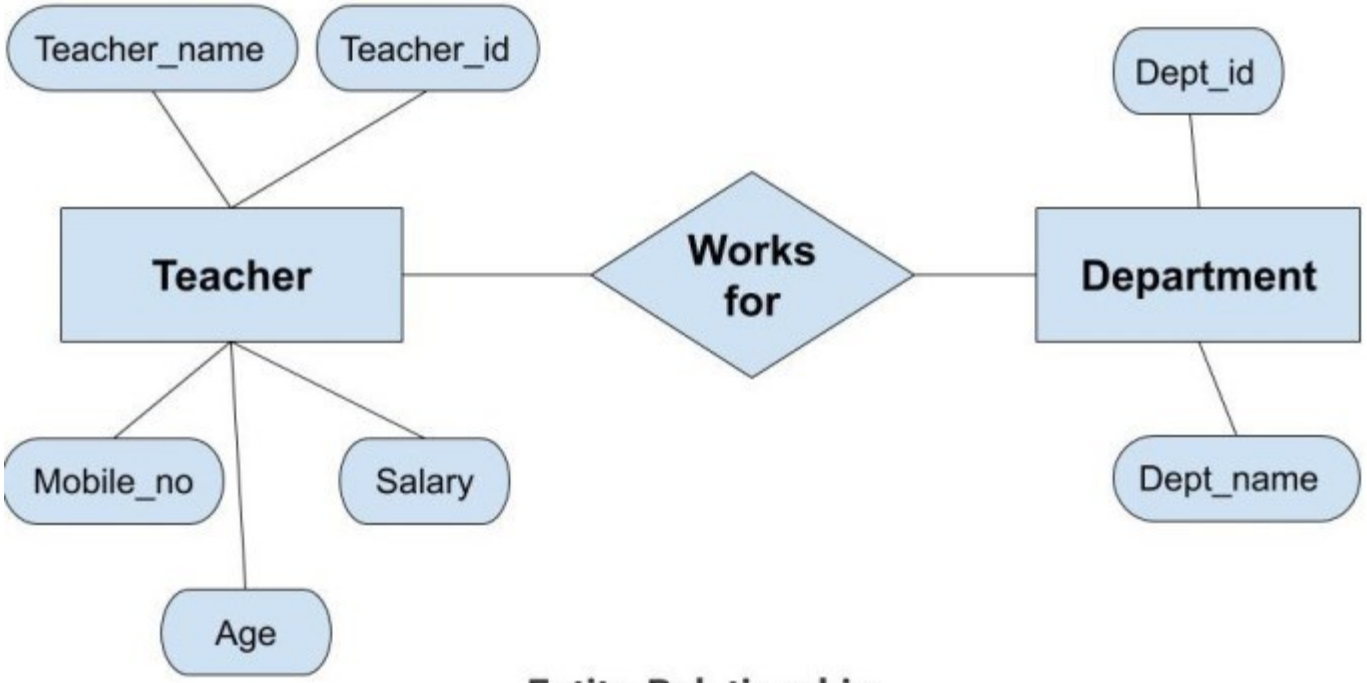
In the network model, the entities are organized in a graph, in which some entities can be accessed through several path



An example of the network model representing a university

E R Database Model

- Relationships are created by dividing object of interest into entity & its characteristics into attributes. Diff. entities are related using relationships.



Entity-Relationship Model

Relational Database Model

- Basic structures of data in the relational model is tables.
- All the info. related to a particular type is stored in rows of that table. So, tables are also known as relations & all the same type of info. are stored in a column

student_Id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_Id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_Id	subject_Id	marks
1	1	98
1	2	78
2	1	76
3	2	88

Instances & Database Schema

- Databases change over time as info. is inserted & deleted. The collection of info stored in the database at a particular moment is called an instance of the database
- Analogous to the value of a variable
- Overall design of the database is called database schema
- Schemas are changed infrequently

Database Engine

- A database engine (or storage engine) is the underlying software component that a database management system (DBMS) uses to create, read, update and delete (CRUD) data from a database.
- The functional components of a database system can be broadly divided into
  - Storage manager
  - Query processing
  - Transaction manager

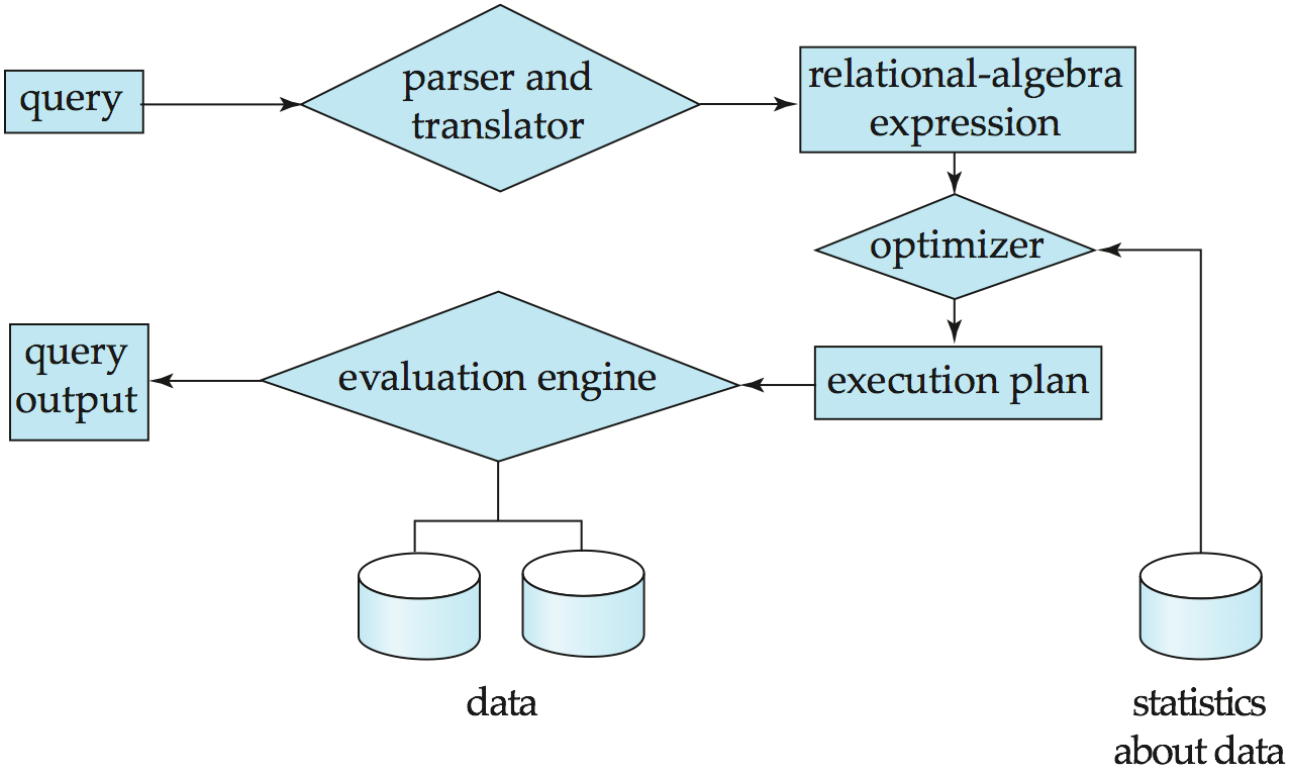
Storage Management

- **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 to the following tasks:
  - Interaction with the OS file manager
  - Efficient storing, retrieving and updating of data

What is Query Processing?

- Query processing: Activities involved in extracting data from a database.
  - Translation of queries in high-level DB languages into expressions that can be used at physical level of file system.
  - Includes query optimization and query evaluation.
- Three basic steps:
  1. Parsing and Translation
  2. Optimization
  3. Evaluation

Query Processing



Parsing and translation

- Translate the query into its internal form.
  - This is then translated into relational algebra.
- Parser checks syntax, verifies relations.
- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to

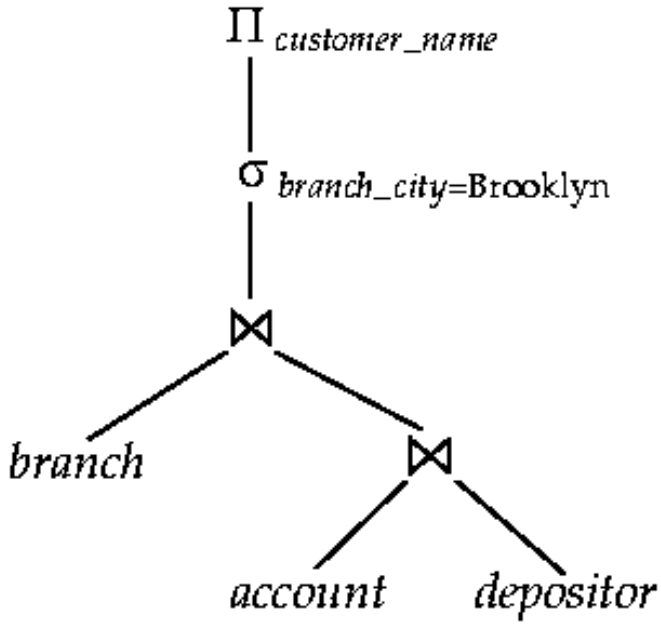
$$\Pi_{balance}(\sigma_{balance<2500}(account))$$

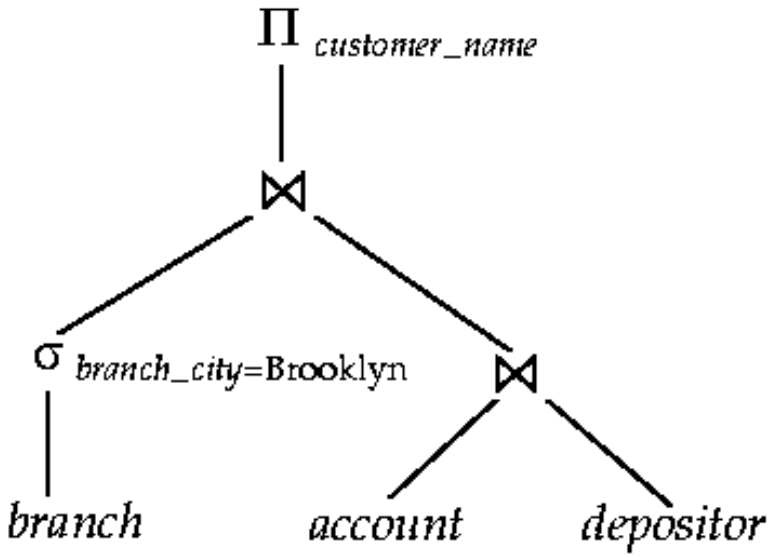
Parsing and translation (cont.)

- Each relational algebra operation can be evaluated using one of several different algorithms
- Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Evaluation-plan: Annotated expression specifying detailed evaluation strategy.
  - e.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance $\geq$ 2500

Query Optimization

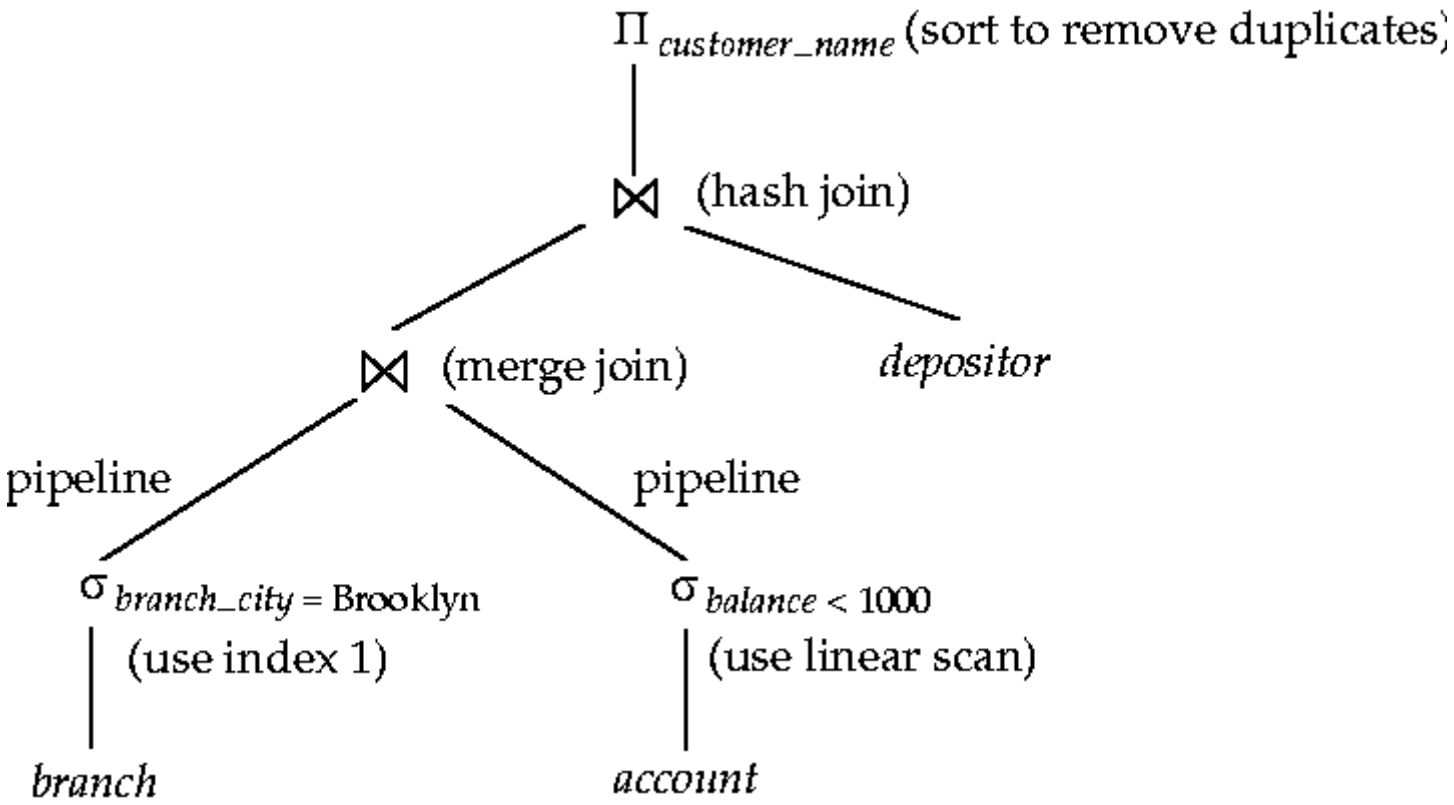
- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation
  - Select the best query





Query Optimization

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



Query Optimization

- Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.
    - How to measure query costs
    - How to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Query Optimization

- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

Query Optimization

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in cost-based query optimization
  - Generate logically equivalent expressions using equivalence rules
  - Annotate resultant expressions to get alternative query plans
  - Choose the cheapest plan based on estimated cost

Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
- Parsed execution plan for previously executed SQL statements is stored in Shared pool (a portion of memory or buffer).
  - If a new SQL statement (query) is exactly the same string as the one in the shared pool, no need to call optimizer and recalculate the execution plan for the SQL statement.

Example

- Select book, author, price from Book where price>300;
- $\sigma_{price>300}(\Pi_{book,\ author,\ price}(\text{Book}))$
- $\Pi_{book,\ author,\ price}(\sigma_{price>300}(\text{Book}))$
- *Query tree*
- *Query optimizer choose the least costly plan*
- Query-execution engine takes the plan, executes, and returns the answers to the query in low level language

Transaction Management

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

Concurrent execution of user programs

- Why?
  - Essential for better performance of DBMS, as concurrent running of several user program keep utilizing CPU time efficiently, since disk accesses are frequent and are relatively slow in case of DBMS.
  - Utilize CPU while waiting for disk I/O
    - (database programs make heavy use of disk)
  - Also, a user's program may carry out many operations on the data returned from DB, but DBMS is only concerned about what data is read/written from/to the database.
  - Avoid short programs waiting behind long ones
    - e.g. ATM withdrawal while bank manager sums balance across all accounts

Concurrent execution

- Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed.
- DBMS ensures such problems don't arise: users can pretend they are using a single-user system.
- Example:
  - Bill transfers $100 from savings to checking

    Savings –= 100; Checking += 100
  - Meanwhile, Bill's wife requests account info.

  Bad interleaving:
    - Savings –= 100
    - Print balances
    - Checking += 100

  – Printout is missing $100 !

Key concept: **Transaction**

- A transaction is action, or series of actions, carried out by user or application, which accesses or updates contents of database.
- A sequence of database actions (reads/writes) executed atomically by DBMS
- Should take DB from one consistent state to another

Example

- Here, *consistency* is based on our knowledge of banking "semantics"
- DBMS provides (limited) automatic enforcement, via integrity constraints
  - e.g., balances must be >= 0

Locking example

- T1 (Bill): *Savings –= 100; Checking* += *100*
- T2 (Bill's wife): *Print(Checking); Print(Savings)*

– T1 and T2 both lock Savings and Checking objects

– If T1 locks Savings & Checking first, T2 must wait

- T1 (Bill): *Savings –= 100; Checking += 100*
- T2 (Bill's wife): *Print(Checking); Print(Savings)*

Suppose:

1. T1 locks Savings
2. T2 locks Checking

• Now neither transaction can proceed!

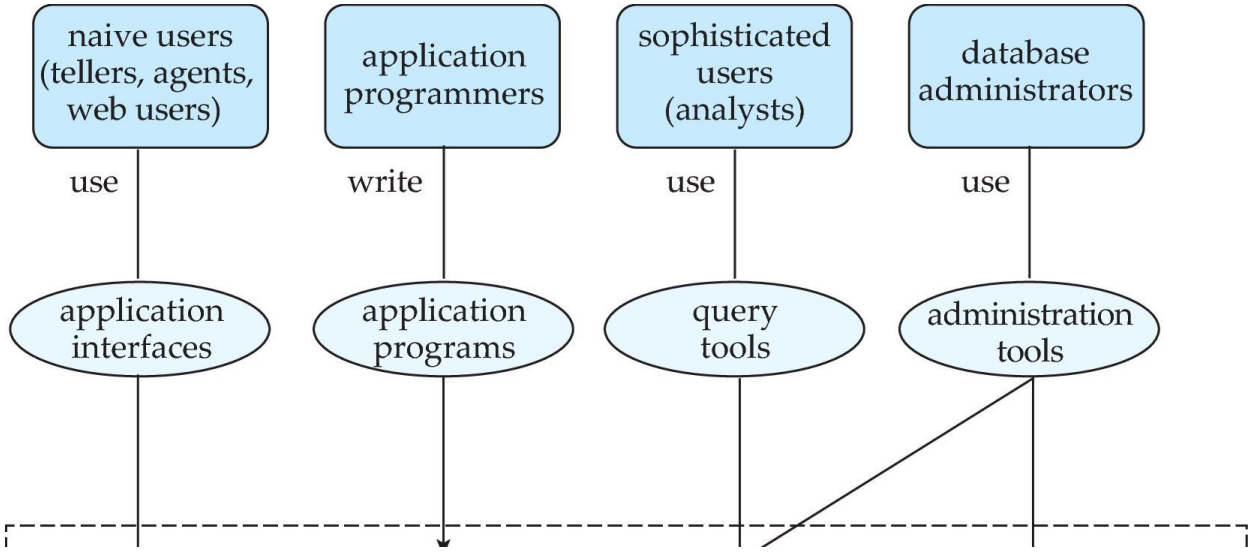
– called "deadlock"

– DBMS will abort and restart one of T1 and T2

– Need "undo" mechanism that preserves consistency

– Undo mechanism also necessary if system crashes between "Savings –= 100" and "Checking += 100" …

Database Users and Administrators



Database Users

- Users are differentiated by the way they expect to interact with the system
- Application programmers – interact with system through DML calls
- Sophisticated users – form requests in a database query language
- Specialized users – write specialized database applications that do not fit into the traditional data processing framework
- Naïve users – invoke one of the permanent application programs that have been written previously
  - E.g. people accessing database over the web, bank tellers, clerical staff

Database Administrator

- Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
- Database administrator's duties include:
  - Schema definition
  - Storage structure and access method definition
  - Granting user authority to access the database
  - Routine Maintenance
  - Monitoring performance and responding to changes in requirements
  - Responsible for recovery from failure

Database Manager

- Central s/w component of DBMS
- Database manager's duties include:
  - Interacts with file manager
  - Enforcing constraints or checks
  - Enforcing security (only authorized access)
  - Concurrency control
  - Backup & recovery

Data Dictionary

- A **data dictionary/ data repository/ system catalogue** is a file or a set of files that contains a **database's** metadata. The **data dictionary** contains records about other objects in the **database**, such as **data** ownership, **data** relationships to other objects, and other **data**.
- Data dictionary includes:
  - Name of the tables in the database
  - Constraints of a table i.e. keys, relationships, etc.
  - Columns of the tables that related to each other
  - Owner of the table
  - Last accessed information of the object
  - Last updated information of the object

Data Dictionary

An example of Data Dictionary can be personal details of a student −

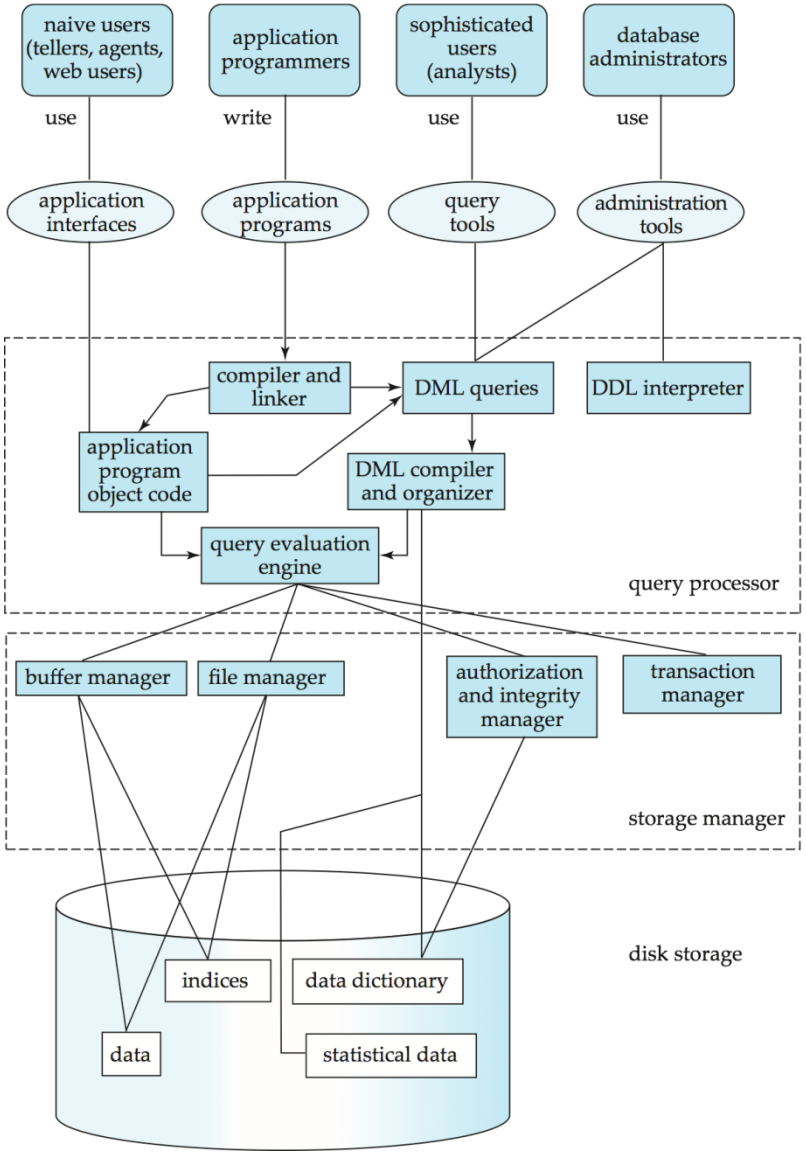
Example

<StudentPersonalDetails>

The following is the data dictionary for the above fields −

Student_ID	Student_Name	Student_Address	Student_City

Database System Internals



History of Database Systems

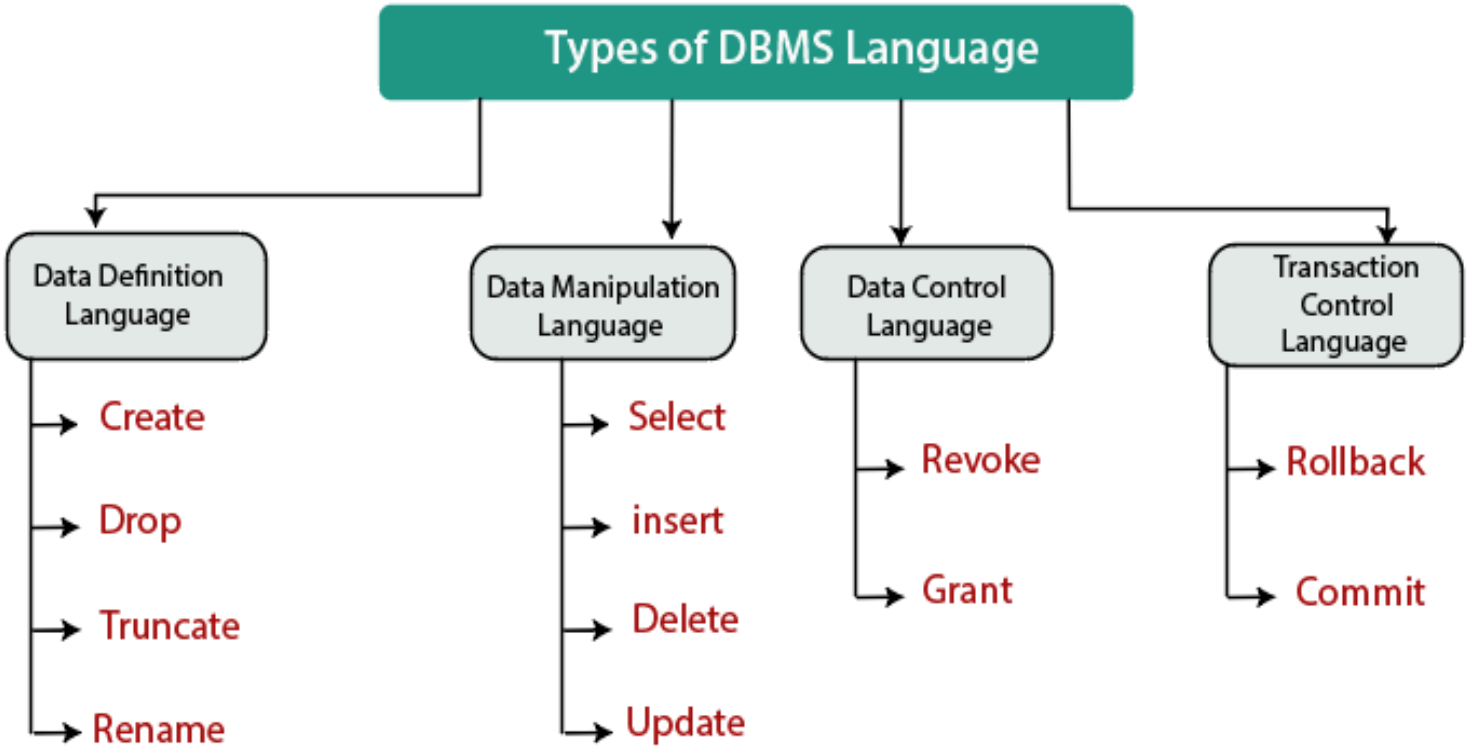
- 1950s and early 1960s:
  - Data processing using magnetic tapes for storage
    - Tapes provided only sequential access
  - Punched cards for input
- Late 1960s and 1970s:
  - Hard disks allowed direct access to data
  - Network and hierarchical data models in widespread use
  - Ted Codd defines the relational data model
    - Would win the ACM Turing Award for this work
    - IBM Research begins System R prototype
    - UC Berkeley begins Ingres prototype
  - High-performance (for the era) transaction processing

History (cont.)

- 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, ..

DBMS Language

- A DBMS has appropriate languages and interfaces to express database queries and updates.
- Database languages can be used to read, store and update the data in the database.



SQL

- Structure Query Language(SQL) is a database query language used for storing and managing data in Relational DBMS.
- SQL was the first commercial language introduced for E.F Codd's Relational model of database.
- Today almost all RDBMS(MySql, Oracle, Infomix, Sybase, MS Access) use SQL as the standard database query language.
- SQL is used to perform all types of data operations in RDBMS.
- It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables.
- To be able to compute complex functions SQL is usually embedded in some higher-level language
- Application programs generally access databases through one of
  - Language extensions to allow embedded SQL
  - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database

Data Definition Language (DDL)

- Specification notation for defining the database schema

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

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

Data Definition Language (DDL)

- DDL actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in database.
- Examples of DDL commands:
  - CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
  - DROP – is used to delete objects from the database.
  - ALTER-is used to alter the structure of the database.
  - TRUNCATE–is used to remove all records from a table, including all spaces allocated for the records are removed.
  - COMMENT –is used to add comments to the data dictionary.
  - RENAME –is used to rename an object existing in the database.

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
  - DML also known as query language
- Two classes of languages
  - **Pure** – used for proving properties about computational power and for optimization
    - Relational Algebra
    - Tuple relational calculus
    - Domain relational calculus
  - **Commercial** – used in commercial systems
    - SQL is the most widely used commercial language

Data Manipulation Language (DML)

- The SQL commands that deals with the manipulation of data present in database belong to DML and this includes most of the SQL statements.
- Examples of DML:
  - SELECT – is used to retrieve data from the database.
  - INSERT – is used to insert data into a table.
  - UPDATE – is used to update existing data within a table.
  - DELETE – is used to delete records from a database table.

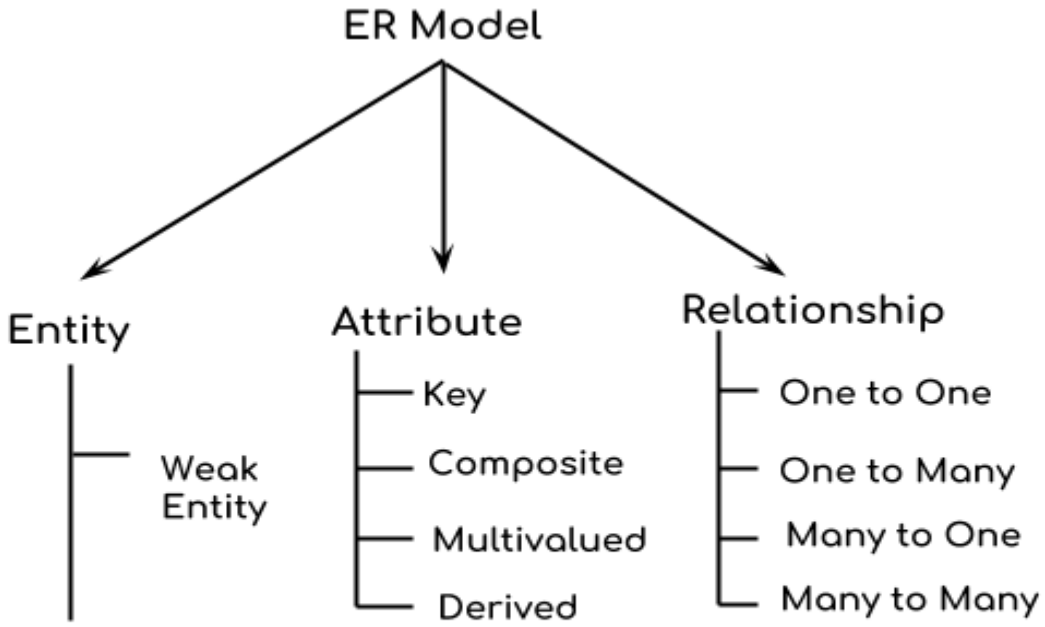
Data Control Language (DCL)

- DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
- Examples of DCL commands:
  - GRANT-gives user's access privileges to database.
  - REVOKE-withdraw user's access privileges given by using the GRANT command.

Transaction Control Language (TCL)

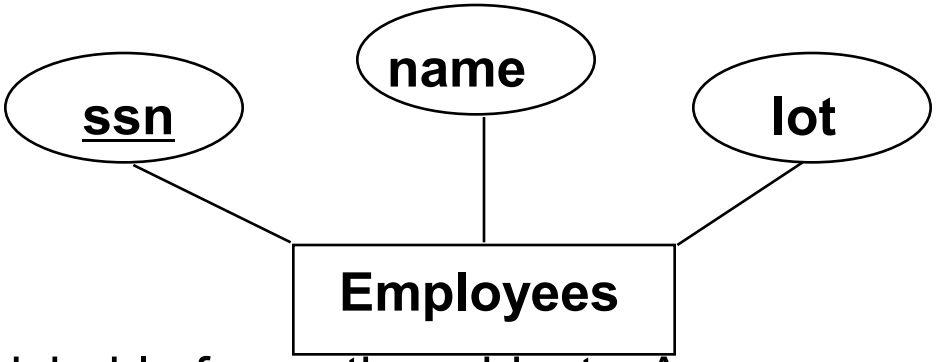
- TCL commands deals with the transaction within the database.
- Examples of TCL commands:
  - COMMIT– commits a Transaction.
  - ROLLBACK– rollbacks a transaction in case of any error occurs.
  - SAVEPOINT–sets a savepoint within a transaction.
  - SET TRANSACTION–specify characteristics for the transaction.

Components of ER Diagram



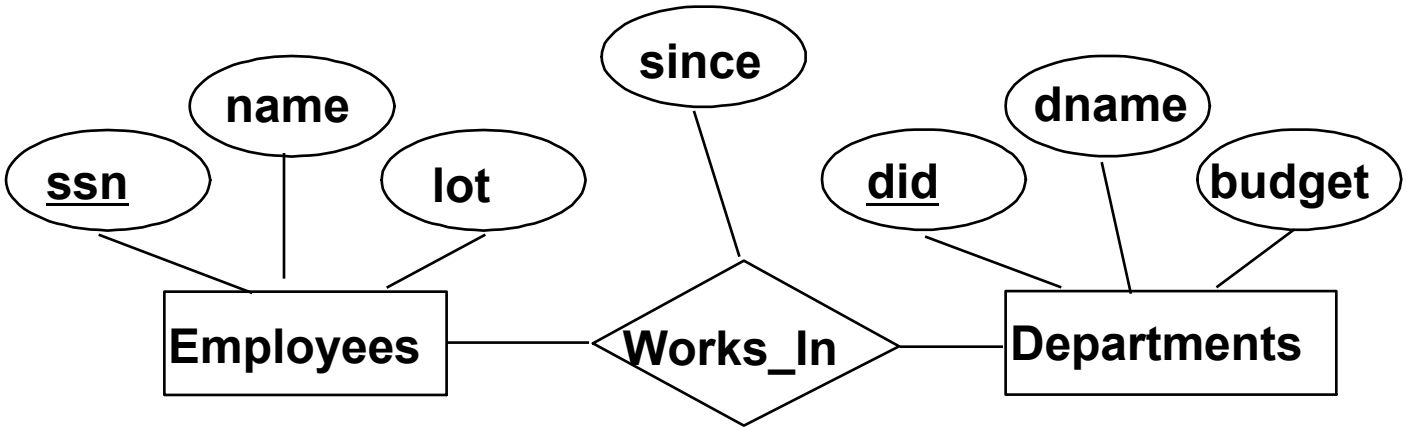
Components of ER Diagram

ER Model Basics



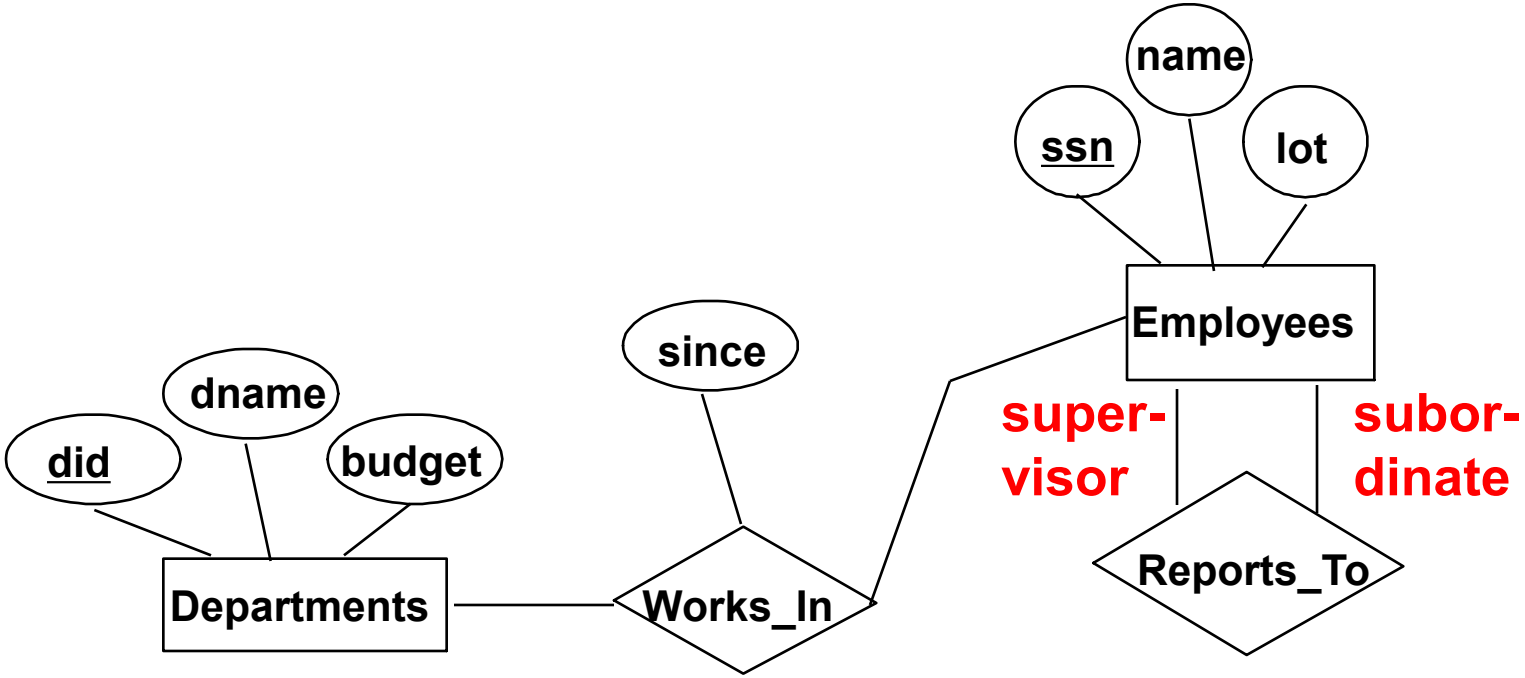
- *Entity:* Real-world object, distinguishable from other objects. An entity is described using a set of *attributes*.
  - Examples:
    - Person: PROFESSOR, STUDENT
    - Place: STORE, UNIVERSITY
    - Object: MACHINE, BUILDING
    - Event: SALE, REGISTRATION
    - Concept: ACCOUNT, COURSE
- *Entity Set*: A collection of similar entities. E.g., all employees.
  - All entities in an entity set have the same set of attributes. (Until we consider hierarchies, anyway!)
  - Each entity set has a *key (underlined)*.

ER Model Basics (Contd.)



- *Relationship*: Association among two or more entities. E.g., Attishoo works in Pharmacy department.
  - relationships can have their own attributes, these are descriptive attributes.
- *Relationship Set*: Collection of similar relationships.
  - An *n*-ary relationship set *R* relates *n* entity sets $E_1 \ldots E_n$; each relationship in *R* involves entities $e_1 \in E_1, \ldots, e_n \in E_n$

ER Model Basics (Cont.)



- Same entity set can participate in different relationship sets, or in different “roles” in the same set.

Attributes

- Each Entity has a set of Attributes
- ***Attribute*** describe the property or characteristic of an entity that is of interest to the organization.
  - Example:
    - STUDENT: Student_ID, Student_Name, Phone_Number, Course
- ***Domain*** range of values or the set of allowed values for each attribute is called the **domain** of the attribute

Attributes
(Naming Guidelines)

- ***An attribute name***:
  - Should be a *noun* and *capitalize the first letter of each word*. (Example: Student_ID.)
  - Should be *unique*.
  - Should follow a *standard format*. (Example: Student_GPA, not GPA_of_Student.)
- Similar attributes of different entity types should use similar but distinguished names.
  - Example: Faculty_Residence_City_Name and Student_Residence_City_Name

Attribute Types

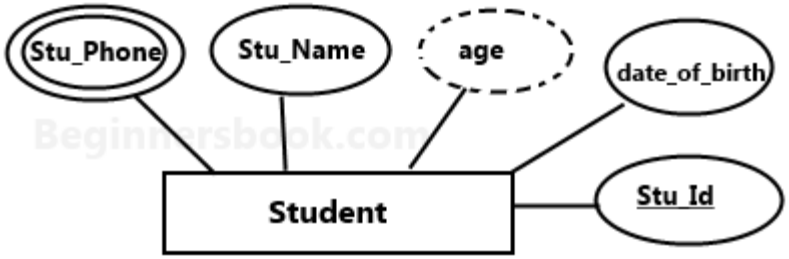
- Attribute values are (normally) required to be **atomic**; that is, indivisible
- The special value ***null*** is a member of every domain. Indicated that the value is "unknown"
- Simple attribute
- Composite attribute
- Derived attributes
- Single-valued attribute
- Multi-valued attribute

Simple/Composite attribute

- A simple attribute cannot be subdivided.
  - Examples: Age, Gender, and Marital status
- A composite attribute can be further subdivided to yield additional attributes.
  - Examples:
    - ADDRESS --→ Street, City, State, Zip
    - PHONE NUMBER --→ Area code, Exchange number

Derived attribute

- A derived attribute is one whose value is dynamic and derived from another attribute.
- is not physically stored within the database.
  - Example 1: Late Charge of 2%
    - MS Access: InvoiceAmt * 0.02
  - Example 2: AGE can be derived from the date of birth and the current date. It is changes over time.
    - MS Access: int(Date() – Emp_Dob)/365)



Single-valued attribute

- can have only a single (atomic) value.
  - Examples:
    - A person can have only one social security number.
    - A manufactured part can have only one serial number.
  - **A single-valued attribute is not necessarily a simple attribute.**
    - Part No: CA-08-02-189935
    - Location: CA, Factory#:08, shift#: 02, part#: 189935

Multi-valued attributes

- An attribute that can hold multiple values.
  - Examples:
    - A person may have several college degrees.
    - A household may have several phones with different numbers
    - A car color

Constraints

- Constraints enforce limits to the data or type of data that can be inserted/updated/deleted from a table.
- The whole purpose of constraints is to maintain the data integrity during an update/delete/insert into a table.
- Types of constraints
  - Mapping constraints
  - Key or Uniqueness constraints
  - Domain constraints

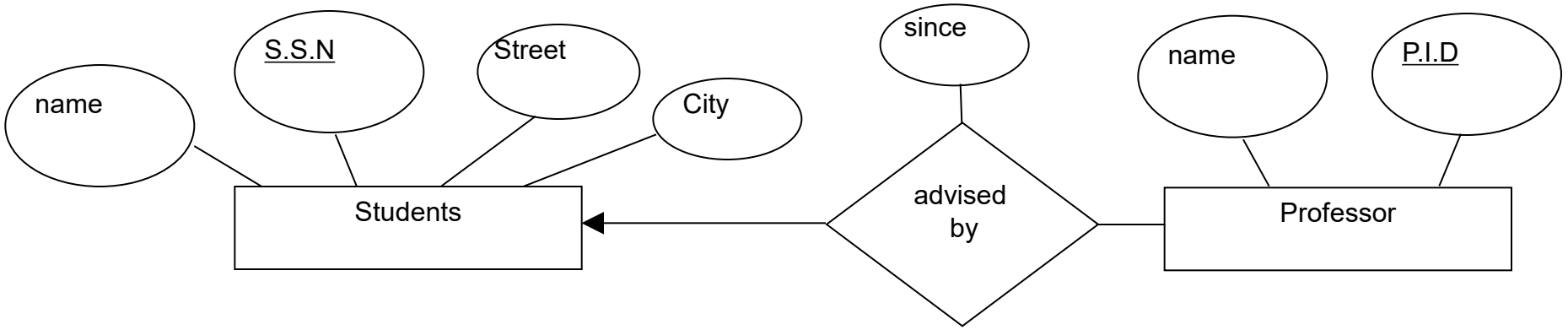
    data type(integer / character/date / time / string / etc.) + Constraints(NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK / DEFAULT)
  - Entity Integrity constraints
  - Referential Integrity constraints

Mapping Constraints

We can also use arrows to indicate **constraints**

Suppose the university has the following rule: A professor is allowed to advise at most one student. However two or more professors are allowed to advise the same student. (I.e Dr. Keogh and Dr. Lonardi both advise Isaac).

This is an example of a **many-to-one constraints**, that is *many* professors can advise a *one* single student. We can represent this with an arrow as shown below.



Mapping Constraints (Contd..)

•There are four possible **key constraints,** they express the number of entities to which another entity can be associated via a relationship. For binary relationship sets between entity sets A and B, the mapping cardinality must be one of:

1.One-to-one: When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship.

2.One-to-many: When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship.

Mapping Constraints (Contd..)

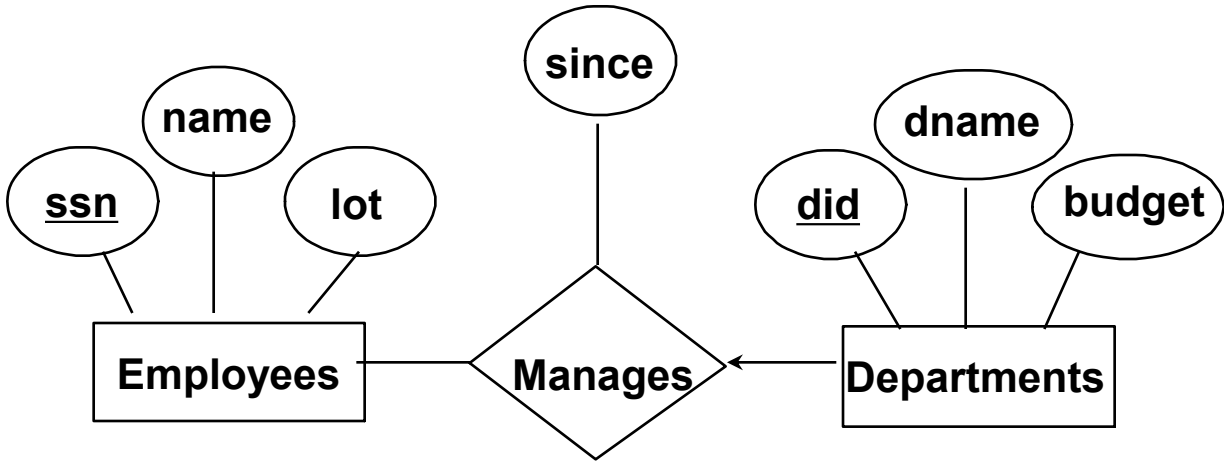
3.Many-to-one: When more than one instances of an entity is associated with a single instance of another entity then it is called many to one relationship.

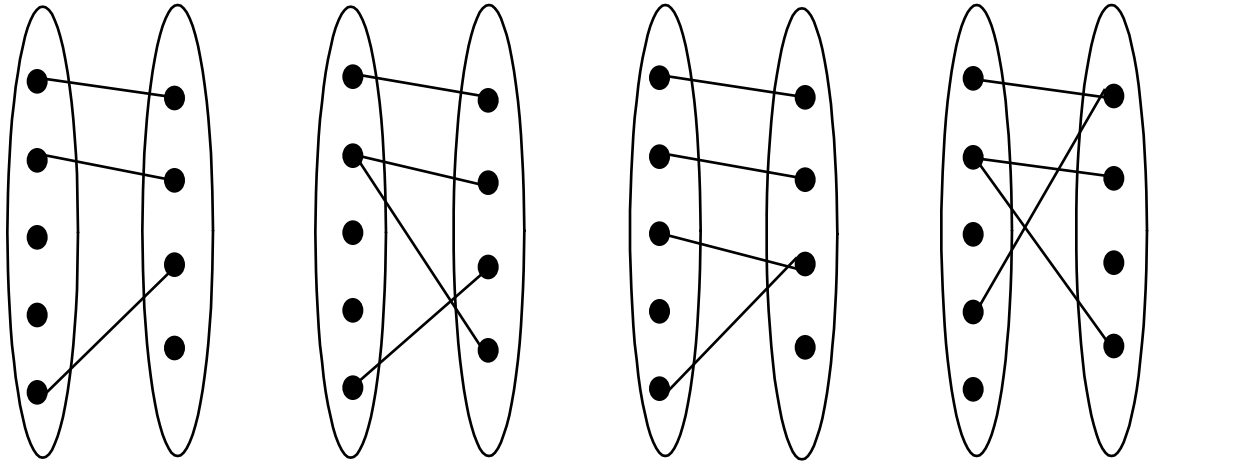
4.Many-to-many: When more than one instances of an entity is associated with more than one instances of another entity then it is called many to many relationship.

The appropriate **constraint** for a particular relationship set depends on the real world being modeled.

Mapping Constraints

- An employee can work in many departments; a dept can have many employees.
- In contrast, Each dept has at most one manager, according to the *constraint* on Manages.

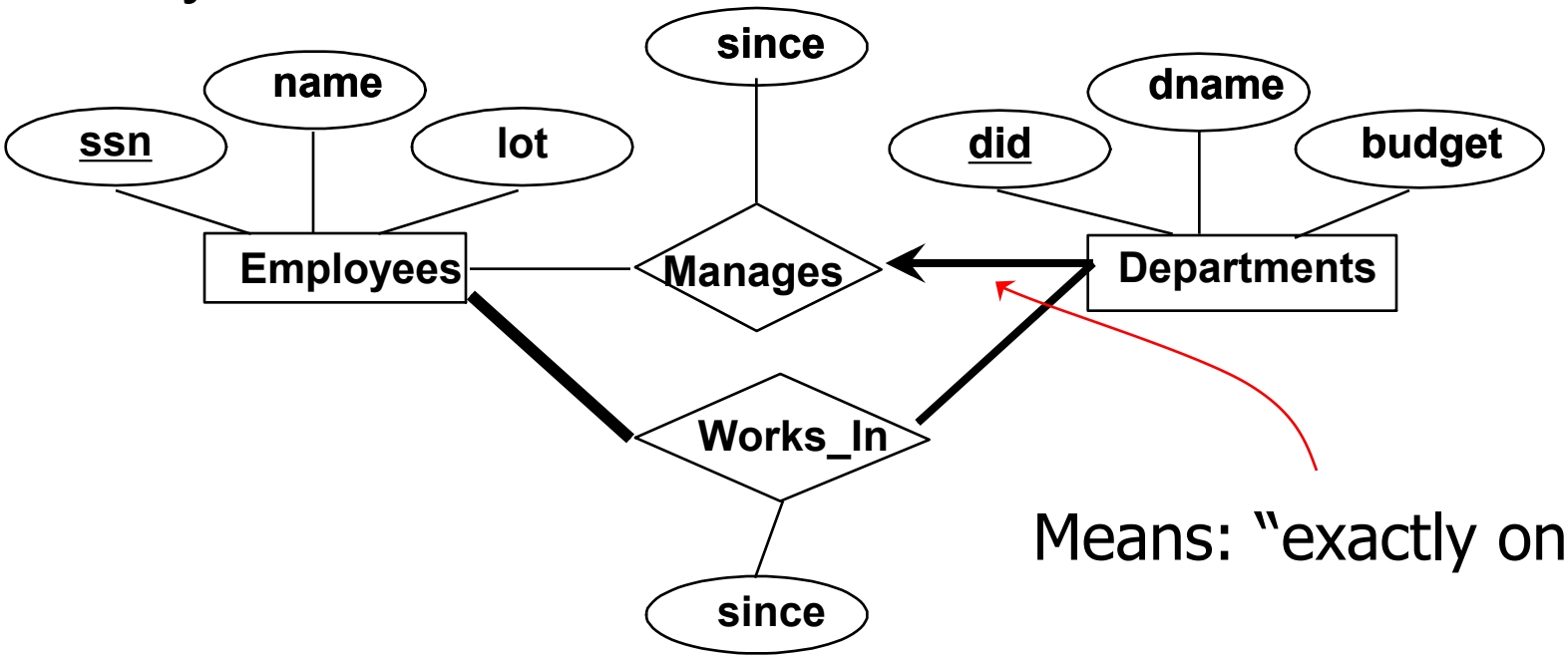




Translation to relational model?

Participation Constraints

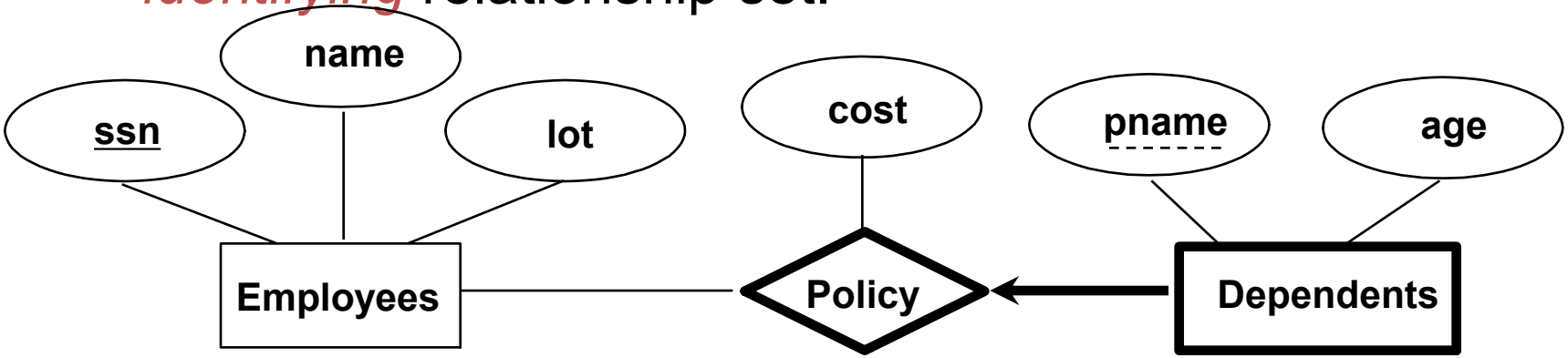
- Does every employee work in a department?
- If so, this is a *participation constraint*
  - the participation of Employees in Works_In is said to be *total* (vs. *partial*)
  - What if every department has an employee working in it?
- Basically means "at least one"



Weak Entities

A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.

- Always depends on the strong entity for its existence
- Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
- Weak entity set must have total participation in this *identifying* relationship set.



Weak entities have only a "partial key" (dashed underline)

Relationships

- **Relationships** are associations between one or more entity types.
- ***The degree of a relationship*** = is the number of entity types that participate in a relationship.
  - There are 3 common relationships:
    1. Unary (degree one)
    2. Binary (degree two)
    3. Ternary (degree three)

Relationships
(Naming Guidelines)

- A relationship name should:
  - Be a verb phrase, such as Is_assigned_to.
  - Avoid vague names, such as “Has”.

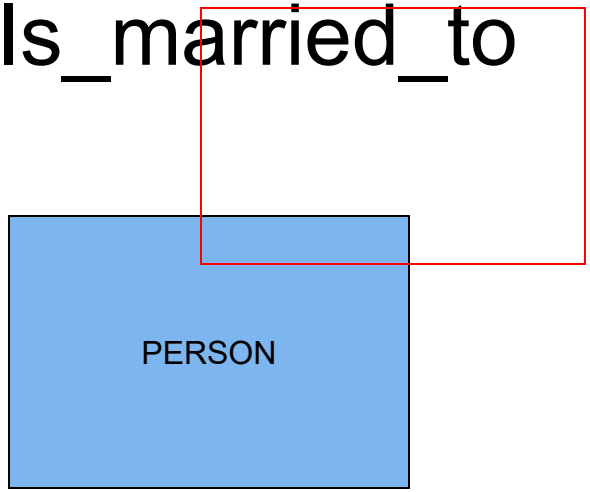
Relationships
(Naming Guidelines)

- A relationship definition should Explain:
  - What action is being *taken* and *why* it is important.
  - If there is any *optional* participation.
  - What any restrictions on participation in the relationship.
    - For example: An EMPLOYEE may only be able to participate in two PROJECTS.

Unary Relationship

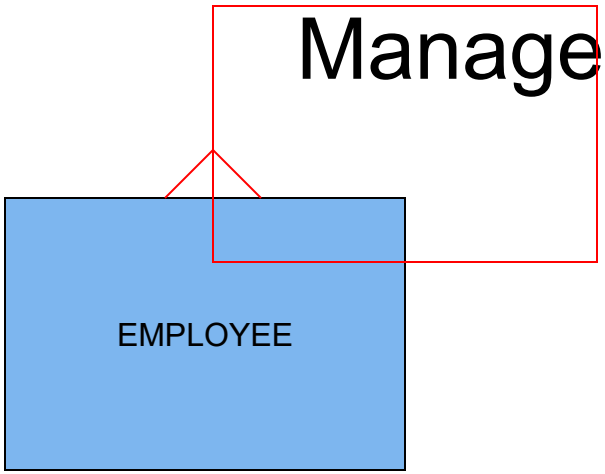
- Relationship between the instances of one entity type.

Is_married_to



One-to-one

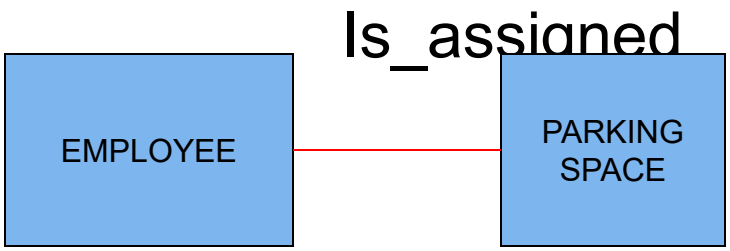
Manages



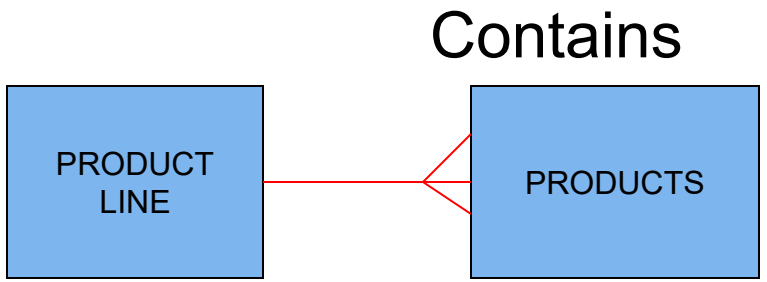
One-to-many

Binary Relationship

- Relationship between the instances of two entity type.



One-to-One

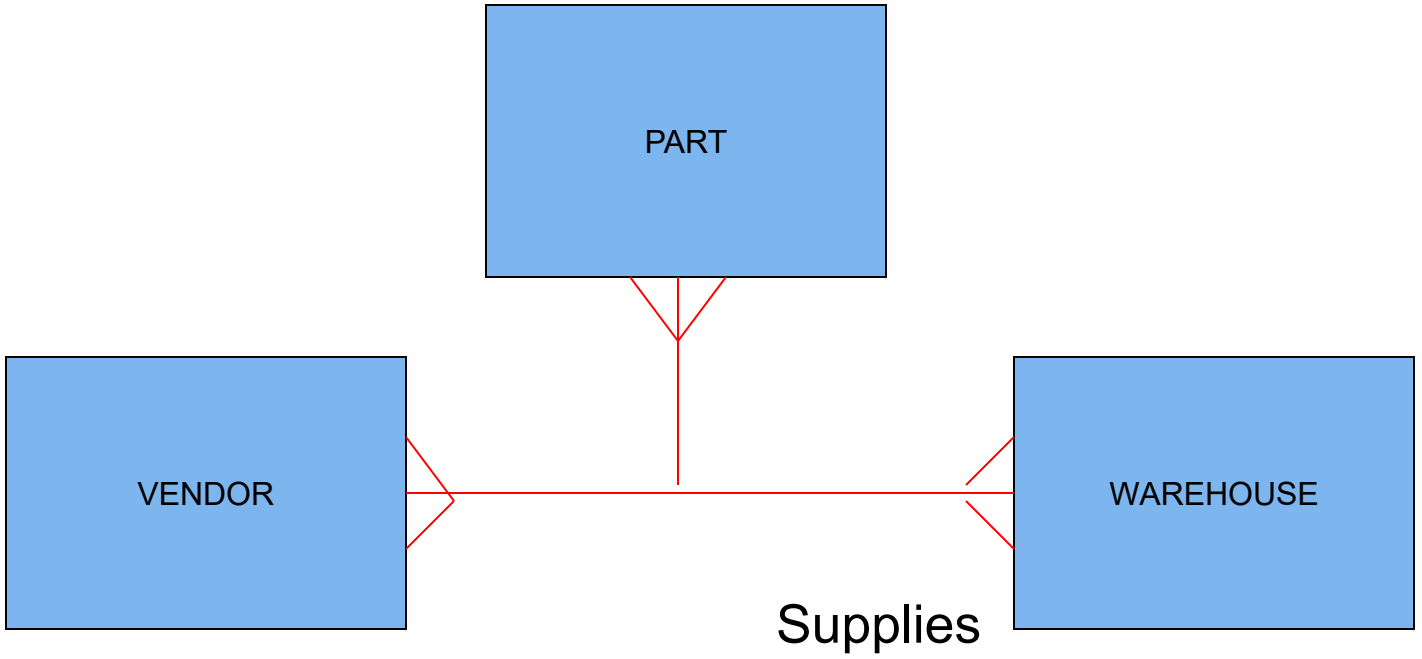


One-to-Many

Can also have many to many!

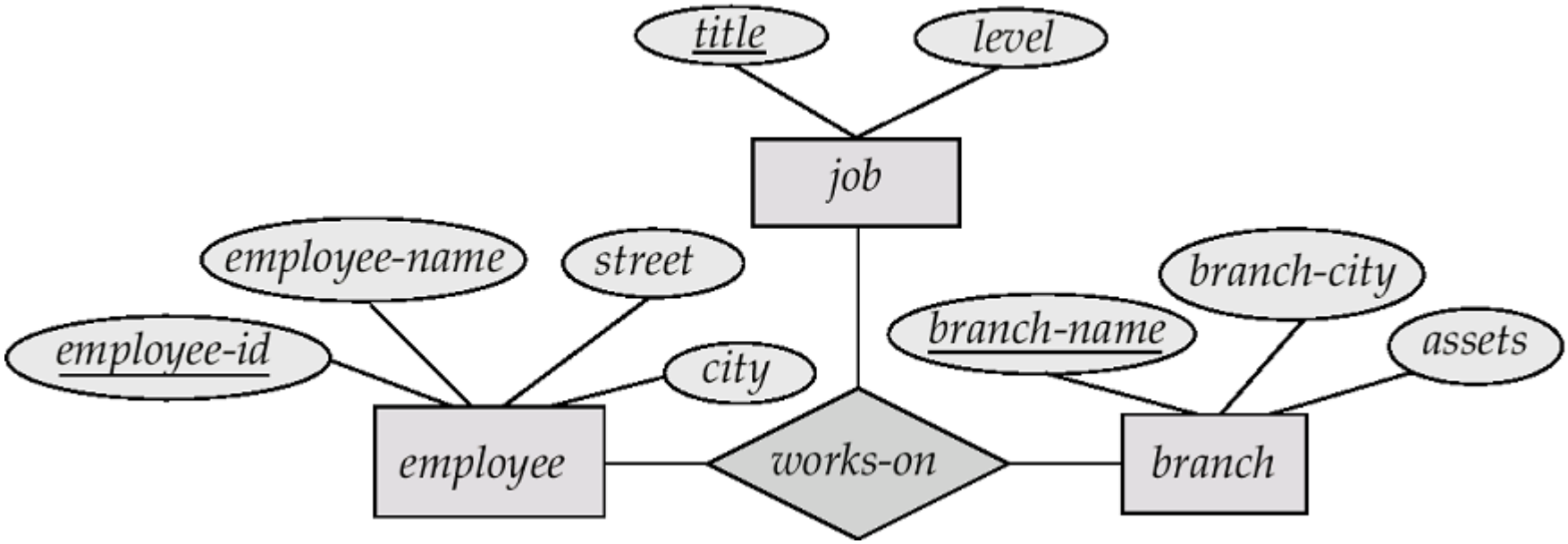
Ternary Relationship

- A simultaneous relationship among instances of three entity types.



Ternary Relationships

So far, we have only considered binary relationships, however it is possible to have **higher order** relationships, including **ternary** relationships. Consider the following example that describes the fact that employees at a bank work in one or more bank branches, and have one or more job descriptions.



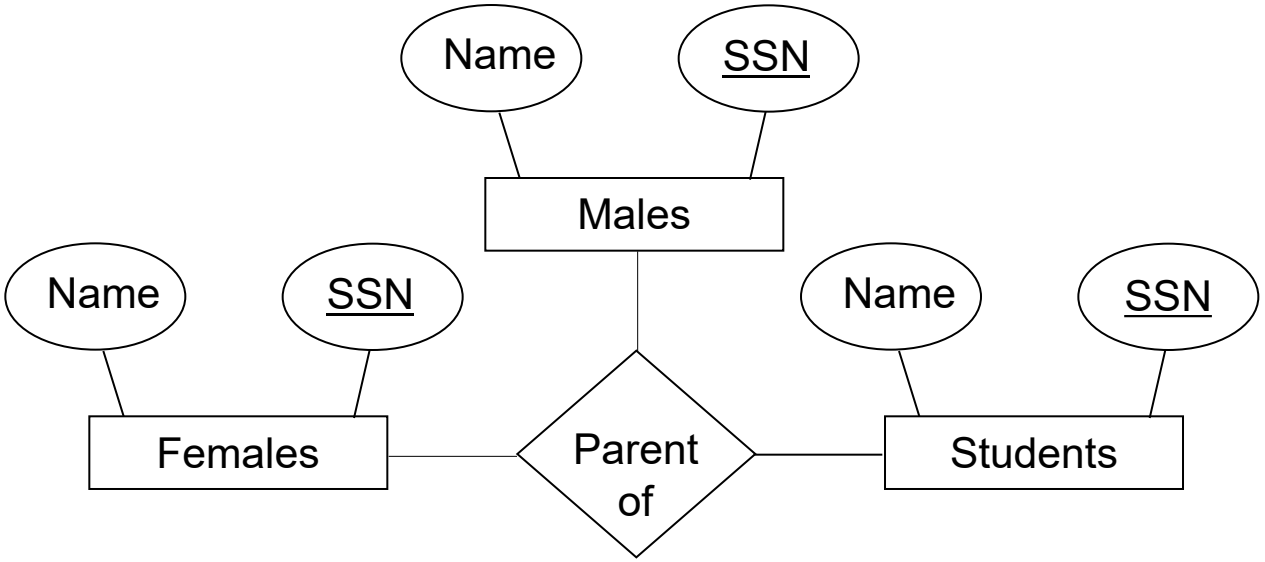
Why not remove the *job* entity by placing the *title* and *level* attributes as part of employee?

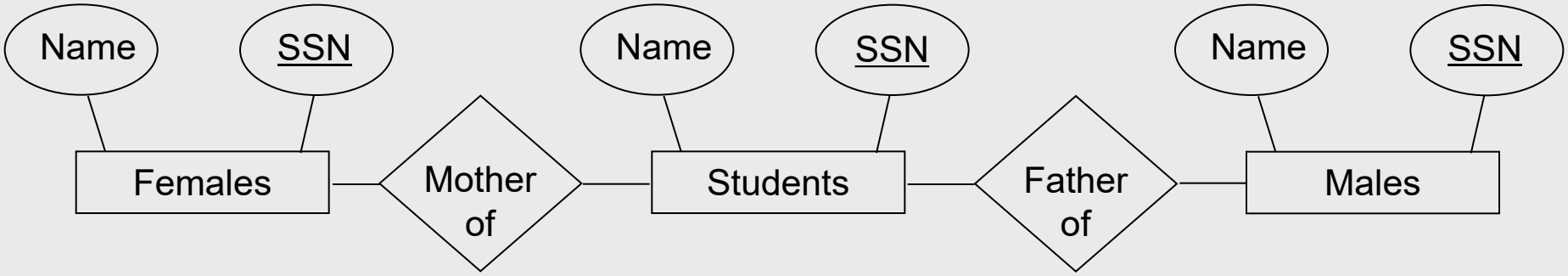
Ternary Relationships

Sometimes you have a choice of a single ternary relationship or two binary relationships…

In general, unless you really need a ternary relationship, use binary relationships.

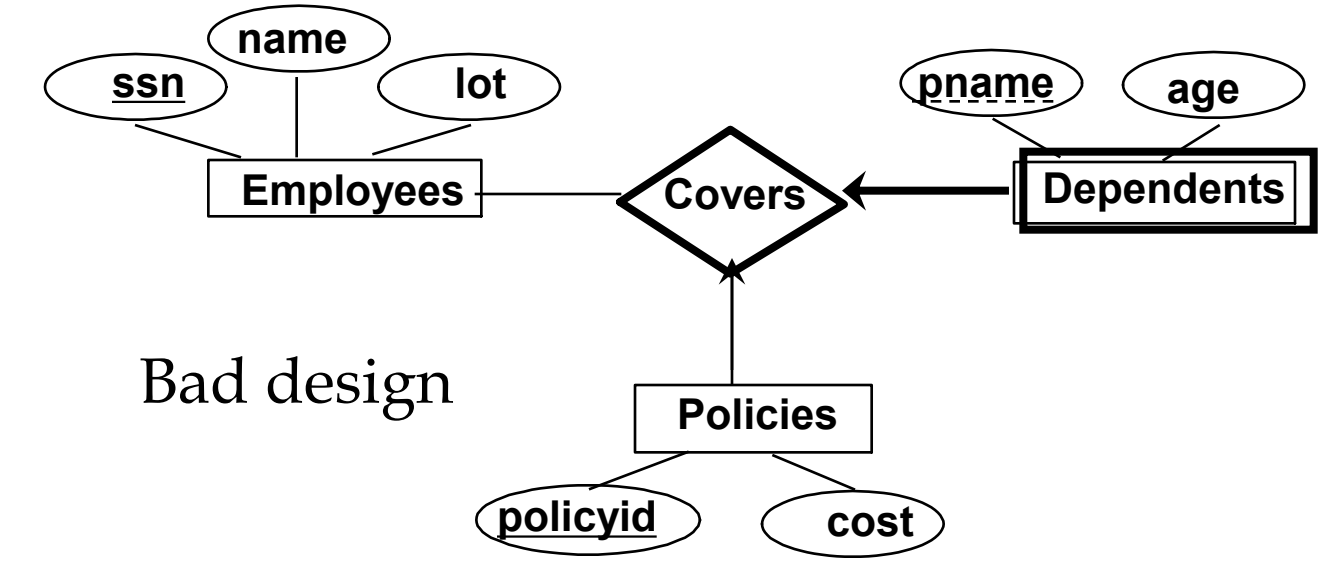
FACT: Every ternary (and higher order) relationship can be converted into a set of binary relationships.





Binary vs. Ternary Relationships

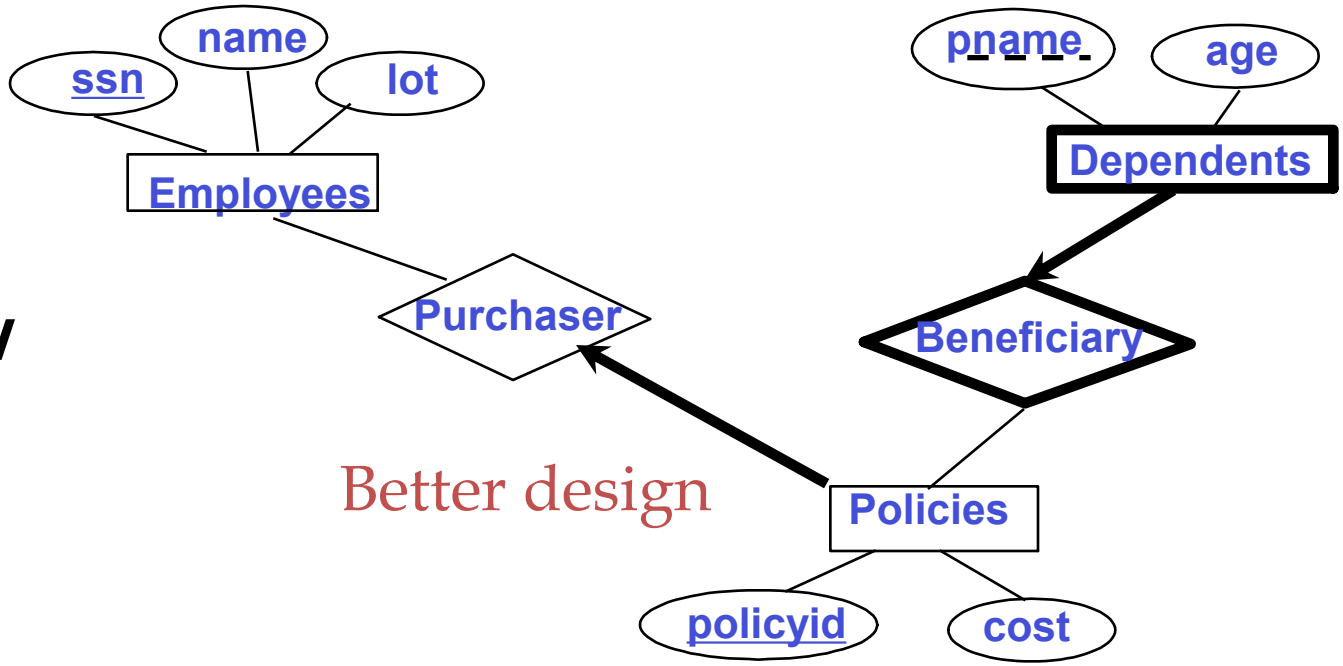
If each policy is owned by just 1 employee:



Bad design

Constraint on Policies would mean policy can only cover 1 dependent!

- **Think through *all* the constraints in the 2nd diagram!**



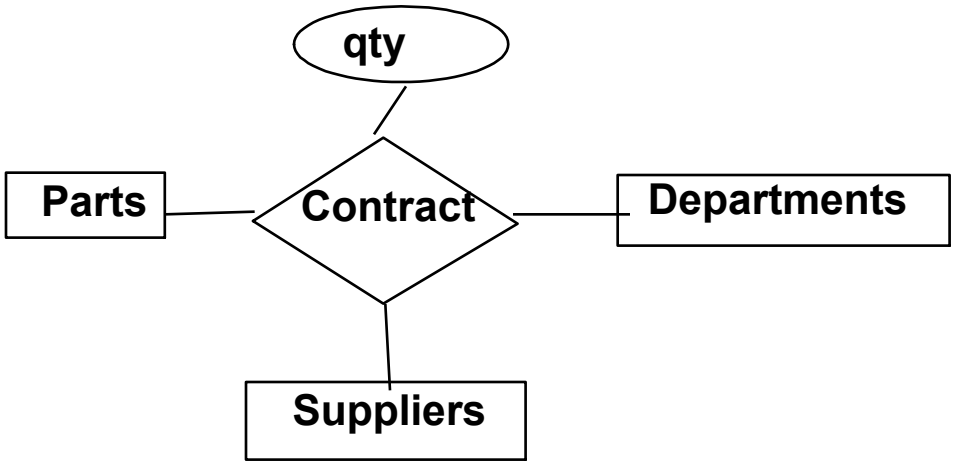
Better design

Binary vs. Ternary Relationships (Contd.)

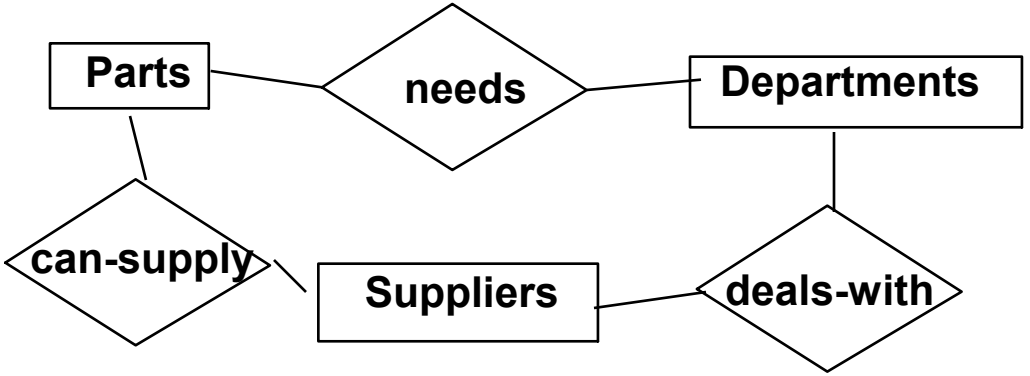
- Previous example illustrated a case when two binary relationships were better than one ternary relationship.

- An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*. No combination of binary relationships is an adequate substitute. (With no new entity sets!)

Binary vs. Ternary Relationships (Contd.)



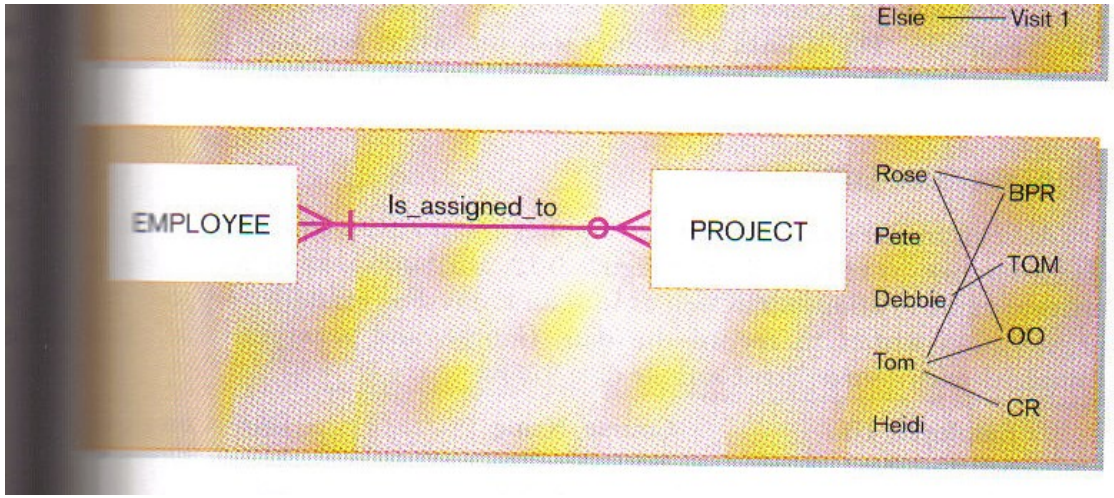
vs.



- S “can-supply” P, D “needs” P, and D “deals-with” S does not imply that D has agreed to buy P from S.
- How do we record *qty*?

Relationship Cardinalities

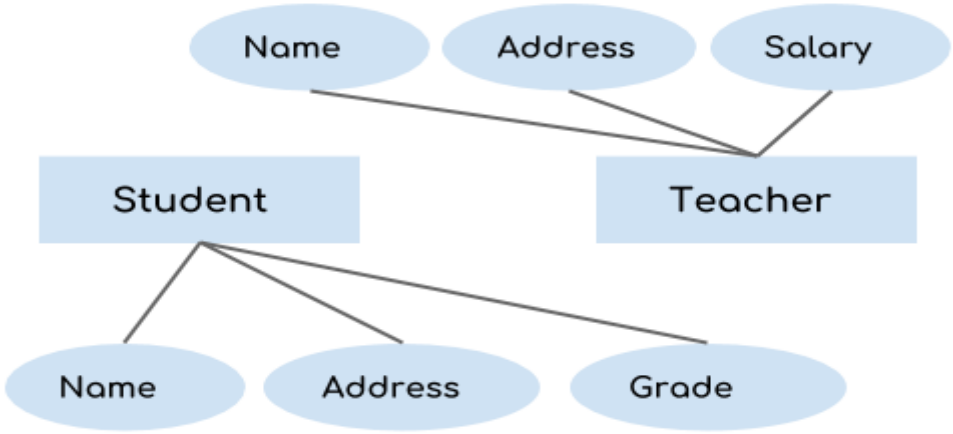
- ***Mapping Cardinalities*** =no. of entities in one entity set which can be associated with the no. of entities in other set via relationship set
- ***Mandatory Cardinalities*** = The entity must participate in another entity.
- ***Optional Cardinalities*** = The entity has a the option to participate in another entity.



(b) One optional, one m cardinality

DBMS Generalization

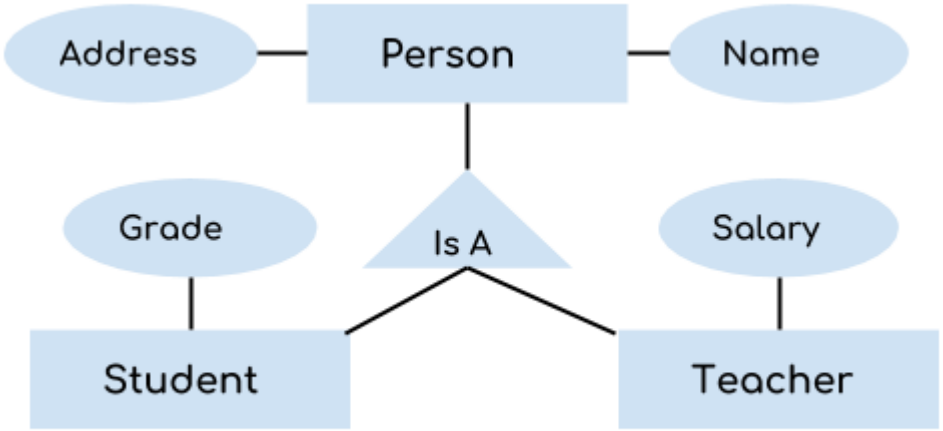
- Generalization is a process in which the common attributes of more than one entities form a new entity. This newly formed entity is called generalized entity.
- Generalization Example
- Lets say we have two entities Student and Teacher.
- Attributes of Entity Student are: Name, Address & Grade
- Attributes of Entity Teacher are: Name, Address & Salary



Before Generalization

ER diagram after generalization:

- We have created a new generalized entity Person and this entity has the common attributes of both the entities.
- After the generalization process the entities Student and Teacher only has the specialized attributes Grade and Salary respectively and their common attributes (Name & Address) are now associated with a new entity Person which is in the relationship with both the entities (Student & Teacher).

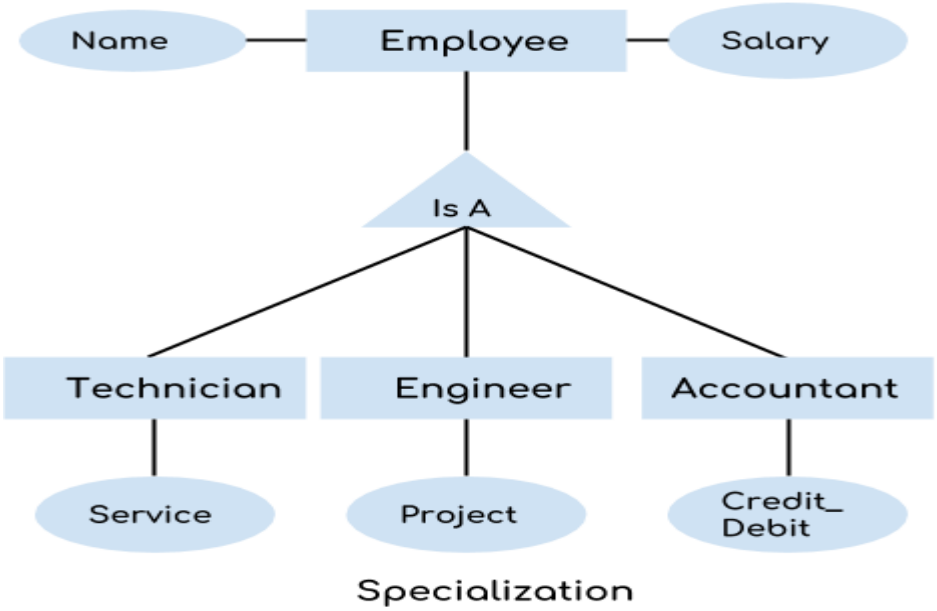


Generalization

- Generalization uses bottom-up approach where two or more lower level entities combine together to form a higher level new entity.
- The new generalized entity can further combine together with lower level entity to create a further higher level generalized entity.

DBMS Specialization

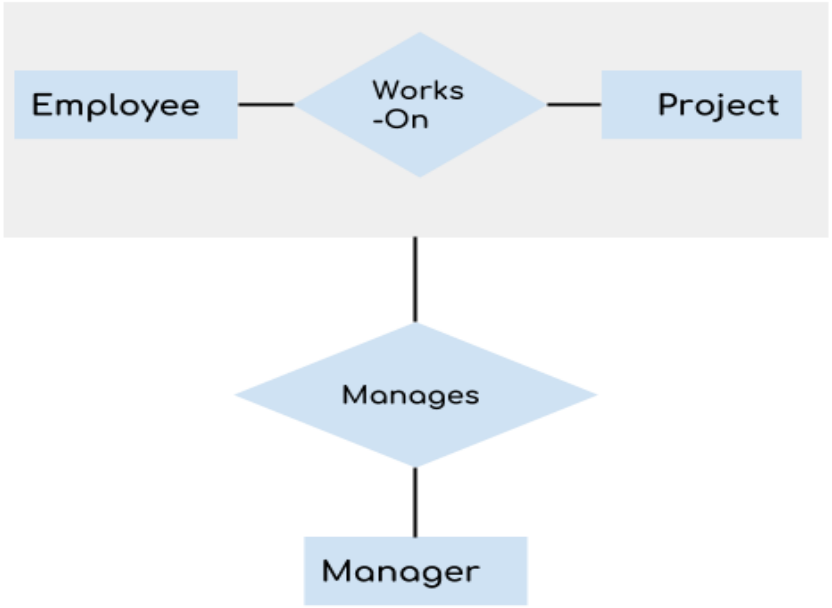
- Specialization is a process in which an entity is divided into sub-entities. You can think of it as a reverse process of generalization, in generalization two entities combine together to form a new higher level entity. Specialization is a top-down process.
- The idea behind Specialization is to find the subsets of entities that have few distinguish attributes.
- For example – Consider an entity employee which can be further classified as sub-entities Technician, Engineer & Accountant because these sub entities have some distinguish attributes.



Specialization

- Here, higher level entity “Employee” which we have divided in sub entities “Technician”, “Engineer” & “Accountant”.
- All of these are just an employee of a company, however their role is completely different and they have few different attributes.
- For example, Technician handles service requests, Engineer works on a project and Accountant handles the credit & debit details.
- All of these three employee types have few attributes common such as name & salary which we had left associated with the parent entity “Employee” as shown in the above diagram.

DBMS Aggregration



- Process in which a single entity alone is not able to make sense in a relationship so the relationship of two entities acts as one entity.
- A manager not only manages the employee working under them but he has to manage the project as well.
- If entity “Manager” makes a “manages” relationship with either “Employee” or “Project” entity alone then it will not make any sense because he has to manage both.
- In these cases the relationship of two entities acts as one entity.

Keys

- Key plays an important role in relational database
- Attribute or collection of attributes uniquely identifies enitity (rows) from entity set (table). It also establishes relationship among tables.
- Types of keys in DBMS
- Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.
  - Two rows can't have the same primary key value
  - It must for every row to have a primary key value.
  - The primary key field cannot be null.
  - The value in a primary key column can never be modified or updated if any foreign key refers to that primary key.
- In the EMPLOYEE (Employee_ID, Employee_Name, Employee_Address, SSN, Passport_Number, License_Number) table, Employee_ID is best suited for the primary key. Rest of the attributes like SSN, Passport_Number, and License_Number, etc. are considered as a candidate key.

Keys

- Super Key – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table.
- Consider the following Student schema-
  - Student (roll , name , sex , age , address , class , section)
- Given below are the examples of super keys since each set can uniquely identify each student in the Student table-
  - (roll , name , sex , age , address , class , section)
  - (class , section , roll)
  - (class , section , roll , sex)
  - (name , address)
- All the attributes in a super key are definitely sufficient to identify each tuple uniquely in the given relation but all of them may not be necessary.

Keys

- Candidate Key – A minimal super key with no redundant attribute is known as candidate key
- Consider the following Student schema-
  - Student (roll , name , sex , age , address , class , section)
- Given below are the examples of candidate keys since each set consists of minimal attributes required to identify each student uniquely in the Student table-
  - (class , section , roll)
  - (name , address)
- All the attributes in a candidate key are sufficient as well as necessary to identify each tuple uniquely.
- Removing any attribute from the candidate key fails in identifying each tuple uniquely.
- The value of candidate key must always be unique.
- The value of candidate key can never be NULL.
- It is possible to have multiple candidate keys in a relation.
- Those attributes which appears in some candidate key are called as prime attributes.

Keys

- Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

Keys

- Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.
  - Sales table has four columns- cust_Id, order_Id, product_code & product_count.
  - cust_Id alone cannot become a key as a same customer can place multiple orders, thus the same customer can have multiple entires.
  - order_Id alone cannot be a primary key as a same order can contain the order of multiple products, thus same order_Id can be present multiple times.
  - product_code cannot be a primary key as more than one customers can place order for the same product.
  - product_count alone cannot be a primary key because two orders can be placed for the same product count.
  - Based on this, it is safe to assume that the key should be having more than one attributes:
  - Composite Key in above table: {cust_id, product_code}
    - cust_Id order_Id product_code product_count
    - -------- -------- ------------ -------------
    - C01 O001 P007 23
    - C02 O123 P007 19
    - C02 O123 P230 82
    - C01 O001 P890 42

Keys

- Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. Set of fields in one relation that is used to `refer' to a tuple in another relation. Like a `logical pointer'. They act as a cross-reference between tables.
  - An attribute 'X' is called as a foreign key to some other attribute 'Y' when its values are dependent on the values of attribute 'Y'.
  - The attribute 'X' can assume only those values which are assumed by the attribute 'Y'.
  - Here, the relation in which attribute 'Y' is present is called as the referenced relation (master table or primary table).
  - The relation in which attribute 'X' is present is called as the referencing relation (foreign table).
  - The attribute 'Y' might be present in the same table or in some other table.
  - Foreign key references the primary key of the table.
  - Foreign key can take only those values which are present in the primary key of the referenced relation.
  - Foreign key may have a name other than that of a primary key.
  - Foreign key can take the NULL value.
  - There is no restriction on a foreign key to be unique.
  - In fact, foreign key is not unique most of the time..

- CREATE TABLE Customer

(CustID INTEGER PRIMARY KEY,

CustName CHAR(35))

  – CREATE TABLE Orders

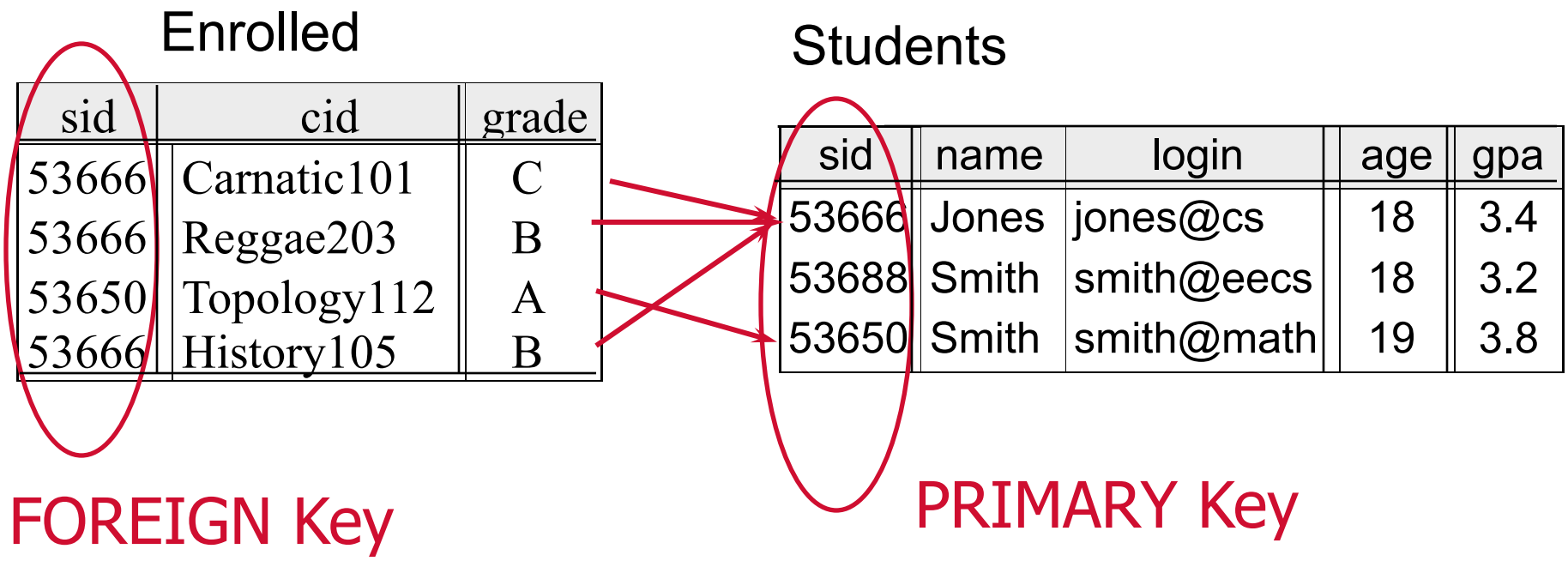
  (OrderID INTEGER PRIMARY KEY,

  CustID INTEGER REFERENCES Customer(CustID),

  OrderDate DATETIME)

Keys

- Keys are a way to associate tuples in different relations
- Keys are one form of integrity constraint (IC)



Primary and Candidate Keys in SQL

- Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.
- Keys must be used carefully!
- "For a given student and course, there is a single grade."

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))
```

VS.

```
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade))
```

"Students can take only one course, and no two students in a course receive the same grade."

Foreign Keys in SQL

- **E.g. Only students listed in the Students relation should be allowed to enroll for courses.**
  - *sid* is a foreign key referring to Students:

```
CREATE TABLE Enrolled
(sid CHAR(20),cid CHAR(20),grade CHAR(2),
 PRIMARY KEY (sid,cid),
 FOREIGN KEY (sid) REFERENCES Students )
```

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B
~~11111~~	~~English102~~	~~A~~

Students

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

. For example, if you have an Orders table with a foreign key referencing a Customers table, you cannot have an order that references a customer ID that does not exist in the Customers table2.

Referential integrity in a Database Management System (DBMS) ensures the accuracy and consistency of data within relationships between tables

Referential Integrity

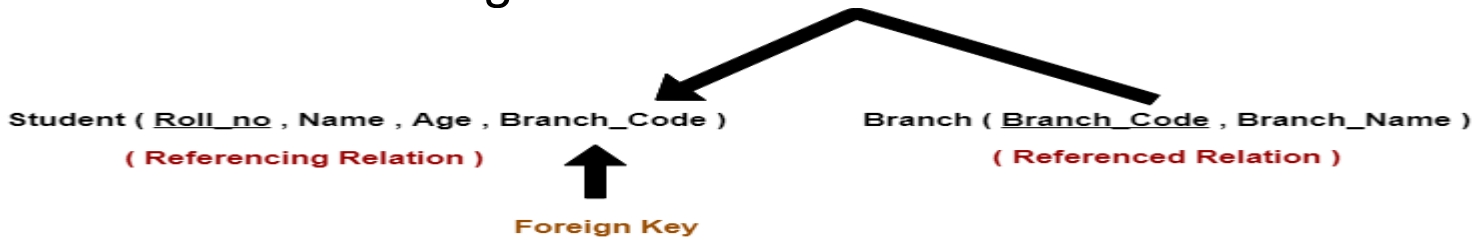
it is a crucial aspect of maintaining data integrity.

Referential integrity is maintained by using foreign keys. A foreign key in one table points to a primary key in another table.

- Referential Integrity constraint is enforced when a foreign key references the primary key of a relation.
- If all foreign key constraints are enforced, *referential integrity* is achieved.
- It specifies that all the values taken by the foreign key must either be available in the relation of the primary key or be null.
- 3 possible causes of violation of referential integrity constraint-
  - Cause-01: Insertion in a referencing relation
  - Cause-02: Deletion from a referenced relation
  - Cause-03: Updation in a referenced relation

Referential Integrity

- It is allowed to insert only those values in the referencing attribute which are already present in the value of the referenced attribute.
- Inserting a value in the referencing attribute which is not present in the value of the referenced attribute violates the referential integrity constraint.
- Example-
- Consider the following two schemas-



- Here, relation “Student” references the relation “Branch”
- In relation “Student”, we can not insert any student having branch code ME (Mechanical Engineering).
- This is because branch code ME is not present in the relation “Branch”.

To maintain referential integrity, DBMSs often support cascading actions. For instance, if a record in the parent table is deleted, the corresponding records in the child table can also be automatically deleted (cascading delete), or the foreign key values can be set to null (cascading nullify)

Referential Integrity

- It is not allowed to delete a row from the referenced relation if the referencing attribute uses the value of the referenced attribute of that row.
- Such a deletion violates the referential integrity constraint.
- Example-
- Consider the above two relations,
- We can not delete a tuple from the relation “Branch” having branch code ‘CS’.
- This is because the referencing attribute “Branch_Code” of the referencing relation “Student” references the value ‘CS’.
- However, we can safely delete a tuple from the relation “Branch” having branch code ‘CE’.
- This is because the referencing attribute “Branch_Code” of the referencing relation “Student” does not uses this value.

Referential Integrity

- Method-01:
  - This method involves simultaneously deleting those tuples from the referencing relation where the referencing attribute uses the value of referenced attribute being deleted.
  - This method of handling the violation is called as On Delete Cascade.
- Method-02:
  - This method involves aborting or deleting the request for a deletion from the referenced relation if the value is used by the referencing relation.
- Method-03:
  - This method involves setting the value being deleted from the referenced relation to NULL or some other value in the referencing relation if the referencing attribute uses that value.

Referential Integrity

- It is not allowed to update a row of the referenced relation if the referencing attribute uses the value of the referenced attribute of that row.
- Such an updation violates the referential integrity constraint.
- Example-
- Consider the above relation.
- We can not update a tuple in the relation “Branch” having branch code ‘CS’ to the branch code ‘CSE’.
- This is because referencing attribute “Branch_Code” of the referencing relation “Student” references the value ‘CS’.
- Method-01:
  - This method involves simultaneously updating those tuples of the referencing relation where the referencing attribute uses the referenced attribute value being updated.
  - This method of handling the violation is called as On Update Cascade.
- Method-02:
  - This method involves aborting or deleting the request for an updation of the referenced relation if the value is used by the referencing relation.
- Method-03:
  - This method involves setting the value being updated in the referenced relation to NULL or some other value in the referencing relation if the referencing attribute uses that value.

Enforcing Referential Integrity

student id

table ka naam hai

- Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.

rtuple=row

- What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)
- What should be done if a Students tuple is deleted?
  - Also delete all Enrolled tuples that refer to it?
  - Disallow deletion of a Students tuple that is referred to?
  - Set sid in Enrolled tuples that refer to it to a *default sid*?
  - (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting *`unknown'* or *`inapplicable'*.)
- Similar issues arise if primary key of Students tuple is updated.

Integrity Constraints (ICs)

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
  - ICs are specified when schema is defined.
  - ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
  - DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.
  - Avoids data entry errors, too!
- Domain Constraint = data type(integer / character/date / time / string / etc.) + Constraints(NOT NULL / UNIQUE / PRIMARY KEY / FOREIGN KEY / CHECK / DEFAULT)

- Example 1:
  - create domain id_value int
  - constraint id_test
  - check(value > 100);
  - create table student_info (
  - stu_id id_value PRIMARY KEY,
  - stu_name varchar(30),
  - stu_age int
  -);
- Example 2:
  - create domain account_type char(12)
  - constraint acc_type_test
  - check(value in ("Checking", "Saving"));
  - create table bank_account (
  - account_nbr int PRIMARY KEY,
  - account_holder_name varchar(30),
  - account_type account_type
  -);

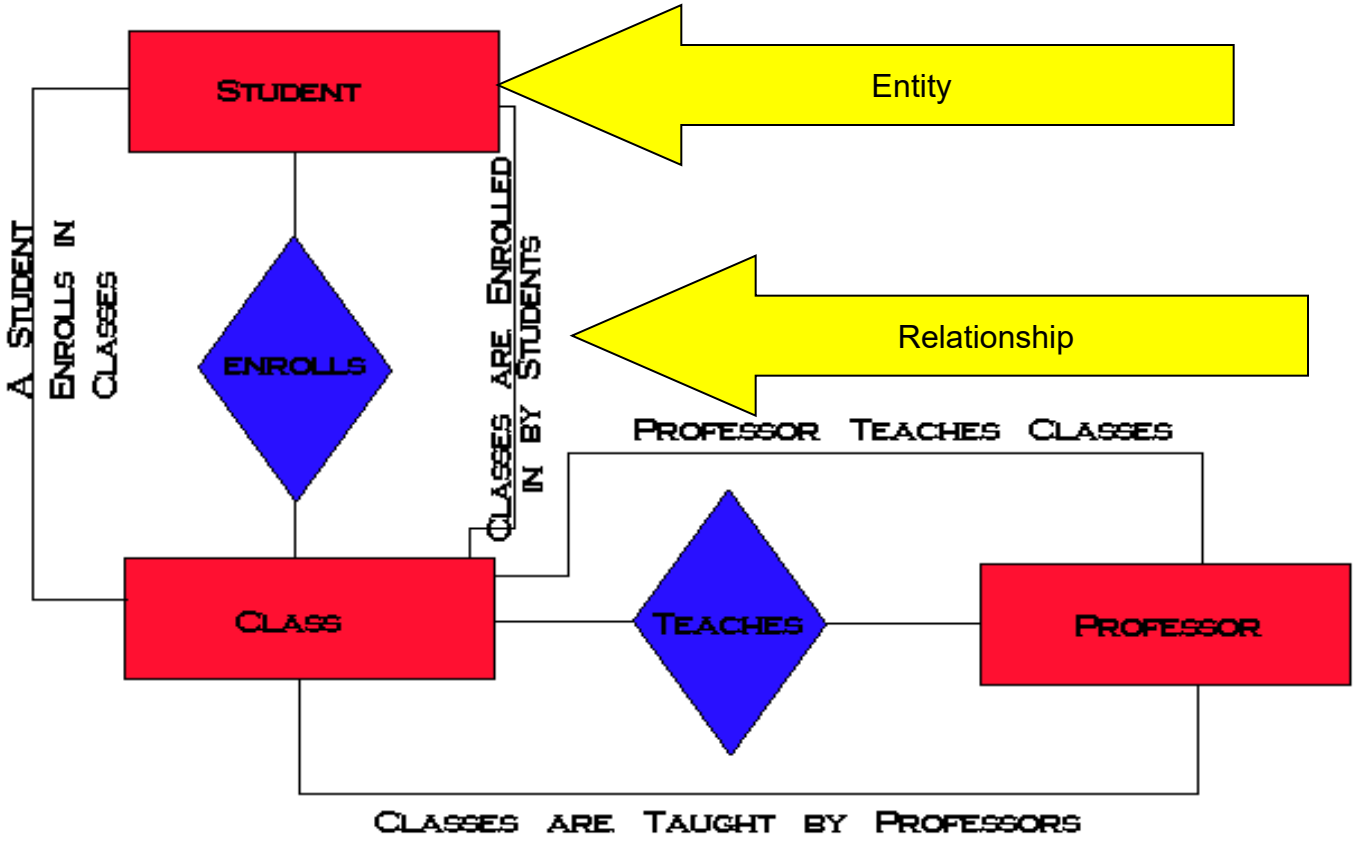
Starting an ERD

1. Define the Entities.
2. Define the Relationships.
3. Add attributes to the relationships.
4. Add cardinality to the relationships.
5. Don’t forget to use proper naming conventions and symbol representation.

Guidelines for Drawing ERDs

- Lay out the diagram with minimal line crossing.
- Place subject entity types on the top of the diagram.
- Place plural entity types below a single entity type in a one-to-many relationship.
- Place entity types participating in one-to-one and many-to-many relationships alongside each other.
- Group closely related entity types when possible. Try to keep the length of relationship lines as short as possible. Also try to minimize the number of changes of direction in a single line.
- Show the most relevant relationship name. One name must always be shown.

University Entity-Relationship Diagram



Exercise

- Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.
- A university registrar's office maintains data about the following entities:
  - (a)courses, including number, title, credits, syllabus, and prerequisites;
  - (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom;
  - (c) students, including student-id, name, and program; and
  - (d) instructors, including identification number, name, department, and title. Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.
  - Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.
- Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

Codd's 12 Rules

- E.F Codd invented the Relational model for Database management. Based on relational model, the Relational database was created.
- Codd proposed 13 rules popularly known as Codd's 12 rules to test DBMS's concept against his relational model.
- Codd's rule actualy define what quality a DBMS requires in order to become a Relational Database Management System(RDBMS).
- Till now, there is hardly any commercial product that follows all the 13 Codd's rules. Even Oracle follows only eight and half(8.5) out of 13. The Codd's 12 rules are as follows.

- Rule zero (Foundation / Base Rule)
  - This rule states that for a system to qualify as an RDBMS, it must be able to manage database entirely through the relational capabilities.

- Rule 1: Information rule
  - All information(including metadata) is to be represented as stored data in cells of tables. The rows and columns have to be strictly unordered.

- Rule 2: Guaranted Access
  - Each unique piece of data(atomic value) should be accesible by : Table Name + Primary Key(Row) + Attribute(column).
  - NOTE: Ability to directly access via POINTER is a violation of this rule.
- Rule 3: Systematic treatment of NULL values
  - Null has several meanings, it can mean missing data, not applicable or no value. It should be handled consistently. Also, Primary key must not be null, ever. Expression on NULL must give null.
- Rule 4: Active Online Catalog
  - Database dictionary(catalog) is the structure description of the complete Database and it must be stored online. The Catalog must be governed by same rules as rest of the database. The same query language should be used on catalog as used to query database.

- Rule 5: Powerful and Well-Structured Language
  - One well structured language must be there to provide all manners of access to the data stored in the database. Example: SQL, etc. If the database allows access to the data without the use of this language, then that is a violation.

- Rule 6: View Updation Rule
  - All the view that are theoretically updatable should be updatable by the system as well.

jo v view thoretically update ho jate hai that should updated by the system as well

- Rule 7: Relational Level Operation
  - There must be Insert, Delete, Update operations at each level of relations. Set operation like Union, Intersection and minus should also be supported.

- Rule 8: Physical Data Independence
  - The physical storage of data should not matter to the system. If say, some file supporting table is renamed or moved from one disk to another, it should not effect the application.

- Rule 9: Logical Data Independence
  - If there is change in the logical structure(table structures) of the database the user view of data should not change. Say, if a table is split into two tables, a new view should give result as the join of the two tables. This rule is most difficult to satisfy.
- Rule 10: Integrity Independence
  - The database should be able to enforce its own integrity rather than using other programs. Key and Check constraints, trigger etc, should be stored in Data Dictionary. This also make RDBMS independent of front-end.
- Rule 11: Distribution Independence
  - A database should work properly regardless of its distribution across a network. Even if a database is geographically distributed, with data stored in pieces, the end user should get an impression that it is stored at the same place. This lays the foundation of distributed database.
- Rule 12: Nonsubversion Rule
  - If low level access is allowed to a system it should not be able to subvert or bypass integrity rules to change the data. This can be achieved by some sort of locking or encryption.

Normalization

- In 1972 Codd first proposed the concept of normalization
- Process of organizing the data in the database.
- Process of removing or minimizing redundancy from a relation or set of relations.
- Redundancy in relation may cause insertion, deletion and updation anomalies. So, it helps to minimize the redundancy in relations.
- Process of designing a consistent database by reducing redundancy and ensuring data integrity through lossless decomposition.
- Process of analyzing the given relation schema based on their FD & PK to achieve the desirable properties of
  - minimizing redundancy, insertion,deletion & updation anomalies.
- Normalization is used for mainly,
  - Ensures data integrity
    - Entity Constraints
    - Domain Constraints
    - Referential integrity Constraints
  - Prevent redundancy in data: Direct & Indirect redundancy
  - To avoid data anomaly: Updation, Insertion & Deletion

Data Anomalies

Sid	Sname	Cid	Cname	Fid	Fname	Salary
1	Ram	C1	DBMS	F1	Sachin	30000
2	Shyam	C2	Java	F2	Boby	28000
3	Ankit	C1	DBMS	F1	Sachin	30000
4	saurabh	C1	DBMS	F1	Sachin	30000

- Add details of new student- no problem
- Add a new course or faculty- problem
- Delete database of Sid=1- no problem
- Delete database of Sid=2- problem
- Change name of Sid=2-no problem
- Change salary of F1 30k to 40k will change 3 times only for one faculty
- Disadvantages
  - Increasees no. of relation
  - Reduces performance

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units
- Examples of nonatomic 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 if the domains of all attributes of R are atomic, i.e., every table contains exactly one value for each attribute
- Nonatomic values complicate storage and encourage redundant (repeated) storage of data
- Rules: Table column should contain atomic value

  There should not be any repeating group of data

First Normal Form

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

Second Normal Form

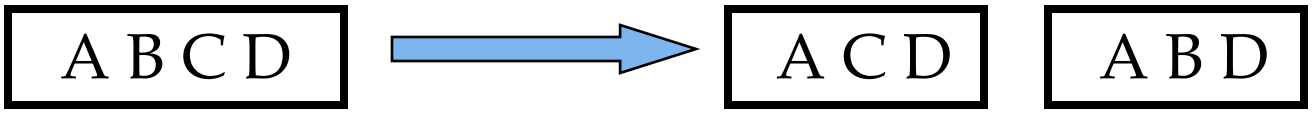
- Table or relation must be in1NF
- All the non-prime attributes should be fully functional dependent on CK.
- There should be no P.D. in the relation

TABLE_PURCHASE_DETAIL

Customer ID	Store ID	Purchase Location
1	1	Los Angeles
1	3	San Francisco
2	1	Los Angeles
3	2	New York
4	3	San Francisco

Redundancy

- *Redundancy:* root of several problems with relational schemas:
  - redundant storage, *insert/delete/update anomalies*
- *Functional dependencies:*
  - *integrity constraints* that can identify redundancy and suggest refinements.
- Main refinement technique: *decomposition*
  - replacing ABCD with, say, AB and BCD, or ACD and ABD.



- Decomposition should be used judiciously:
  - Is there reason to decompose a relation?
  - What problems (if any) does the decomposition cause?

- A **functional dependency** (FD) is a relationship that exists between two sets of attributes in a relation from a database.
- These dependencies are restrictions imposed on the data in DB
- Used to define various normal forms
- Given a relation *R*, a set of attributes *X* in *R* is said to **functionally determine** another attribute *Y*, also in *R*, (written $X \rightarrow Y$) if and only if each *X* value is associated with precisely one *Y* value.
- Customarily we call *X* the *determinant set* and *Y* the *dependent attribute*.
- L.H.S. attributes functionally determines R.H.S. attributes or R.H.S. attributes is a functionally dependent on L.H.S. attributes
- Each value of X is associated precisely with one Y value
- For example, in an "Employee" table that includes the attributes
- "Employee ID" and "Employee Date of Birth", the functional dependency {Employee ID} $\rightarrow$ {Employee Date of Birth} would hold.
- 2 attributes of a table is said to be dependent on each other when an attribute of a table uniquely identifies another attribute of the same table

- **Full functional dependency**
  - An attribute is fully functionally dependent on a set of attributes X if it is functionally dependent on X, and not functionally dependent on any proper subset of X. {Employee Address} has a functional dependency on {Employee ID, Skill},
  - but not a *full* functional dependency, because it is also dependent on {Employee ID}.

Functional Dependencies (FDs)

- A <u>functional dependency</u> $X \rightarrow Y$ holds over relation schema R if, for every allowable instance *r* of R:

$$t1 \in r,\ \ t2 \in r,\ \ \pi_X(t1) = \pi_X(t2)$$

$$\text{implies} \quad \pi_Y(t1) = \pi_Y(t2)$$

(where t1 and t2 are tuples;X and Y are sets of attributes)

- Explanation:
  - $X \rightarrow Y$ means:

    If for 2 tuples X is the same, then Y must also be the same.

CAUTION: The opposite is not true.

- Read "$\rightarrow$" as "determines"

FD's Continued

- An FD is a statement about *all* allowable relations.
  - Identified based on semantics, NOT instances
  - Given an instance of R, we can disprove a FD, but we cannot verify the validity of a FD.

- Question: Are FDs related to keys?
- if "K $\rightarrow$ all attributes of R" then K is a *superkey* for R

  (does not require K to be *minimal*.)
- FDs are a generalization of keys.

- Advantages:
  - Avoids data redundancy
  - Maintains the quality of data in DB
  - Allows clearly defined meaning & constraints of DB
  - Identifying bad designs

Functional dependencies are denoted by an arrow (). For instance, (EmployeeID \rightarrow EmployeeName) means that knowing the EmployeeID allows you to determine the EmployeeName.

Types

- Trivial

Trivial Functional Dependency: When an attribute is functionally dependent on itself or a subset of itself. For example, ({A} \rightarrow {A}).

  - A functional dependency FD: X → Y is called **trivial** if Y is a subset of X.
  - Example:
  - {EmployeeID, EmployeeAddress} → {EmployeeAddress}
  - Is trivial because {EmployeeAddress} → {EmployeeAddress}

- Non-trivial

Non-Trivial Functional Dependency: When (X \rightarrow Y) and (Y) is not a subset of (X). For example, (EmployeeID \rightarrow EmployeeName).

  - If a functional dependency X->Y holds true where Y is not a subset of X then this dependency is called non trivial Functional dependency.
  - Example :
  - An employee table with attributes: emp_id, emp_name, emp_address.
  - The following functional dependencies are non-trivial:
  - emp_id -> emp_name (emp_name is not a subset of emp_id)
  - emp_id -> emp_address (emp_address is not a subset of emp_id)
  - On the other hand, the following dependencies are trivial:
  - {emp_id, emp_name} -> emp_name [emp_name is a subset of {emp_id,emp_name}]

Types

- Completely non trivial FD:
- If a Functional dependency X->Y holds true where X intersection Y is Null then this dependency is said to be completely non trivial function dependency.
- Multivalued
  - Multivalued dependency occurs when there are more than one independent multivalued attributes in a table.
  - a multivalued dependency is a full constraint between two sets of attributes in a relation. In contrast to the functional dependency, the multivalued dependency requires that certain tuples be present in a relation.

Multivalued Dependency: When one attribute determines multiple independent attributes.

Types

- Consider a bike manufacture company, which produces two colors (Black and white) in each model every year.

bike_model	manuf_year	color
M1001	2007	Black
M1001	2007	Red
M2012	2008	Black
M2012	2008	Red
M2222	2009	Black
M2222	2009	Red

Types

- Here columns manuf_year and color are independent of each other and dependent on bike_model. In this case these two columns are said to be multivalued dependent on bike_model.
- These dependencies can be represented like this:
- bike_model ->> manuf_year
- bike_model ->> color

Types

- Transitive Transitive Dependency: If (X \rightarrow Y) and (Y \rightarrow Z), then (X \rightarrow Z)
  - A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies.
  - X -> Z is a transitive dependency if the following three functional dependencies hold true:
  - X->Y but Y does not ->X
  - Y->Z
  - A transitive dependency can only occur in a relation of three of more attributes. This dependency helps us normalizing the database in 3NF (3rd Normal Form).

Book	Author	Author_age
Game of Thrones	George R. R. Martin	66
Harry Potter	J. K. Rowling	49
Dying of the Light	George R. R. Martin	66

Types

- {Book} ->{Author} (if we know the book, we knows the author name)
- {Author} does not ->{Book}
- {Author} -> {Author_age}
- Therefore as per the rule of transitive dependency:
- {Book} -> {Author_age} should hold, that makes sense because if we know the book name we can know the author's age.
- Join dependency
  - A table *T* is subject to a join dependency if *T* can always be recreated by joining multiple tables each having a subset of the attributes of *T*.
  - If a relation has JD, then it can be decomposed into smaller relation. Such that one can rejoin these to get original relation.

Example: Constraints on Entity Set

- Consider relation obtained from Hourly_Emps:

  Hourly_Emps (*ssn, name, lot, rating, wage_per_hr, hrs_per_wk*)

- We sometimes denote a relation schema by listing the attributes: e.g., SNLRWH
- This is really the *set* of attributes {S,N,L,R,W,H}.

What are some FDs on Hourly_Emps?

ssn is the key: S → SNLRWH

rating determines *wage_per_hr*: R → W

lot determines *lot*: L → L ("trivial" dependency)

Decomposing a Relation

- Redundancy can be removed by "chopping" the relation into pieces (vertically!)
- FD's are used to drive this process.

  R $\rightarrow$ W is causing the problems, so decompose SNLRWH into what relations?

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Hourly_Emps2

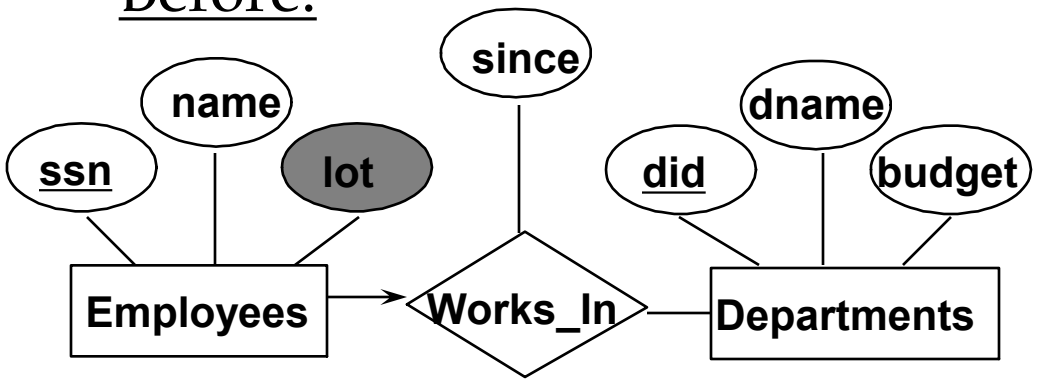
R	W
8	10
5	7

Wages

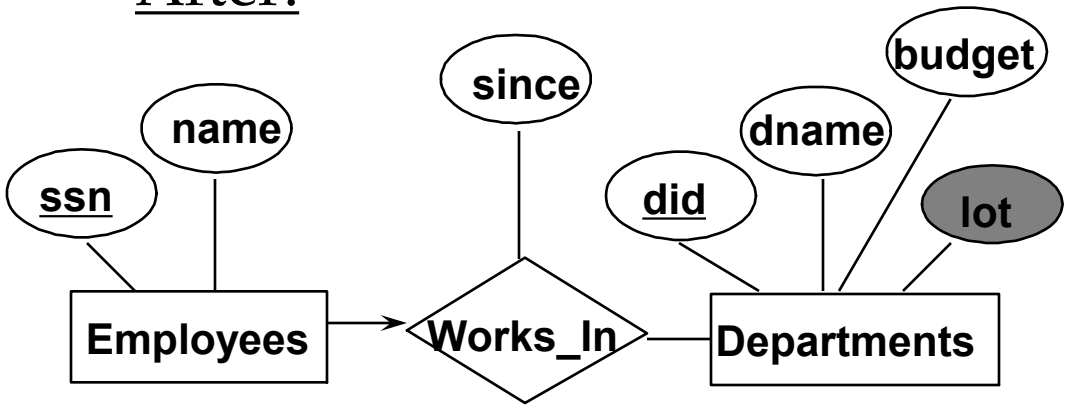
Refining an ER Diagram

- 1st diagram becomes:
  Workers(S,N,L,D,Si)
  Departments(D,M,B)
  - Lots associated with workers.
- Suppose all workers in a dept are assigned the same lot: D → L
- Redundancy; fixed by:
  Workers2(S,N,D,Si)
  Dept_Lots(D,L)
  Departments(D,M,B)
- Can fine-tune this:
  Workers2(S,N,D,Si)
  Departments(D,M,B,L)

Before:



After:



Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:

  title → *studio, star* implies *title* → *studio* and *title* → *star*

  title → *studio* and *title* → *star* implies *title* → *studio, star*

  title → *studio*, *studio* → *star* implies *title* → *star*

  But,

  title, star → *studio* does NOT necessarily imply that *title* → *studio* or that *star* → *studio*

- An FD *f* is *implied by* a set of FDs *F* if *f* holds whenever all FDs in *F* hold.

- F^+ = *closure of F* is the set of all FDs that are implied by *F*. (includes "trivial dependencies")

Rules of Inference

- The most important properties are Armstrong's axioms, which are used in database normalization(X, Y, Z are sets of attributes):
- **Subset Property** (Axiom of Reflexivity): If Y is a subset of X, then $X \rightarrow Y$
- **Augmentation** (Axiom of Augmentation): If $X \rightarrow Y$, then $XZ \rightarrow YZ$
- **Transitivity** (Axiom of Transitivity):If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These are *sound* and *complete* inference rules for FDs!
  - i.e., using AA you can compute all the FDs in F+ and only these FDs.

- Some additional (secondary) rules (that follow from AA):
  - *Union*: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - Composition: If $X \rightarrow Y$ and $A \rightarrow B$, then $AX \rightarrow BY$
  - *Decomposition*: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - Pseudo transitivity: : If $X \rightarrow Y$, and $YZ \rightarrow P$ then $XZ \rightarrow P$

Example

- Contracts(*cid,sid,jid,did,pid,qty,value*), and:
  - C is the key: $C \rightarrow CSJDPQV$
  - Proj purchases each part using single contract: $JP \rightarrow C$
  - Dept purchases at most 1 part from a supplier: $SD \rightarrow P$
- Problem: Prove that SDJ is a key for Contracts
- $JP \rightarrow C$, $C \rightarrow CSJDPQV$ imply $JP \rightarrow CSJDPQV$
  (by transitivity) (shows that JP is a key)
- $SD \rightarrow P$ implies $SDJ \rightarrow JP$ (by augmentation)
- $SDJ \rightarrow JP$, $JP \rightarrow CSJDPQV$ imply $SDJ \rightarrow CSJDPQV$
  (by transitivity) thus SDJ is a key.

Q: can you now infer that $SD \rightarrow CSDPQV$ (i.e., drop J on both sides)?

No! FD inference is not like arithmetic multiplication.

Attribute Closure

- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure of a set of FDs *F*. An efficient check:
  - Compute *attribute closure* of X (denoted X^+) wrt *F.*
    X^+ = Set of all attributes A such that $X \rightarrow A$ is in F^+
    - $X^+ := X$
    - Repeat until no change: if there is an FD $U \rightarrow V$ in *F* such that U is in X^+, then add V to X^+
  - Check if Y is in X^+
  - Approach can also be used to find the keys of a relation.
    - If all attributes of R are in the closure of X then X is a superkey for R.
    - Q: How to check if X is a "candidate key"?

Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive functional dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.
- A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency $X \rightarrow Y$.
- X is a super key.
- Y is a prime attribute, i.e., each element of Y is part of some candidate key.

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.
- Example: Let's assume there is a company where employees work in more than one department.
- EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

Fourth Normal Form (4NF)

- For a table to satisfy the Fourth Normal Form, it should satisfy the following two conditions:
  - It should be in the Boyce-Codd Normal Form.
  - And, the table should not have any Multi-valued Dependency.
- What is Multi-valued Dependency?
  - A table is said to have multi-valued dependency, if the following conditions are true,
  - For a dependency A → B, if for a single value of A, multiple value of B exists, then the table may have multi-valued dependency.
  - Also, a table should have at-least 3 columns for it to have a multi-valued dependency.
  - And, for a relation R(A,B,C), if there is a multi-valued dependency between, A and B, then B and C should be independent of each other.
  - If all these conditions are true for any relation(table), it is said to have multi-valued dependency.

- s_id course hobby
- 1 Science Cricket
- 1 Maths Hockey
- 2 C# Cricket
- 2 Php Hockey
- Here, student with s_id 1 has opted for two courses, Science and Maths, and has two hobbies, Cricket and Hockey.

5th Normal Form(5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Attribute Closure (example)

- R = {A, B, C, D, E}
- F = { B →CD, D → E, B → A, E → C, AD →B }
- Is B → E in F^+ ?

  $B^+ = B$

  $B^+ = BCD$

  $B^+ = BCDA$

  $B^+ = BCDAE$ … Yes!

  and B is a key for R too!
- Is D a key for R?

  $D^+ = D$

  $D^+ = DE$

  $D^+ = DEC$

  … Nope!

- **Is AD a key for R?**

  $AD^+ = AD$

  $AD^+ = ABD$ and B is a key, so Yes!
- **Is AD a *candidate* key for R?**

  $A^+ = A$, D+ = DEC

  … A,D not keys, so Yes!
- **Is ADE a *candidate* key for R?**

  … No! AD is a key, so ADE is a superkey, but not a cand. key

Normal Forms

- Recall FDs, Armstrong's Axioms, Attr. Closures.
- Q1: is any refinement needed??!
- If a relation is in a *normal form* (BCNF, 3NF etc.):
  - we know that certain problems are avoided/minimized.
  - helps decide whether decomposing a relation is useful.
- Role of FDs in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC.
    - No (non-trivial) FDs hold: There is no redundancy here.
    - Given $A \rightarrow B$: If A is not a key, then several tuples could have the same A value, and if so, they'll all have the same B value!
- 1st Normal Form – all attributes are atomic
  - i.e. the relational model
- 1st $\supset$ 2nd (of historical interest) $\supset$ 3rd $\supset$ Boyce-Codd $\supset$ …

Boyce-Codd Normal Form (BCNF)

- Reln R with FDs *F* is in BCNF if, for all $X \rightarrow A$ in F^+
  - $A \in X$ (called a *trivial* FD), or
  - X is a superkey for R.
- In other words: "R is in BCNF if the only non-trivial FDs over R are *key constraints*."
- If R in BCNF, then every field of every tuple records information that cannot be inferred using FDs alone.
  - Say we know FD $X \rightarrow A$ holds this example relation:

• Can you guess the value of the missing attribute?

•Yes, so relation is not in BCNF

X	Y	A
5	y1	7
5	y2	?

Decomposition of a Relation Schema

- If a relation is not in a desired normal form, it can be *decomposed* into multiple relations that each are in that normal form.
- Suppose that relation R contains attributes *A1 ... An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R, and
  - Every attribute of R appears as an attribute of at least one of the new relations.

Example (same as before)

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Hourly_Emps

- SNLRWH has FDs $S \rightarrow$ SNLRWH and $R \rightarrow W$
- Q: Is this relation in BCNF?

No, The second FD causes a violation;
W values repeatedly associated with R values.

Decomposing a Relation

- Easiest fix is to create a relation RW to store these associations, and to remove W from the main schema:

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Hourly_Emps2

R	W
8	10
5	7

Wages

- Q: Are both of these relations are now in BCNF?
- **Decompositions should be used only when needed.**
  - Q: potential problems of decomposition?

Problems with Decompositions

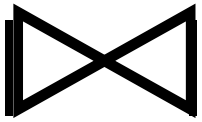
- There are three potential problems to consider:
  1) May be impossible to reconstruct the original relation! (Lossy Decomposition)
     - Fortunately, not in the SNLRWH example.
  2) Dependency checking may require joins (not Dependency Preserving)
     - Fortunately, not in the SNLRWH example.
  3) Some queries become more expensive.
     - e.g., How much does Guldu earn?

Tradeoff: **Must consider these issues vs. redundancy.**

(Well, not usually #1)

Lossless Decomposition (example)

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40



R	W
8	10
5	7

=

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

Lossy Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

$A \rightarrow B;\ C \rightarrow B$

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

Lossless Join Decompositions

- Decomposition of R into X and Y is *lossless* w.r.t. a set of FDs F if, for every instance *r* that satisfies F:

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem #1)*

More on Lossless Decomposition

- The decomposition of R into X and Y is lossless with respect to F *if and only if* the closure of F contains:

$$X \cap Y \rightarrow X, \quad \textbf{or}$$
$$X \cap Y \rightarrow Y$$

i.e. the common attributes form a **superkey** for one side or the other

in example: decomposing ABC into AB and BC is lossy, because intersection (i.e., "B") is not a key of either resulting relation.

- Useful result: If $W \rightarrow Z$ holds over R and $W \cap Z$ is empty, then decomposition of R into R-Z and WZ is loss-less.

Lossless Decomposition (example)

A	B	C
1	2	3
4	5	6
7	2	8

A	C
1	3
4	6
7	8

B	C
2	3
5	6
2	8

$A \rightarrow B$; $C \rightarrow B$

A	C
1	3
4	6
7	8

B	C
2	3
5	6
2	8

=

A	B	C
1	2	3
4	5	6
7	2	8

But, now we can't check $A \rightarrow B$ without doing a join!

Dependency Preserving Decomposition

- Dependency preserving decomposition (Intuitive):
  - If R is decomposed into X, Y and Z, and we enforce the FDs that hold individually on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem #2 on our list.)*
    - Why do we care??
- *Projection of set of FDs F* : If R is decomposed into X and Y the projection of F on X (denoted F_X) is the set of FDs $U \rightarrow V$ in F^+ (*closure of F , not just F*) such that all of the attributes U, V are in X. *(same holds for Y of course)*

Dependency Preserving Decompositions (Contd.)

- Decomposition of R into X and Y is *dependency preserving* if $(F_X \cup F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.
- Important to consider F^+ in this definition:
  - ABC, $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \rightarrow A$ preserved?????
    - note: F^+ contains $F \cup \{A \rightarrow C, B \rightarrow A, C \rightarrow B\}$, so…

Decomposition into BCNF

- Consider relation R with FDs F. If $X \rightarrow Y$ violates BCNF, decompose R into R - Y and XY (guaranteed to be loss-less).
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, $JP \rightarrow C$, $SD \rightarrow P$, $J \rightarrow S$
  - *{contractid, supplierid, projectid,deptid,partid, qty, value}*
  - To deal with $SD \rightarrow P$, decompose into SDP, CSJDQV.
  - To deal with $J \rightarrow S$, decompose CSJDQV into JS and CJDQV
  - So we end up with: SDP, JS, and CJDQV
- Note: several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS $\rightarrow$ Z, Z $\rightarrow$ C
  - Can't decompose while preserving 1st FD; not in BCNF.
- Similarly, decomposition of CSJDPQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP $\rightarrow$ C, SD $\rightarrow$ P and J $\rightarrow$ S).
- *{contractid, supplierid, projectid,deptid,partid, qty, value}*
  - However, it is a lossless join decomposition.
  - In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.
    - but JPC tuples are stored only for checking the f.d. (*Redundancy!*)

Third Normal Form (3NF)

- Reln R with FDs *F* is in 3NF if, for all $X \rightarrow A$ in F^+

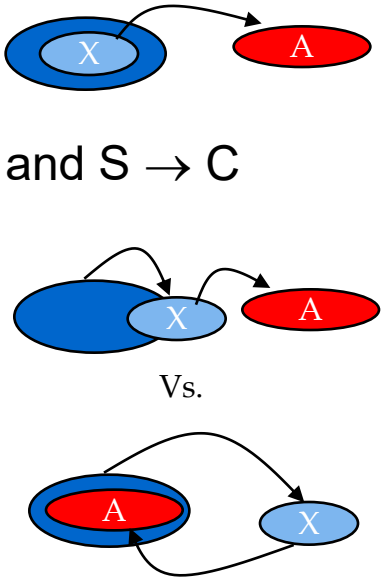
  $A \in X$ (called a *trivial* FD), or

  X is a superkey of R, or

  A is part of some candidate key (not superkey!) for R. (sometimes stated as "A is *prime*")
- *Minimality* of a key is crucial in third condition above!
- If R is in BCNF, obviously in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
  - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

What Does 3NF Achieve?

- If 3NF violated by $X \rightarrow A$, one of the following holds:
  - X is a subset of some key K ("partial dependency")
    - We store (X, A) pairs redundantly.
    - e.g. Reserves SBDC (C is for credit card) with key SBD and $S \rightarrow C$
  - X is not a proper subset of any key. ("transitive dep.")
    - There is a chain of FDs $K \rightarrow X \rightarrow A$
    - So we can't associate an X value with a K value unless we also associate an A value with an X value (different K's, same X implies same A!) – problem with initial SNLRWH example.



- But: even if R is in 3NF, these problems could arise.
  - e.g., Reserves SBDC (note: "C" is for credit card here), $S \rightarrow C$, $C \rightarrow S$ is in 3NF (why?)
  - Even so, for each reservation of sailor S, same (S, C) pair is stored.
- Thus, 3NF is indeed a compromise relative to BCNF.
  - You have to deal with the partial and transitive dependency issues in your application code!

An Aside: Second Normal Form

- Like 3NF, but allows transitive dependencies:
  - Reln R with FDs *F* is in 2NF if, for all $X \rightarrow A$ in F^+

    $A \in X$ (called a *trivial* FD), or

    X is a superkey of R, or

    X is not part of any candidate key for R.
    (i.e. "X is *not prime*")
- There's no reason to use this in practice
  - And we won't expect you to remember it

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier) but does not ensure dependency preservation.
- To ensure dependency preservation, one idea:
  - If $X \rightarrow Y$ is not preserved, add relation XY.

  Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' $JP \rightarrow C$. What if we also have $J \rightarrow C$?
- Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.

Minimal Cover for a Set of FDs

- *Minimal cover* G for a set of FDs F:
  - Closure of F = closure of G.
  - Right hand side of each FD in G is a single attribute.
  - If we modify G by deleting an FD or by deleting attributes from an FD in G, the closure changes.
- Intuitively, every FD in G is needed, and ``*as small as possible*'' in order to get the same closure as F.
- e.g., $A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow GH$, $ACDF \rightarrow EG$ has the following minimal cover:
  - $A \rightarrow B$, $ACD \rightarrow E$, $EF \rightarrow G$ and $EF \rightarrow H$
- M.C. implies Lossless-Join, Dep. Pres. Decomp!!!

Summary of Schema Refinement

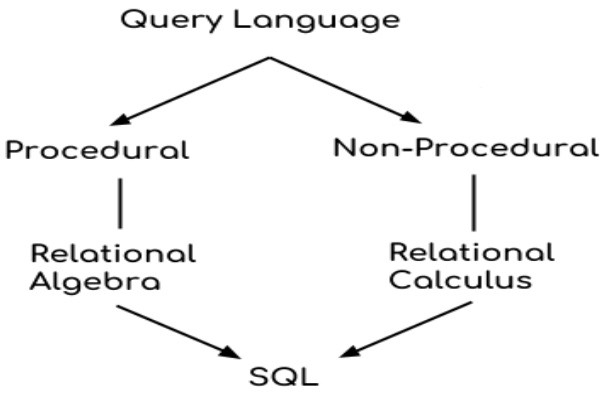
- BCNF: each field contains information that cannot be inferred using only FDs.
  - ensuring BCNF is a good heuristic.
- Not in BCNF? Try decomposing into BCNF relations.
  - Must consider whether all FDs are preserved!
- Lossless-join, dependency preserving decomposition into BCNF impossible? Consider 3NF.
  - Same if BCNF decomp is unsuitable for typical queries
  - Decompositions should be carried out and/or re-examined while keeping *performance requirements* in mind.

Review: Database Design

- Requirements Analysis
  - user needs; what must database do?
- Conceptual Design
  - high level descr (often done w/ER model)
- Logical Design
  - translate ER into DBMS data model
- Schema Refinement
  - consistency,normalization
- Physical Design - indexes, disk layout
- Security Design - who accesses what

Relational algebra & Relational calculus

- For designing of database we just use Relational model, E-R diagram and normalization. Now that we have designed the database, we need to store and retrieve data from the database, for this purpose we need to understand the concept of Relational algebra and relational calculus.
- Query Language
  - a Language which is used to store and retrieve data from database is known as query language. For example – SQL
- There are two types of query language:
  - 1.Procedural Query language
  - 2.Non-procedural query language



- 1. Procedural Query language:
  - In procedural query language, user instructs the system to perform a series of operations to produce the desired results. Here users tells what data to be retrieved from database and how to retrieve it.
  - For example – RA is conceptual procedural query language used on relational model.
- 2. Non-procedural query language:
  - In Non-procedural query language, user instructs the system to produce the desired result without telling the step by step process. Here users tells what data to be retrieved from database but doesn't tell how to retrieve it.
  - RC is conceptual non procedural query language used on relational model
- Relational algebra and calculus are the theoretical concepts used on relational model.
- RDBMS is a practical implementation of relational model.
- SQL is a practical implementation of relational algebra and calculus.

Relational Algebra

- Relational algebra is a procedural query language that works on relational model. The purpose of a query language is to retrieve data from database or perform various operations such as insert, update, delete on the data.
- Relational algebra is a procedural query language, it means that it tells what data to be retrieved and how to be retrieved.
- On the other hand relational calculus is a non-procedural query language, which means it tells what data to be retrieved but doesn't tell how to retrieve it.

Types of operations in relational algebra

- We have divided these operations in two categories:
  - 1. Basic Operations
  - 2. Derived Operations
- Basic/Fundamental Operations:
  - 1. Select (σ)
  - 2. Project (∏)
  - 3. Union (∪)
  - 4. Set Difference (-)
  - 5. Cartesian product (X)
  - 6. Rename (ρ)
- Derived Operations:
  - 1. Natural Join (⋈)
  - 2. Left, Right, Full outer join (⟕, ⟖, ⟗)
  - 3. Intersection (∩)
  - 4. Division (÷)

Select Operator (σ)

- Select Operator (σ)
  - Select Operator is denoted by sigma (σ) and it is used to find the tuples (or rows) in a relation (or table) which satisfy the given condition.
- Syntax of Select Operator (σ)
  - σ Condition/Predicate(Relation/Table name)
- Select Operator (σ) Example
  - Table: CUSTOMER
  - Customer_Id Customer_Name Customer_City
  - C10100 Steve Agra
  - C10111 Raghu Agra
  - C10115 Chaitanya Noida
  - C10117 Ajeet Delhi
  - C10118 Carl Delhi
- Query:
  - σ Customer_City="Agra" (CUSTOMER)
- Output:
  - Customer_Id Customer_Name Customer_City
  - C10100 Steve Agra
  - C10111 Raghu Agra

Project Operator (∏)

- Project Operator (∏)
  - Project operator is denoted by ∏ symbol and it is used to select desired columns (or attributes) from a table (or relation).
  - Project operator in relational algebra is similar to the Select statement in SQL.
- Syntax of Project Operator (∏)
  - ∏ column_name1, column_name2, ..., column_nameN(table_name)
- Project Operator (∏) Example
  - we have a table CUSTOMER with three columns, we want to fetch only two columns of the table, which we can do with the help of Project Operator ∏.
  - Table: CUSTOMER
  - Customer_Id Customer_Name Customer_City
  - C10100 Steve Agra
  - C10111 Raghu Agra
  - C10115 Chaitanya Noida
  - C10117 Ajeet Delhi
  - C10118 Carl Delhi
- Query:
  - ∏ Customer_Name, Customer_City (CUSTOMER)
- Output:
  - Customer_Name Customer_City
  - Steve Agra
  - Raghu Agra
  - Chaitanya Noida
  - Ajeet Delhi
  - Carl Delhi

Union Operator (∪)

- Union operator is denoted by ∪ symbol and it is used to select all the rows (tuples) from two tables (relations).
- two relations R1 and R2 both have same columns and we want to select all the tuples(rows) from these relations then we can apply the union operator on these relations.
- Note: The rows (tuples) that are present in both the tables will only appear once in the union set. In short you can say that there are no duplicates present after the union operation.
- Syntax of Union Operator (∪)
  - table_name1 ∪ table_name2
- Union Operator (∪) Example
  - Table 1: COURSE
  - Course_Id Student_Name Student_Id
  - C101 Aditya S901
  - C104 Aditya S901
  - C106 Steve S911
  - C109 Paul S921
  - C115 Lucy S931
- Table 2: STUDENT
  - Student_Id Student_Name Student_Age
  - S901 Aditya 19
  - S911 Steve 18
  - S921 Paul 19
  - S931 Lucy 17
  - S941 Carl 16
  - S951 Rick 18

Query:
∏ Student_Name (COURSE) ∪ ∏ Student_Name (STUDENT)

Output:
Student_Name
Aditya
Carl
Paul
Lucy
Rick
Steve

Intersection Operator (∩)

- Intersection operator is denoted by ∩ symbol and it is used to select common rows (tuples) from two tables (relations).
  - two relations R1 and R2 both have same columns and we want to select all those tuples(rows) that are present in both the relations, then in that case we can apply intersection operation on these two relations R1 ∩ R2.
  - Note: Only those rows that are present in both the tables will appear in the result set.
- Syntax of Intersection Operator (∩)
  - table_name1 ∩ table_name2
- Intersection Operator (∩) Example
  - Table 1: COURSE
  - Course_Id Student_Name Student_Id
  - C101 Aditya S901
  - C104 Aditya S901
  - C106 Steve S911
  - C109 Paul S921
  - C115 Lucy S931
  - Table 2: STUDENT
  - Student_Id Student_Name Student_Age
  - S901 Aditya 19
  - S911 Steve 18
  - S921 Paul 19
  - S931 Lucy 17
  - S941 Carl 16
  - S951 Rick 18

Output:

Student_Name
Aditya
Steve
Paul
Lucy

- Query:
  - ∏ Student_Name (COURSE) ∩ ∏ Student_Name (STUDENT)

Set Difference (-)

- Set Difference: denoted by – symbol. Lets say we have two relations R1 and R2 and we want to select all those tuples(rows) that are present in Relation R1 but not present in Relation R2, this can be done using Set difference R1 – R2.
- Syntax of Set Difference (-)
  - table_name1 - table_name2
- Set Difference (-) Example
- Query:
  - Lets write a query to select those student names that are present in STUDENT table but not present in COURSE table.
  - ∏ Student_Name (STUDENT) - ∏ Student_Name (COURSE)
- Output:
  - Student_Name
  - Carl
  - Rick

Cartesian product (X)

- Cartesian Product: denoted by X symbol. Lets say we have two relations R1 and R2 then the cartesian product of these two relations (R1 X R2) would combine each tuple of first relation R1 with the each tuple of second relation R2.
- Syntax of Cartesian product (X)
  - R1 X R2
- Cartesian product (X) Example
  - Table 1: R
  - Col_A Col_B
  - AA 100
  - BB 200
  - CC 300
  - Table 2: S
  - Col_X Col_Y
  - XX 99
  - YY 11
  - ZZ 101
- Query:
  - Lets find the cartesian product of table R and S.
  - R X S
- Output:
  - Col_A Col_B Col_X Col_Y
  - AA 100 XX 99
  - AA 100 YY 11
  - AA 100 ZZ 101
  - BB 200 XX 99
  - BB 200 YY 11
  - BB 200 ZZ 101
  - CC 300 XX 99
  - CC 300 YY 11
  - CC 300 ZZ 101

Rename (ρ)

- Rename (ρ) operation can be used to rename a relation or an attribute of a relation.
- Rename (ρ) Syntax:
  - ρ(new_relation_name, old_relation_name)
- Rename (ρ) Example
  - Lets say we have a table customer, we are fetching customer names and we are renaming the resulted relation to CUST_NAMES.
- Table: CUSTOMER
  - Customer_Id Customer_Name Customer_City
  - C10100 Steve Agra
  - C10111 Raghu Agra
  - C10115 Chaitanya Noida
  - C10117 Ajeet Delhi
  - C10118 Carl Delhi
- Query:
  - ρ(CUST_NAMES, ∏(Customer_Name)(CUSTOMER))
- Output:
  - CUST_NAMES
  - Steve
  - Raghu
  - Chaitanya
  - Ajeet
  - Carl

Join Operations

- Join operation is essentially a cartesian product followed by a selection criterion.
- Join operation denoted by ⋈.
- JOIN operation also allows joining variously related tuples from different relations.
- Types of JOIN:
- Various forms of join operation are:
- Inner Joins:
  - Theta join
  - EQUI join
  - Natural join
- Outer join:
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join

Inner Join

- In an inner join, only those tuples that satisfy the matching criteria are included, while the rest are excluded. Let's study various types of Inner Joins:
- **Theta Join:**
- The general case of JOIN operation is called a Theta join. It is denoted by symbol **θ**
- Example
- `A ⋈θ B`
- Theta join can use any conditions in the selection criteria.
- For example:
- `A ⋈ A.column 2 >  B.column 2 (B)`

column 1	column 2
1	2

- **EQUI join:**
- When a theta join uses only equivalence condition, it becomes a equi join.
- For example:
- `A ⋈ A.column 2 =  B.column 2 (B)`

A ⋈ A.column 2 = B.column 2 (B)	
column 1	**column 2**
1	1

- EQUI join is the most difficult operations to implement efficiently using SQL in an RDBMS and one reason why RDBMS have essential performance problems.

- NATURAL JOIN ($\bowtie$)
- $C \bowtie D$
- Natural join can only be performed if there is a common attribute (column) between the relations. The name and type of the attribute must be same.
- Example
- Consider the following two tables

C	
Num	**Square**
2	4
3	9

D	
Num	**Cube**
2	8
3	27

C ⋈ D		
Num	**Square**	**Cube**
2	4	8
3	9	27

OUTER JOIN

- In an outer join, along with tuples that satisfy the matching criteria, we also include some or all tuples that do not match the criteria.
- Left Outer Join(A⟕B)
- In the left outer join, operation allows keeping all tuple in the left relation. However, if there is no matching tuple is found in right relation, then the attributes of right relation in the join result are filled with null values.

- Consider the following 2 Tables

A	
Num	**Square**
2	4
3	9
4	16

B	
Num	**Cube**
2	8
3	18
5	75

A ⋈ B		
Num	**Square**	**Cube**
2	4	8
3	9	18
4	16	-

- Right Outer Join: (A ⟖ B)
- In the right outer join, operation allows keeping all tuple in the right relation. However, if there is no matching tuple is found in the left relation, then the attributes of the left relation in the join result are filled with null values.

A ⟖ B

A ⋈ B		
Num	**Cube**	**Square**
2	8	4
3	18	9
5	75	-

- **Full Outer Join: (A ⟗ B)**
- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.
- A ⟗ B

A ⋈ B		
Num	**Cube**	**Square**
2	4	8
3	9	18
4	16	-
5	-	75

Division Operator (÷)

- Division operator A÷B can be applied if and only if:
  - Attributes of B is proper subset of Attributes of A.
  - The relation returned by division operator will have attributes = (All attributes of A – All Attributes of B)
  - The relation returned by division operator will return those tuples from relation A which are associated to every B's tuple.
- Consider the relation STUDENT_SPORTS and ALL_SPORTS
- To apply division operator as
  - STUDENT_SPORTS÷ ALL_SPORTS
- The operation is valid as attributes in ALL_SPORTS is a proper subset of attributes in STUDENT_SPORTS.
- The attributes in resulting relation will have attributes {ROLL_NO,SPORTS}-{SPORTS}=ROLL_NO
- The tuples in resulting relation will have those ROLL_NO which are associated with all B's tuple {Badminton, Cricket}. ROLL_NO 1 and 4 are associated to Badminton only. ROLL_NO 2 is associated to all tuples of B. So the resulting relation will be:
  - ROLL_NO
  - 2

Relational Calculus

- Relational calculus is a non-procedural query language that tells the system what data to be retrieved but doesn't tell how to retrieve it.
- Tuple Relational Calculus (TRC)
  - Tuple relational calculus is used for selecting those tuples that satisfy the given condition.
- Table: Student
  - First_Name Last_Name Age
  - Ajeet Singh 30
  - Chaitanya Singh 31
  - Rajeev Bhatia 27
  - Carl Pratap 28
- Query to display the last name of those students where age is greater than 30
  - { t.Last_Name | Student(t) AND t.age > 30 }
- The result of the above query would be:
  - Last_Name
  - Singh
- Query to display all the details of students where Last name is 'Singh'
  - { t | Student(t) AND t.Last_Name = 'Singh' }
  - Output:
  - First_Name Last_Name Age
  - Ajeet Singh 30
  - Chaitanya Singh 31

Domain Relational Calculus (DRC)

- In domain relational calculus the records are filtered based on the domains.
- Again we take the same table to understand how DRC works.
- Table: Student
  - First_Name Last_Name Age
  - Ajeet Singh 30
  - Chaitanya Singh 31
  - Rajeev Bhatia 27
  - Carl Pratap 28
- Query to find the first name and age of students where student age is greater than 27
  - {< First_Name, Age > | ∈ Student ∧ Age > 27}
- Note:
  - The symbols used for logical operators are: ∧ for AND, ∨ for OR and ¬ for NOT.
- Output:
- First_Name Age
- Ajeet 30
- Chaitanya 31
- Carl 28

Table-1: Customer

Customer name	Street	City
Saurabh	A7	Patiala
Mehak	B6	Jalandhar
Sumiti	D9	Ludhiana
Ria	A5	Patiala

Table-2: Branch

Branch name	Branch city
ABC	Patiala
DEF	Ludhiana
GHI	Jalandhar

Table-3: Account

Account number	Branch name	Balance
1111	ABC	50000
1112	DEF	10000
1113	GHI	9000
1114	ABC	7000

Table-4: Borrower

Customer name	Loan number
Saurabh	L33
Mehak	L49
Ria	L98

Table-5: Depositor

Customer name	Account number
Saurabh	1111
Mehak	1113
Sumiti	1114

Table-6: Loan

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L49	GHI	9000
L98	DEF	65000

- Queries-1: Find the loan number, branch, amount of loans of greater than or equal to 10000 amount.
  - {t| t ∈ loan ∧ t[amount]>=10000}
  - Resulting relation:
  - In the above query, t[amount] is known as tupple variable.
- Queries-2: Find the loan number for each loan of an amount greater or equal to 10000.
  - {t| ∃ s ∈ loan(t[loan number] = s[loan number] ∧ s[amount]>=10000)}

- Queries-3: Find the names of all customers who have a loan and an account at the bank.
  - {t | ∃ s ∈ borrower(t[customer-name] = s[customer-name])
  - ∧ ∃ u ∈ depositor(t[customer-name] = u[customer-name])}
- Queries-4: Find the names of all customers having a loan at the “ABC” branch.
  - {t | ∃ s ∈ borrower(t[customer-name] = s[customer-name]
  - ∧ ∃ u ∈ loan(u[branch-name] = “ABC” ∧ u[loan-number] = s[loan-number]))}

Loan number	Branch name	Amount
L33	ABC	10000
L35	DEF	15000
L98	DEF	65000

Loan number
L33
L35
L98

Customer name
Saurabh
Mehak

Customer name
Saurabh

- Assume the following relations:
- BOOKS(DocId, Title, Publisher, Year)
- STUDENTS(StId, StName, Major, Age)
- AUTHORS(AName, Address)
- borrows(DocId, StId, Date)
- has-written(DocId, AName)
- describes(DocId, Keyword)

- *List the year and title of each book.*

  $\pi_{\text{Year, Title}}(\text{BOOKS})$

- *List all information about students whose major is CS.*

  $\sigma_{\text{Major = 'CS'}}(\text{STUDENTS})$

- *List all students with the books they can borrow.*

  $\text{STUDENTS} \times \text{BOOKS}$

- *List all books published by McGraw-Hill before 1990.*

  $\sigma_{\text{Publisher = 'McGraw-Hill'} \wedge \text{Year} < 1990}(\text{BOOKS})$

- *List the name of those authors who are living in Davis.*

  $\pi_{\text{AName}}(\sigma_{\text{Address like '\%Davis\%'}}(\text{AUTHORS}))$

- *List the name of students who are older than 30 and who are not studying CS.*

  $\pi_{\text{StName}}(\sigma_{\text{Age}>30}(\text{STUDENTS})) -$
  $\pi_{\text{StName}}(\sigma_{\text{Major='CS'}}(\text{STUDENTS}))$

- *Rename* `AName` *in the relation* `AUTHORS` *to* `Name`.

  $\rho_{\text{AUTHORS(Name, Address)}}(\text{AUTHORS})$

1. *List the names of all students who have borrowed a book and who are CS majors.*

$$\pi_{\text{StName}}(\sigma_{\text{STUDENTS.StId}=\text{borrows.StId}} (\sigma_{\text{Major}=\text{'CS'}}(\text{STUDENTS}) \times \text{borrows}))$$

2. *List the title of books written by the author 'Silberschatz'.*

$$\pi_{\text{Title}}(\sigma_{\text{AName}=\text{'Silberschatz'}} (\sigma_{\text{has-written.DocId}=\text{BOOKS.DocID}}(\text{has-written} \times \text{BOOKS})))$$

or

$$\pi_{\text{Title}}(\sigma_{\text{has-written.DocId}=\text{BOOKS.DocID}} (\sigma_{\text{AName}=\text{'Silberschatz'}}(\text{has-written}) \times \text{BOOKS}))$$

3. *As 2., but not books that have the keyword 'database'.*

. . . as for 2. . . .

$$- \ \pi_{\text{Title}}(\sigma_{\text{describes.DocId}=\text{BOOKS.DocId}} (\sigma_{\text{Keyword}=\text{'database'}}(\text{describes}) \times \text{BOOKS}))$$

4. *Find the name of the youngest student.*

$$\pi_{\text{StName}}(\text{STUDENTS}) - \pi_{\text{S1.StName}}(\sigma_{\text{S1.Age}>\text{S2.Age}}(\rho_{\text{S1}}(\text{STUDENTS}) \times \rho_{\text{S2}}(\text{STUDENTS})))$$

5. *Find the title of the oldest book.*

$$\pi_{\text{Title}}(\text{BOOKS}) - \pi_{\text{B1.Title}}(\sigma_{\text{B1.Year}>\text{B2.Year}}(\rho_{\text{B1}}(\text{BOOKS}) \times \rho_{\text{B2}}(\text{BOOKS})))$$

1. *List each book with its keywords.*

   $BOOKS \bowtie Descriptions$

   Note that books having no keyword are not in the result.

2. *List each student with the books s/he has borrowed.*

   $BOOKS \bowtie (borrows \bowtie STUDENTS)$

3. *List the title of books written by the author 'Ullman'.*

   $\pi_{Title}(\sigma_{AName='Ullman'}(BOOKS \bowtie has\text{-}written))$

   or

   $\pi_{Title}(BOOKS \bowtie \sigma_{AName='Ullman'}(has\text{-}written))$

4. *List the authors of the books the student 'Smith' has borrowed.*

   $\pi_{AName}(\sigma_{StName='Smith'}(has\text{-}written \bowtie (borrows \bowtie STUDENTS))$

5. *Which books have both keywords 'database' and 'programming'?*

   $BOOKS \bowtie (\pi_{DocId}(\sigma_{Keyword='database'}(Descriptions)) \cap \pi_{DocId}(\sigma_{Keyword='programming'}(Descriptions)))$

   or

   $BOOKS \bowtie (Descriptions \div \{('database'), ('programming')\})$

   with $\{('database'), ('programming')\})$ being a constant relation.

6. Query 4 using assignments.

   $temp1 \longleftarrow borrows \bowtie STUDENTS$

   $temp2 \longleftarrow has\text{-}written \bowtie temp1$

   $result \longleftarrow \pi_{AName}(\sigma_{StName='Smith'}(temp2))$

- Retrieve the name and address of all employees who work for the ’Research’ department.
- RESEARCH DEPT ←− σDNAME=0Research0(DEPARTMENT)
- RESEARCH EMPS ←− (RESEARCH DEPT ./DNUMBER=DNO (EMPLOY EE))
- RESULT ←− πF NAME,LNAME,ADDRESS(RESEARCH EMPS)
- Find the names of employees who work on all the projects controlled by department
- number 5.
- DEPT5 PROJS(PNO) ←− πP NUMBER(σDNUM=5(PROJECT))
- EMP PROJ(SSN, PNO) ←− πESSN,P NO(WORKS ON)
- RESULT EMP SSNS ←− EMP PROJ ÷ DEPT PROJS
- RESULT ←− πLNAME,F NAME(RESULT EMP SSNS ∗ EMPLOY EE)

- EXAMPLE 1: Restrict the S relation with the condition that supplier is in London, and store the result in a temporary relation T1.
- T1 := S WHERE CITY = 'London'
- Note: In the above, ':=' is the assignment operator.

- EXAMPLE 2: Project T1 to attribute S# and store the result in a temporary relation T2.
- T2 := T1 [S#]

- EXAMPLE 3: Find suppliers in London and store the result in a temporary relation T2.
- T2 := (S WHERE CITY = 'London') [S#]

- EXAMPLE 4: Find parts supplied by suppliers in London.
- (T2 JOIN SP) [P#]
- or equivalently
- ((S WHERE CITY = 'London') JOIN SP) [P#]
- Note: Why we did not project on P# before the join?

- EXAMPLE 5: Create snapshot relation SR containing parts supplied by suppliers in London.
- SR := ((S WHERE CITY = 'London') JOIN SP) [P#]

- EXAMPLE 6: What new parts have been added by the suppliers in London during the last six months?
- If SR1 is the snapshot relation created six months ago, and SR2 is the new snapshot relation, then the new parts can be obtained by:
- SR2 MINUS SR1

- EXAMPLE 7: What parts are no longer supplied by the suppliers in London?
- SR1 MINUS SR2

- EXAMPLE 8: What supplier supplies all the parts no longer supplied by the suppliers in London?
- (SP [S#, P#]) DIVIDE_BY (SR1 MINUS SR2)
- Note: For the DIVIDE_BY operator to work, the two relations must be binary and unary.

- EXAMPLE 9: Where are the suppliers supplying all the parts no longer supplied by the suppliers in London?

- (((SP [S#, P#]) DIVIDE_BY (SR1 MINUS SR2)) JOIN S) [CITY]

SQL

- SQL stands for Structured Query Language. It is used for storing and managing data in relational database management system (RDMS).
- It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables.
- All the RDBMS like MySQL, Informix, Oracle, MS Access and SQL Server use SQL as their standard database language.
- SQL allows users to query the database in a number of ways, using English-like statements.
- Rules:
  - Structure query language is not case sensitive. Generally, keywords of SQL are written in uppercase.
  - Statements of SQL are dependent on text lines. We can use a single SQL statement on one or multiple text line.
  - Using the SQL statements, you can perform most of the actions in a database.
  - SQL depends on tuple relational calculus and relational algebra.

- Characteristics of SQL
  - SQL is easy to learn.
  - SQL is used to access data from relational database management systems.
  - SQL can execute queries against the database.
  - SQL is used to describe the data.
  - SQL is used to define the data in the database and manipulate it when needed.
  - SQL is used to create and drop the database and table.
  - SQL is used to create a view, stored procedure, function in a database.
  - SQL allows users to set permissions on tables, procedures, and views.

- SQL CREATE DATABASE Statement
- The CREATE DATABASE statement is used to create a new SQL database.
- Syntax
  - CREATE DATABASE databasename;
- The following SQL statement creates a database called "testDB":
  - CREATE DATABASE testDB;
- DROP DATABASE statement is used to drop an existing SQL database.
- Syntax
  - DROP DATABASE databasename;
- The following SQL statement drops the existing database "testDB":
  - DROP DATABASE testDB;

Data Type

- 1. Binary Datatypes
- There are Three types of binary Datatypes which are given below:
  - Data Type Description
  - binary It has a maximum length of 8000 bytes. It contains fixed-length binary data.
  - varbinary It has a maximum length of 8000 bytes. It contains variable-length binary data.
  - image It has a maximum length of 2,147,483,647 bytes. It contains variable-length binary data.
- 2. Approximate Numeric Datatype :
  - Data type From To Description
  - float -1.79E + 308 1.79E + 308 It is used to specify a floating-point value e.g. 6.2, 2.9 etc.
  - real -3.40e + 38 3.40E + 38 It specifies a single precision floating point number

- 3. Exact Numeric Datatype
  - Data type Description
  - int It is used to specify an integer value.
  - smallint It is used to specify small integer value.
  - bit It has the number of bits to store.
  - decimal It specifies a numeric value that can have a decimal number.
  - numeric It is used to specify a numeric value.
- 4. Character String Datatype
  - Data type Description
  - char It has a maximum length of 8000 characters. It contains Fixed-length non-unicode characters.
  - varchar It has a maximum length of 8000 characters. It contains variable-length non-unicode characters.
  - text It has a maximum length of 2,147,483,647 characters. It contains variable-length non-unicode characters.
- 5. Date and time Datatypes
  - Datatype Description
  - date It is used to store the year, month, and days value.
  - time It is used to store the hour, minute, and second values.
  - timestamp It stores the year, month, day, hour, minute, and the second value.

- Operation on Table
  - Create table
  - Drop table
  - Delete table
  - Rename table
- SQL Create Table
- SQL create table is used to create a table in the database. To define the table, you should define the name of the table and also define its columns and column's data type.
- Syntax
  - create table "table_name"
  - ("column1" "data type",
  - "column2" "data type",
  - "column3" "data type",
  - ...
  - "columnN" "data type");
- Example
  - SQL> CREATE TABLE EMPLOYEE (
  - EMP_ID INT NOT NULL,
  - EMP_NAME VARCHAR (25) NOT NULL,
  - PHONE_NO INT NOT NULL,
  - ADDRESS CHAR (30),
  - PRIMARY KEY (ID)
  -);

- If you create the table successfully, you can verify the table by looking at the message by the SQL server. Else you can use DESC command as follows:
- SQL> DESC EMPLOYEE;
- Field Type Null Key Default Extra
- EMP_IDint(11) NO PRI NULL
- EMP_NAME varchar(25) NO NULL
- PHONE_NO NO int(11) NULL
- ADDRESS YES NULL char(30)
- 4 rows in set (0.35 sec)
- Now you have an EMPLOYEE table in the database, and you can use the stored information related to the employees.
- Drop table
- used to delete a table definition and all the data from a table. When this command is executed, all the information available in the table is lost forever, so you have to very careful while using this command.
- Syntax
- DROP TABLE "table_name";
- Firstly, you need to verify the EMPLOYEE table using the following command:
- SQL> DESC EMPLOYEE;
- Field Type Null Key Default Extra
- EMP_ID int(11) NO PRI NULL
- EMP_NAME varchar(25) NO NULL
- PHONE_NO NO int(11) NULL
- ADDRESS YES NULL char(30)
- 4 rows in set (0.35 sec)
- This table shows that EMPLOYEE table is available in the database, so we can drop it as follows:

- SQL>DROP TABLE EMPLOYEE;
- Now, we can check whether the table exists or not using the following command:
- Query OK, 0 rows affected (0.01 sec)
- As this shows that the table is dropped, so it doesn't display it.
- SQL DELETE table
- used to delete rows from a table. We can use WHERE condition to delete a specific row from a table. If you want to delete all the records from the table, then you don't need to use the WHERE clause.
- Syntax
- DELETE FROM table_name WHERE condition;
- Example
- Suppose, the EMPLOYEE table having the following records:

EMP_ID	EMP_NAME	CITY	PHONE_NO	SALARY
1	Kristen	Chicago	9737287378	150000
2	Russell	Austin	9262738271	200000
3	Denzel	Boston	7353662627	100000
4	Angelina	Denver	9232673822	600000
5	Robert	Washington	9367238263	350000
6	Christian	Los angels	7253847382	260000

- The following query will DELETE an employee whose ID is 3.
- SQL> DELETE FROM EMPLOYEE
- WHERE EMP_ID = 3;
- Now, the EMPLOYEE table would have the following records.

EMP_ID	EMP_NAME	CITY	PHONE_NO	SALARY
1	Kristen	Chicago	9737287378	150000
2	Russell	Austin	9262738271	200000
4	Angelina	Denver	9232673822	600000
5	Robert	Washington	9367238263	350000
6	Christian	Los angels	7253847382	260000

- If you don't specify the WHERE condition, it will remove all the rows from the table.
- DELETE FROM EMPLOYEE;
- Now, the EMPLOYEE table would not have any records.

Commit, Rollback and Savepoint

- Transaction Control Language(TCL) commands are used to manage transactions in the database.
- COMMIT command is used to permanently save any transaction into the database.
- When we use any DML command like INSERT, UPDATE or DELETE, the changes made by these commands are not permanent, until the current session is closed, the changes made by these commands can be rolled back.
- To avoid that, we use the COMMIT command to mark the changes as permanent.
- COMMIT;
- ROLLBACK command
- This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction.
- If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.
- ROLLBACK TO savepoint_name;
- SAVEPOINT command
- SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.

- SAVEPOINT savepoint_name;
- In short, using this command we can name the different states of our data in any table and then rollback to that state using the ROLLBACK command whenever required.
- Using Savepoint and Rollback
- Following is the table class,
  - id name
  - 1 Abhi
  - 2 Adam
  - 4 Alex
  - INSERT INTO class VALUES(5, 'Rahul');
  - COMMIT;
  - UPDATE class SET name = 'Abhijit' WHERE id = '5';
  - SAVEPOINT A;
  - INSERT INTO class VALUES(6, 'Chris');
  - SAVEPOINT B;
  - INSERT INTO class VALUES(7, 'Bravo');
  - SAVEPOINT C;
  - SELECT * FROM class;

- id name
- 1 Abhi
- 2 Adam
- 4 Alex
- 5 Abhijit
- 6 Chris
- 7 Bravo

- Now let's use the ROLLBACK command to roll back the state of data to the savepoint B.
- ROLLBACK TO B;
- SELECT * FROM class;
  - id name
  - 1 Abhi
  - 2 Adam
  - 4 Alex
  - 5 Abhijit
  - 6 Chris
- Now let's again use the ROLLBACK command to roll back the state of data to the savepoint A
- ROLLBACK TO A;
- SELECT * FROM class;
  - id name
  - 1 Abhi
  - 2 Adam
  - 4 Alex
  - 5 Abhijit

GRANT and REVOKE

- Data Control Language(DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables, sequences or views, a user needs privileges. Privileges are of two types,
- System: This includes permissions for creating session, table, etc and all types of other system privileges.
- Object: This includes permissions for any command or query to perform any operation on the database tables.
- In DCL we have two commands,
  - GRANT: Used to provide any user access privileges or other priviliges for the database.
  - REVOKE: Used to take back permissions from any user.
  - Allow a User to create session
- When we create a user in SQL, it is not even allowed to login and create a session until and unless proper permissions/priviliges are granted to the user.
- To grant the session creating priviliges.
- GRANT CREATE SESSION TO username;
- Allow a User to create table
  - To allow a user to create tables in the database, we can use the below command,
  - GRANT CREATE TABLE TO username;
  - Provide user with space on tablespace to store table
  - Allowing a user to create table is not enough to start storing data in that table. We also must provide the user with priviliges to use the available tablespace for their table and data

- ALTER USER username QUOTA UNLIMITED ON SYSTEM;
- The above command will alter the user details and will provide it access to unlimited tablespace on system.
- NOTE: Generally unlimited quota is provided to Admin users.
- Grant all privilege to a User
- sysdba is a set of priviliges which has all the permissions in it. So if we want to provide all the privileges to any user, we can simply grant them the sysdba permission.
- GRANT sysdba TO username
- Grant permission to create any table
- Sometimes user is restricted from creating come tables with names which are reserved for system tables. But we can grant privileges to a user to create any table using the below command,
- GRANT CREATE ANY TABLE TO username
- Grant permission to drop any table
- As the title suggests, if you want to allow user to drop any table from the database, then grant this privilege to the user,
- GRANT DROP ANY TABLE TO username
- To take back Permissions
- And, if you want to take back the privileges from any user, use the REVOKE command.
- REVOKE CREATE TABLE FROM username

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.
- Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:
- X's Account
  - Open_Account(X)
  - Old_Balance = X.balance
  - New_Balance = Old_Balance - 800
  - X.balance = New_Balance
  - Close_Account(X)
- Y's Account
  - Open_Account(Y)
  - Old_Balance = Y.balance
  - New_Balance = Old_Balance + 800
  - Y.balance = New_Balance
  - Close_Account(Y)

Operations of Transaction

- Main operations of transaction:
  - Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.
  - Write(X): Write operation is used to write the value back to the database from the buffer.
- Let's take an example to debit transaction from an account which consists of following operations:
  - 1. R(X);
  - 2. X = X - 500;
  - 3. W(X);
- Let's assume the value of X before starting of the transaction is 4000.
  - The first operation reads X's value from database and stores it in a buffer.
  - The second operation will decrease the value of X by 500. So buffer will contain 3500.
  - The third operation will write the buffer's value to the database. So X's final value will be 3500.
  - But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.
- For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.
- Commit: It is used to save the work done permanently.
- Rollback: It is used to undo the work done.

Transaction property

- Atomicity
  - It states that all operations of the transaction take place at once if not, the transaction is aborted.
  - There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.
  - Atomicity involves the following two operations:
  - Abort: If a transaction aborts then all the changes made are not visible.
  - Commit: If a transaction commits then all the changes made are visible.
- Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.
  - T1 T2
  - Read(A)
  - A:= A-100
  - Write(A) Read(B)
  - Y:= Y+100
  - Write(B)
- After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.
  - If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

- Consistency
  - The integrity constraints are maintained so that the database is consistent before and after the transaction.
  - The execution of a transaction will leave a database in either its prior stable state or a new stable state.
  - The consistent property of database states that every transaction sees a consistent database instance.
  - The transaction is used to transform the database from one consistent state to another consistent state.
- For example: The total amount must be maintained before or after the transaction.
  - Total before T occurs = 600+300=900
  - Total after T occurs= 500+400=900
  - Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.
- Isolation
  - It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
  - In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
  - The concurrency control subsystem of the DBMS enforced the isolation property.

- Durability
  - The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
  - They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
  - The recovery subsystem of the DBMS has the responsibility of Durability property.

Stored Procedure

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.
- You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.
- Stored Procedure Syntax
  - CREATE PROCEDURE procedure_name
  - AS
  - sql_statement
  - GO;
- Execute a Stored Procedure
  - EXEC procedure_name;

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country	
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany	
2	Ana Trujillo	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico	
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico	
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK	
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden	

- CREATE PROCEDURE SelectAllCustomers
- AS
- SELECT * FROM Customers
- GO;
- EXEC SelectAllCustomers;

- Stored Procedure With One Parameter
- The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:
- CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
- AS
- SELECT * FROM Customers WHERE City = @City
- GO;
- Execute the stored procedure above as follows:
- Example
- EXEC SelectAllCustomers @City = 'London';
- Stored Procedure With Multiple Parameters
- Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.
- The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

- CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
- AS
- SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
- GO;
- Execute the stored procedure above as follows:
- EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';

- The main advantages of stored procedure are given below:
- Better Performance –
- The procedure calls are quick and efficient as stored procedures are compiled once and stored in executable form.Hence the response is quick. The executable code is automatically cached, hence lowers the memory requirements.
- Higher Productivity –
- Since the same piece of code is used again and again so, it results in higher productivity.
- Ease of Use –
- To create a stored procedure, one can use any Java Integrated Development Environment (IDE). Then, they can be deployed on any tier of network architecture.
- Scalability –
- Stored procedures increase scalability by isolating application processing on the server.
- Maintainability –Maintaining a procedure on a server is much easier then maintaining copies on various client machines, this is because scripts are in one location.
- Security –Access to the Oracle data can be restricted by allowing users to manipulate the data only through stored procedures that execute with their definer's privileges.

- Disadvantages :
- Testing –Testing of a logic which is encapsulated inside a stored procedure is very difficult. Any data errors in handling stored procedures are not generated until runtime.
- Debugging –Depending on the database technology, debugging stored procedures will either be very difficult or not possible at all. Some relational databases such as SQL Server have some debugging capabilities.
- Versioning –Version control is not supported by the stored procedure.
- Cost –An extra developer in the form of DBA is required to access the SQL and write a better stored procedure. This will automatically incur added cost.
- Portability –Complex stored procedures will not always port to upgraded versions of the same database. This is specially true in case of moving from one database type(Oracle) to another database type(MS SQL Server).

View

- SQL CREATE VIEW Statement
- In SQL, a view is a virtual table based on the result-set of an SQL statement.
- A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
- You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
- CREATE VIEW Syntax
  - CREATE VIEW view_name AS
  - SELECT column1, column2, ...
  - FROM table_name
  - WHERE condition;
- Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.
- SQL CREATE VIEW Examples
- The following SQL creates a view that shows all customers from Brazil:
  - CREATE VIEW [Brazil Customers] AS
  - SELECT CustomerName, ContactName
  - FROM Customers
  - WHERE Country = 'Brazil';
- We can query the view above as follows:
  - SELECT * FROM [Brazil Customers];
- The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:
  - CREATE VIEW [Products Above Average Price] AS
  - SELECT ProductName, Price
  - FROM Products
  - WHERE Price > (SELECT AVG(Price) FROM Products);
- We can query the view above as follows:
  - SELECT * FROM [Products Above Average Price];

View

- View from multiple tables can be created by simply include multiple tables in the SELECT statement.
- In the given example, a view is created named MarksView from two tables Student_Detail and Student_Marks.
  - CREATE VIEW MarksView AS
  - SELECT Student_Detail.NAME, Student_Detail.ADDRESS, Student_Marks.MARKS
  - FROM Student_Detail, Student_Mark
  - WHERE Student_Detail.NAME = Student_Marks.NAME;

- SQL Updating a View
- A view can be updated with the CREATE OR REPLACE VIEW command.
- SQL CREATE OR REPLACE VIEW Syntax
  - CREATE OR REPLACE VIEW view_name AS
  - SELECT column1, column2, ...
  - FROM table_name
  - WHERE condition;
- The following SQL adds the "City" column to the "Brazil Customers" view:
  - CREATE OR REPLACE VIEW [Brazil Customers] AS
  - SELECT CustomerName, ContactName, City
  - FROM Customers
  - WHERE Country = 'Brazil';
- SQL Dropping a View
- A view is deleted with the DROP VIEW command.
- SQL DROP VIEW Syntax
  - DROP VIEW view_name;
- The following SQL drops the "Brazil Customers" view:
  - DROP VIEW [Brazil Customers];

View

- Inserting a row in a view:
- We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.Syntax:
  - INSERT INTO view_name(column1, column2 , column3,..)
  - VALUES(value1, value2, value3..);
- In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.
  - INSERT INTO DetailsView(NAME, ADDRESS)
  - VALUES("Suresh","Gurgaon");
- Deleting a row from a View:
- Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.Syntax:
  - DELETE FROM view_name
  - WHERE condition;
- In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.
  - DELETE FROM DetailsView
  - WHERE NAME="Suresh";

View

- The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.
- The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
- If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.
- In the below example we are creating a View SampleView from StudentDetails Table with WITH CHECK OPTION clause.
  - CREATE VIEW SampleView AS
  - SELECT S_ID, NAME
  - FROM StudentDetails
  - WHERE NAME IS NOT NULL
  - WITH CHECK OPTION;
- In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.
- For example,though the View is updatable but then also the below query for this View is not valid:
  - INSERT INTO SampleView(S_ID)
  - VALUES(6);
- NOTE: The default value of NAME column is null.
- Uses of a View : A good database should contain views due to the given reasons:
- Restricting data access –Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- Hiding data complexity –A view can hide the complexity that exists in a multiple table join.
- Simplify commands for the user –Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.
- Store complex queries –Views can be used to store complex queries.
- Rename Columns –Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.
- Multiple view facility –Different views can be created on the same table for different users.

Trigger

- A trigger is a stored procedure or stored programs in database which automatically executed or fired whenever a special event in the database occurs. Triggers are, in fact, written to be executed in response to any of the following events −
- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP).
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
- Triggers can be defined on the table, view, schema, or database with which the event is associated.
  - create trigger [trigger_name]
  - [before | after]
  - {insert | update | delete}
  - on [table_name]
  - [for each row]
  - [trigger_body]
- create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
- [before | after]: This specifies when the trigger will be executed.
- {insert | update | delete}: This specifies the DML operation.
- on [table_name]: This specifies the name of the table associated with the trigger.
- [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
- [trigger_body]: This provides the operation to be performed as trigger is fired
- BEFORE and AFTER of Trigger:
- BEFORE triggers run the trigger action before the triggering statement is run.
- AFTER triggers run the trigger action after the triggering statement is run.
- Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and average of specified marks is automatically inserted whenever a record is insert.
- Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Trigger

- Benefits of Triggers
  - Generating some derived column values automatically
  - Enforcing referential integrity
  - Event logging and storing information on table access
  - Auditing
  - Synchronous replication of tables
  - Imposing security authorizations
  - Preventing invalid transactions
- Suppose the database Schema –

```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30) | YES  |     | NULL    |                |
| subj1 | int(2)      | YES  |     | NULL    |                |
| subj2 | int(2)      | YES  |     | NULL    |                |
| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
```

- SQL Trigger to problem statement.
- create trigger stud_marks
- before INSERT
- on
- Student
- for each row
- set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total * 60 / 100;
- Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,
- insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);

- Select * from customers;
- +----+----------+-----+-----------+----------+
- | ID | NAME | AGE | ADDRESS | SALARY |
- +----+----------+-----+-----------+----------+
- | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
- | 2 | Khilan | 25 | Delhi | 1500.00 |
- | 3 | kaushik | 23 | Kota | 2000.00 |
- | 4 | Chaitali | 25 | Mumbai | 6500.00 |
- | 5 | Hardik | 27 | Bhopal | 8500.00 |
- | 6 | Komal | 22 | MP | 4500.00 |
- +----+----------+-----+-----------+----------+
- The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −
  - CREATE OR REPLACE TRIGGER display_salary_changes
  - BEFORE DELETE OR INSERT OR UPDATE ON customers
  - FOR EACH ROW
  - WHEN (NEW.ID > 0)
  - DECLARE
  - sal_diff number;
  - BEGIN
  - sal_diff := :NEW.salary - :OLD.salary;
  - dbms_output.put_line('Old salary: ' || :OLD.salary);
  - dbms_output.put_line('New salary: ' || :NEW.salary);
  - dbms_output.put_line('Salary difference: ' || sal_diff);
  - END;

- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (7, 'Kriti', 22, 'HP', 7500.00);
- When a record is created in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −
- Old salary:
- New salary: 7500
- Salary difference:
- Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −
- UPDATE customers
- SET salary = salary + 500
- WHERE id = 2;
- When a record is updated in the CUSTOMERS table, the above create trigger, display_salary_changes will be fired and it will display the following result −

- Old salary: 1500
- New salary: 2000
- Salary difference: 500

Cursor

- Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.
- A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.
- You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −
  - Implicit cursors
  - Explicit cursors
- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.
- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.
- In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT. The SQL cursor has additional attributes, %BULK_ROWCOUNT and %BULK_EXCEPTIONS, designed for use with the FORALL statement. The following table provides the description of the most used attributes −
- S.No Attribute & Description
- 1 %FOUND Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
- 2 %NOTFOUND The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
- 3 %ISOPEN Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
- 4 %ROWCOUNT Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Cursor

- Any SQL cursor attribute will be accessed as sql%attribute_name as shown below in the example.
- Example
- We will be using the CUSTOMERS table we had created and used in the previous chapters.
- Select * from customers;
  - +----+----------+-----+-----------+----------+
  - | ID | NAME | AGE | ADDRESS | SALARY |
  - +----+----------+-----+-----------+----------+
  - | 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
  - | 2 | Khilan | 25 | Delhi | 1500.00 |
  - | 3 | kaushik | 23 | Kota | 2000.00 |
  - | 4 | Chaitali | 25 | Mumbai | 6500.00 |
  - | 5 | Hardik | 27 | Bhopal | 8500.00 |
  - | 6 | Komal | 22 | MP | 4500.00 |
  - +----+----------+-----+-----------+----------+
- The following program will update the table and increase the salary of each customer by 500 and use the SQL%ROWCOUNT attribute to determine the number of rows affected −
  - DECLARE
  - total_rows number(2);
  - BEGIN
  - UPDATE customers
  - SET salary = salary + 500;
  - IF sql%notfound THEN
  - dbms_output.put_line('no customers selected');
  - ELSIF sql%found THEN
  - total_rows := sql%rowcount;
  - dbms_output.put_line(total_rows || ' customers selected ');
  - END IF;
  - END;
- /

Cursor

- When the above code is executed at the SQL prompt, it produces the following result −
- 6 customers selected
- PL/SQL procedure successfully completed.
- If you check the records in customers table, you will find that the rows have been updated −
- Select * from customers;
  - +----+----------+-----+-----------+----------+
  - | ID | NAME | AGE | ADDRESS | SALARY |
  - +----+----------+-----+-----------+----------+
  - | 1 | Ramesh | 32 | Ahmedabad | 2500.00 |
  - | 2 | Khilan | 25 | Delhi | 2000.00 |
  - | 3 | kaushik | 23 | Kota | 2500.00 |
  - | 4 | Chaitali | 25 | Mumbai | 7000.00 |
  - | 5 | Hardik | 27 | Bhopal | 9000.00 |
  - | 6 | Komal | 22 | MP | 5000.00 |
  - +----+----------+-----+-----------+----------+
- Explicit Cursors
- Explicit cursors are programmer-defined cursors for gaining more control over the context area. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.
- The syntax for creating an explicit cursor is −
- CURSOR cursor_name IS select_statement;
- Working with an explicit cursor includes the following steps −
  - Declaring the cursor for initializing the memory
  - Opening the cursor for allocating the memory
  - Fetching the cursor for retrieving the data
  - Closing the cursor to release the allocated memory
  - Declaring the Cursor
- Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −
  - CURSOR c_customers IS
  - SELECT id, name, address FROM customers;

Cursor

- Opening the Cursor
- Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −
  - OPEN c_customers;
- Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −
  - FETCH c_customers INTO c_id, c_name, c_addr;
- Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −
  - CLOSE c_customers;
- Following is a complete example to illustrate the concepts of explicit cursors;
  - DECLARE
  - c_id customers.id%type;
  - c_name customer.name%type;
  - c_addr customers.address%type;
  - CURSOR c_customers is
  - SELECT id, name, address FROM customers;
  - BEGIN
  - OPEN c_customers;
  - LOOP
  - FETCH c_customers into c_id, c_name, c_addr;
  - EXIT WHEN c_customers%notfound;
  - dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  - END LOOP;
  - CLOSE c_customers;
  - END;
  - /
- When the above code is executed at the SQL prompt, it produces the following result −
  - 1 Ramesh Ahmedabad
  - 2 Khilan Delhi
  - 3 kaushik Kota
  - 4 Chaitali Mumbai
  - 5 Hardik Bhopal
  - 6 Komal MP