UNIT - I
INTRODUCTION to DBMS (Part -I)

- ***Introduction***
  - What is a database management system?
  - Why study databases? Why not use file systems?
  - The three-level architecture
  - Schemas and instances
- ***Overview***
  - Data models, E-R model, Relational model
  - Data Definition Language, Data Manipulation Language
  - SQL
  - Transaction Management, Storage Management
  - User types, database administrator
  - System Structure

1.1 What is a Database Management System?

Data

- Data is raw fact or figures or entity.
- When activities in the organization takes place, the effect of these activities need to be recorded which is known as Data.

Information

- Processed data is called information
- The purpose of data processing is to generate the information required for carrying out the business activities.

Database

- Database may be defined in simple terms as a collection of data
- A database is a collection of related data.

Database Management System

- A Database Management System (DBMS) is a collection of program that enables user to create and maintain a database.
- The DBMS is hence a general purpose software system that facilitates the process of defining constructing and manipulating database for various applications.

- ***History***
  - 1950s-60s: magnetic tape and punched cards
  - 1960s-70s: hard disks, random access, file systems
  - 1970s-80s: relational model becoming competitive
  - 1980s-90s: relational model dominant, object-oriented databases
  - 1990s-00s: web databases and XML

Why Study Databases?

- They touch every aspect of our lives
- ***Applications:***
  - Banking: all transactions
  - Airlines: reservations, schedules
  - Universities: registration, course enrolment, grades
  - Sales: customers, products, purchases
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
  - Telecommunications: subscribers, usage, routing
  - Computer accounts: privileges, quotas, usage

- Records: climate, stock market, library holdings
- ***Explosion of unstructured data on the web:***
  - Large document collections
  - Image databases, streaming media

1.2 Why not use file systems?

- ***Data redundancy and inconsistency***
  - Multiple file formats
  - Duplication of information in different files
- ***Difficulty in accessing data***
  - Need to write a new program to carry out each new task
- ***Data isolation***
  - Multiple files and formats
- ***Integrity problems***
  - Integrity constraints (e.g. account balance > 0) become part of program code
  - Hard to add new constraints or change existing ones
- ***Maintenance problems***
  - When we add a new field, all existing applications must be modified to ignore it
- ***Atomicity of updates***
  - Failures may leave database in an inconsistent state with partial updates carried out
  - E.g. transfer of funds from one account to another should either complete or not happen at all
- ***Concurrent access by multiple users***
  - Concurrent accessed needed for performance
  - Uncontrolled concurrent accesses can lead to inconsistencies
    - E.g. two people reading a balance and updating it at the same time
  - Security problems

Database systems offer solutions to all the above problems

Advantages of DBMS.

- ✓ Due to its centralized nature, the database system can overcome the disadvantages of the file system-based system

1. **Data independency**: Application program should not be exposed to details of data representation and storage DBMS provides the abstract view that hides these details.

2. **Efficient data access**: DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently.

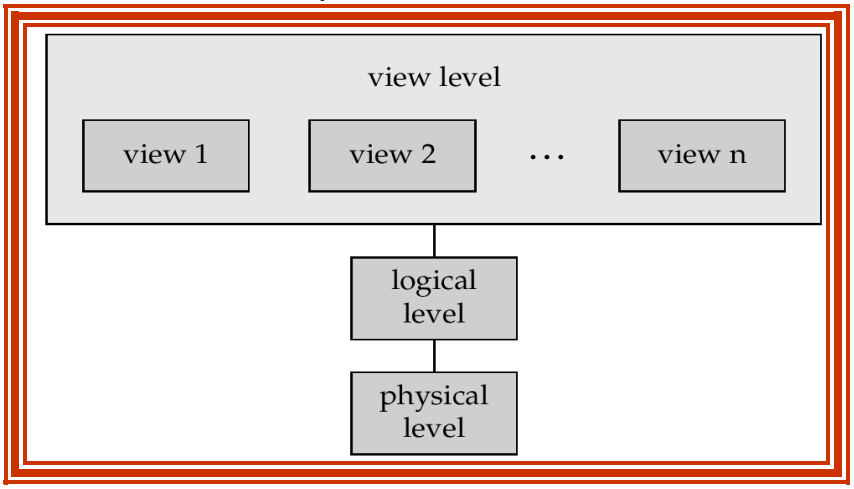
3. **Data integrity and security**: Data is accessed through DBMS, it can enforce integrity constraints. E.g.: Inserting salary information for an employee.

4. **Data Administration**: When users share data, centralizing the data is an important task, Experience professionals can minimize data redundancy and perform fine tuning which reduces retrieval time.

5. **Concurrent access and Crash recovery:** DBMS schedules concurrent access to the data. DBMS protects user from the effects of system failure.

1.3 The Levels of Abstraction

- ***Physical level:*** how a record is stored on disk
- ***Logical level:*** describes data stored in database, and the relationships among the data.
- type customer = record
  name : string;
  street : string;
  city : integer;
  end;
- ***View level:*** application-specific selections and arrangements of the data
- hide details of data types
- Views can also hide information for security reasons



1.4 Schemas *vs.* Instances

- ***Schema***
  - the logical structure of the database.
  - e.g., the database consists of information about a set of customers and accounts and the relationship between them.
  - Analogous to type information of a variable in a program.
- ***Instance***
  - the actual content of the database at a particular point in time.
  - Analogous to the value of a variable.
- ***Data Independence***
  - the ability to modify a schema in one-level (i.e. Internal Schema or Conceptual Schema) without affecting a schema in the next-higher-level (i.e. Conceptual or External schema)
  - Applications depend on the logical schema
  - Database engines take care of efficient storage and query processing
  - *Data independence are of two types:*
    - Physical Data Independence: Physical data independence is the ability to modify the physical schema – i.e. internal schema which describes the physical storage devices or structure to store the data – without affecting the conceptual schema – application programs.
    - Logical Data Independence: Logical data is the ability to modify the logical schema – i.e. Conceptual Schema, which decides what information is to be kept in the database – without affecting the next higher level schema – i.e., External Schema – application program.

1.5 Data Models

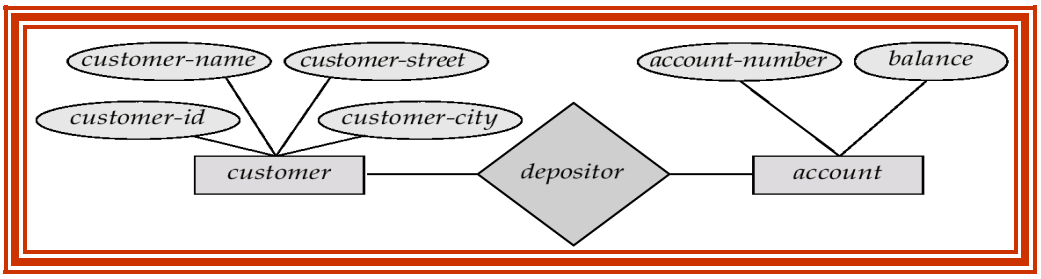
- A collection of tools for describing
  - data
  - data relationships
  - data semantics
  - data constraints
- The data models are divided into different groups
  - Object-Based Logical Data Models
  - Record-Based Logical Data Models

➢ Object-Based Logical Model

- Object – based logical models are used in describing data at logical level and view level. Logical and view levels are used to retrieve the data.
- *Object-Based Logical Models are described in the different following models:*
  - ✓ The Entity-Relationship Model
  - ✓ Object-Oriented Model

■ Entity-Relationship Model

- An entity is a thing or object in the real world that is distinguishable from other objects.
- The Entity – Relationship Model is based on a collection of basic objects, called entities, and the relationship among these objects.
- Example of schema in the entity-relationship model



- ♦ Rectangles represent entities
- ♦ Diamonds represent relationship among entities
- ♦ Ellipse represents attributes
- ♦ Lines represents links of attributes to entities to relations

■ Object-Oriented Model

- Like the E-R model, the Object-Oriented Model is based on a collection of objects. An object contains values stored in instance variables, within the object also contains bodies of code that operates on the object. These bodies of code are called as methods.
- Objects that contain the same types of values and the same methods are grouped together into classes. A class may be viewed as a definition for objects. This combination of data and methods comprising a definition is similar to a Programming-language abstract data type.
- The only way in which one object can access the data of another object is by invoking a method of that other object. This action is called sending a message to the object.

➢ Record-Based Logical Models

- Describes data at logical and view levels.
- Compared with object-based data models, the record-based logical models specify the overall logical structure of the database and provide higher level implementation.
  - Relational Model

■ Relational Model

- The relational model represents both data (entities) and relationships among in the form of tables. Each table has multiple columns and each column has a unique name.

■ Example of tabular data in the relational model

Attributes

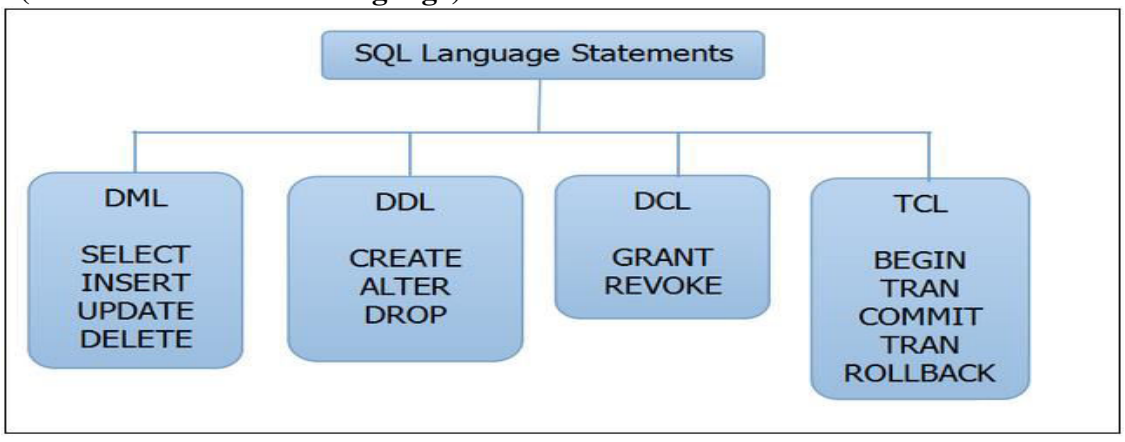
Customer-id	*customer-name*	*customer-street*	*customer-city*	*account-number*
192-83-7465	Johnson	Alma	Palo Alto	A-101
019-28-3746	Smith	North	Rye	A-215
192-83-7465	Johnson	Alma	Palo Alto	A-201
321-12-3123	Jones	Main	Harrison	A-217
019-28-3746	Smith	North	Rye	A-201

- The description of data in terms of tables is called as relations, from the above Customer and Accounts relations, we can make a condition that customer details are maintained in Customer table and their deposit details are maintained in the account table database.

1.6 DATABASE LANGUAGES

SQL language is divided into four types of primary language statements: DML, DDL, DCL and TCL. Using these statements, we can define the structure of a database by creating and altering database objects, and we can manipulate data in a table through updates or deletions. We also can control which user can read/write data or manage transactions to create a single unit of work.

- The four main categories of SQL statements are as follows:
  1. **DML (Data Manipulation Language)**
  2. **DDL (Data Definition Language)**
  3. **DCL (Data Control Language)**
  4. **TCL (Transaction Control Language)**



Here we Discuss only DDL and DML

■ Data Definition Language (DDL)

- Specification notation for defining the database schema
  - E.g.
    create table *account* (
    account-number char(10),
    balance integer)

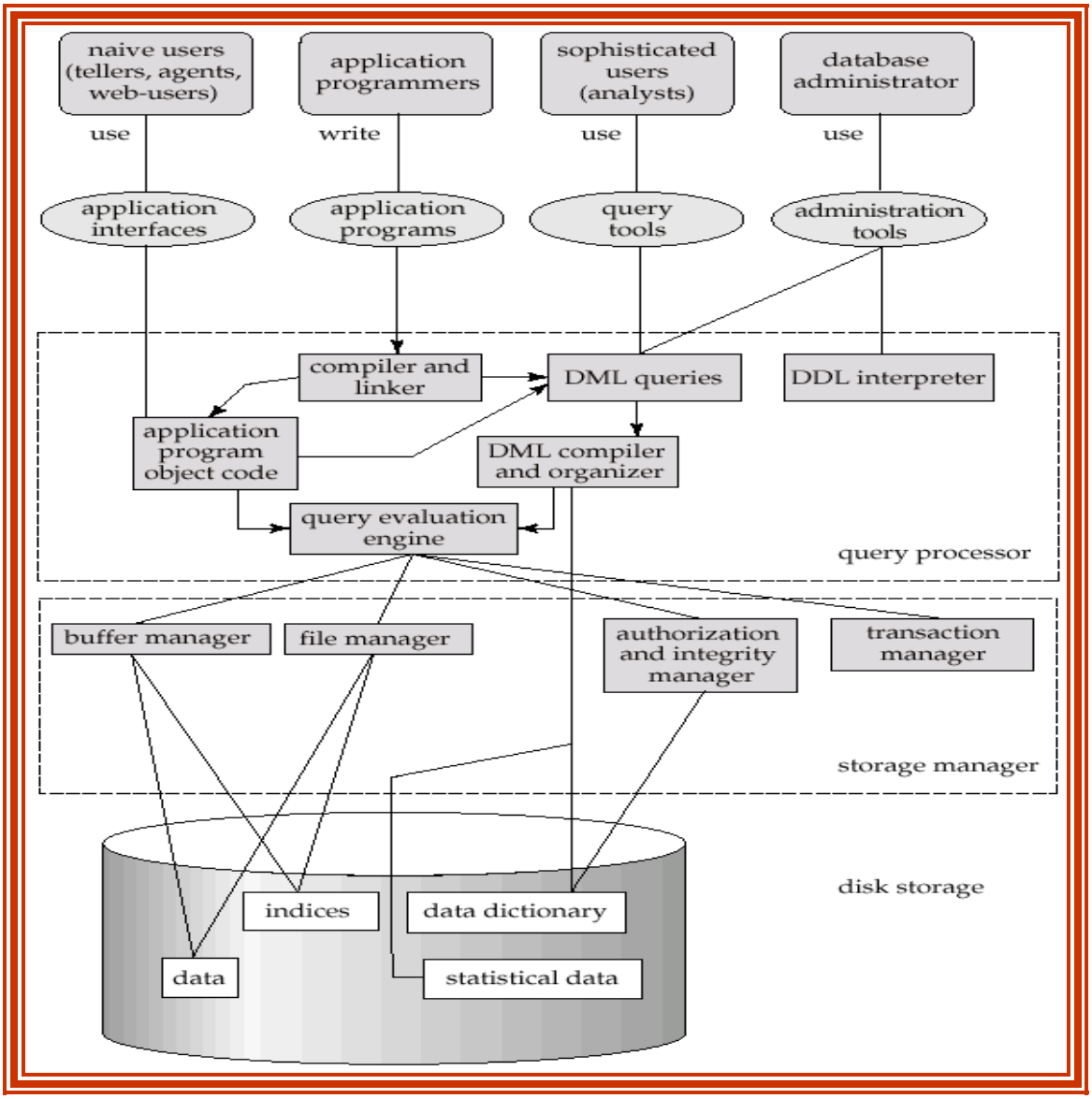
- DDL compiler generates a set of tables stored in a *data dictionary:*
  - Database schema
  - Specification of storage structures and access methods

■ 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
  - Procedural – user specifies what data is required and how to get those data
  - Nonprocedural – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language

1.7 Overall System Structure of DBMS

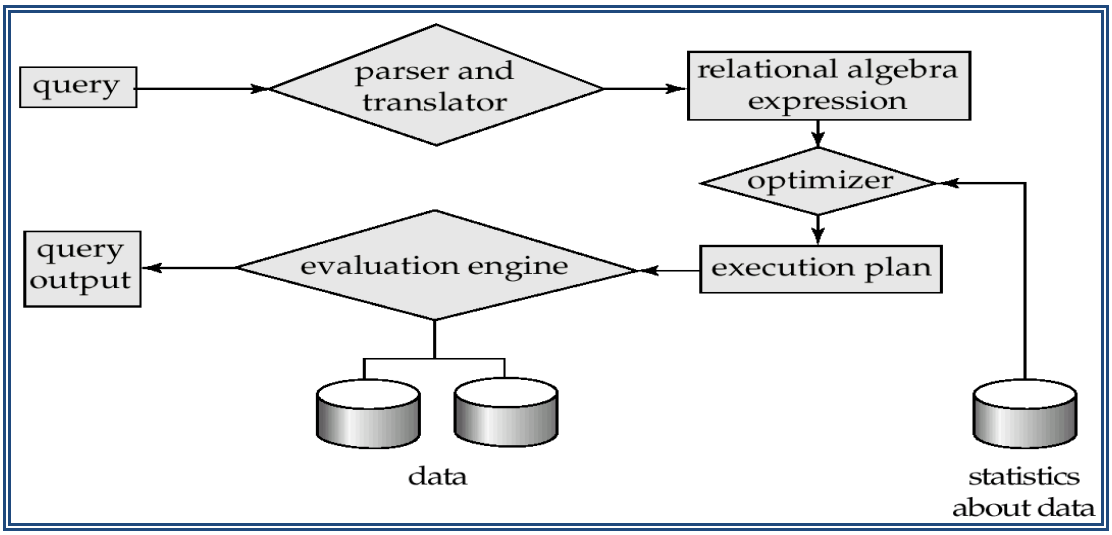
- The following figure shows the structure supporting parts of a DBMS with some simplification based on the relation data model.
- *A DBMS is divided into <u>two</u> modules (parts)*
  - Query processor
  - Storage Manager



■ Query Processor

The Query processor components are:

- **DDL Interpreter:** This interprets DDL statements and records the definitions in the data dictionary.
- **DML Compiler:** as any other compiler, DML Compiler converts the DML Statements into low-level instructions.
- **Query Evaluation:** This executes low-level instructions generated by the DML compiler.



■ Storage Manager

- A storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible for storing, retrieving and updating data in the database.
- The storage manager components include:
  - **Authorization and integrity Manager:** This tests for the satisfaction of integrity constraints and checks the authority of users to access data.
  - **Transaction Manager:** This ensures that the database remains in a consistent state despite system failures, and that concurrent transaction executions proceed without conflicting.
  - **File Manager:** This manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
  - **Buffer Manager:** This is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory.
- The storage manager implements several data structures as part of the physical system implementation:
  - **Data Files:** This store the database itself.
  - **Data Dictionary:** This stores metadata about the structure of the database, in particular the schema of the database.
  - **Indices:** This provides fast access to data items that hold particular values.

➢ 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
  - Schema and physical organization modification
  - Granting user authority to access the database
  - Specifying integrity constraints
  - Acting as liaison with users
  - Monitoring performance and responding to changes in requirements

- **Transaction Management**

- A *transaction* is a collection of operations that performs a single logical function in a database application
  - **E.g. transfer funds from one account to another**
- Transaction-management component ensures that the database remains in a consistent state despite system failures
- Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.
  - **E.g. simultaneous withdrawals**

• ACID Properties:
A - Atomicity / Accessing the Data C - Concurrency Access I - Integrity Problems / Inconsistency D - Data Redundancy

UNIT - I
ENTITY-RELATIONSHIP MODEL (Part -II)

Topics :

- ✓ Entity Sets
- ✓ Relationship Sets
- ✓ Mapping Constraints
- ✓ Keys
- ✓ E-R Diagram
- ✓ Extended E-R Features
- ✓ Design of an E-R Database Schema
- ✓ Reduction of an E-R Schema to Tables

■ Entity Sets

- A *database* can be modeled as:
  - a collection of entities,
  - Relationship among entities.
- An *entity* is an object that exists and is distinguishable from other objects.
  - E.g. specific person, company, event, plant
- Entities have *attributes*
  - E.g: people have *names* and *addresses*
- An *entity set* is a set of entities of the same type that share the same properties.
  - Example: set of all persons, companies, courses, books

- **Entity Sets *customer* and *loan***

321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

customer

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

loan

- **Attributes**

An entity is represented by a set of attributes that is descriptive properties possessed by all members of an entity set.

Domain – the set of permitted values for each attribute

Attribute types:

***Simple* and *composite* attributes.**

Simple Attribute

***Single-valued* and *multi-valued* attributes**

E.g. multi valued attribute: *phone-numbers*

Multi valued Attribute

***Derived* attributes**

Can be computed from other attributes

E.g. *age*, given date of birth

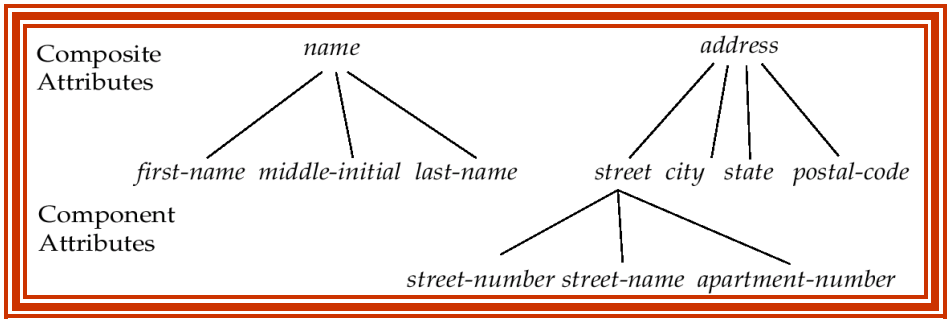
Derived attributes

Key Attribute

Represents primary key. (main characteristics of an entity). It is an attribute, that has distinct value for each entity/element in an entity set. For example, Roll number in a Student Entity Type.

Key Attribute

- **Composite Attributes**



Relationship and Relationship Set :

Relationships connect the entities and represent meaningful dependencies between them. It represents an association among several entities.

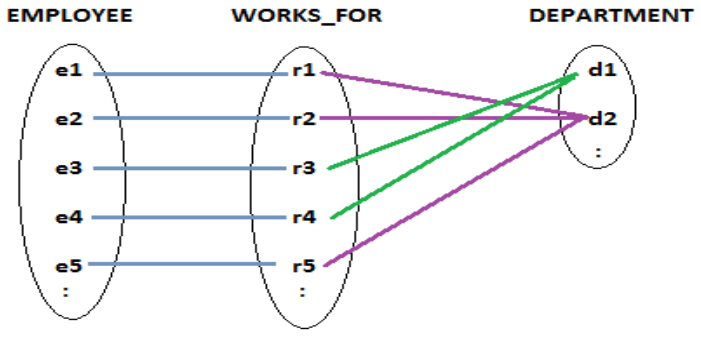
Relationships sets is a set of relationships of the same type. It is a mathematical relation on entity sets (n>=2). Relationship set R is a subset of –

$\{(r1,r2,r3,....r_n) | r1 \in E1, r2 \in E2, r_n \in E_n\}$

where r1,r2,….r_n are called relationships and E1,E2,….E_n are entity sets.

The way in which two or more entity types are related is called **relation type.**

For example, consider a relationship type **WORKS_FOR** between the two entity types EMPLOYEE and DEPARTMENT, which associates or links each employee with the department the employee works for. The WORKS_FOR relation type is shown as –



In the above figure, each instance of relation type WORKS_FOR i.e.(r1, r2,…,r5) is connected to instances of employee and department entities. Employee e1, e2 and e5 work for department d2 and employee e3 and e4 work for department d1.

Notation to Represent Relation Type in ER Diagram-

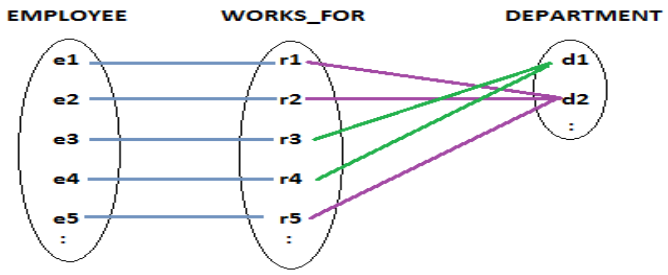
Relation types are represented as diamond shaped boxes.

Degree of a Relationship Type-

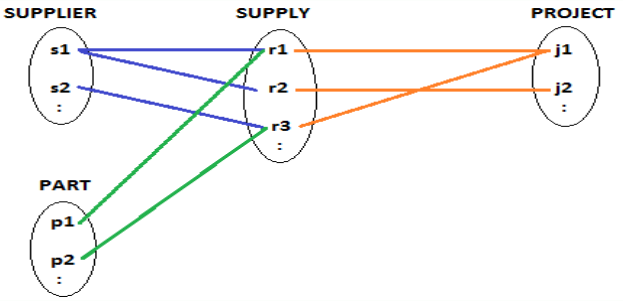
The number of participating entity types is known as the degree of relationship type.

Types of Relationship Type Based on Degree –

- **Binary Relationship** – A relationship type of degree two is called binary relationship. The WORKS_FOR in above figure is a binary relationship as there are two participating entities- employee and department.

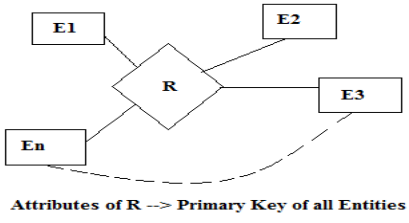


- **Ternary Relationship**- A relationship type of degree three is a ternary relationship for example, in the below figure **supply** relationship connects three entities SUPPLIER, PART AND PROJECT.



The above diagram can be read as – a supplier supplies the parts to projects

- **N-ary Relationship Set –** A relationship type of degree n is called n ary relationship . For example



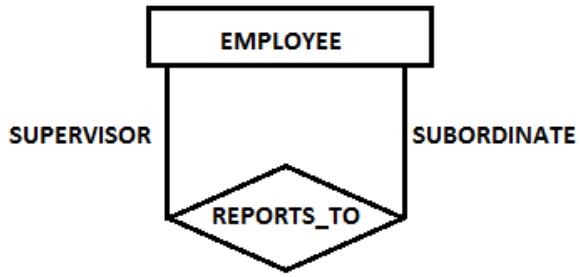
Role Names-

A relationship type has a name which signifies what role a participating entity plays in that relationship instance. The role names helps to explain what the relationship means.

In the first example WORKS_FOR relationship type, employee plays the role of worker and department plays the role of employee(because a department consists of a number of employees.

Recursive Relationship

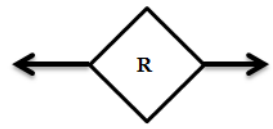
If the same entity type participate more than once in a relationship type in different roles then such relationship types are called recursive relationship. For example, in the below figure REPORTS_TO is a recursive relationship as the Employee entity type plays two roles – 1) Supervisor and 2) Subordinate.



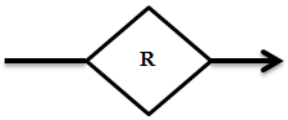
- **Mapping Cardinalities**
  - Express the number of entities to which another entity can be associated via a relationship set.
  - Most useful in describing binary relationship sets.
  - For a binary relationship set the mapping cardinality must be one of the following types:

    - One-to-One Cardinality (1:1)
    - One-to-Many Cardinality (1:m)
    - Many-to-One Cardinality (m:1)
    - Many-to-Many Cardinality (m:n)

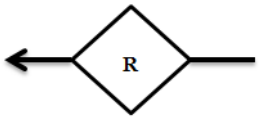
- **Notations of Different Types of Cardinality In ER Diagram –**



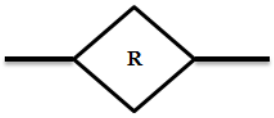
ONE-TO-ONE CARDINALITY



MANY-TO-ONE CARDINALITY



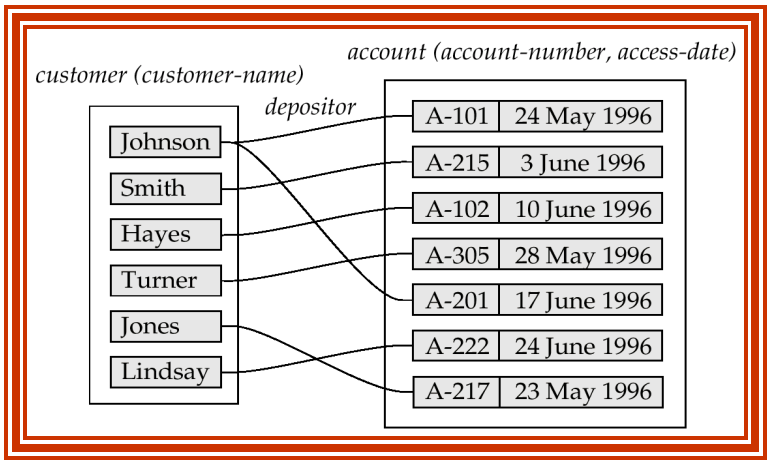
ONE-TO-MANY CARDINALITY



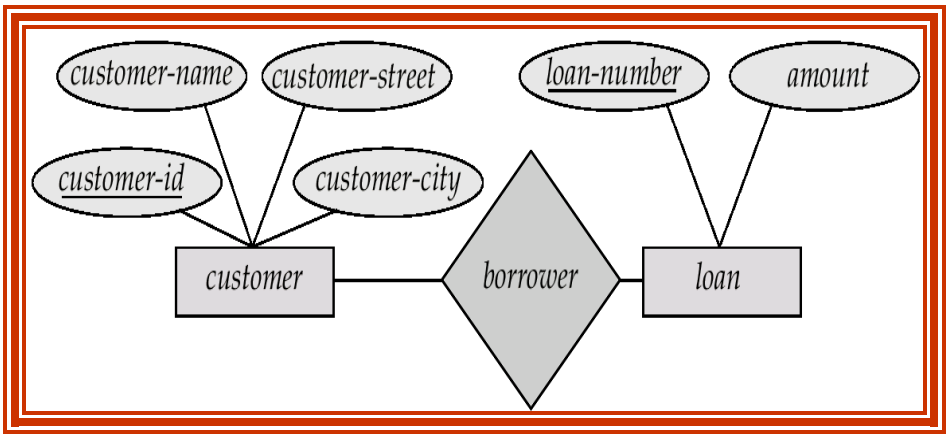
MANY-TO-MANY CARDINALITY

- **Mapping Cardinalities affect ER Design**

- Can make *access-date* an attribute of account, instead of a relationship attribute, if each account can have only one customer
  - I.e., the relationship from account to customer is many to one,



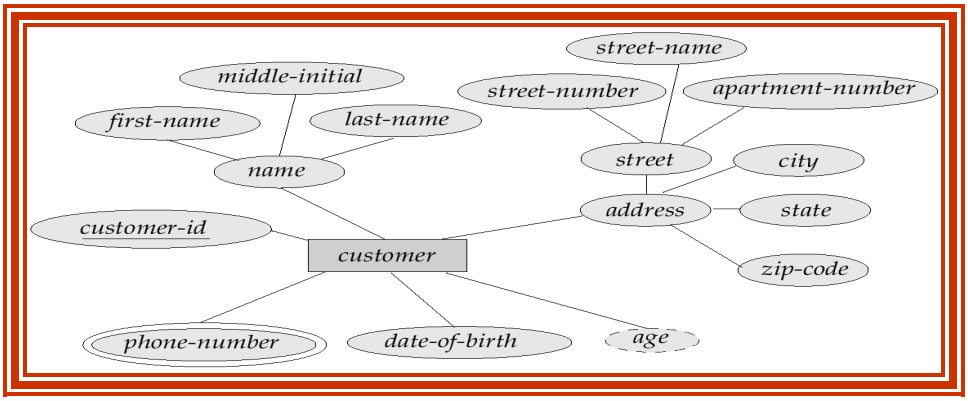
- **E-R Diagrams**



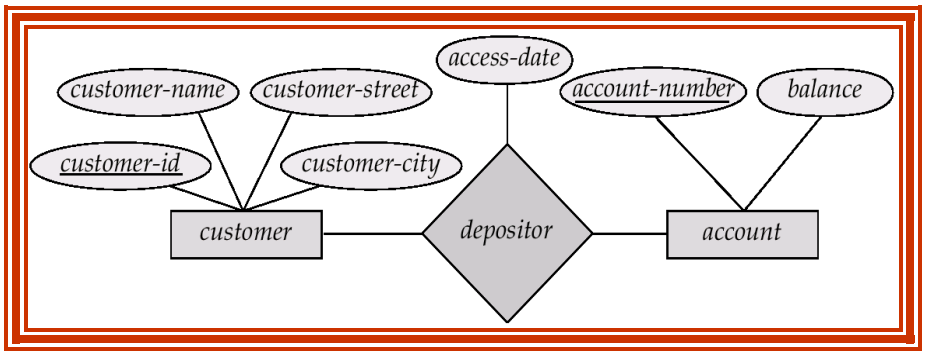
  - **Rectangles** represent entity sets.
  - **Diamonds** represent relationship sets.
  - **Lines** link attributes to entity sets and entity sets to relationship sets.
  - **Ellipses** represent attributes
    - **Double ellipses** represent multivalued attributes.
    - **Dashed ellipses** denote derived attributes.
  - **Underline** indicates primary key attributes

- **E-R Diagram with Composite, Multi valued, and Derived Attributes**

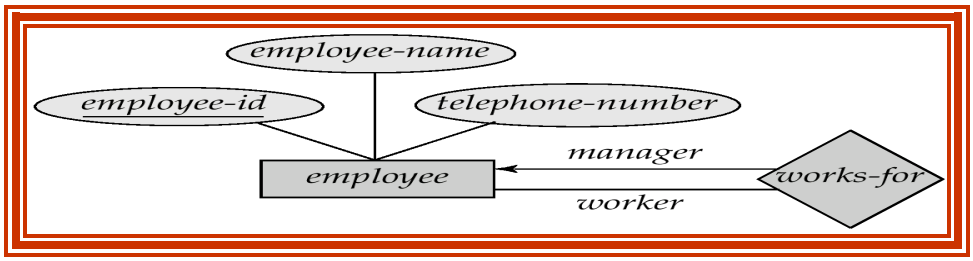
  - **Composite attributes:** The attributes that can be divided into subparts are known as composite attributes. Ex: name can be divided into first name, middle name and last name.
  - **Multi valued attributes:** The attributes that have many values for a particular entity. Ex: name. There can be more than one name for customer.
  - **Derived attribute:** The value for this type of attribute can be derived from the values of other related attributes or entities.



- **Relationship Sets with Attributes**

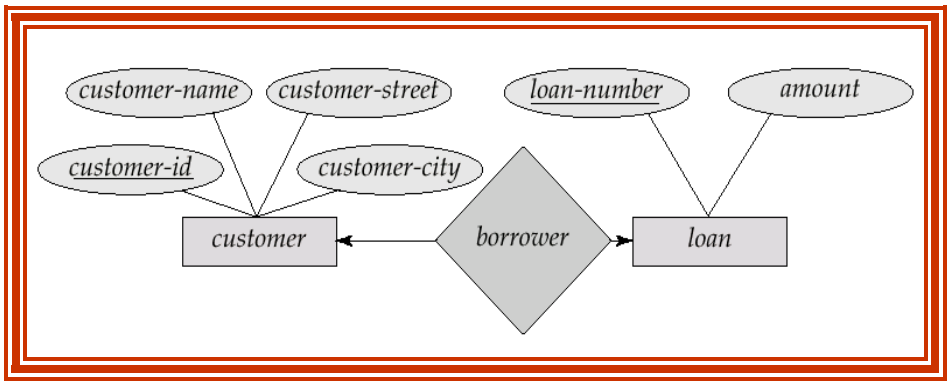


- **Roles**
  - Entity sets of a relationship need not be distinct
  - The labels "manager" and "worker" are called roles; they specify how employee entities interact via the works-for relationship set.
  - Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
  - Role labels are optional, and are used to clarify semantics of the relationship.



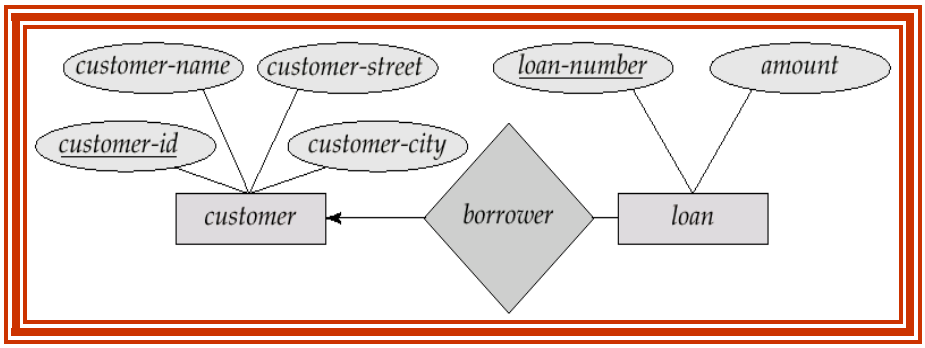
■ Cardinality Constraints

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



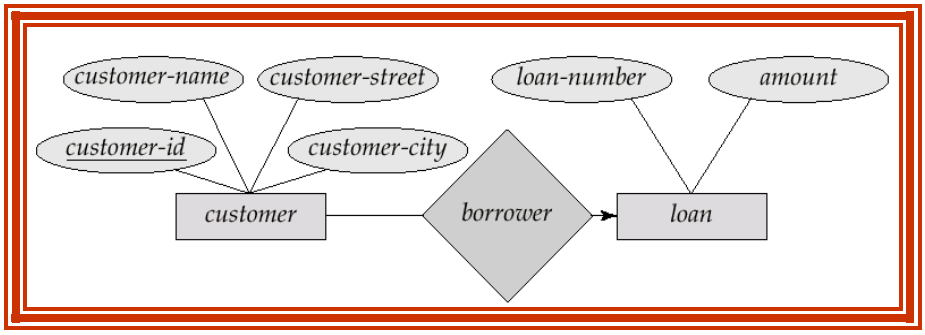
♦ One-To-Many Relationship

- In the one-to-many relationship a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*

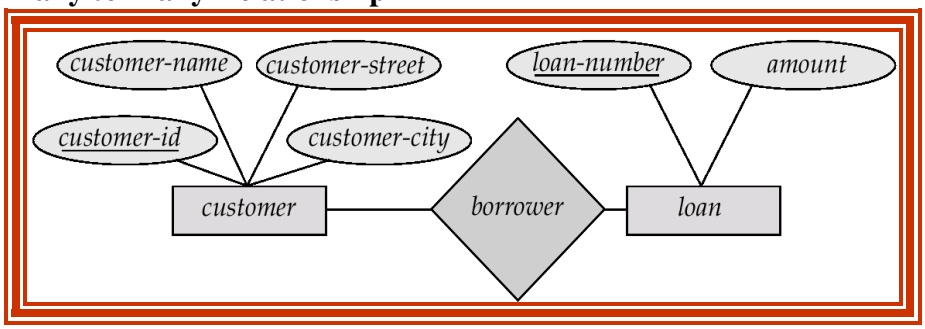


♦ Many to One Relationship

- In a many-to-one relationship a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower*



♦ **Many to Many Relationship**

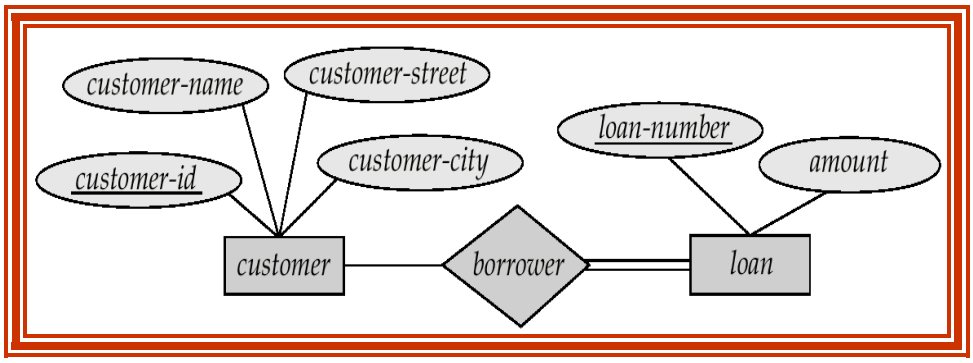


- A customer is associated with several (possibly 0) loans via borrower
- A loan is associated with several (possibly 0) customers via borrower

♦ **Participation of an Entity Set in a Relationship Set**

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - E.g. participation of *loan* in *borrower* is total
    - Every loan must have a customer associated to it via borrower.

- Partial participation: Some entities may not participate in any relationship in the relationship set.
  - E.g. participation of *customer* in *borrower* is partial



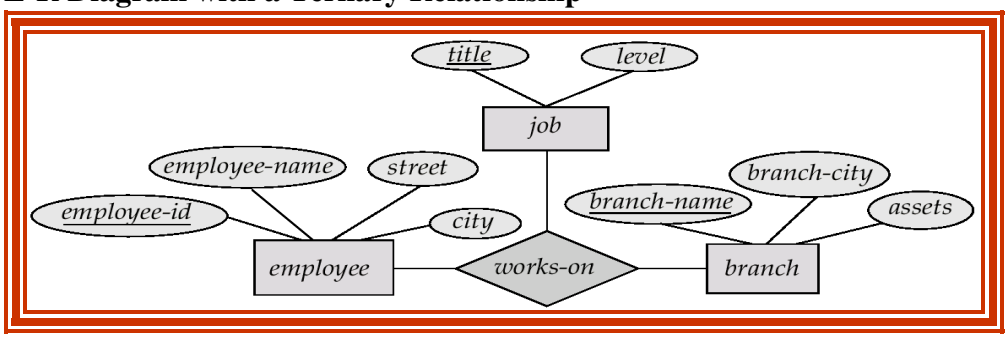
■ **Keys**

- A *super key* of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- A *candidate key* of an entity set is a minimal super key
  - ***Customer-id* is candidate key of *customer***
  - ***account-number* is candidate key of *account***
- Although several candidate keys may exist, one of the candidate keys is selected to be the *primary key*.

♦ **Keys for Relationship Sets**

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - **(*customer-id, account-number*) is the super key of *depositor***
  - ***NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.***
    - E.g. if we wish to track all access-dates to each account by each customer, we cannot assume a relationship for each access. We can use a multi valued attribute though
- Must consider the mapping cardinality of the relationship set when deciding the what are the candidate keys
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key

▪ **E-R Diagram with a Ternary Relationship**



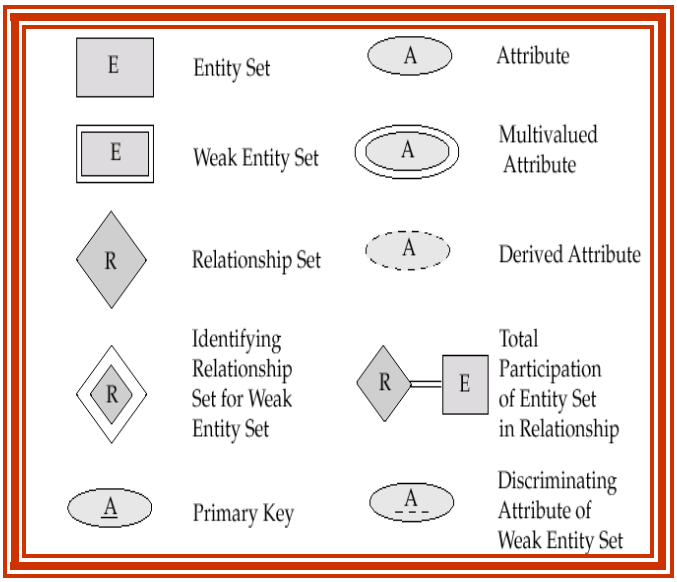
▪ **Cardinality Constraints on Ternary Relationship**

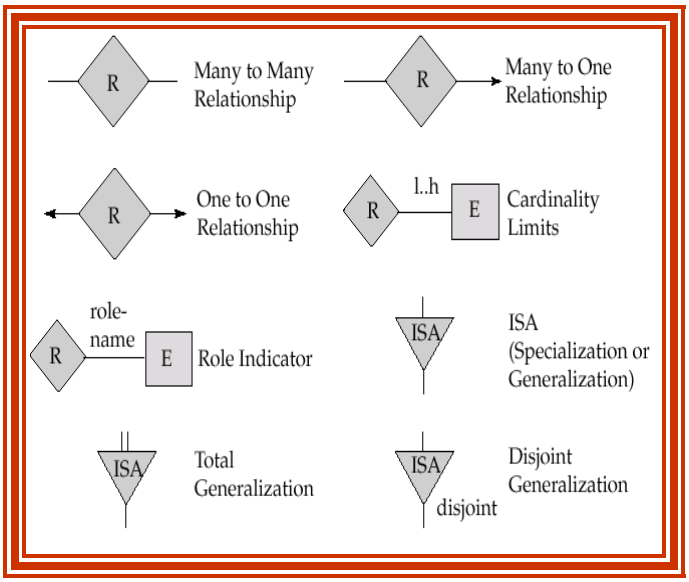
- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g. an arrow from *works-on* to *job* indicates each employee works on at most one job at any branch.
- If there is more than one arrow, there are two ways of defining the meaning.
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

■ **Design Issues**

- *Use of entity sets vs. attributes* : Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.
- *Use of entity sets vs. relationship sets*: Possible guideline is to designate a relationship set to describe an action that occurs between entities.
- *Binary versus n-ary relationship sets* : Although it is possible to replace any nonbinary (*n*-ary, for *n* > 2) relationship set by a number of distinct binary relationship sets, a *n*-ary relationship set shows more clearly that several entities participate in a single relationship.
- *Placement of relationship attributes*

♦ **Summary of Symbols Used in E-R Notation**





■ **Extended E-R Features**

- Weak entity sets
- Specialization
- Generalization
- Aggregation

♦ **Weak Entity Sets**

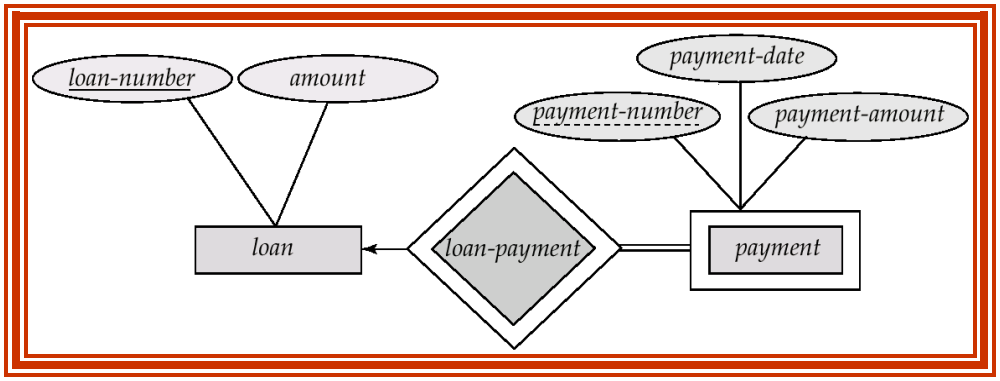
Assumption: entity sets always have a key

- ***This is not always true***

- Examples:
  - ***Dependents covered by an employee's insurance policy***
  - ***Film crews working at a movie studio***
  - ***Species within a genus***
- Properties
  - ***Weak entity set lacks a key***
  - ***Existence of weak entities depends on existence of corresponding entities in the "identifying entity set"***
    - i.e. the participation of the weak entity in the database is only by virtue of its relationship to the identifying entity
    - E.g. we're not interested in film crews except insofar as they are associated with a movie studio (an idiosyncratic property of our enterprise).
- Definition: An entity set that does not have a primary key

- The existence of a weak entity set depends on
  - ***the existence of a identifying entity set***
  - ***must relate to the identifying entity set via a total, many-to-one relationship set***
  - ***Identifying relationship depicted using a double diamond***

  - We depict a weak entity set by double rectangles.
  - We underline the discriminator of a weak entity set with a dashed line.
  - payment-number – discriminator of the payment entity set
  - Primary key for payment – (loan-number, payment-number)



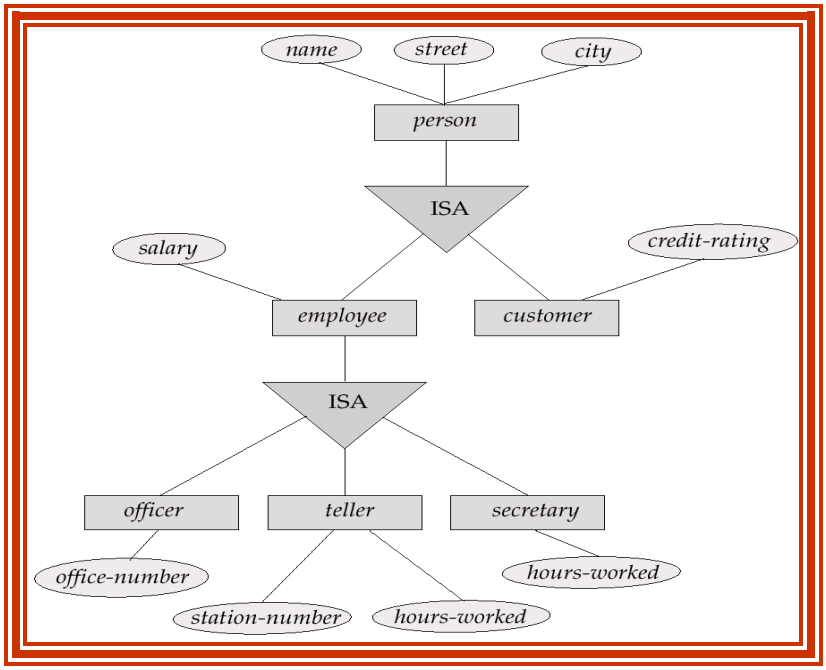
- Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- If *loan-number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan-number* common to *payment* and *loan.*

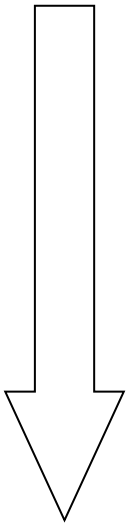
■ Specialization

Top-down design process
Start with few entity sets having many attributes
E.g. person entity may have attributes suitable for students, lecturers, employees, employers, etc.

we identify distinctive sub-groupings within an entity set These sub-groupings become lower-level entity sets
They have attributes or participate in relationships that do not apply to the higher-level entity set
Depicted by a *triangle* component labeled ISA
E.g. *customer* "is a" *person*
Inheritance
a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

Specialization Example





■ Generalization

- A bottom-up design process
  - **start with lots of distinct entities that share attributes**
  - **Combine a number of entity sets that share the same attributes into a higher-level entity set.**
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.

♦ Specialization and Generalization

- Can have multiple specializations of an entity set based on different features.
- E.g. *permanent-employee* vs. *temporary-employee*, in addition to *officer* vs. *secretary* vs. *teller.*
- Each particular employee would be
  - **a member of one of *permanent-employee* or *temporary-employee*,**
  - **and also a member of one of *officer*, *secretary*, or *teller***

- The ISA relationship also referred to as **super-class - subclass** relationship.

♦ Design Constraints on a Specialization/Generalization

- Constraint on which entities can be members of a given lower-level entity set.
  - **condition-defined**
    - E.g. all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
  - **user-defined**

- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**
    - an entity can belong to only one lower-level entity set
    - write *disjoint* next to the ISA triangle
  - **Overlapping**
    - an entity can belong to more than one lower-level entity set

- Completeness constraint
  - ***Does an entity in the higher-level entity set have to belong to at least one of the lower-level entity sets?***

- Total
  - **an entity must belong to one of the lower-level entity sets**

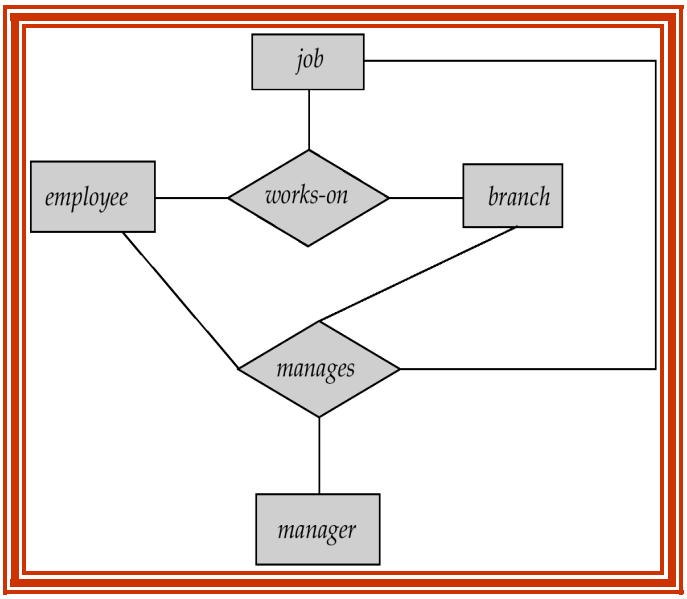
- Partial
  - **an entity need not belong to one of the lower-level entity sets**

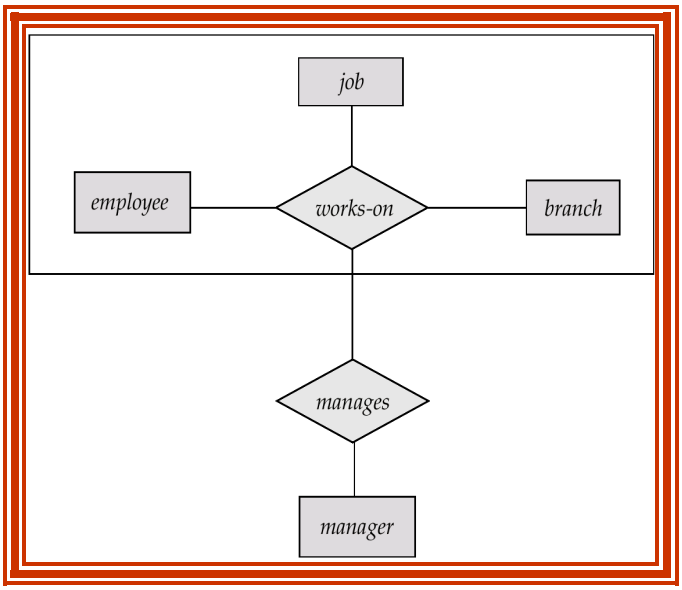
■ Aggregation

- Consider the ternary relationship works-on
- Suppose we want to record managers for tasks performed by an employee at a branch.
- *works-on* and *manages* represent overlapping information
  - **Every *manages* relationship corresponds to a *works-on* relationship**
  - **some *works-on* relationships may not correspond to any *manages* relationships**
  - **we can't discard the *works-on* relationship**

- Eliminate this redundancy via *aggregation*
  - **Treat *works-on* relationship as an abstract entity**
  - **Allow relationships between relationships!**
- *Abstraction of relationship into new entity*
- Without introducing redundancy, the following diagram represents:
  - **An employee works on a particular job at a particular branch**
  - **An employee, branch, job combination may have an associated manager**

▪ **E-R Diagram with Aggregation**

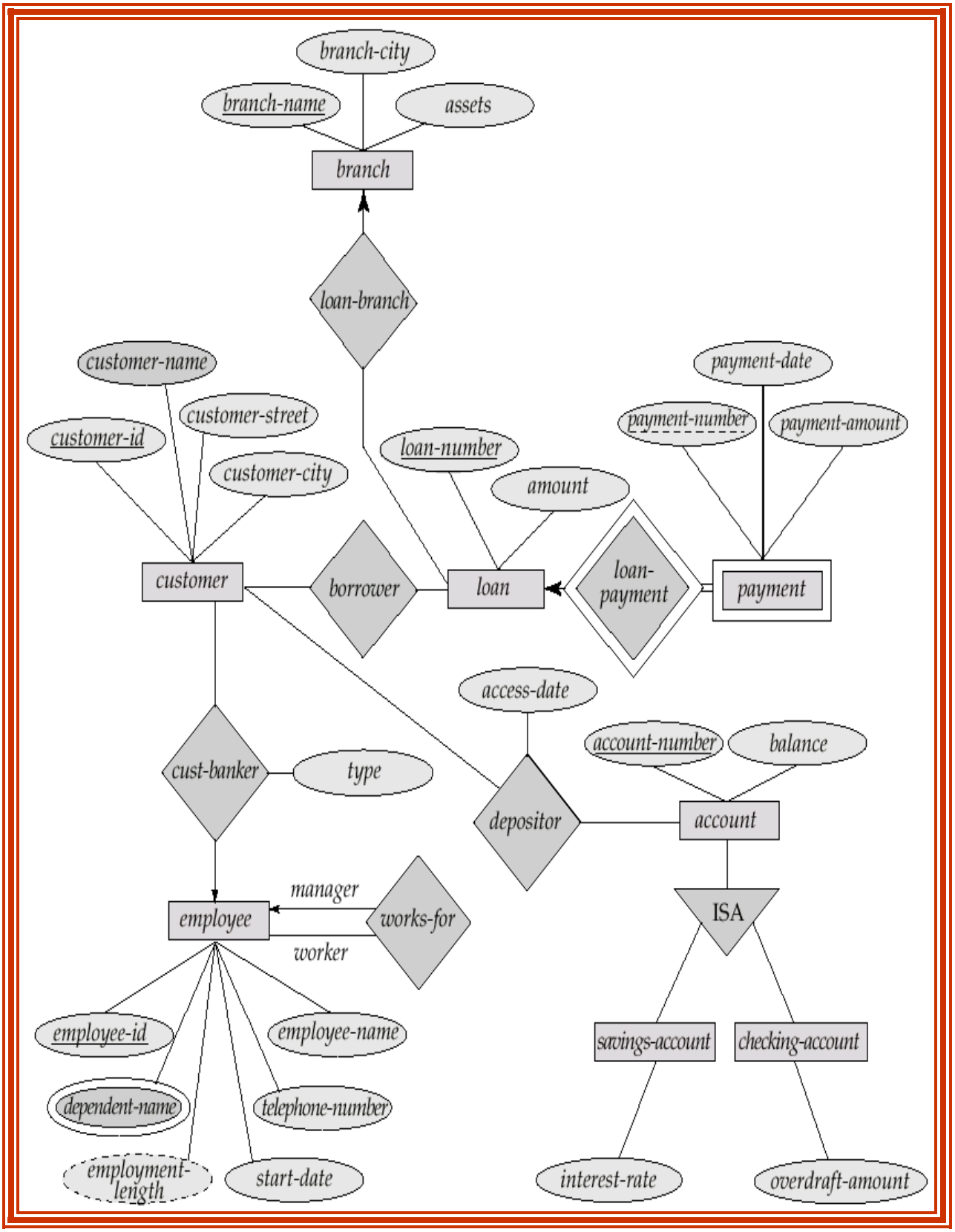




■ **E-R Design Principles**

- Faithfulness
  - **Entities, attributes and relationships should reflect reality**
  - **Sometimes the correct approach is not obvious**
    - E.g. course and instructor entities and teaching relationship
    - What are the cardinality constraints? It depends…
- Avoiding Redundancy
  - **No information should be repeated**
    - Wastes space, leads to consistency problems
- Simplicity
  - **Some relationships may be unnecessary**
    - E.g. student member-of student-body attends course vs student attends course
- Choosing the right kind of element
  - **The use of an attribute or entity set to represent an object**
  - **Whether a real-world concept is best expressed by an entity set or a relationship set**
- Choosing the right relationships
  - The use of a ternary relationship versus a pair of binary relationships
  - The use of a strong or weak entity set.
  - The use of specialization/generalization – contributes to modularity in the design.
  - The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.

▪ **E-R Diagram for a Banking Enterprise**



■ Reduction of an E-R Schema to Tables

- Primary keys allow entity sets and relationship sets to be expressed uniformly as *tables* which represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of tables.
- For each entity set and relationship set there is a unique table which is assigned the name of the corresponding entity set or relationship set.
- Each table has a number of columns (generally corresponding to attributes), which have unique names.

- Converting an E-R diagram to a table format is the basis for deriving a relational database design from an E-R diagram.

- **Representing Entity Sets as Tables**

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

customer-id	*customer-name*	*customer-street*	*customer-city*
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

- **Composite and Multivalued Attributes**

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - **E.g. given entity set *customer* with composite attribute *name* with component attributes *first-name* and *last-name* the table corresponding to the entity set has two attributes *name.first-name* and *name.last-name***

- A multivalued attribute M of an entity E is represented by a separate table EM
  - **Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M**
  - **E.g. Multivalued attribute *dependent-names* of *employee* is represented by a table *employee-dependent-names*(*employee-id, dname*)**
  - **Each value of the multivalued attribute maps to a separate row of the table EM**
    - E.g., an employee entity with primary key John and dependents Johnson and Johndotir maps to two rows: (John, Johnson) and (John, Johndotir)

- **Representing Weak Entity Sets**

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

loan-number	*payment-number*	*payment-date*	*payment-amount*
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

♦ **Representing Relationship Sets as**

- A many-to-many relationship set is represented as a table with columns for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- E.g.: table for relationship set *borrower*

customer-id	*loan-number*
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

1.13 Conceptual Design with ER Model

- Developing an ER diagram presents several design issues, including the following:
  - Entity versus Attribute.
  - Entity versus Relationship
  - Binary versus Ternary Relationships.
  - Aggregation versus Ternary Relationships.

■ Entity versus Attribute

- While identifying the attributes of an entity set, it is sometimes not clear, whether a property should be modeled as an attribute or as an entity set.
- Example: consider the entity set employee with attributes employees name and telephone number. It can easily be said that a telephone is an entity in its own right with attributes telephone number and location. If we take this point of views, the employee entity set must be redefined as follows:
  - The employee entity set with attribute employee name.
  - The telephone entity set with attributes telephone number and location.
  - The relationship set employee telephone, which denotes the association between employees and the telephones that they have.
- The main difference between these two definitions of an employee is as follows:
  - In the first case, the definition implies that every employee has one telephone number associated with him.
  - In the second case, the definition implies that all employees may have several telephone number associated with them.
- Thus, the second definition is more general than the first one, and may more accurately reflect the real world situation. Even if we are given that each employee has only one telephone number associated with him, the second definition may still be more appropriate because the telephone is shared among several employees.
- However, it is appropriate to have employees-name as an attribute of the employee entity set instead of an entity because most of the employees have single name.

■ Entity versus Relationship

- It is not always clear whether an object is best expressed by an entity set or a relationship set.

- Example: assume that, a bank loan is modeled as an entity. An alternative is to model a loan not as an entity, but rather as a relationship between customers and branches, with loan number and amount as descriptive attributes. Each loan is represented by a relationship between a customer and a branch.

- If every loan is held by exactly one customer and customer is associated with exactly one branch, we may find satisfactory the design, where a loan is represented as a relationship. But, with this design, we cannot represent conveniently a situated in which several customers hold a loan jointly. We must define a separate relationship for each holder of the joint loan. Then, we must replicate the values for the descriptive attributes loan-number and amount in each such relationship. Each such relationship must of course, have the same value for the descriptive attributes loan number and amount.

- Two problems arise as a result of the replication:

    - The data are stored multiple times, wasting storage space.
    - Updates leave the data in an inconsistent state.

- One possible guideline is determining whether to use an entity set or a relationship set to designate a relationship set, an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

■ **Binary versus ternary Relationships**

- It is always possible to replace a non-binary (n-ary, for n>2) relationship set by a number of distinct binary relationship sets.

- Example: for simplicity, consider the abstract ternary (n=3) relationship set R, relating entity sets A, B, C. We replace the relationship set R by an entity set E, and create three relationship sets:

    * R_A, relating E and A
    * R_B, relating E and B
    * R_C, relating E and C

- If the relationship set R has any attributes, these are assigned to entity set E; otherwise, a special identifying attribute is created for E. For each relationship (a_i, b_i, c_i) in the relationship set R, we create a new entity e; in the entity set E.
- Then, in each of the three new relationship sets, we insert a relationship as follows:
    * (e_i, a_i) in R_A
    * (e_i, b_i) in R_B
    * (e_i, c_i) in R_C

- We can generalize this process in a straight forward manner to n-ary relationship sets. Thus, conceptually, we can restrict the E-R model to include only binary relationship sets.

■ **Aggregation versus Ternary Relationships**

- The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set. The choice may also be guided by certain integrity constraints that we want to express.

- Example: consider the constraint that each sponsorship be monitored by at most one employee. We can express this constraint in terms of the sponsors relationship set. On the other hand, we can easily express the constraint by drawing an arrow from the aggregated relationship sponsors to the relationship

Monitors. Thus, the presence of such a constraint servers as another reason for using aggregation rather than a ternary relationship set.

1.14 Conceptual Design for Large Database

- Designing database for large organization takes efforts of more than a single designer. It diagrammatically represents the complete database and enables the user who provides inputs to database, to understand the complete functionality of database.

- Large databases are modeled in two methodologies.

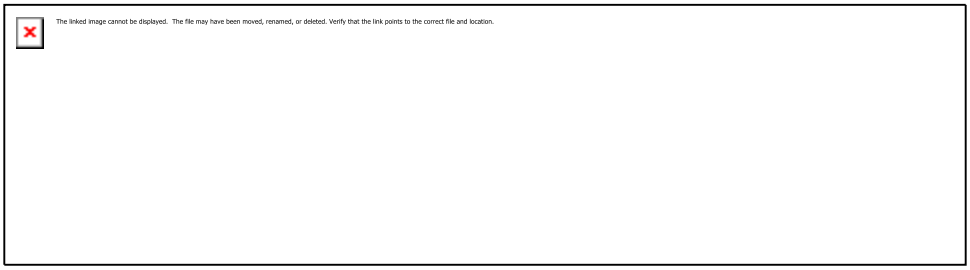
    - The requirements of all the users are collected. The conflicting requirements are resolved and a final conceptual view is generated to satisfy the requirements of all users.

    - In the other method, the user provides his requirements; the designer generates a conceptual view for the requirements. Likewise all the conceptual views from all user requirements are generated and a comprehensive conceptual view that satisfies all the requirements is generated.

CASE STUDY

How to Draw ER Diagram ??

We have read all the basic terms of E-R Diagram. Now, let's understand how to draw E-R diagram? In ER Model, objects of similar structures are collected into an **entity set**. The relationships between an entity sets is represented by a named **E-R relationship**, which may be **(one-to-one, one-to-many, many-to-one, many-to-many)**, which maps one entity set to another entity set. A General ER Diagram is shown as-



In Figure, there are two entities **ENTITY-1** and **ENTITY-2** having attributes **(Atr_{11}, Atr_{12}, ... Atr_{1m})** and **(Atr_{21}, Atr_{22}, ... Atr_{2n})** respectively, connected via many to many relationship **(M:N).** The attributes of **RELATIONSHIP** are **(Atr_{R1}, Atr_{R2}, ... Atr_{RO}).**

Steps - How to Draw ER Diagram -

1. Identify all the entities of the given problem
2. Identify all the attributes of the entities identified in step 1.
3. Identify the Primary Keys of entities identified in Step 1.
4. Identify the Attribute Types of attributes identified in step 2
5. Identify relationship between the entities and constraints on the entities and implement them.

Need of ER Diagram -

The ER Diagrams are useful in representing the relationship among entities. It helps to show basic data structures in a way that different people can understand. Many types of people are involved in the database environment, including programmers, designers, managers and end users. But not all of these people work with database and might not be as skilled as others to understand the making of a software or a program etc, so, a conceptual model like the ERD helps show the design to many different people in a way they can all understand.

Example of drawing ER Diagram -

How to draw E-R diagram of a company database if the following requirements are given : Question : Make an ER Diagram for the company database with the following description :

1. **The company is organized into departments. Each department has a unique name and a unique number. A department may have several locations.**
2. **A department controls a number of projects, each of which has a unique name, a unique number and a single location.**
3. **We store each employee's name, social security number, address and salary. An employee is assigned to one department but may work on several projects, which are not necessarily controlled by the same departments.**
4. **We want to keep track of the departments of each employee for insurance purposes. We keep each dependent's name, age and relationship to the employee.**

Answer :

Step 1 : Identifies Entities of the given problem.

Entities :

1. DEPARTMENT (From 1st Point)
2. PROJECT (From 2nd Point)
3. EMPLOYEE (From 3rd Point)
4. DEPENDENT (From 4th Point)

Step 2 : Identify the attributes of the above entities.

Attributes :

1. DEPARTMENT : Name, Number, Location;
2. PROJECT : Name, Number, Location;
3. EMPLOYEE : SSN, Name, Address, Salary
4. DEPENDENT : Name, Age, Relationship

Step 3 : Identify the Primary Keys of all entities identified in Step 1.

Primary Keys :

1. DEPARTMENT : Name, Number, Location; (Unique Name and Unique Number)
2. PROJECT : Name, Number, Location; (Unique Name and Unique Number)
3. EMPLOYEE : SSN, Name, Address, Salary (Since, Social Security Number will be Unique, so SSN is selected as primary Key)
4. DEPENDENT : Name, Age, Relationship (No Unique Attribute to identify DEPENDENT entity. So, It is referred as a Weak Entity)

Step 4: Identify the Attribute Types -

Attributes Types :

1. Location Attribute of DEPARTMENT entity :
   Multi valued Attribute (Since there are several locations)
2. Name Attribute of EMPLOYEE entity :
   Composite Attribute (since a name consists of first name, middle name and last name)
3. Address Attribute of EMPLOYEE entity :
   Composite Attribute (Address consists of H.no, Street, City, State, Country)

Step 5 : Identify the Relationships and relationships attributes between the entities.

Relationships and their Attributes :

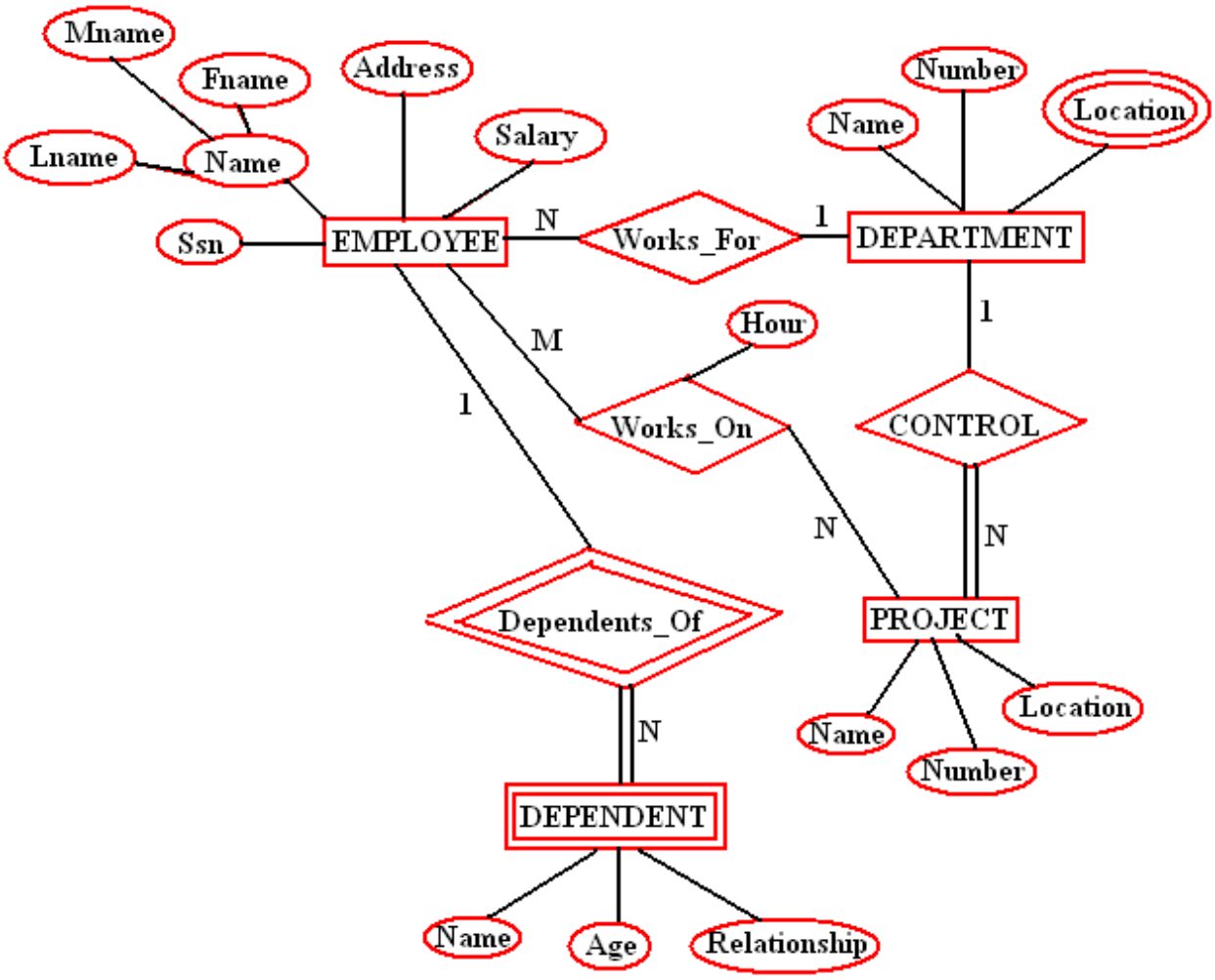
Relationship Name	*Entities Name having relationships among them*	*Attributes*
1. Works_For	EMPLOYEE and DEPARTMENT	-
2. Control	DEPARTMENT and PROJECT	-
3. Works_On	EMPLOYEE and PROJECT	Hours
4. Dependents_Of	EMPLOYEE and DEPENDENT	-

Step 6: Identify the constraints on the entities.

Cardinality Constraints :

Relationship	*Cardinality*	*Reason*
1. Works_For	N:1	Since N employees Works_For a Department.
2. Works_On	M:N	Since different employees works on different projects.
3. Control	1:N	Since a department Controls a number of projects.
4. Dependents_Of	1:N	Since each Dependents has name, age and relationship to the employee.

Implementation of ER Diagram of the given problem on the basis of above steps :



Relational Algebra

Chapter 4, Part A

Relational Query Languages

- *Query languages*: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages != programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:

1. *Relational Algebra*: More operational, very useful for representing execution plans.
2. *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)

☛ *Understanding Algebra & Calculus is key to*
☛ *understanding SQL, query processing!*

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Relational Algebra

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$\pi_{sname,rating}(S2)$

age
35.0
55.5

$\pi_{age}(S2)$

Selection

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$\sigma_{rating>8}(S2)$

sname	rating
yuppy	9
rusty	10

$\pi_{sname,rating}(\sigma_{rating>8}(S2))$

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

☛ *Renaming operator*: $\rho\,(C(1 \to sid1, 5 \to sid2), S1 \times R1)$

Joins

- *Condition Join*: $R \bowtie_c S = \sigma_c(R \times S)$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$S1 \bowtie_{S1.sid < R1.sid} R1$

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join.*

Joins

- *Equi-Join*: A special case of condition join where the condition *c* contains only ***equalities.***

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$S1 \bowtie_{sid} R1$

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equijoin on *all* common fields.

Division

- Not supported as a primitive operator, but useful for expressing queries like:
  *Find sailors who have reserved **all** boats.*
- Let A have 2 fields, x and y; B have only field y:
  - $A/B = \{\langle x\rangle \mid \exists \langle x, y\rangle \in A \ \forall \langle y\rangle \in B\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of y values (boats) associated with an x value (sailor) in A contains all y values in B, the x value is in A/B.
- In general, x and y can be any lists of fields; y is the list of fields in B, and $x \cup y$ is the list of fields of A.

Examples of Division A/B

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

A

pno
p2

B1

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

A/B2

pno
p1
p2
p4

B3

sno
s1

A/B3

Expressing A/B Using Basic Operators

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea*: For A/B, compute all x values that are not `disqualified' by some y value in B.
  - x value is *disqualified* if by attaching y value from B, we obtain an xy tuple that is not in A.

Disqualified x values: $\pi_x((\pi_x(A) \times B) - A)$

A/B: $\pi_x(A)$ − all disqualified tuples

Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103}\text{Reserves}) \rtimes Sailors)$
- Solution 2: $\rho(Temp1, \sigma_{bid=103}\text{Reserves})$

  $\rho(Temp2, Temp1 \rtimes Sailors)$

  $\pi_{sname}(Temp2)$
- Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \rtimes Sailors))$

Find names of sailors who've reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

$\pi_{sname}((\sigma_{color='red'}, Boats) \rtimes \text{Reserves} \rtimes Sailors)$

- A more efficient solution:

$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}, Boats) \rtimes \text{Res}) \rtimes Sailors)$

☛ *A query optimizer can find this given the first solution!*

Find sailors who've reserved a red or a green boat

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

$\rho(Tempboats, (\sigma_{color='red' \vee color='green'}, Boats))$

$\pi_{sname}(Tempboats \rtimes \text{Reserves} \rtimes Sailors)$

- Can also define Tempboats using union! (How?)
- What happens if $\vee$ is replaced by $\wedge$ in this query?

Find sailors who've reserved a red and a green boat

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\,(Tempred, \pi_{sid}((\sigma_{color='red'}, Boats) \times Reserves))$$

$$\rho\,(Tempgreen, \pi_{sid}((\sigma_{color='green'}, Boats) \times Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \times Sailors)$$

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\,(Tempsids, (\pi_{sid,bid}Reserves) / (\pi_{bid}Boats))$$

$$\pi_{sname}(Tempsids \times Sailors)$$

- To find sailors who've reserved all 'Interlake' boats:

$$..... / \pi_{bid}(\sigma_{bname='Interlake'}, Boats)$$

Summary

- The relational model has rigorously defined query languages that are simple and powerful.
- Relational algebra is more operational; useful as internal representation for query evaluation plans.
- Several ways of expressing a given query; a query optimizer should choose the most efficient version.

Relational Calculus

Chapter 4, Part B

Relational Calculus

- Comes in two flavours: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
- Calculus has *variables*, *constants*, *comparison ops*, *logical connectives* and *quantifiers*.
  - TRC: Variables range over (i.e., get bound to) *tuples*.
  - DRC: Variables range over *domain elements* (= field values).
  - Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Domain Relational Calculus

- *Query* has the form:

$$\{\langle x1,x2,...,xn\rangle \mid p(\langle x1,x2,...,xn\rangle)\}$$

- *Answer* includes all tuples $\langle x1,x2,...,xn\rangle$ that make the *formula* $p(\langle x1,x2,...,xn\rangle)$ be *true*.
- Formula is recursively defined, starting with simple *atomic formulas* (getting tuples from relations or making comparisons of values), and building bigger and better formulas using the *logical connectives*.

DRC Formulas

- *Atomic formula:*
  - $\langle x1,x2,...,xn\rangle \in$ ***Rname*** , or X *op* Y, or X *op* constant
  - *op* is one of $<,>,=,\leq,\geq,\neq$
- *Formula:*
  - an atomic formula, or
  - $\neg p, p \wedge q, p \vee q$, where p and q are formulas, or
  - $\exists X(p(X))$, where variable X is *free* in p(X), or
  - $\forall X(p(X))$, where variable X is *free* in p(X)
- The use of quantifiers $\exists X$ and $\forall X$ is said to bind X.
  - A variable that is not bound is free.

Free and Bound Variables

- The use of quantifiers $\exists X$ and $\forall X$ in a formula is said to bind X.
  - A variable that is not bound is free.
- Let us revisit the definition of a query:

$$\{\langle x1,x2,...,xn\rangle \mid p(\langle x1,x2,...,xn\rangle)\}$$

- There is an important restriction: the variables x1, ..., xn that appear to the left of `|' must be the *only* free variables in the formula p(...).

Find all sailors with a rating above 7

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7\}$$

- The condition $\langle I,N,T,A\rangle \in Sailors$ ensures that the domain variables *I*, *N*, *T* and *A* are bound to fields of the same Sailors tuple.
- The term $\langle I,N,T,A\rangle$ to the left of `|' (which should be read as *such that*) says that every tuple $\langle I,N,T,A\rangle$ that satisfies *T*>7 is in the answer.
- Modify this query to answer:
  - Find sailors who are older than 18 or have a rating under 9, and are called 'Joe'.

Find sailors rated > 7 who've reserved boat #103

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7 \wedge \exists\, Ir,Br,D\ (\langle Ir,Br,D\rangle \in \text{Re}serves \wedge Ir=I \wedge Br=103)\}$$

- We have used $\exists\, Ir, Br, D\ (\ldots)$ as a shorthand for $\exists\, Ir\,(\exists\, Br\,(\exists\, D\,(\ldots)))$
- Note the use of $\exists$ to find a tuple in Reserves that `joins with' the Sailors tuple under consideration.

Find sailors rated > 7 who've reserved a red boat

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T>7 \wedge \exists\, Ir,Br,D\,(\langle Ir,Br,D\rangle \in \text{Re}serves \wedge Ir=I \wedge \exists\, B,BN,C\,(\langle B,BN,C\rangle \in Boats \wedge B=Br \wedge C=\text{'}red\text{'}))\}$$

- Observe how the parentheses control the scope of each quantifier's binding.
- This may look cumbersome, but with a good user interface, it is very intuitive. (Wait for QBE!)

Find sailors who've reserved all boats

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\, B,BN,C\,(\neg(\langle B,BN,C\rangle \in Boats) \vee (\exists\, Ir,Br,D\ (\langle Ir,Br,D\rangle \in \text{Re}serves \wedge I=Ir \wedge Br=B)))\}$$

- Find all sailors I such that for each 3-tuple $\langle B,BN,C\rangle$ either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor I has reserved it.

Find sailors who've reserved all boats (again!)

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\, \langle B,BN,C\rangle \in Boats\ (\exists\langle Ir,Br,D\rangle \in \text{Re}serves(I=Ir \wedge Br=B))\}$$

- Simpler notation, same query. (Much clearer!)
- To find sailors who've reserved all red boats:

..... $(C \neq \text{'}red\text{'} \vee \exists\langle Ir,Br,D\rangle \in \text{Re}serves(I=Ir \wedge Br=B))\}$

Unsafe Queries, Expressive Power

- It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called *unsafe*.
  - e.g., $\{S \mid \neg(S \in Sailors)\}$
- It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.
- *Relational Completeness*: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

Summary

- Relational calculus is non-operational, and users define queries in terms of what they want, not in terms of how to compute it. (Declarativeness.)
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance 53 of Sailors

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance *R2* of Reserves

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance *B1* of Boats

1. Find the' names and ages of all sailors.

SELECT DISTINCT S.sname, S.age FROM Sailors S

2. Find all sailors with a rating above 7.

SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

3. Find the names of sailors 'Who have reserved boat number 103.

SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103

(Or)

SELECT Sname FROM Sailors, Reserves WHERE Sailors.sid = Reserves.sid AND bid=103

4.Find the sids of sa'iloTs who have TeseTved a Ted boat.

SELECT R.sid FROM Boats B, Reserves R WHERE B.bid = R.bid AND 8.color = 'red'

5. Find the names of sailors Who have Reserved a Red boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE
S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

6. Find the colors of boats reserved by Lubber.

SELECT B.color FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'

7. Find the names of sailors who have reserved at least one boat.

SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid

UNION, INTERSECT, AND EXCEPT

8. Find the names of sailors who have reserved a red or a green boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE
S.sid = R.sid AND R.bid = B.bid AND (B.color = 'red' OR B.color = 'green')

(Or)

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid
AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid =
R2.sid AND R2.bid = B2.bid AND B2.color = 'green'

9. Find the names of sailors who have reserved both a red and a green boat.

SELECT S.sname FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats
B2 WHERE S.sid = Rl.sid AND R1.bid = Bl.bid AND S.sid = R2.sid AND
R2.bid =B2.bid AND B1.color='red' AND B2.color = 'green'

(or)

```
SELECT S.snarne FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid
AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

.

(Q 19) Find the sids of all sailor's who have reserved red boats but not green boats.

```
SELECT S.sid FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND
R.bid = B.bid AND B.color = 'red'

EXCEPT

SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE
S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Nested Queries

A nested query is a query that has another query embedded within it; the embedded query is called a suhquery. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible.

1. Find the names of sailors who have reserved boat 103.

```
SELECT S.sname FROM Sailors S WHERE S.sid IN ( SELECT  R.sid FROM
Reserves R WHERE R.bid = 103 )
```

2. Find the names of sailors who have reserved a red boat.

```
SELECT S.sname FROM Sailors S WHERE S.sid  IN
        ( SELECT R.sid FROM Reserves R WHERE R. bid IN
                (SELECT B.bid FROM Boats B WHERE B.color = 'red'))
```

3. Find the names of sailors who have not reserved a red boat.

```
SELECT S.sname FROM Sailors S WHERE S.sid NOT IN
         ( SELECT R.sid  FROM Reserves R WHERE R.bid IN
                ( SELECT B.bid FROM Boats B WHERE B.color = 'red' ))
```

Correlated Nested Queries

In nested query subquery is executed only once but in correlated nested query sub query is executed as many number of times as many rows are there in relation of main query.

Q.Find the names of sailors who have reserved boat number 103.

SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid = 103
AND R.sid = S.sid)

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each Sailor row 5, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty.

Set-Comparison Operators

set-comparison operators are EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}.

AGGREGATE OPERATORS

SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:
1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

NULL VALUES

SQL provides a special column value called *null* to use in situations when the column value is either unknown or inapplicable.
Eg:- Suppose the Sailor table definition was modified to include a rnaiden-name column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the rmaiden_name colun are inapplicable.

Comparisons Using Null Values

An issue in the presence of *'null* values is the definition of when two rows in a relation instance are regarded as *duplicates.* The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null.* Contradiction to this definition with the fact that if

we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

SQL provides a special comparison operator ISNULL to fint out null value for a column.

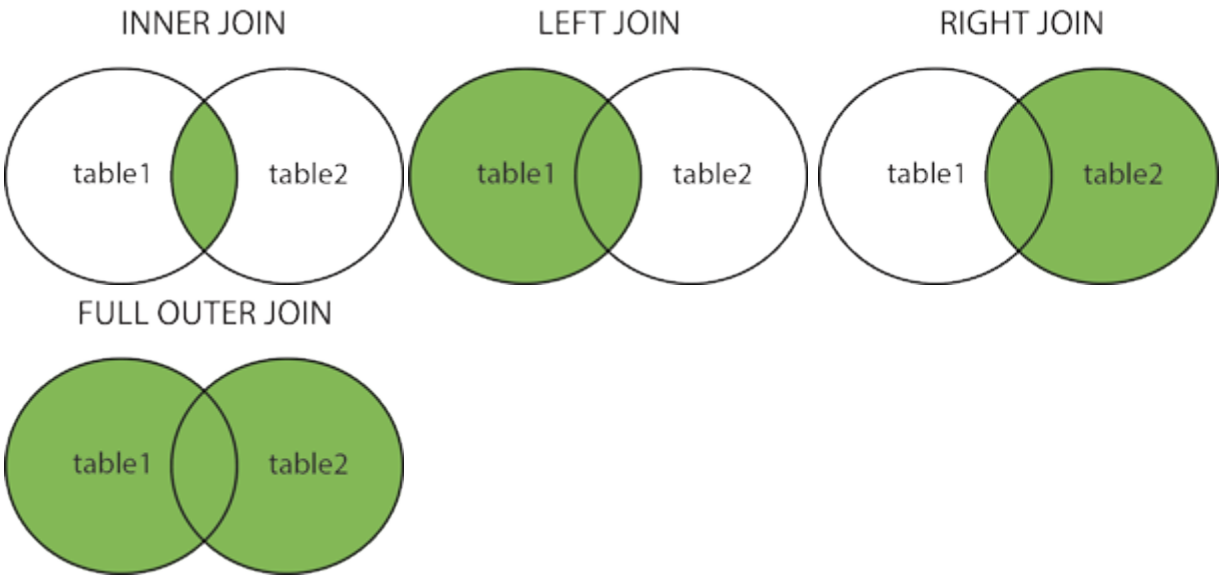
Disallowing Null Values

We can disallow *null* values by specifying NOT NULL as part of the field definition; for example, *sname* CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on *null* values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

JOINS

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table



Employee table

LastName	DepartmentID
Rafferty	31
Jones	33
Heisenberg	33
Robinson	34
Smith	34
Williams	NULL

Department table

DepartmentID	DepartmentName
31	Sales
33	Engineering
34	Clerical
35	Marketing

Left outer join

The result of a *left outer join* (or simply **left join**) for tables A and B always contains all rows of the "left" table (A), even if the join-condition does not find any matching row in the "right" table (B). This means that if the `ON` clause matches 0 (zero) rows in B (for a given row in A), the join will still return a row in the result (for that row)—but with NULL in each column from B. A **left outer join** returns all the values from an inner join plus all values in the left table that do not match to the right table, including rows with NULL (empty) values in the link column.

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Jones	33	Engineering	33
Rafferty	31	Sales	31
Robinson	34	Clerical	34
Smith	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33

Right outer join

A **right outer join** (or **right join**) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those rows that have no match in B.

A right outer join returns all the values from the right table and matched values from the left table

(NULL in the case of no matching join predicate). For example, this allows us to find each employee and his or her department, but still show departments that have no employees.

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

Full outer join[edit]

Conceptually, a **full outer join** combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those rows that do match, a single row will be produced in the result set (containing columns populated from both tables).

For example, this allows us to see each employee who is in a department and each department that has an employee, but also see each employee who is not part of a department and each department which doesn't have an employee.

Employee.LastName	Employee.DepartmentID	Department.DepartmentName	Department.DepartmentID
Smith	34	Clerical	34
Jones	33	Engineering	33
Robinson	34	Clerical	34
Williams	NULL	NULL	NULL
Heisenberg	33	Engineering	33
Rafferty	31	Sales	31
NULL	NULL	*Marketing*	*35*

UNIT – 3

Normalisation or Schema Refinement or Database design

- Normalisation or Schema Refinement is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
- The Schema Refinement refers to refine the schema by using some technique. The best technique of schema refinement is decomposition.
- The Basic **Goal of Normalisation** is used to **eliminate redundancy**.
- Redundancy refers to repetition of same data or duplicate copies of same data stored in different locations.

Normalization is used for mainly two purpose :

- Eliminating redundant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Anomalies or Problems Facing without Normalisation :

Anomalies refers to the problems occurred after poorly planned and unnormalised databases where all the data is stored in one table which is sometimes called a flat file database. Let us consider such type of schema –

SID	Sname	CID	Cname	FEE
S1	A	C1	C	5k
S2	A	C1	C	5k
S1	A	C2	C++	10k
S3	B	C2	C++	10k
S3	B	C3	JAVA	15k
Primary Key(SID,CID)				

Here all the data is stored in a single table which causes redundancy of data or say anomalies as SID and Sname are repeated once for same CID . Let us discuss anomalies one bye one.

Types of Anomalies : (Problems because of Redundancy)

There are three types of Anomalies produced in the database because of redundancy –

- **Updation/Modification Anomaly**
- **Insertion Anomaly**
- **Deletion Anomaly**

1. **Problem in updation / updation anomaly –** If there is updation in the fee from 5000 to 7000, then we have to update **FEE** column in all the rows, else data will become inconsistent.

SID	Sname	CID	Cname	FEE
S1	A	C1	C	~~5k~~ 7k
S2	A	C1	C	~~5k~~ 7k
S1	A	C2	C	10k
S3	B	C2	C	10k
S3	B	C2	JAVA	15k

Costly Operation

More IO Cost

2. **Insertion Anomaly and Deleteion Anomaly-** These anamolies exist only due to redundancy, otherwise they do not exist.

- **Insertion Anomaly :**

New course is introduced C4, But no student is there who is having C4 subject.

SID	Sname	CID	Cname	FEE
S1	A	C1	C	5k
S2	A	C1	C	5k
S1	A	C2	C	10k
S3	B	C2	C	10k
S3	B	C2	JAVA	15k

NULL	NULL	CA	DB	12k

To Insert that Row, It is Required to Put Dummy Data..

Therefore,

xx	xx	CA	DB	12k

Because of insertion of some data, It is forced to insert some other dummy data.

3.

- **Deletion Anomaly :**

Deletion of S3 student cause the deletion of course.

Because of deletion of some data forced to delete some other useful data.

SID	Sname	CID	Cname	FEE
S1	A	C1	C	5k
S2	A	C1	C	5k
S1	A	C2	C	10k
~~S3~~	~~B~~	~~C2~~	~~C~~	~~10k~~
~~S3~~	~~B~~	~~C2~~	~~JAVA~~	~~15k~~

Deleting student S3 will permanently delete the course B.

Solutions To Anomalies : Decomposition of Tables – Schema Refinement

SID	Sname	CID	Cname	FEE
S1	A	C1	C	5k
S2	A	C1	C	5k
S1	A	C2	C	10k
S3	B	C2	C	10k
S3	B	C2	JAVA	15k

SID	Sname	CID
S1	A	C1
S2	A	C1
S1	A	C2
~~S3~~	~~B~~	~~C2~~
~~S3~~	~~B~~	~~C3~~

PK(SID,CID)

Deletion Anamoly Removed

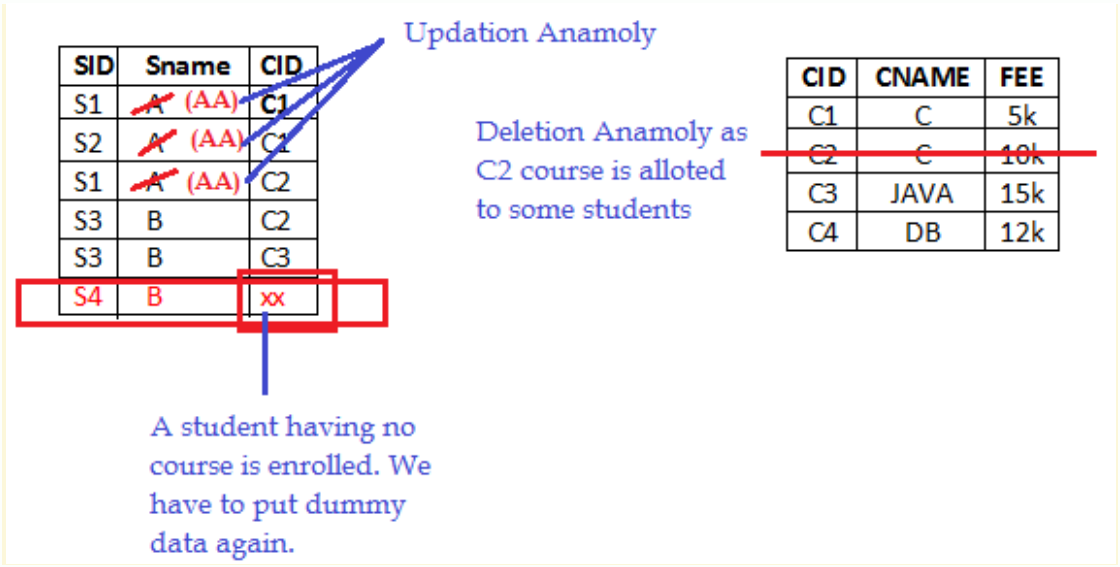
CID	CNAME	FEE
C1	C	~~5k~~ 7k
C2	C	10k
C3	JAVA	15k
C4	DB	12k

(Updation Anamoly Removed

Insertion Anamoly Removed

PK(CID)

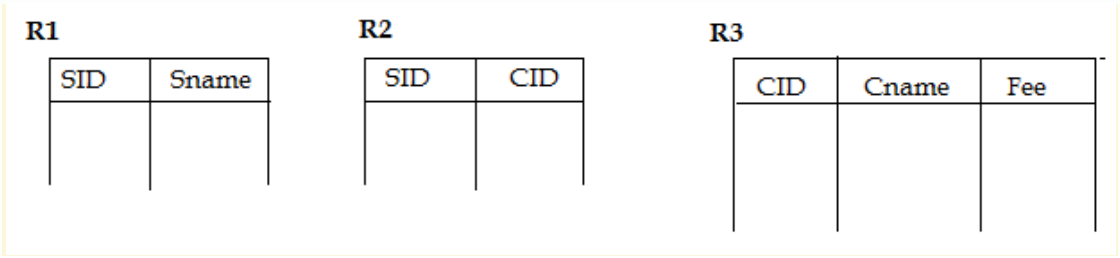
There are some Anomalies in this again –



SID	Sname	CID
S1	A (AA)	C1
S2	A (AA)	C1
S1	A (AA)	C2
S3	B	C2
S3	B	C3
S4	B	xx

CID	CNAME	FEE
C1	C	5k
C2	C	10k
C3	JAVA	15k
C4	DB	12k

What is the Solution ??
Solution :



Functional dependency in DBMS

The 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.

For example: Suppose we have a student table with attributes: Stu_Id, Stu_Name, Stu_Age. Here Stu_Id attribute uniquely identifies the Stu_Name attribute of student table because if we know the student id we can tell the student name associated with it. This is known as functional dependency and can be written as Stu_Id->Stu_Name or in words we can say Stu_Name is functionally dependent on Stu_Id.

Formally:
If column A of a table uniquely identifies the column B of same table then it can represented as A->B (Attribute B is functionally dependent on attribute A)

Types of Functional Dependencies

- **Trivial functional dependency**
- **non-trivial functional dependency**
- **Multivalued dependency**
- **Transitive dependency**

Trivial functional dependency

The dependency of an attribute on a set of attributes is known as trivial functional dependency if the set of attributes includes that attribute.

Symbolically: A ->B is trivial functional dependency if B is a subset of A.

The following dependencies are also trivial: A->A & B->B

For example: Consider a table with two columns Student_id and Student_Name.

{Student_Id, Student_Name} -> Student_Id is a trivial functional dependency as Student_Id is a subset of {Student_Id, Student_Name}. That makes sense because if we know the values of Student_Id and Student_Name then the value of Student_Id can be uniquely determined.

Also, Student_Id -> Student_Id & Student_Name -> Student_Name are trivial dependencies too.

Non trivial functional dependency

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.

For example:
An employee table with three 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}]
Refer: trivial functional dependency.

Completely non trivial FD:
If a FD X->Y holds true where X intersection Y is null then this dependency is said to be completely non trivial function dependency.

Multivalued dependency

Multivalued dependency occurs when there are more than one **independent** multivalued attributes in a table.

For example: 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

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

Transitive dependency

A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies. For e.g.

X -> Z is a transitive dependency if the following three functional dependencies hold true:

- X->Y
- Y does not ->X
- Y->Z

Note: A transitive dependency can only occur in a relation of three of more attributes. This dependency helps us normalizing the database in 3NF (3rdNormal Form).

Inference Rules

Armstrong's axioms are a set of axioms (or, more precisely, inference rules) used to infer all the functional dependencies on a relational database. They were developed by William W. Armstrong.
Let R(U) be a relation scheme over the set of attributes U. We will use the letters X, Y, Z to represent any subset of and, for short, the union of two sets of attributes and by instead of the usual X U Y.

- The **Armstrong's axioms** are **very *intuitive***
- **Consider the relation:**

```
Employee-Department

   SSN            fname       lname     DNO        DName
+-------------+----------+----------+----------+----------+
| 111-11-1111 | John     | Smith    |    5     | Research |
+-------------+----------+----------+----------+----------+
| 222-22-2222 | Jane     | Doe      |    4     | Payroll  |
+-------------+----------+----------+----------+----------+
| 333-33-3333 | Pete     | Pan      |    5     | Research |
+-------------+----------+----------+----------+----------+
```

- **Examples of Armostrong axioms:**

1. **Reflexivity rule**: if $Y \subseteq X$ then $X \rightarrow Y$

```
{fname, lname} ⟶ {fname}

  What it says is: if I see that same values for
{fname, lname}
  I must also see that same value for {fname} -
kinda obvious :-)
```

2. **Augmentation rule**: if $X \rightarrow Y$ then $XZ \rightarrow YZ$

```
If {SSN} ⟶ {fname}   then:  {SSN, DName} ⟶
{fname, DName}
```

3. **Transitivity rule**: if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$

```
If:

   {SSN} ⟶ {DNO}

   {DNO} ⟶ {DName}
Then also:

   {SSN} ⟶ {DName}
```

The **Decomposition rule:**

- if $X \rightarrow YZ$ then: $X \rightarrow Y$ and $X \rightarrow Z$

Union rule:

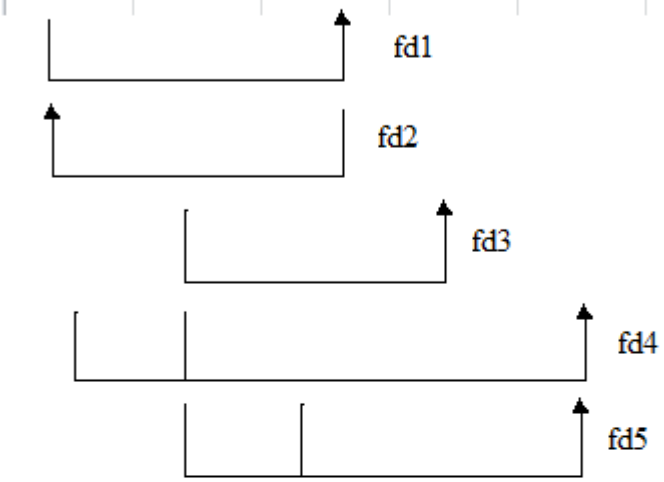
- if $X \rightarrow Y$ and $X \rightarrow Z$ then: $X \rightarrow YZ$

***Psuedo* transitivity rule**:

- if $X \rightarrow Y$ and $YW \rightarrow Z$ then: $XW \rightarrow Z$

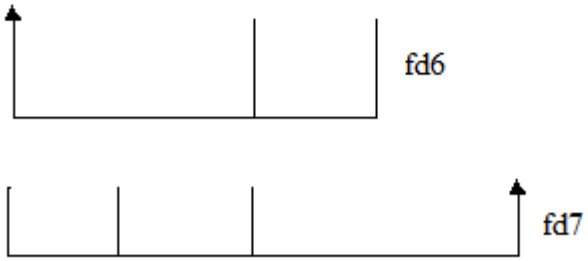
Sample Relation:

A	B	C	D	E
a	b	z	w	q
e	b	r	w	p
a	d	z	w	t
e	d	r	w	q
a	f	z	s	t
e	f	r	s	t



A ⟶ C (fd1)
C ⟶ A (fd2)
B ⟶ D (fd3)
A,B ⟶ E (fd4)
B,C ⟶ E (fd5)

True



C, D ⟶ A (fd6)
A, B, C ⟶ E (fd7)

Not true

How to Find Candidate Key using Functional Dependencies –

In the previous post (How to Find Super Key from Functional Dependencies), we identify all the superkeys using functional dependencies. To identify Candidate Key, Let R be the relational schema, and X be the set of attributes over R. X^+ determine all the attributes of R, and therefore X is said to be superkey of R. If there are no superflous attributes in the Super key, then it will be Candidate Key.
In short, a minimal Super Key is a Candidate Key.

Example/Question 1 : Let R(ABCDE) is a relational schema with following functional dependencies. AB → C DE → B CD → E

Step 1: Identify the SuperKeys –

ACD, ABD, ADE, ABDE, ACDB, ACDE, ACDBE. {From Previous Post Eg.}

Step 2: Find minimal super key -

Neglecting the last four keys as they can be trimmed down, so, checking

the first three keys (ACD, ABD and ADE)

For SuperKey : ACD

$(A)^+$ = {A} - {Not determine all attributes of R}

$(C)^+$ = {C} - {Not determine all attributes of R}

$(D)^+$ = {D} - {Not determine all attributes of R}

For SuperKey : ABD

$(A)^+$ = {A} - {Not determine all attributes of R}

$(B)^+$ = {B} - {Not determine all attributes of R}

$(D)^+$ = {D} - {Not determine all attributes of R}

For SuperKey : ADE

$(A)^+$ = {A} - {Not determine all attributes of R}

$(D)^+$ = {D} - {Not determine all attributes of R}

$(E)^+$ = {E} - {Not determine all attributes of R}

Hence none of proper sets of SuperKeys is not able to determine all attributes of R, So ACD, ABD, ADE all are minimal superkeys or candidate keys.

Example/Question 2 : Let R(ABCDE) is a relational schema with following functional dependencies - AB → C C → D B → EA Find Out the Candidate Key ?

Step 1: Identify the super key

(AB^+) : {ABCDE} ⇒ Superkey

(C^+) : {CD} ⇒ Not a Superkey

(B^+) : {BEACD} ⇒ Superkey

So, Super Keys will be B, AB, BC, BD, BE, BAC, BAD, BAE, BCD, BCE, BDE,

BACD, BACE, BCDE, ABDE, ABCDE

Step 2: Find minimal super key -

Taking the first one key, as all other keys can be trimmed down -

(B^+) : {EABCD} {determine all the attributes of R}

Since B is a minimal SuperKey ⇒ B is a Candidate Key.

So, the Candidate Key of R is - B.

Functional Dependency Set Closure (F+)

Functional Dependency Set Closure of F is the set of all functional dependencies that are determined by it.

Example of Functional Dependency Set Closure :

Consider a relation R(ABC) having following functional dependencies :

F = { A → B, B → C }

To find the Functional Dependency Set closure of F^+ :

$(\Phi)^+ = \{\Phi\}$

$\Rightarrow \Phi \rightarrow \Phi$

$\Rightarrow$ 1 FD

$(A)^+ = \{ABC\}$

$\Rightarrow A \rightarrow \Phi,\quad A \rightarrow A,\quad A \rightarrow B,\quad A \rightarrow C,$

$A \rightarrow BC,\quad A \rightarrow AB,\quad A \rightarrow AC,\quad A \rightarrow ABC$

$\Rightarrow$ 8 FDs = $(2)^3$

... where 3 is number of attributes in closure

$(B)^+ = \{BC\}$

$\Rightarrow\ B \rightarrow \Phi,\ B \rightarrow B,\ B \rightarrow C,\ B \rightarrow BC$

$\Rightarrow$ 4 FDs = $(2)^2$

$(C)^+ = \{C\}$

$\Rightarrow\ C \rightarrow \Phi,\ C \rightarrow C$

$\Rightarrow$ 2 FDs = $(2)^1$

$(AB)^+ = \{ABC\}$

$\Rightarrow\ AB \rightarrow \Phi,\ AB \rightarrow A,\ AB \rightarrow B,\ AB \rightarrow C,$

AB → AB, AB → BC, AB → AC, AB → ABC

⇒ 8 FDs = $(2)^3$

$(BC)^+$ = {BC}

⇒ BC → Φ, BC → B, BC → C, BC → BC

⇒ 4 FDs = $(2)^2$

$(AC)^+$ = {ABC}

⇒ AC → Φ, AC → A, AC → C, AC → C,

AC → AC, AC → AB, AC → BC, AC → ABC

⇒ 8 FDs = $(2)^3$

$(ABC)^+$ = {ABC}

⇒ ABC → Φ, ABC → A, ABC → B, ABC → C,

ABC → BC, ABC → AB, ABC → AC, ABC → ABC

⇒ 8 FDs = $(2)^3$

So, the Functional Dependency Set Closure of $(F)^+$ will be :

F^+ = {

Φ → Φ, A → Φ, A → A, A → B, A → C, A → BC, A → AB, A → AC, A → ABC,

B → Φ, B → B, B → C, B → BC, C → Φ, C → C, AB → Φ, AB → A, AB → B,

AB → C, AB → AB, AB → BC, AB → AC, AB → ABC, BC → Φ, BC → B,

BC → C, BC → BC, AC → Φ, AC → A, AC → C, AC → C, AC → AC, AC → AB,

AC → BC, AC → ABC, ABC → Φ, ABC → A, ABC → B, ABC → C, ABC → BC,

ABC → AB, ABC → AC, ABC → ABC

}

The Total FDs will be :

```
1 + 8 + 4 + 2 + 8 + 4 + 8 + 8 = 43 FDs
```

Consider another relation R(AB) having following functional dependencies :

F = { A → B, B → A }

To find the Functional Dependency Set closure of F^+ :

$(\Phi)^+ = \{\Phi\} \Rightarrow 1$

$(A)^+ = \{AB\} \Rightarrow 4 = (2)^2$

$(B)^+ = \{AB\} \Rightarrow 4 = (2)^2$

$(AB)^+ = \{AB\} \Rightarrow 4 = (2)^2$

Total = 13

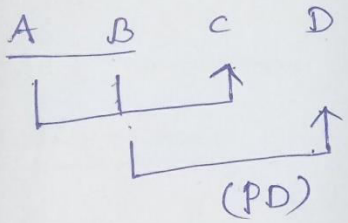
Types of FD

① Partial FD:-

If nonkey attributes depends only on a part of the key then it leads to the "partial dep" & causes redund prob.

eg: R(ABCD)

F = {AB → C, B → D} key: AB

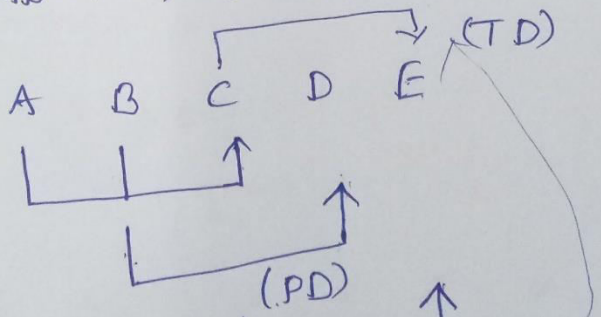


② Transitive dep :-

If there is a FD b/w 2 or more non-key attributes then it is called as "tran dep".

eg: R(ABCDE)

F = { AB → C, B → D, C → E } key: AB



Full func dep :

If FD $X \rightarrow y$ is a full FD, if removal of any attribute 'a' from X, then dep does not holds any more.

(or)

If all non-key attributes are totally depending on key-attribute then it is full func dep.

Eg: R(ABCD)

FD : $AB \rightarrow C$

$B \rightarrow D$

key : AB

$AB \rightarrow C$ is full func dep

$B \rightarrow D$ is pd

First normal form

First normal form (**1NF**) is a property of a relation in a relational database. A relation is in first normal form if and only if the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain.

Designs that Violate 1NF-Below is a table that stores the names and telephone numbers of customers. One requirement though is to retain *multiple* telephone numbers for some customers. The simplest way of satisfying this requirement is to allow the "Telephone Number" column in any given row to contain more than one value:

Customer

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

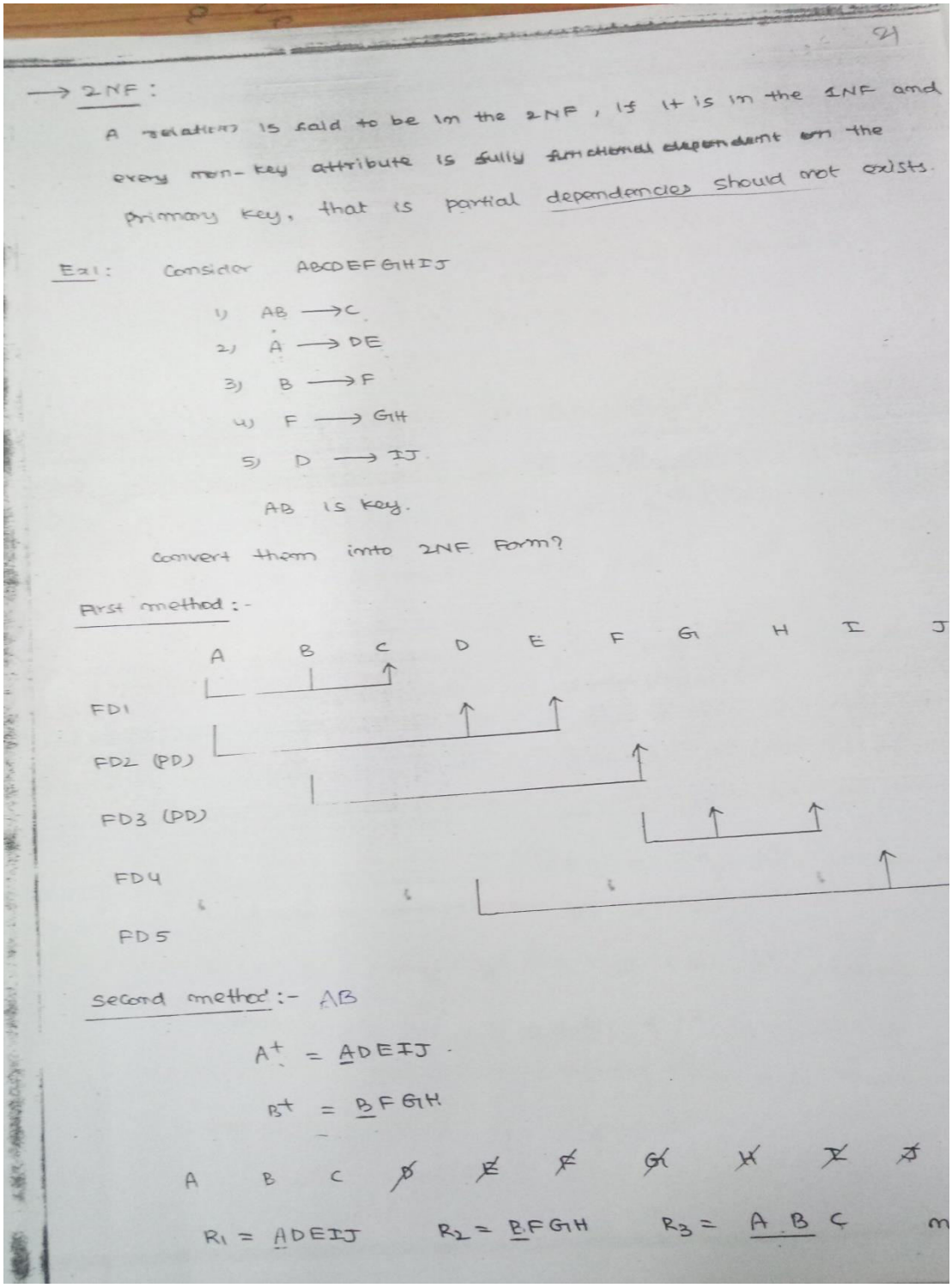
Designs that Comply with 1NF-To bring the model into the first normal form, we split the strings we used to hold our telephone number information into "atomic" (i.e. indivisible) entities: single phone numbers. And we ensure no row contains more than one phone number.

Customer

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

Second normal form

A relation is in 2NF if it is in 1NF and no non-prime attribute is dependent on any proper subset of any candidate key of the relation. **A non-prime attribute of a relation** is an attribute that is not a part of any candidate key of the relation.



If there is a partial dependency then remove all partially dependent attributes and place them in a new relation along with a copy of their determinent.

26)
Ex2:

A B C D E F

FD's $A \longrightarrow FC$

$C \longrightarrow D$

$B \longrightarrow E$

(i) AB is key

(ii) $A^+ = \underline{A}CDF$

$B^+ = \underline{B}E$

(iii) A B ~~C~~ ~~D~~ ~~E~~ ~~F~~

$R_1 = \underline{A}CDF$ $R_2 = \underline{B}E$ $R_3 = \underline{AB}$

27)
Ex3:

A B C D E

FD's $B \longrightarrow E$

$C \longrightarrow D$

$A \longrightarrow B$

(i) Key is AC

(ii) $A^+ = \underline{A}BE$

$C^+ = \underline{C}D$

(iii) A B C ~~D~~ ~~E~~

$R_1 = \underline{A}BE$

$R_2 = \underline{C}D$

$R_3 = \underline{AC}$

28) Ex 4:

AB ⟶ C PD

BD ⟶ EF PD

AD ⟶ GH PD

A ⟶ I PD

H ⟶ J

(i) ABD is the key (we are only find the key)

(ii) AB^+ = ABCI

BD^+ = BDEF

AD^+ = ADGHIJ

A B C D E F G H I J

(iii)

R_1 = AI

R_2 = ABC

R_3 = BDEF

R_4 = ADGHJ

R_5 = ABD

ABCI

BDEF

AFGHIJ

ABD

A^+ : AI ✓

B^+ : B ✗

A B C ✗ ✓

Date 3-09-05

→ 3NF :-

If there is a x such that x ⟶ y it must satisfies the following conditions to allow it in 3NF.

(i) x must be a super key.

(ii) y must be part of the key.

(iii) If x is a subset of the key then it is treated as partial dependency.

(iv) If x is a not a part of the key then it is treated as transit dependency.

Ex1:)

R = ABCDEFGHIJ

(1)

AB → C

A → DE

B → F

F → GH : TD

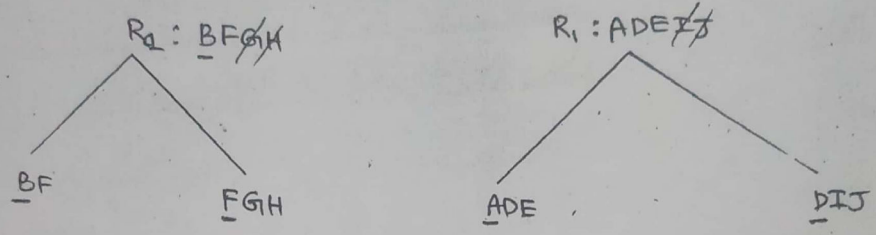
D → IJ : TD

Key is AB.

After 2NF the relations are.

R_1 : ADEIJ R_2 : BFGH R_3 : ABC

R_3 is contain only one non-key attribute. so it is in 3NF.



3NF Relations :

i) ADE
ii) DIJ
iii) FGH
iv) BF
v) ABC

Note :- If there is Transitive Dependent remove dependent attributes and place it in a separate table allowing with a copy of its determinent.

Ex1:)

(2)

R = ABCDEF

A → FC

C → D : TD

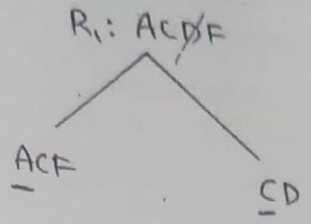
B → E

Key : AB.

After 2NF the relations are.

R_1: ACDF R_2: BE R_3: AB.

**) If table contain 2 attributes it satisfies the 3NF.



3NF Relations are:-

i) CD
ii) ACF
iii) BE
iv) AB.

Ex3:)

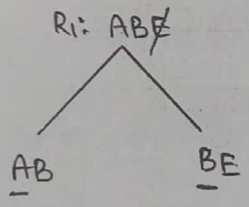
R = ABCDE.

B → E TD
C → D
A → B

Key is AC.

After 2NF the relations are.

R_1: ABE R_2: CD R_3: AC.



3NF Relations are:-

i) AB
ii) BE
iii) CD
iv) AC

Boyce - Codd Normal Form (BCNF) :-

(3.5 NF)

- Raymond. F. Boyce
- Edgar F. Codd

} 1974

If there is a functional dependency

$$\alpha \longrightarrow \beta$$

then α : should be Super key.

Ex: $R(A, B, C)$;

$AB \rightarrow C$ $\quad C \rightarrow B$ $\quad$ BCNF?

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

$AB^+ = \{A, B, C\}$, $AC^+ = \{A, B, C\}$, $ABC^+ = \{A, B,$

Super key : AB, AC, ABC ; Ckeys: AB, AC

1NF

$AB \rightarrow C$	$C \rightarrow B$
X	✓
X	X

2NF (no pd)

= 2NF

3NF (no tr. d)

$AB \rightarrow C$	$C \rightarrow B$
X	x
X	X

= 3NF

BCNF ($\alpha \rightarrow \beta$; α : Super key)

$AB \rightarrow C$	$C \rightarrow B$
✓	X

= Not in BCNF

$R_1(A, C)$ → fk

$R_2(C, B)$ → pk : Decompos

3.38.3
28

Ex: $R(A, B, C, D)$ BCNF?

$A \to BCD$ $BC \to AD$ $D \to B$

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

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

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

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

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

Superkey : ABCD, ABC, AB, BCD, BC, AD, AC, ACD, ..

Candidate key : A, BC, CD

2NF? (no pd)

$A \to BCD$	$BC \to AD$	$D \to B$
x x	x x	x x

Yes 2NF

3NF? (no tr d + no pd)

$A \to BCD$	$BC \to AD$	$D \to B$
x x	x x	✓ x

Yes 3NF

BCNF ($\alpha \to \beta$, α: Superkey)

$A \to BCD$	$BC \to AD$	$D \to B$
✓	✓	x

NO BCNF

$A \rightarrow BCD$ $BC \rightarrow AD$ $D \rightarrow B$ ✗

Convert into BCNF

R1 (A C D) — D: fk

R2 (D, B) — D: pk

Why not?

R1 (A B C) ← ? D — R2 (D, B), $D \rightarrow B$, D: pk.

✗

∴ R1 (A, C, D) R2 (D, B)

= Yes in BCNF.

3NF Decomposition

Ex R(A, B, C, D, E, F, G, H) ACDB ABD

{A → B ABCD → E EF → GH ACDF → EG}

3NF?

Sol) Directly check in 3NF & Solve for 3NF.

not sk / not ck → non key → To D

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

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

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

$ACDF^+ = \{A, B, C, D, E, F, G, H\}$

X 3NF

Perform 3NF Decomposition method.

So, ① Find Minimal Cover / Canonical cover.

(a) LHS → RHS (single Attr)

(b) Remove Redundat Attrb

(c) Remove Redut FD's

{A → B ABCD → E EF → GH ACDF → EG}

Step1

① A → B

② ABCD → E

③ EF → G

④ EF → H

⑤ ACDF → E

⑥ ACDF → G

Step2 (LHS > 1 Attr)

① A → B

② ACD → E

③ EF → G

④ EF → H

⑤ ACDF → E ✗

⑥ ACDF → G ✗

ACDFEGH

Step3 (remo red. A

① A → B

② ACD → E

③ EF → G

④ EF → H

$ACDF^+ = \{A, C, D, F, E\}$

$ACDF^+ = \{A, C, D, F, E, G\}$

$F_c = \{A \to B, ACD \to E, EF \to G, EF \to H\}$.

↓ R1(A, B) ↓ R2(ACDE) ↓ R3(EFG) ↓ R4(EFH)

BCNF Decomposition:

Ex. $R(A, B, C)$ $F = \{A \to B, B \to C\}$. BCNF?

Sol)

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

R(A B C)

$F = \{A \to B, B \to C\}$.

CK = A

BCNF? $(B \to C)$ x violates.

BCNF Decomposition

$B \to C$

$R_1(B, C)$

$F_1 = \{B \to C\}$

$CK = \{B\}$.

BCNF? True.

$R_2(A, B)$

$F_2 = \{A \to B\}$.

CK : A

BCNF? T

$R(A, B, C, D, E)$, $F = \{A \to B, BC \to D\}$.

BCNF?

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

$F = \{A \to B, BC \to D\}$.

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

CK = ACE

BCNF? $A \to B.X$ $BC \to D.X$ Pseudo Transit rule.

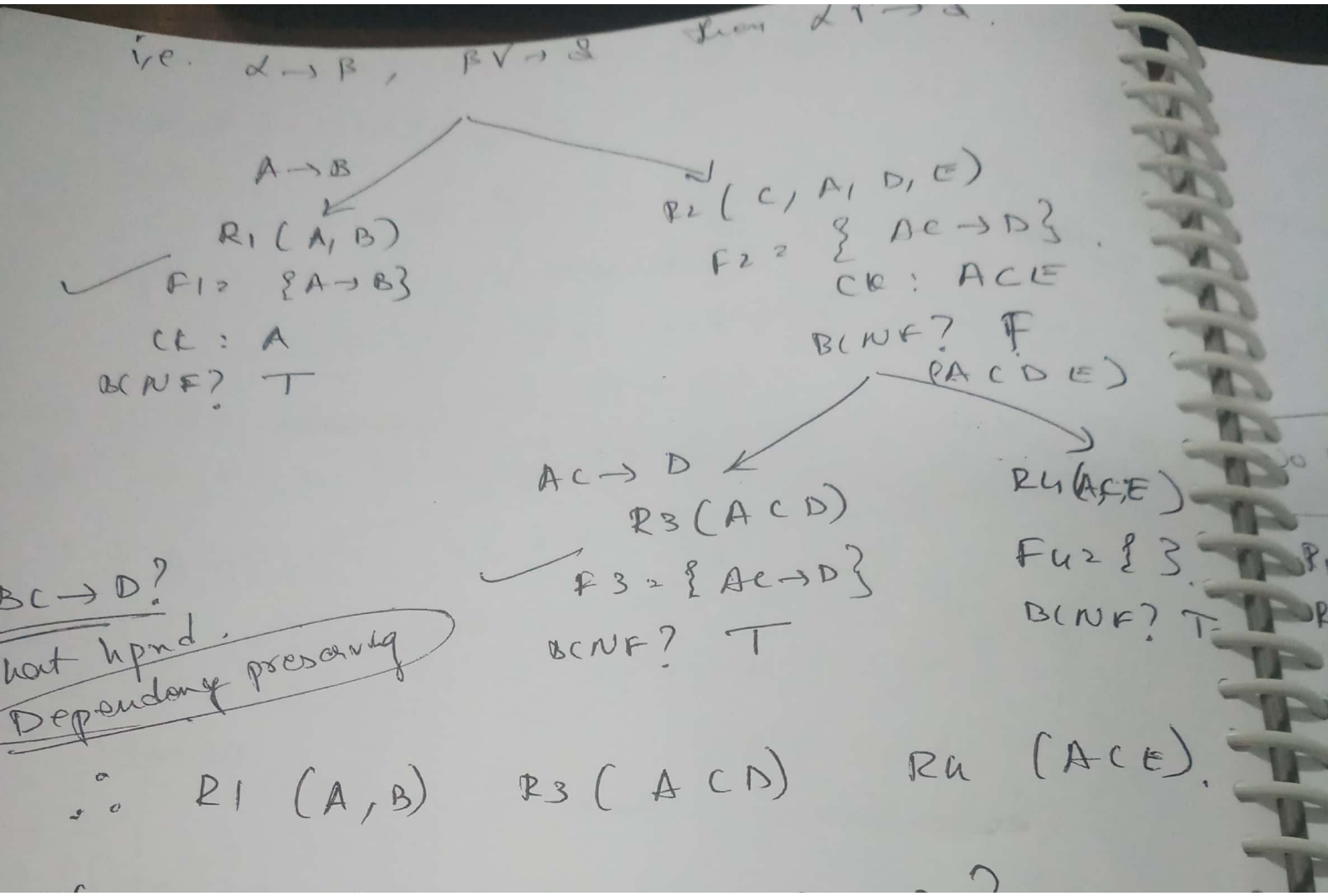
$A \to B$.

$R_1(A, B)$

$F_1 = \{A \to B\}$

$R_2(C, A, D, E)$

$F_2 = \{AC \to D\}$



Properties of Decomposition

Lossless - Join Decomposition

Let R be a relation schema & let F be a set of FD's over R. A decomposition is said to be lossless join decomposition with respect to F if, for every instance γ of R that satisfies the dependencies in F,

$$\Pi_X(\gamma) \bowtie \Pi_Y(\gamma) = \gamma.$$

i.e., we can recover the original relat from the decomposed relations.

S	P	D
S1	P1	d1
S2	P2	d2
S3	P1	d2

Ins γ

S	P
S1	P1
S2	P2
S3	P1

$\Pi_{SP}(\gamma)$

P	D
P1	d1
P2	d2
P1	d2

$\Pi_{PD}(\gamma)$

S	P
S1	P1
S2	P2
S3	P1
S1	P1
S3	P1

$\Pi_{SP}(\gamma)$

Here the last two rows are not

which is lossy join which is illegal.

⇒ All decompositions used to eliminat

Conditions for lossless join decomposition

1. $R_1 \cup R_2 = R$
2. $R_1 \cap R_2 \neq \phi$
3. $R_1 \cap R_2$ should be sk of either R_1 or R_2 or both

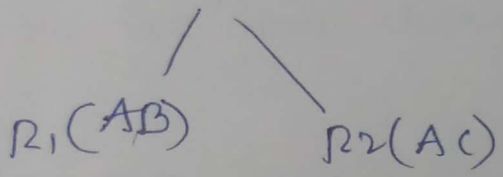
i.e., $R_1 \cap R_2 \longrightarrow R_1$

or

$R_1 \cap R_2 \longrightarrow R_2$

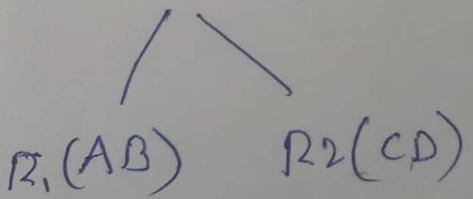
$\longrightarrow$ uniqueness property

Ex:- (1) $R(ABCD)$



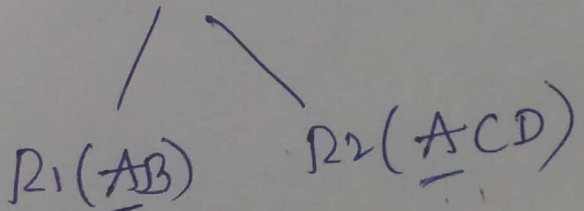
Rule 1 failed

(2) $R(ABCD)$



Rule 2 failed

(3) $R(ABCD)$



Ex:- $R = (A, B, C, D, E)$, we decompose it into $R1 = (A, B, C)$, $R2 = (A, D, E)$. The set of FD's are

(Dependency - Preserving Decomposition):

$A \rightarrow BC$, $CD \rightarrow E$, $B \rightarrow D$, $E \rightarrow A$. Show that this is a loss less join decomposition.

$FD\,R1 = \{ A \rightarrow BC \}$ $\quad$ $FD\,R2 = \{ E \rightarrow A \}$

$R1 \cap R2 = A$

$A \rightarrow ABC$ $\quad$ or $\quad$ $A \rightarrow ADE$

If any one holds true, then it is.

$A \rightarrow BC$ by Augu Rule

$AA \rightarrow ABC \iff A \rightarrow ABC$

so, it is LLJD

Theorem 3: Let R be a relation & F be a set of FDs that hold over R. The decomposition of R into R's with attr sets R1 & R2 is lossless if & only if F^+ contains

① FD $R1 \cap R2 \longrightarrow R1$ (or)

② FD $R1 \cap R2 \longrightarrow R2$

II If an FD $X \rightarrow Y$ holds over a relation R & $X \cap Y$ is empty, the decomposition of R into R-Y & XY is loseless.

i.e., X appears in both R-Y & XY, & it is a key for XY.

Another important observation, with out proof is

If R$\not$ < R1 < R11, R12 > R2 > then

$R11 \bowtie R12 \Rightarrow R1$

$R1 \bowtie R2 \Rightarrow R$

→ If the decomps are loss less decomp's.

Ex: $R(A, B, C, D)$ $R_1(A, B)$ $R_2(B, C)$ $R_3(C, D)$

$A \to B$

$B \to C$

$C \to D$

j : col
i : row

Sol)

	A	B	C	D
R_1	a_1	a_2	a_3 / $b_{1,3}$	a_4 / $b_{1,4}$
R_2	$b_{2,1}$	a_2	a_3	a_4 / $b_{2,4}$
R_3	$b_{3,1}$	$b_{3,2}$	a_3	a_4

a col

b r, col

a_j

$b_{i,j}$

When to stop this Alg?

① When no modification to the table can be done any more.

② When any one row in the table contains all 'a's.

Note: Lossless join Decomp: One Row with all 'a's

(else)

Lossy

Ex: $R(A, B, C, D, E)$

$R_1(A, D)$ $R_2(A, B)$ $R_3(B, E)$ $R_4(C, D, E)$ $R_5(A, E)$

$A \to C$ $B \to C$ $C \to D$ $DE \to C$ $CE \to A$

Sol)

	A	B	C	D	E	
R_1	a_1	$b_{1,2}$	$b_{1,3}$	a_4	$b_{1,5}$	
R_2	a_1	a_2	~~$b_{2,3}$~~ $b_{1,3}$	~~$b_{2,4}$~~ a_4	$b_{1,5}$	
R_3	~~$b_{3,1}$~~ a_1	a_2	~~$b_{3,3}$~~ a_3 $b_{1,3}$	~~$b_{3,4}$~~ a_4	a_5	← stop
R_4	~~$b_{4,1}$~~ a_1	$b_{4,2}$	a_3	a_4	a_5	
R_5	a_1	$b_{5,2}$	~~$b_{5,3}$~~ a_3 $b_{1,3}$	~~$b_{5,4}$~~ a_4	a_5	

∴ It is a Lossless join decomposition.

Ex: $R(A, B, C, D)$

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

$C \to D$ $B \to C$ $A \to B$

Ex: $R(A, B, C, D, E)$

$R_1(A, B)$ $R_2(B, C)$ $R_3(C, D)$ $R_4(D, E)$

$A \to B$ $C \to B$ $C \to D$ $E \to D$

Dependency - Preserving Decomposition

Let R is rel. schema & it is decomposed into R1 and R2 with attri sets X & Y. F is the set of FD's holds over R, then it is a dependency preserving decomposition if every FD in F appears in one of the decomposed relations.

(or)

R - Relation
- R1 - X attis $X \subset R$
- R2 - Y " $Y \subset R$

F - Set of FD's

F_X = FD's that involve only attis's in

F_Y = " " " " " "

Then, the decom is D-P-D if

$$(F_X \cup F_Y)^+ = F^+$$

TVW305

56767

Question 1: R(ABCD) F = {A → B, B → C, C → D, D → A} D = {AB, BC, CD} Check whether the decomposition is preserving dependency or not ?

Solution :
The following dependencies can be projected into the following decomposition :

R1(AB)	R2(BC)	R3(CD)
A→B	B→C	C → D

Inferring reverse FDs which fits into the decomposition-
B+ w.r.t F = {BCDA} ⇒ B → A
C+ w.r.t F = {CDAB} ⇒ C → B
D+ w.r.t F = {DABC} ⇒ D → C

So, the table will be updated as -

R1(AB)	R2(BC)	R3(CD)
A→B B→A	B→C C→B	C → D D→C

Checking D → A preserves dependency or not -
Compute D+ w.r.t updated table FDs :
D+ = {DCBA}
as closure of D w.r.t updated table FDs contains A. So D→A preserves dependency.

Question 3: R(ABCDEG) F = {AB → C, AC → B, BC → A, AD → E, B → D, E → G} D = {ABC, ACDE, ADG} Check whether the decomposition is preserving dependency or not ?

Solution :
The following dependencies can be projected into the following decomposition :

R1(ABC)	R2(ACDE)	R3(ADG)
AB→C AC→B BC→A	AD→E	

Inferring reverse FDs which fits into the decomposition-
C+ w.r.t F = {C}
B+ w.r.t F = {BD}
A+ w.r.t F = {A}
E+ w.r.t F = {EG}
No reverse FDs can be derived.

Checking B → D preserves dependency or not -
Compute B+ w.r.t updated table FDs :
B+ = {B}
as closure of B w.r.t table FDs doesn't contains D. So B→D doesn't preserves dependency.

Checking E → G preserves dependency or not -
Compute E+ w.r.t updated table FDs :
E+ = {E}
as closure of E w.r.t table FDs doesn't contains G. So E→G doesn't preserves dependency.

Question 4: Let R(ABCD) be a relational schema with the following functional dependencies : F = {A → B, B → C, C → D, D → B}. The decomposition of R into D = {AB, BC, BD} Check whether the decomposition is preserving dependency or not ?

Other kind of Dependencies

FDs are probably the most common & imp kind of Constr's from the pt. of view of db design.

Multivalued Dependencies

Suppose, we have a relation with attri's

Course	Teacher	Book
P	G	M
P	G	O
P	B	M
P	B	O
M	G	M
M	G	V
M	G	G

⇒ The R schema CTB is in BCNF, ∴ we would not consider decomposing it further.

> There is a redundancy i.e., G can teach is recorded with text M & " " "

" " " " O

This redundancy can be eliminated by decomposing R into CT & CB.

Let R be a relation schema & let X & Y be subsets of attri's of R.

MVD $X \rightarrow\rightarrow Y$ is said to hold over R, if in every legal instance γ of R, each x variable is also with a set of y values & this set is indepen of the values in other attri's.

Formally, if the MVD $X \rightarrow\rightarrow Y$ holds ov R & $Z = R - XY$, the following must be true fo every legal instance γ of R.

If $t_1 \in \gamma$, $t_2 \in \gamma$ & $t_1.X = t_2.X$, then th must be some $t_3 \in \gamma$ such that $t_1.XY = t_3.XY$ & $t_2.Z = t_3.Z$

X	Y	Z	
a	b1	c1	— t1
a	b2	c2	— t2
a	b1	c2	— t3
a	b2	c1	— t4

MVD's has 1A's & ~~five~~ additional rules. Multiva

① MVD Complementation : if $X \to\to Y$ then $X \to\to R-XY$ attrib

② " Augmentation : " " & $W \supseteq Z$ then $WX \to\to YZ$ →

③ " Transitivity : " " & $Y \to\to Z$ then $X \to\to (Z-Y)$

④ " Replication : If $X \to Y$ then $X \to\to Y$

Fourth Normal forms

" " " is direct generalization of BCNF.

Let R be a relation schema, X & Y be non empty subsets of the attri's of R, & F be a set of dependencies that includes both FDs & MVDs. R is said to be in 4NF, if for every $X \to\to Y$ that ~~holds~~ over R, one of the ~~following~~ is true

① $Y \subseteq X$ or $XY = R$ or

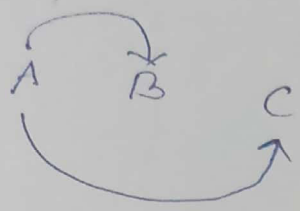
② X is a superkey

5NF : R is in 3NF & each of its keys consist of a single attri, it is also in 5NF.

Multivalued Dependency (MVD)

MVD arise when a relation R having non-atomic attributes is converted to a normalized form.

→ for MVD there should be atleast 3 attributes.



If A determines multiple values of B &

" A " " " " C &

there is no connection b/w B & C, then

$A \rightarrow\!\!\rightarrow B$ $A \rightarrow\!\!\rightarrow C$ is called MVD.

Course	Instr	Textbook
OS	A	T1
	B	T2
	C	T3

- Course can be taught by multiple instructors
- " has " textbooks &

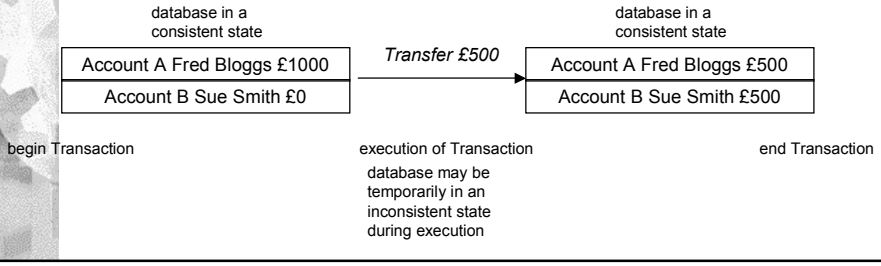
b/w Instructors & textbook there is no corr.

∴ $C \rightarrow\!\!\rightarrow I$, $C \rightarrow\!\!\rightarrow T$

Transaction Processing
Recovery & Concurrency Control

What is a transaction

- A transaction is the basic logical unit of execution in an information system. A transaction is a sequence of operations that must be executed as a whole, taking a consistent (& correct) database state into another consistent (& correct) database state;
- A collection of actions that make consistent transformations of system states while preserving system consistency
- An indivisible unit of processing



Desirable Properties of ACID Transactions

A *Atomicity*: a transaction is an atomic unit of processing and it is either performed entirely or not at all

C *Consistency* Preservation: a transaction's correct execution must take the database from one correct state to another

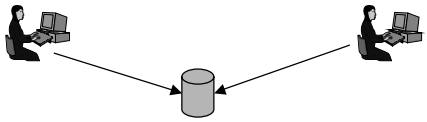
I *Isolation/Independence*: the updates of a transaction must not be made visible to other transactions until it is committed (solves the temporary update problem)

D *Durability* (or Permanency): if a transaction changes the database and is committed, the changes must never be lost because of subsequent failure

o *Serialisability*: transactions are considered serialisable if the effect of running them in an interleaved fashion is equivalent to running them serially in some order

Requirements for Database Consistency

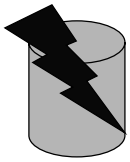
- Concurrency Control
  - Most DBMS are multi-user systems.
  - The concurrent execution of many different transactions submitted by various users must be organised such that each transaction does not interfere with another transaction with one another in a way that produces incorrect results.
  - The concurrent execution of transactions must be such that each transaction appears to execute in isolation.



- Recovery
  - System failures, either hardware or software, must not result in an inconsistent database

Transaction as a Recovery Unit

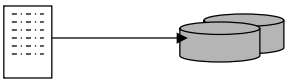
- If an error or hardware/software crash occurs between the begin and end, the database will be inconsistent
  - Computer Failure (system crash)
  - A transaction or system error
  - Local errors or exception conditions detected by the transaction
  - Concurrency control enforcement
  - Disk failure
  - Physical problems and catastrophes



- The database is restored to some state from the past so that a correct state—close to the time of failure—can be reconstructed from the past state.
- A DBMS ensures that if a transaction executes some updates and then a failure occurs before the transaction reaches normal termination, then those updates are undone.
- The statements COMMIT and ROLLBACK (or their equivalent) ensure Transaction Atomicity

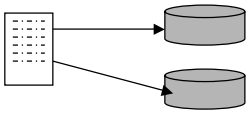
Recovery

- Mirroring
  - keep two copies of the database and maintain them simultaneously



- Backup
  - periodically dump the complete state of the database to some form of tertiary storage

- System Logging
  - the log keeps track of all transaction operations affecting the values of database items. The log is kept on disk so that it is not affected by failures except for disk and catastrophic failures.



Recovery from Transaction Failures

Catastrophic failure

- Restore a previous copy of the database from archival backup
- Apply transaction log to copy to reconstruct more current state by redoing committed transaction operations up to failure point
- Incremental dump + log each transaction

Non-catastrophic failure

- Reverse the changes that caused the inconsistency by *undoing* the operations and possibly *redoing* legitimate changes which were lost
- The entries kept in the system log are consulted during recovery.
- No need to use the complete archival copy of the database.

Transaction States

- For recovery purposes the system needs to keep track of when a transaction starts, terminates and commits.
- Begin_Transaction: marks the beginning of a transaction execution;
- End_Transaction: specifies that the read and write operations have ended and marks the end limit of transaction execution (but may be aborted because of concurrency control);
- Commit_Transaction: signals a successful end of the transaction. Any updates executed by the transaction can be safely committed to the database and will not be undone;
- Rollback (or Abort): signals that the transaction has ended unsuccessfully. Any changes that the transaction may have applied to the database must be undone;
- Undo: similar to **ROLLBACK** but it applies to a single operation rather than to a whole transaction;
- Redo: specifies that certain transaction operations must be redone to ensure that all the operations of a committed transaction have been applied successfully to the database;

Entries in the System Log

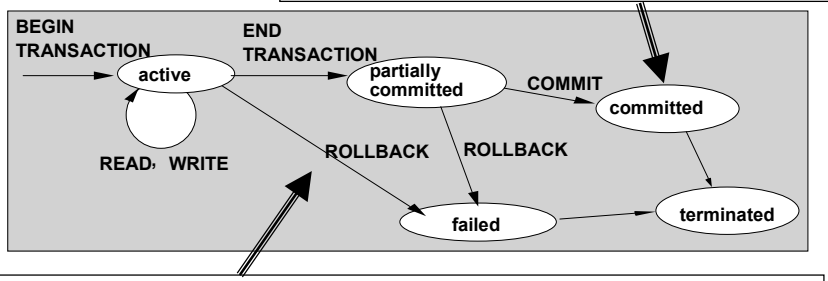
For every transaction a unique transaction-id is generated by the system.

- **[start_transaction, transaction-id]:** the start of execution of the transaction identified by transaction-id
- **[read_item, transaction-id, X]:** the transaction identified by transaction-id reads the value of database item X. Optional in some protocols.
- **[write_item, transaction-id, X, old_value, new_value]:** the transaction identified by transaction-id changes the value of database item X from old_value to new_value
- **[commit, transaction-id]:** the transaction identified by transaction-id has completed all accesses to the database successfully and its effect can be recorded permanently (committed)
- **[abort, transaction-id]:** the transaction identified by transaction-id has been aborted

```
Credit_labmark (sno
NUMBER, cno CHAR, credit
NUMBER)
old_mark  NUMBER;
new_mark  NUMBER;

SELECT labmark INTO
old_mark FROM enrol
WHERE studno = sno and
courseno = cno FOR UPDATE
OF labmark;

new_ mark := old_ mark +
credit;

UPDATE enrol SET labmark
= new_mark WHERE studno =
sno and courseno = cno ;

COMMIT;

EXCEPTION
  WHEN OTHERS THEN
ROLLBACK;

END credit_labmark;
```

Transaction execution

A transaction reaches its *commit point* when all operations accessing the database are completed and the result has been recorded in the log. It then writes a [commit, transaction-id].



If a system failure occurs, searching the log and rollback the transactions that have written into the log a

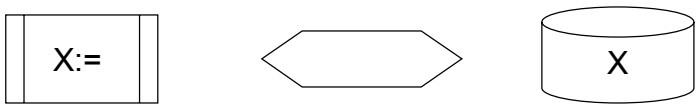
[start_transaction, transaction-id]

[write_item, transaction-id, X, old_value, new_value]

but have not recorded into the log a **[commit, transaction-id]**

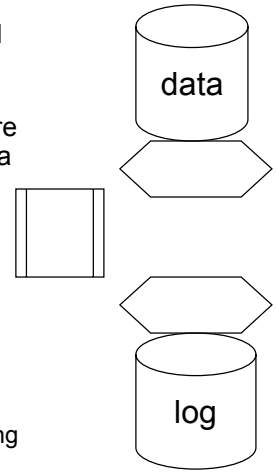
Read and Write Operations of a Transaction

- Specify read or write operations on the database items that are executed as part of a transaction
- read_item(X):
  - reads a database item named X into a program variable also named X.
    1. find the address of the disk block that contains item X
    2. copy that disk block into a buffer in the main memory
    3. copy item X from the buffer to the program variable named
- write_item(X):
  - writes the value of program variable X into the database item named X.
    1. find the address of the disk block that contains item X
    2. copy that disk block into a buffer in the main memory
    3. copy item X from the program variable named X into its current location in the buffer store the updated block in the buffer back to disk (this step updates the database on disk)



Checkpoints in the System Log

- A [checkpoint] record is written periodically into the log when the system writes out to the database on disk the effect of all WRITE operations of committed transactions.
- All transactions whose [commit, transaction-id] entries can be found in the system log will not require their WRITE operations to be redone in the case of a system crash.
- Before a transaction reaches commit point, force-write or flush the log file to disk before commit transaction.
- Actions Constituting a Checkpoint
  - temporary suspension of transaction execution
  - forced writing of all updated database blocks in main memory buffers to disk
  - writing a [checkpoint] record to the log and force writing the log to disk
  - resuming of transaction execution



Write Ahead Logging

"In place" updating protocols: Overwriting data in situ

Deferred Update:

- no actual update of the database until after a transaction reaches its commit point

1. Updates recorded in log
2. Transaction commit point
3. Force log to the disk
4. Update the database

FAILURE!
REDO database from log entries
No UNDO necessary because database never altered

Immediate Update:

- the database may be updated by some operations of a transaction before it reaches its commit point.

1. Update X recorded in log
2. Update X in database
3. Update Y recorded in log
4. Transaction commit point
3. Force log to the disk
4. Update Y in database

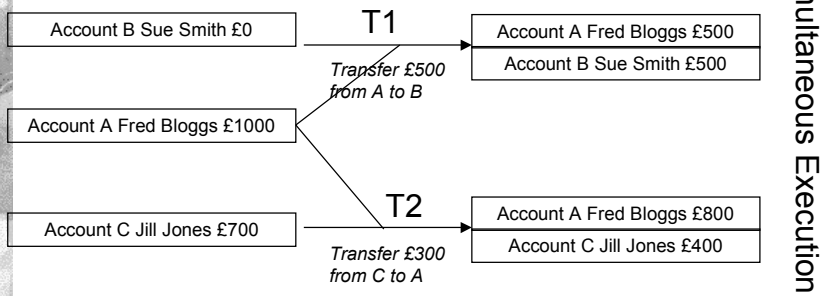
FAILURE!
UNDO X

FAILURE!
REDO Y

- Undo in reverse order in log
- Redo in committed log order
- uses the write_item log entry

Transaction as a Concurrency Unit

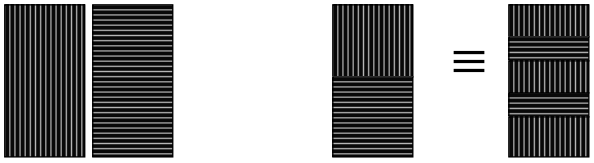
- Transactions must be synchronised correctly to guarantee database consistency



Net result
Account A 800
Account B 500
Account C 400

Transaction scheduling algorithms

- Transaction Serialisability
  - The effect on a database of any number of transactions executing in parallel must be the same as if they were executed one after another



- Problems due to the Concurrent Execution of Transactions
  - The Lost Update Problem
  - The Incorrect Summary or Unrepeatable Read Problem
  - The Temporary Update (Dirty Read) Problem

The Lost Update Problem

- Two transactions accessing the same database item have their operations interleaved in a way that makes the database item incorrect

X=4
Y=8
N=2
M=3

T1: (joe)	T2: (fred)	X	Y
read_item(X);		4	
X:= X - N;		2	
	read_item(X);	4	
	X:= X + M;	7	
write_item(X);		2	
read_item(Y);		8	
	write_item(X);	7	
Y:= Y + N;			10
write_item(Y);			10

- item X has incorrect value because its update from T1 is “lost” (overwritten)
- T2 reads the value of X before T1 changes it in the database and hence the updated database value resulting from T1 is lost

The Incorrect Summary or Unrepeatable Read Problem

- One transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records.
- The aggregate function may calculate some values before they are updated and others after.

T2 reads X after N is subtracted and reads Y before N is added, so a wrong summary is the result

T1:	T2:	T1	T2	Sum
	sum:= 0;			0
	read_item(A);		4	
	sum:= sum + A;			4
read_item(X);.	.	4		
X:= X - N;	.	2		
write_item(X);		2		
	read_item(X);		2	
	sum:= sum + X;			6
	read_item(Y);		8	
	sum:= sum + Y;			14
read_item(Y);		8		
Y:= Y + N;		10		
write_item(Y);		10		

Dirty Read or The Temporary Update Problem

- One transaction updates a database item and then the transaction fails. The updated item is accessed by another transaction before it is changed back to its original value

Joe books seat on flight X

Joe cancels

T1: (joe)	T2: (fred)	Database	Log old	Log new
read_item(X);		4		
X:= X - N;		2		
write_item(X);		2	4	2
	read_item(X);	2		
	X:= X- N;	-1		
	write_item(X);	-1	2	-1
failed write (X)		4	rollback T1 log	

Fred books seat on flight X because Joe was on Flight X

- transaction T1 fails and must change the value of X back to its old value
- meanwhile T2 has read the “temporary” incorrect value of X

Schedules of Transactions

- A schedule S of n transactions is a sequential ordering of the operations of the n transactions.
  - *The transactions are interleaved*
- A schedule maintains the order of operations within the individual transaction.
  - For each transaction T if operation a is performed in T before operation b, then operation a will be performed before operation b in S.
  - *The operations are in the same order as they were before the transactions were interleaved*
- Two operations conflict if they belong to different transactions, AND access the same data item AND one of them is a write.

T1

read x
write x

T2

read x
write x

S

read x
read x
write x
write x

Serial and Non-serial Schedules

- A schedule S is *serial* if, for every transaction T participating in the schedule, all of T's operations are executed consecutively in the schedule; otherwise it is called *non-serial*.
- Non-serial schedules mean that transactions are interleaved. There are many possible orders or schedules.
- *Serialisability* theory attempts to determine the 'correctness' of the schedules.
- A schedule S of n transactions is serialisable if it is equivalent to some serial schedule of the same n transactions.

Example of Serial Schedules

- Schedule A

T1:	T2:
read_item(X);	
X:= X - N;	
write_item(X);	
read_item(Y);	
Y:=Y + N;	
write_item(Y);	
	read_item(X);
	X:= X + M;
	write_item(X);

•Schedule B

T1:	T2:
	read_item(X);
	X:= X + M;
	write_item(X);
read_item(X);	
X:= X - N;	
write_item(X);	
read_item(Y);	
Y:=Y + N;	
write_item(Y);	

Example of Non-serial Schedules

- Schedule C

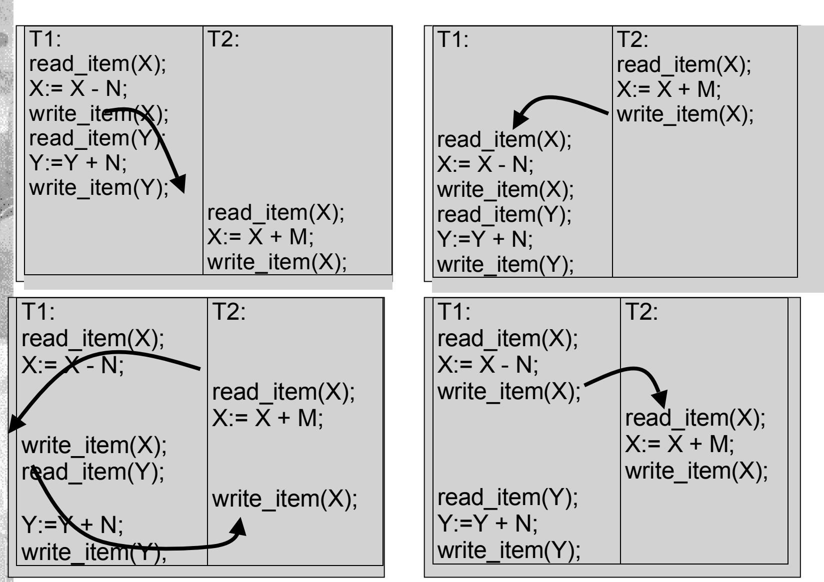
T1:	T2:
read_item(X);	
X:= X - N;	
	read_item(X);
	X:= X + M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y + N;	
write_item(Y);	

•Schedule D

T1:	T2:
read_item(X);	
X:= X - N;	
write_item(X);	
	read_item(X);
	X:= X + M;
	write_item(X);
read_item(Y);	
Y:=Y + N;	
write_item(Y);	

We have to figure out whether a schedule is equivalent to a serial schedule
i.e. the reads and writes are in the right order

Precedence graphs (assuming read X before write X)



View Equivalence and View Serialisability

- View Equivalence:
  - As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
  - The read operations are said to *see the same view* in both schedules
  - The final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules
- A schedule S is view serialisable if it is view equivalent to a serial schedule.
- Testing for view serialisability is NP-complete

Semantic Serialisability

- Some applications can produce schedules that are correct but aren't conflict or view serialisable.
- e.g. Debit/Credit transactions (Addition and subtraction are commutative)

T1	T2
read_item(X);	read_item(Y);
X:=X-10;	Y:=Y-20;
write_item(X);	write_item(Y);
read_item(Y);	read_item(Z);
Y:=Y+10;	Z:+Z+20;
write_item(Y);	write_item(Z);

Schedule

T1	T2
read_item(X);	
X:=X-10;	
write_item(X);	
	read_item(Y);
	Y:=Y-20;
	write_item(Y);
read_item(Y);	
Y:=Y+10;	
write_item(Y);	

Methods for Serialisability

- *Multi-version* Concurrency Control techniques keep the old values of a data item when that item is updated.
- *Timestamps* are unique identifiers for each transaction and are generated by the system. Transactions can then be ordered according to their timestamps to ensure serialisability.
- *Protocols* that, if followed by every transaction, will ensure serialisability of all schedules in which the transactions participate. They may use *locking* techniques of data items to prevent multiple transactions from accessing items concurrently.
- Pessimistic Concurrency Control
  - Check before a database operation is executed by locking data items before they are read and written or checking timestamps

Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronising the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

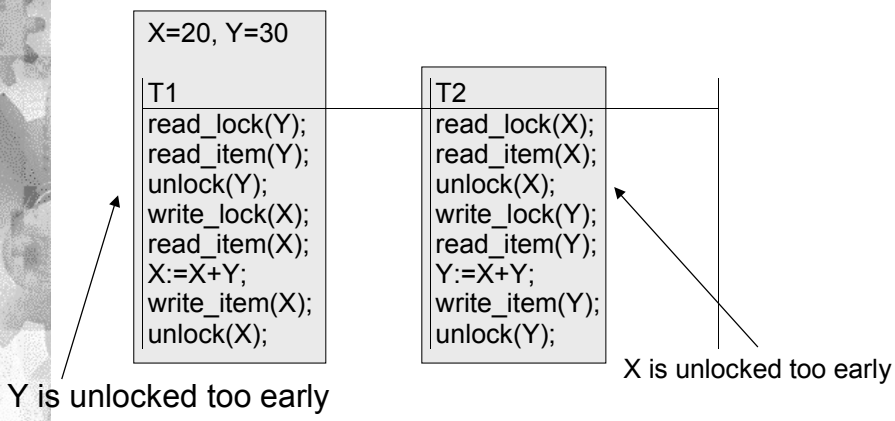
Types of Locks

- Binary locks have two possible states:
  1. locked (lock_item(X) operation) and
  2. unlocked (unlock_item(X) operation
- Multiple-mode locks allow concurrent access to the same item by several transactions. Three possible states:
  1. read locked or shared locked (other transactions are allowed to read the item)
  2. write locked or exclusive locked (a single transaction exclusively holds the lock on the item) and
  3. unlocked.
- Locks are held in a lock table.
  - upgrade lock: read lock to write lock
  - downgrade lock: write lock to read lock

Locks don't guarantee serialisability: Lost Update

T1: (joe)	T2: (fred)	X	Y
write_lock(X)			
read_item(X);		4	
X:= X - N;		2	
unlock(X)			
	write_lock(X)		
	read_item(X);	4	
	X:= X + M;	7	
	unlock(X)		
write_lock(X)			
write_item(X);		2	
unlock(X)			
write_lock(Y)			
read_item(Y);			8
	write_lock(X)		
	write_item(X);	7	
	unlock(X)		
Y:= Y + N;			10
write_item(Y);			10
unlock(Y)			

Locks don't guarantee serialisability



- Schedule 1: T1 followed by T2 $\Rightarrow$ X=50, Y=80
- Schedule 2: T2 followed by T1 $\Rightarrow$ X=70, Y=50

Non-serialisable schedule S that uses locks

X=20
Y=30

T1	T2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
X:=X+Y;	
write_item(X);	
unlock(X);	

result of S $\Rightarrow$ X=50, Y=50

Ensuring Serialisability: Two-Phase Locking

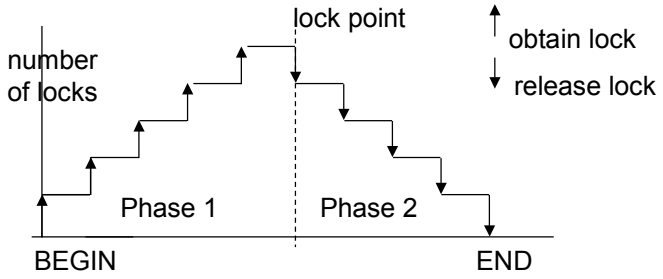
- All locking operations (read_lock, write_lock) precede the first unlock operation in the transactions.
- Two phases:
  - *expanding phase:* new locks on items can be acquired but none can be released
  - *shrinking phase:* existing locks can be released but no new ones can be acquired

X=20, Y=30

T1	T2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Two-Phasing Locking

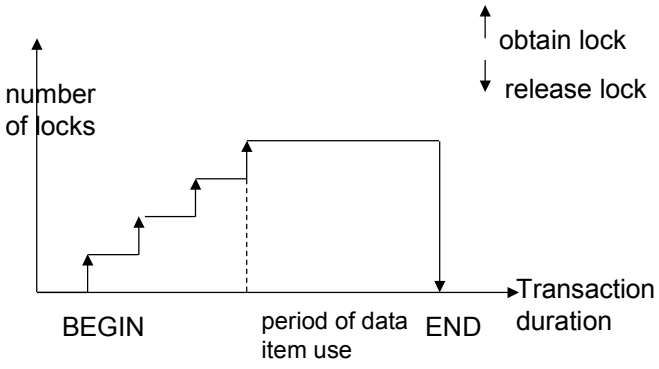
- Basic 2PL
  - When a transaction releases a lock, it may not request another lock



- Conservative 2PL or static 2PL
  - a transaction locks all the items it accesses before the transaction begins execution
  - pre-declaring read and write sets

Two-Phasing Locking

- Strict 2PL a transaction does not release any of its locks until after it commits or aborts
- leads to a strict schedule for recovery



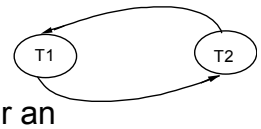
Locking Problems: Deadlock

- Each of two or more transactions is waiting for the other to release an item. Also called a deadly embrace

T1	T2
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

Deadlocks and Livelocks

- Deadlock prevention protocol:
  - conservative 2PL
  - transaction stamping (younger transactions aborted)
    - no waiting
    - cautious waiting
    - time outs
- Deadlock detection (if the transaction load is light or transactions are short and lock only a few items)
  - wait-for graph for deadlock detection
  - victim selection
  - cyclic restarts



- Livelock: a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
  - fair waiting schemes (i.e. first-come-first-served)

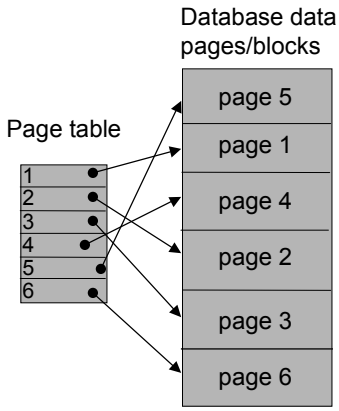
Locking Granularity

- A database item could be
  - a database record
  - a field value of a database record
  - a disk block
  - the whole database
- Trade-offs
  - coarse granularity
    - the larger the data item size, the lower the degree of concurrency
  - fine granularity
    - the smaller the data item size, the more locks to be managed and stored, and the more lock/unlock operations needed.

Other Recovery and Concurrency Strategies

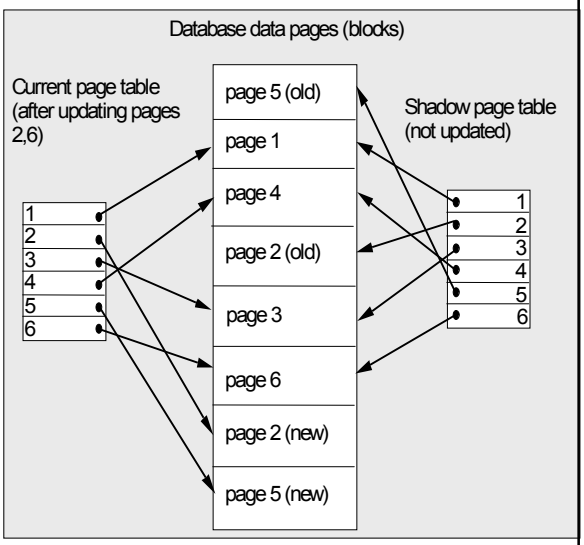
Recovery: Shadow Paging Technique

- Data isn't updated 'in place'
- The database is considered to be made up of a number of n fixed-size disk blocks or pages, for recovery purposes.
- A page table with n entries is constructed where the i^{th} page table entry points to the i^{th} database page on disk.
- Current page table points to most recent current database pages on disk



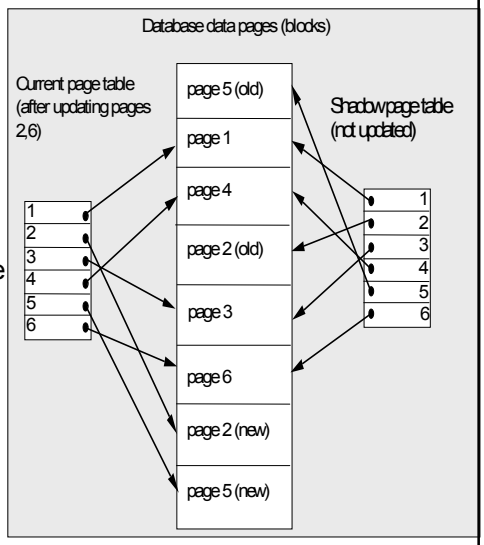
Shadow Paging Technique

- When a transaction begins executing
  - the current page table is copied into a shadow page table
  - shadow page table is then saved
  - shadow page table is never modified during transaction execution
  - writes operations—new copy of database page is created and current page table entry modified to point to new disk page/block



Shadow Paging Technique

- To recover from a failure
  - the state of the database before transaction execution is available through the shadow page table
    - free modified pages
    - discard currrent page table
    - that state is recovered by reinstating the shadow page table to become the current page table once more
- Commiting a transaction
  - discard previous shadow page
  - free old page tables that it references
- Garbage collection



Optimistic Concurrency Control

- No checking while the transaction is executing.
- Check for conflicts after the transaction.
- Checks are all made at once, so low transaction execution overhead
- Relies on little interference between transactions
  - Updates are not applied until end_transaction
  - Updates are applied to *local copies* in a transaction space.

1. *read phase*: read from the database, but updates are applied only to local copies
2. *validation phase*: check to ensure serialisability will not be validated if the transaction updates are actually applied to the database
3. *write phase*: if validation is successful, transaction updates applied to database; otherwise updates are discarded and transaction is aborted and restarted.

Validation Phase

- Use transaction timestamps
- write_sets and read_sets maintained
- Transaction B is committed or in its validation phase
- Validation Phase for Transaction A
- To check that TransA does not interfere with TransB the following must hold:
  - TransB completes its write phase before TransA starts its reads phase
  - TransA starts its write phase after TransB completes its write phase, and the read set of TransA has no items in common with the write set of TransB
  - Both the read set and the write set of TransA have no items in common with the write set of TransB, and TransB completes its read phase before TransA completes its read phase.

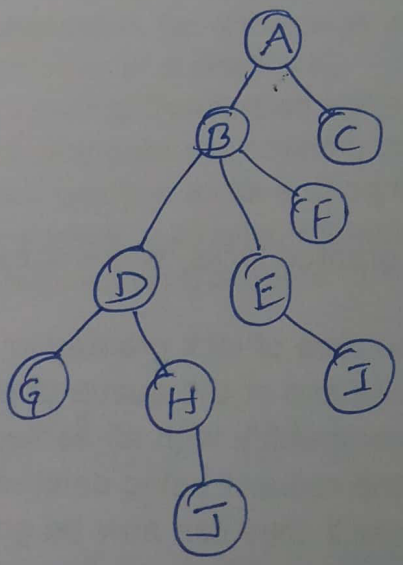
Conclusions

- Transaction management deals with two key requirements of any database system:
- Resilience
  - in the ability of data surviving hardware crashes and software errors without sustaining loss or becoming inconsistent
- Access Control
  - in the ability to permit simultaneous access of data multiple users in a consistent manner and assuring only authorised access

Graph-Based Protocols

when we wish to develop protocols which are not two phase, then we need add inf. & it gets by prior knowledge abt the order in which the db items will be accessed.

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering → on the set D = {d1, d2 ,..., dh} of all data items.
- If di → dj then any transaction accessing both di and dj must access di before accessing dj.
- Implies that the set D may now be viewed as a directed acyclic graph, called a database graph.
- The tree-protocol is a simple kind of graph protocol.



Need to lock A & I
first lock-x (A)
for I, E shd be loc
" E B " " "
" B, A " " "
So, lock E & B
to lock I

Only exclusive locks are allowed.

- The first lock by Ti may be on any data item. Subsequently, a

Conditions

1. Only lock-Ex
2. First lock can be acquired on any data item.
3. Subsequent locks are allowed only if the parent is locked.
4. Unlock at any point.
5. Each data item can be accessed atmost once by a
6. Relocking by same transaction is not allowed.

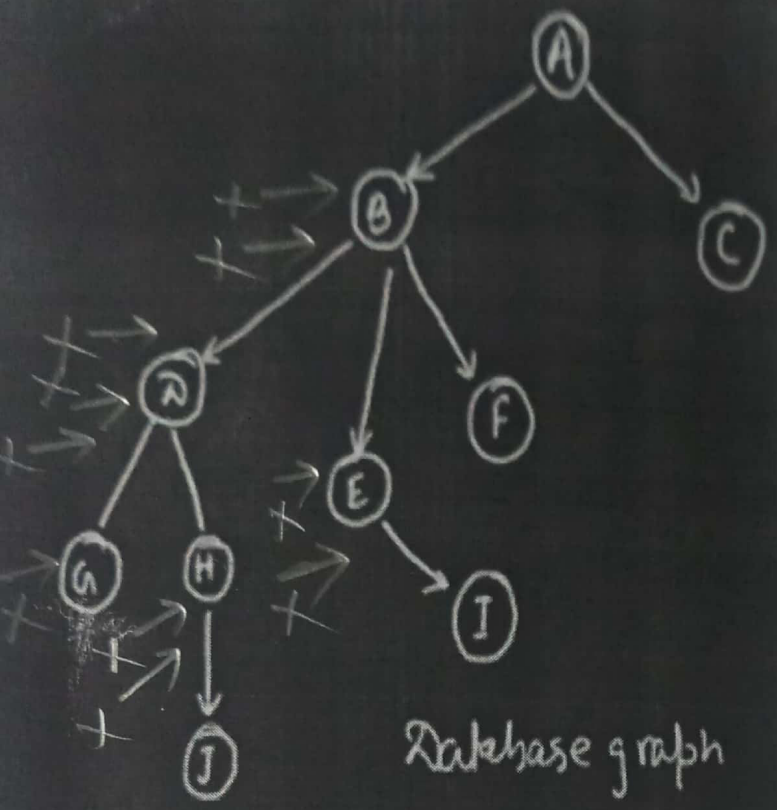
Advantages

1. Deadlock free
2. No rollback
3. Shorter waiting time

Disadvan

1. prior knowledge of data access
2. unnecessary locks.

Illustration on Graph-Based Protocol



Database graph

Schedule S

T1	T2	T3	T4
lock-x(B)			
	lock-x(D)		
	lock-x(H)		
	Unlock(D)		
lock-x(E)			
lock-x(D)			
Unlock(B)			
Unlock(E)			
		lock-x(B)	
		lock-x(E)	
	Unlock(H)		
lock-x(G)			
Unlock(D)			
			lock-x(D)
			lock-x(H)
			Unlock(D)
			Unlock(H)
		Unlock(E)	
		Unlock(B)	
Unlock(G)			

VBP-①

data Q can be locked by Ti only if the parent of Q is currently locked by Ti.

- Data items may be unlocked at any time.

The tree protocol ensures conflict serializability as well as freedom from deadlock.

- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.

- shorter waiting times, and increase in concurrency
- protocol is deadlock-free, no rollbacks are required
- the abort of a transaction can still lead to cascading rollbacks.(this correction has to be made in the book also.)
- However, in the tree-locking protocol, a transaction may have to lock data items that it does not access.
- increased locking overhead, and additional waiting time
- potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

Validation-Based Protocol

Execution of transaction Ti is done in three phases.

1. **Read and execution phase**: Transaction Ti writes only to temporary local variables

2. **Validation phase**: Transaction Ti performs a ``validation test'' to determine if local variables can be written without violating serializability.

3. **Write phase**: If Ti is validated, the updates are applied to the database; otherwise, Ti is rolled back.

- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation

Each transaction Ti has 3 timestamps

VPB -(2)

- **Start(Ti)** : the time when Ti started its execution
- **Validation(Ti)**: the time when Ti entered its validation phase
- **Finish(Ti)** : the time when Ti finished its write phase

- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.

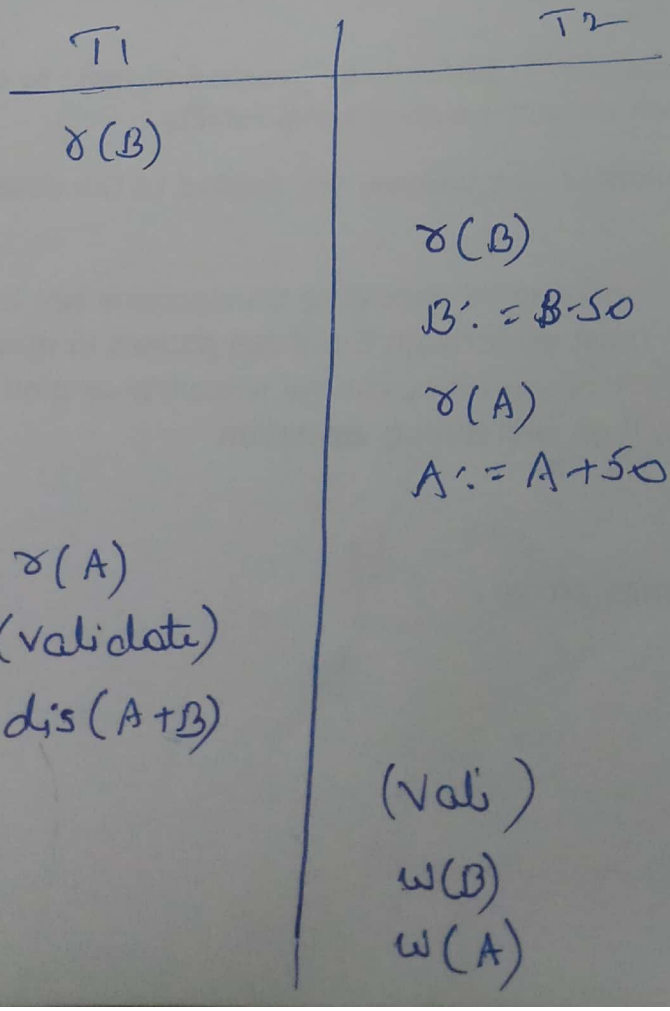
If for all Ti with TS (Ti) < TS (Tj) either one of the following condition holds:

- finish(Ti) < start(Tj)
- start(Tj) < finish(Ti) < validation(Tj) and the set of data items written by Ti does not intersect with the set of data items read by Tj .then validation succeeds and Tj can be committed. Otherwise, validation fails and Tj is aborted.
- Justification: Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and

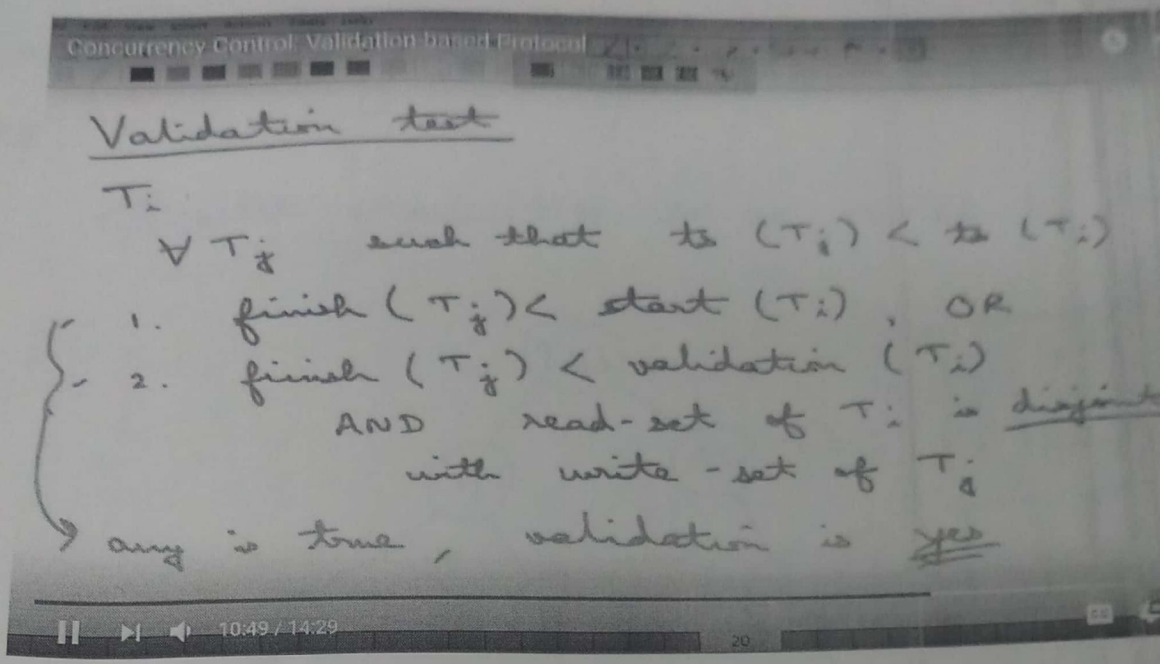
1. the writes of Tj do not affect reads of Ti since they occur after Ti has finished its reads.

2. the writes of Ti do not affect reads of Tj since Tj does not read any item written by Ti

Schedule Produced by Validation



VBP-③



Advantages

1. Cascadeless
2. Deadlock free

Disadvantage

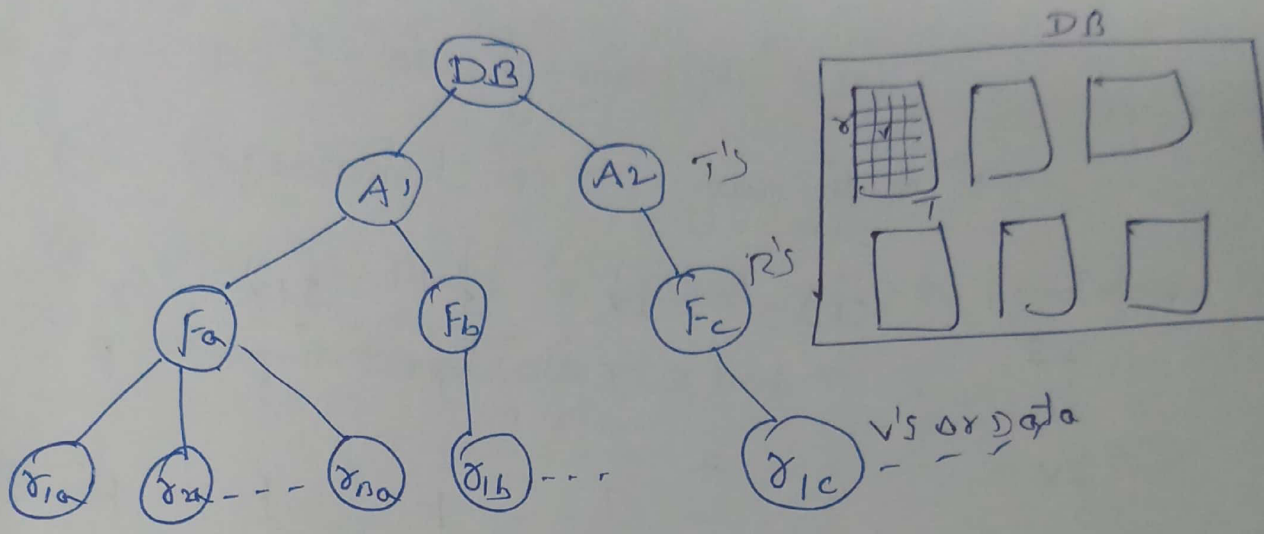
1. Starvation

Multiple Granularity

MG-①

Data items can be grouped in some way and instead of locking ind. dI's, we can lock the complete group.

Smaller granularities can be nested in larger ones



Each node can be locked individually

If T locks a node in S or X mode, all descendents are also locked in same mode.

if F_{1a} is exp locked in only mode by T_1 then all records are implicitly locked in Ex mode.

In this we have intension lock modes.

They are ① Intention - shared IS
② " Excl IX
③ shared & Intent - Excl SIX

MG - ①

	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

Compatability Matrix

A B C . . . .

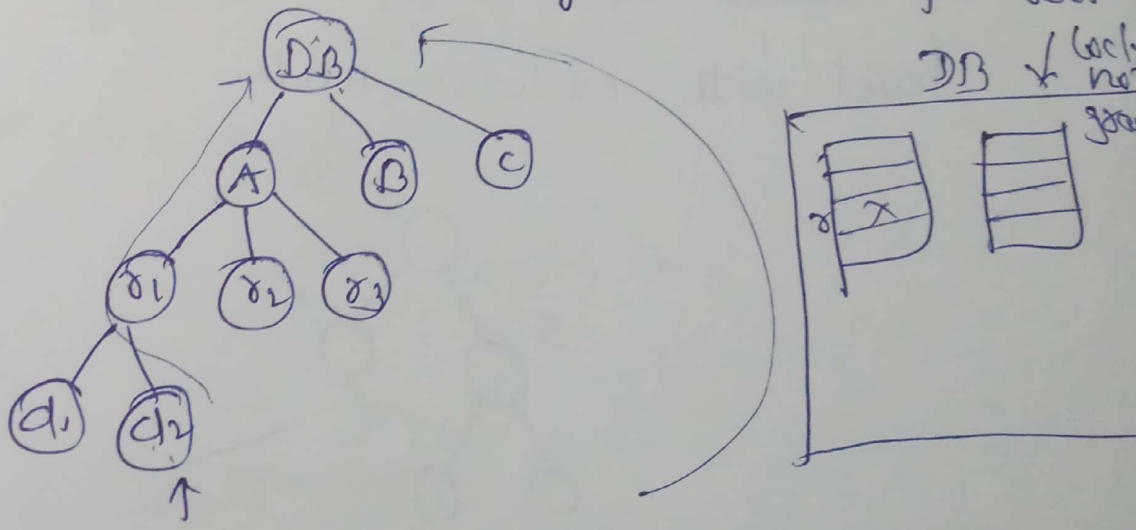
DI's are individual

(A (γ1)) → levels of DI's in DB

Re ① DI's are of multiple sizes
② Represented in the form of a tree.
③ when a node is locked all the children are automatically locked.

Problems

① Case-1 : if we lock a dI in last level, we need to ~~lock~~ check whether all its ancestors are lock free or not upto root.

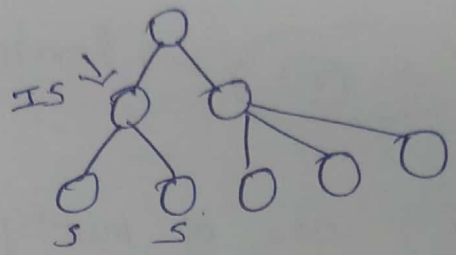


Case-2 : If we want to lock DB itsely, we need to check whether any other Trans locked any T, or γ, d.

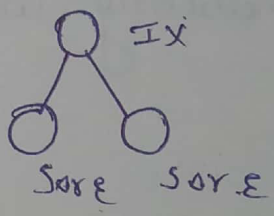
So, Now we introduce Intension lock Modes

IS : Explicit locking at lower level of tree but only with shared lock.

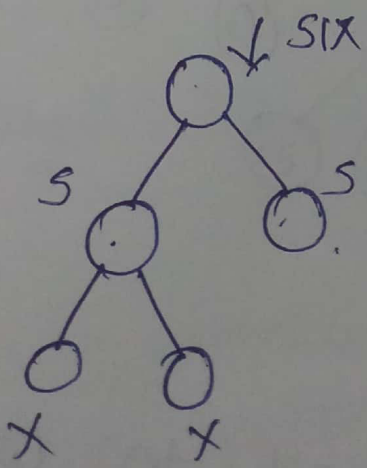
IMG-④



IX : Explicit locking at lower level of tree with exclusive or shared lock.



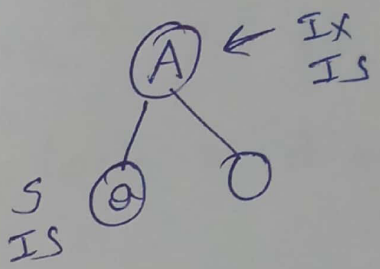
IX : Subtree rooted by that node is locked explicitly in shared mode & explicit locking is done at lower level with exclusive mode.



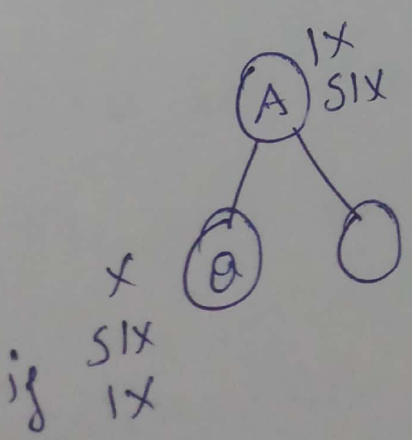
So, a tran T_i that attempts to lock a node Q must follow these rules:

MG-(5)

1. Must observe the lock compatability
2. Root of the tree must be locked first & it can be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i either IX or IS mode.



4. " " " " X, SIX or IX "
" " " " " " " "
" IX or SIX mode.



should be if

⑤ T_i can lock a node only if it has not unlocked any node so far.

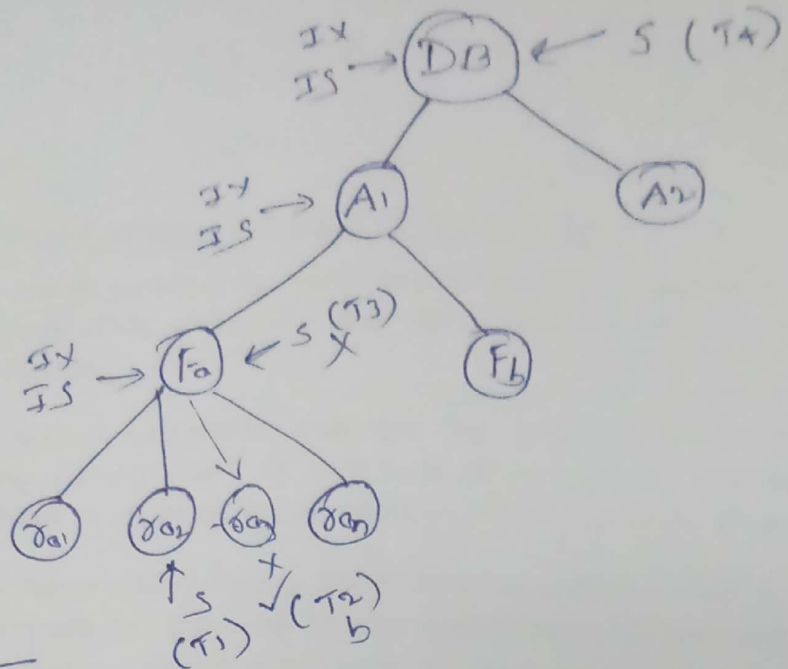
⑥ T_i can unlock a node " " non of the child of a is currently locked by T_i

TMG-⑥

Locking - Root to Leaf

Unlocking - Leaf to Root

MIG-7



(1) T_1 reads ra_2, for this we need to check fo root (no locks should be there) to that node

(2) T_2 modifies ra_2 (IS), on ra_2 - S lo is there so will not get a lock & it has to roll back

if ra_3, it has to get X lock, for it all ancs are lock comp all or no check it from root.

3) T_3 read all records of F_a

S is not comp with IX

It will not exe, roll back

) T_4 reads entire DB

S is not comp with IX

So roll back

TSOP-②

Timestamp-based Protocols

The most commonly used concurrency protocol is the timestamp based protocol. This protocol uses either system time or logical counter as a timestamp.

Lock-based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

Every transaction has a timestamp associated with it, and the ordering is determined by the age of the transaction. A transaction created at 0002 clock time would be older than all other transactions that come after it. For example, any transaction 'y' entering the system at 0004 is two seconds younger and the priority would be given to the older one.

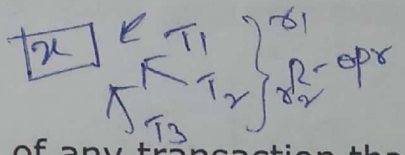
In addition, every data item is given the latest read and write-timestamp. This lets the system know when the last 'read and write' operation was performed on the data item.

Each transaction is issued a timestamp when it enters the system. If an old transaction Ti has time-stamp TS(Ti), a new transaction Tj is assigned time-stamp TS(Tj) such that TS(Ti) <TS(Tj).

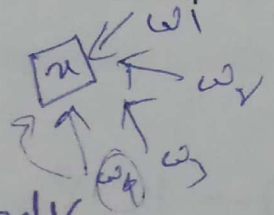
- The protocol manages concurrent execution such that the timestamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:

youngest

- W-timestamp(Q) is the largest time-stamp of any transaction that executed write(Q) successfully.



- R-timestamp(Q) is the largest time-stamp of any transaction that executed read(Q) successfully.

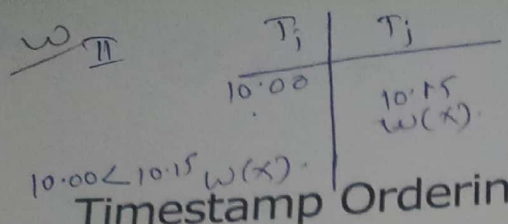


Adv

① NO Deadlock

Disadv

starvation may happen.



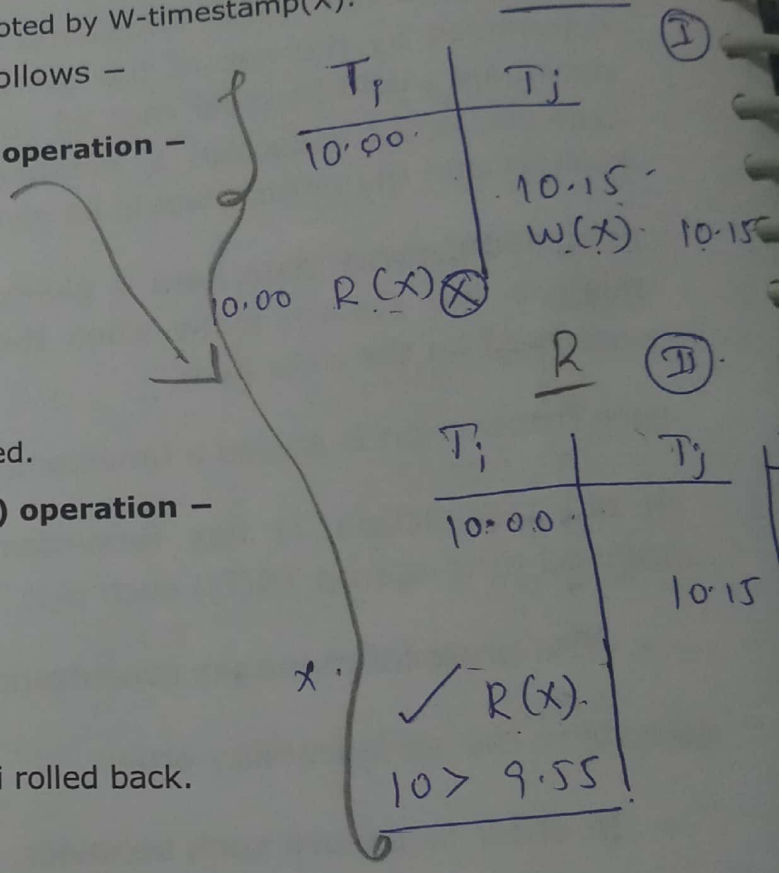
Timestamp Ordering Protocol

The timestamp-ordering protocol ensures serializability among transactions in their conflicting read and write operations. This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- The timestamp of transaction T_i is denoted as $TS(T_i)$.
- Read time-stamp of data-item X is denoted by R-timestamp(X).
- Write time-stamp of data-item X is denoted by W-timestamp(X).

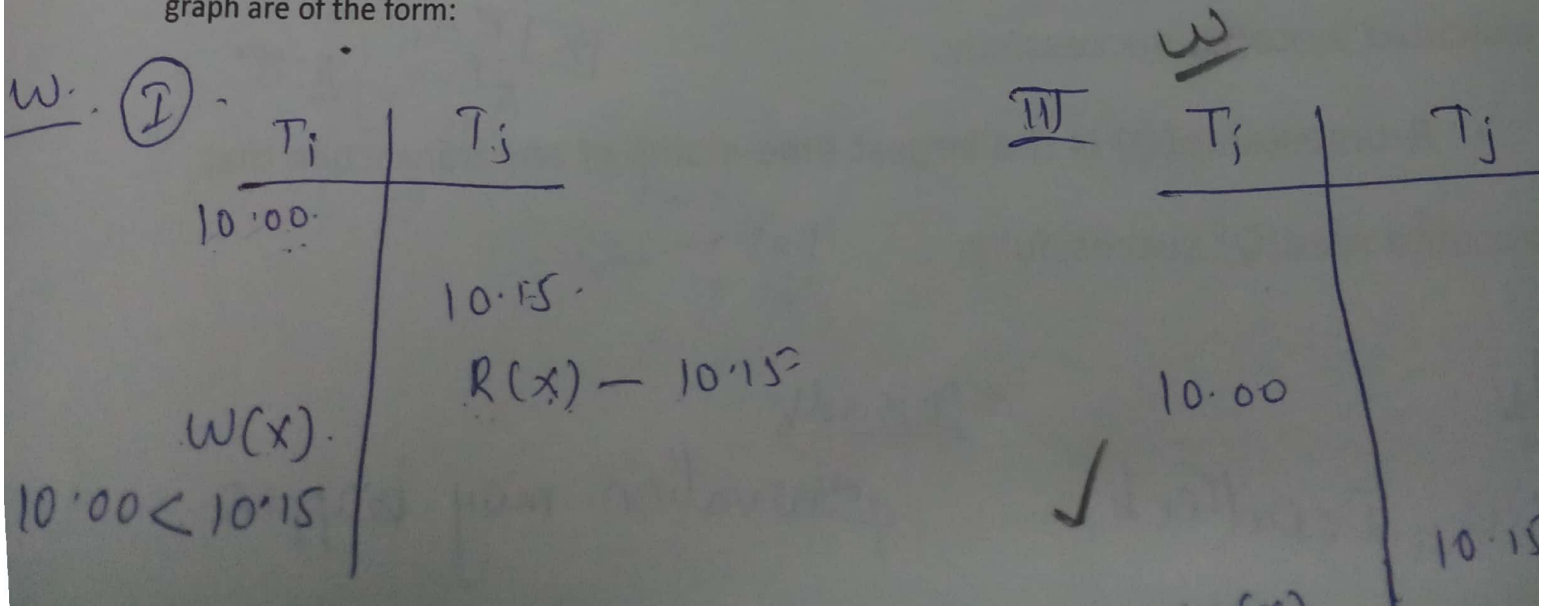
Timestamp ordering protocol works as follows –

- **If a transaction Ti issues a read(X) operation –**
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected.
  - If TS(Ti) >= W-timestamp(X)
    - Operation executed.
  - All data-item timestamps updated.
- **If a transaction Ti issues a write(X) operation –**
  - If TS(Ti) < R-timestamp(X)
    - Operation rejected.
  - If TS(Ti) < W-timestamp(X)
    - Operation rejected and Ti rolled back.
  - Otherwise, operation executed.



Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



- Thus, there will be no cycles in the precedence graph.
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

Problem with timestamp-ordering protocol:

- Suppose Ti aborts, but Tj has read a data item written by Ti
- Then Tj must abort; if Tj had been allowed to commit earlier, the schedule is not recoverable.
- Further, any transaction that has read a data item written by Tj must abort
- This can lead to cascading rollback --- that is, a chain of rollbacks

Solution:

- A transaction is structured such that its writes are all performed at the end of its processing
- All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
- A transaction that aborts is restarted with a new timestamp

Thomas' Write Rule

This rule states if TS(Ti) < W-timestamp(X), then the operation is rejected and T_i is rolled back.

Time-stamp ordering rules can be modified to make the schedule view serializable.

Instead of making T_i rolled back, the 'write' operation itself is ignored.

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.

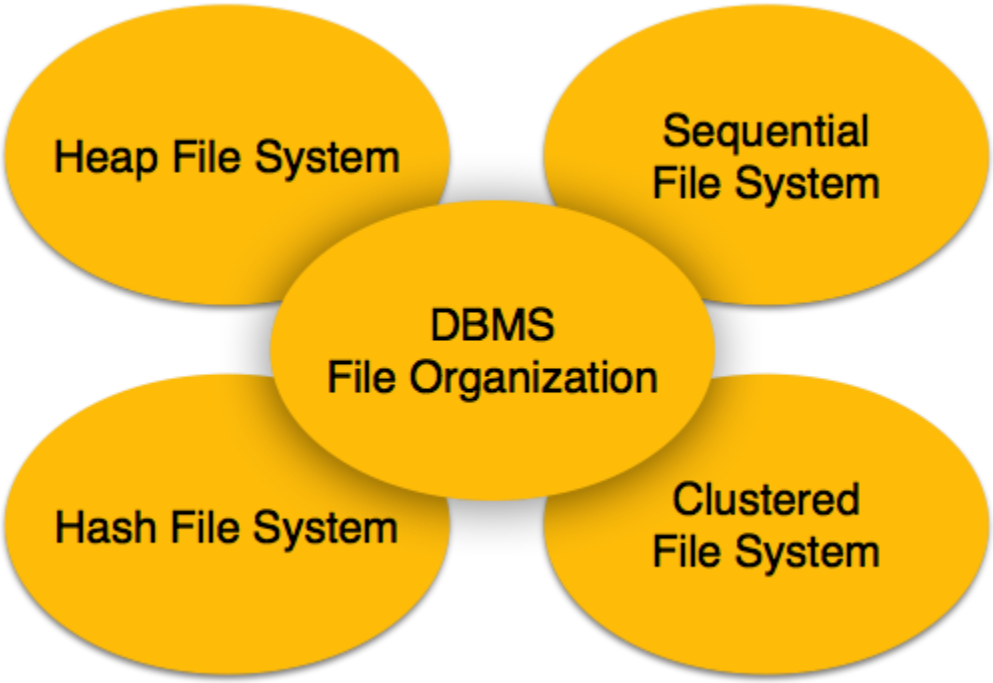
- When Ti attempts to write data item Q, if TS(Ti) < Wtimestamp(Q), then Ti is attempting to write an obsolete value of {Q}. Hence, rather than rolling back Ti as the timestamp ordering protocol would have done, this {write} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

Storage Structure

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.

File Organization

File Organization defines how file records are mapped onto disk blocks. We have four types of File Organization to organize file records –



Heap File Organization

When a file is created using Heap File Organization, the Operating System allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of the software to manage the records. Heap File does not support any ordering, sequencing, or indexing on its own.

Sequential File Organization

Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization, records are placed in the file in some

sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.

Hash File Organization

Hash File Organization uses Hash function computation on some fields of the records. The output of the hash function determines the location of disk block where the records are to be placed.

Clustered File Organization

Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in the same disk block, that is, the ordering of records is not based on primary key or search key.

Indexing

We know that data is stored in the form of records. Every record has a key field, which helps it to be recognized uniquely.

Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to what we see in books.

Indexing is defined based on its indexing attributes. Indexing can be of the following types –

- **Primary Index** – Primary index is defined on an ordered data file. The data file is ordered on a **key field**. The key field is generally the primary key of the relation.
- **Secondary Index** – Secondary index may be generated from a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
- **Clustering Index** – Clustering index is defined on an ordered data file. The data file is ordered on a non-key field.

Ordered Indexing is of two types –

- Dense Index
- Sparse Index

Dense Index

In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index records contain search key value and a pointer to the actual record on the disk.

Sparse Index

In sparse index, index records are not created for every search key. An index record here contains a search key and an actual pointer to the data on the disk. To search a record, we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following the index, then the system starts sequential search until the desired data is found.

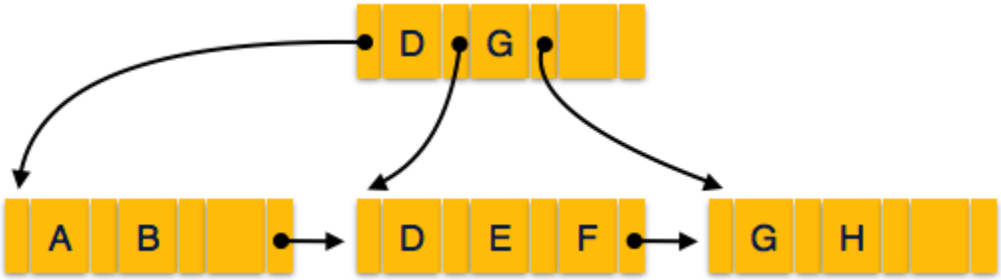
B^+ Tree

A B^+ tree is a balanced binary search tree that follows a multi-level index format. The leaf nodes of a B^+ tree denote actual data pointers. B^+ tree ensures that all leaf nodes remain at the same height, thus balanced.

Additionally, the leaf nodes are linked using a link list; therefore, a B^+ tree can support random access as well as sequential access.

Structure of B^+ Tree

Every leaf node is at equal distance from the root node. A B^+ tree is of the order **n** where **n** is fixed for every B^+ tree.



Internal nodes –

- Internal (non-leaf) nodes contain at least ⌈n/2⌉ pointers, except the root node.
- At most, an internal node can contain **n** pointers.

Leaf nodes –

- Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values.
- At most, a leaf node can contain **n** record pointers and **n** key values.
- Every leaf node contains one block pointer **P** to point to next leaf node and forms a linked list.

B^+ Tree Insertion

- B^+ trees are filled from bottom and each entry is done at the leaf node.
- If a leaf node overflows –
    - Split node into two parts.
    - Partition at $\mathbf{i = \lfloor (m+1)/2 \rfloor}$.
    - First **i** entries are stored in one node.
    - Rest of the entries (i+1 onwards) are moved to a new node.
    - $\mathbf{i^{th}}$ key is duplicated at the parent of the leaf.

- If a non-leaf node overflows –
    - Split node into two parts.
    - Partition the node at **i = ⌈(m+1)/2⌉**.
    - Entries up to **i** are kept in one node.
    - Rest of the entries are moved to a new node.

B^+ Tree Deletion

- B^+ tree entries are deleted at the leaf nodes.
- The target entry is searched and deleted.
    - If it is an internal node, delete and replace with the entry from the left position.
- After deletion, underflow is tested,
    - If underflow occurs, distribute the entries from the nodes left to it.
- If distribution is not possible from left, then
    - Distribute from the nodes right to it.
- If distribution is not possible from left or from right, then
    - Merge the node with left and right to it.

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

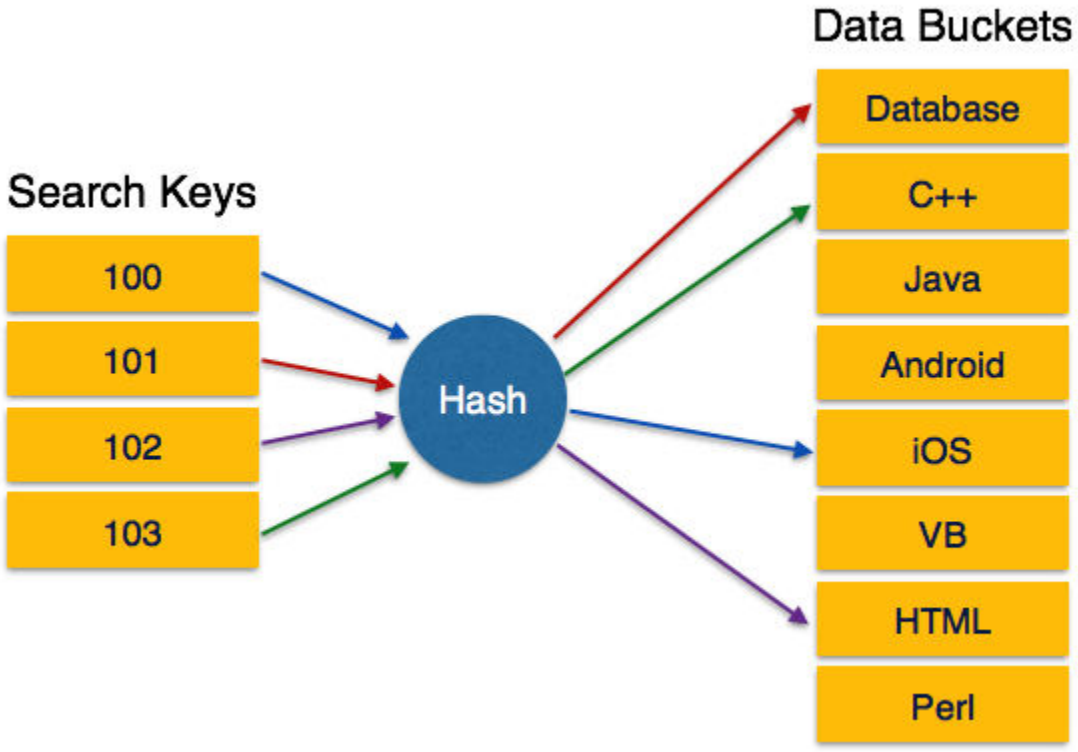
Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

- **Hash Function** – A hash function, **h,** is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.



Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

  Bucket address = h(K)

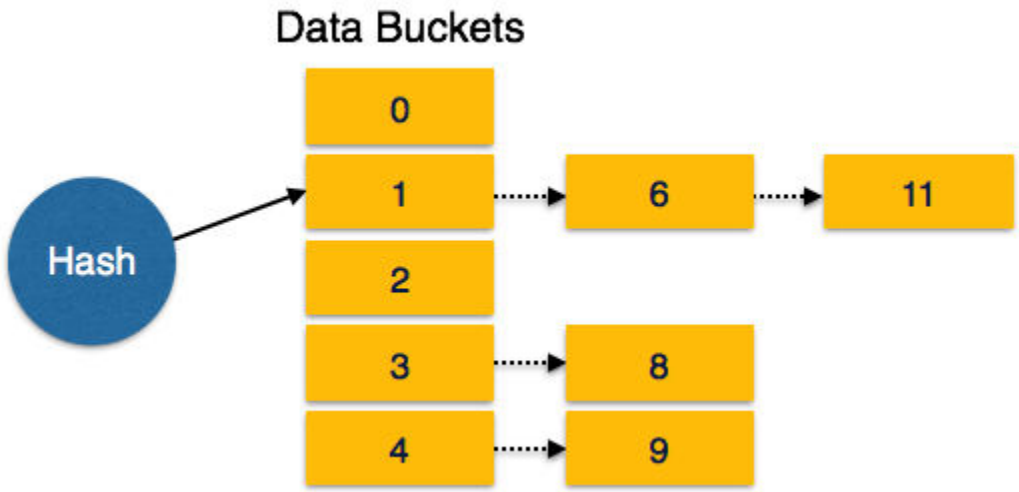
- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.

- **Delete** – This is simply a search followed by a deletion operation.

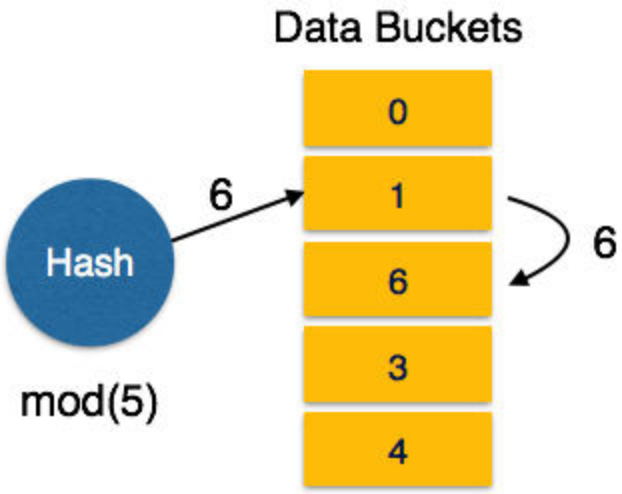
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



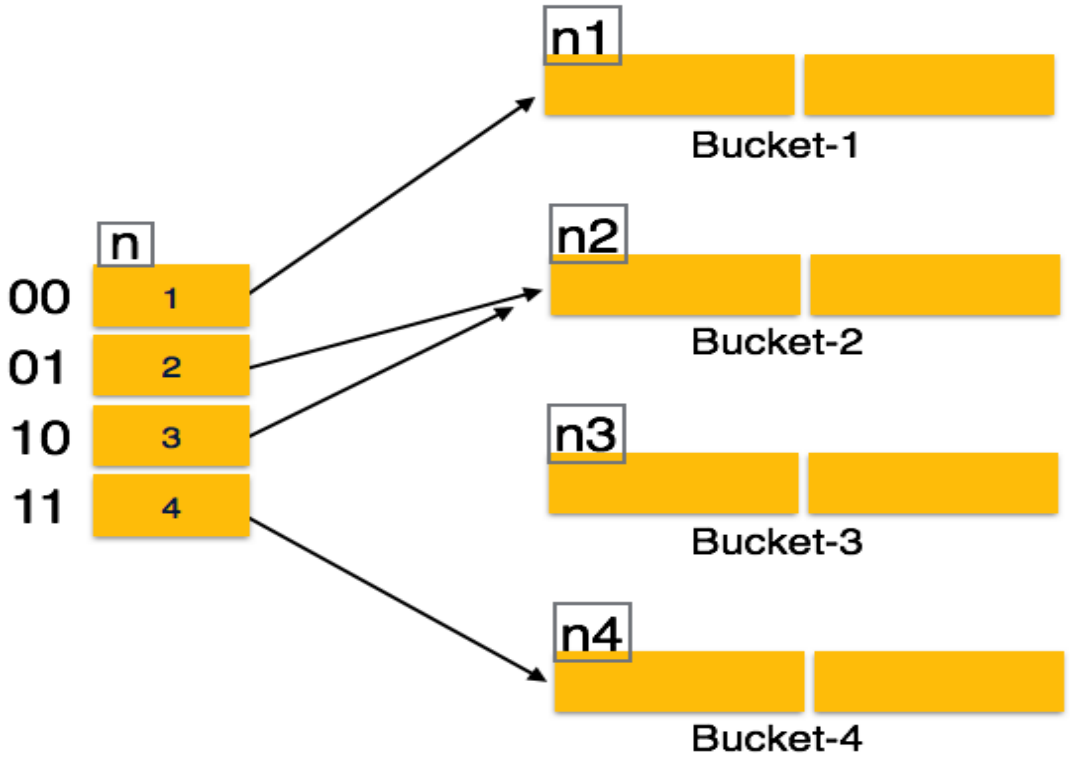
- **Linear Probing** – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



Organization

The prefix of an entire hash value is taken as a hash index. Only a portion of the hash value is used for computing bucket addresses. Every hash index has a depth value to signify how many bits are used for computing a hash function. These bits can address 2n buckets. When all these bits are consumed – that is, when all the buckets are full – then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying** – Look at the depth value of the hash index and use those bits to compute the bucket address.
- **Update** – Perform a query as above and update the data.
- **Deletion** – Perform a query to locate the desired data and delete the same.
- **Insertion** – Compute the address of the bucket
    - If the bucket is already full.

        - Add more buckets.
        - Add additional bits to the hash value.
        - Re-compute the hash function.
    - Else
        - Add data to the bucket,
    - If all the buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and the queries require a range of data. When data is discrete and random, hash performs the best.

Hashing algorithms have high complexity than indexing. All hash operations are done in constant time.

B - Trees

In a binary search tree, AVL Tree, Red-Black tree etc., every node can have only one value (key) and maximum of two children but there is another type of search tree called B-Tree in which a node can store more than one value (key) and it can have more than two children. B-Tree was developed in the year of 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

B-Tree can be defined as follows...

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

Here, number of keys in a node and number of children for a node is depend on the order of the B-Tree. Every B-Tree has order.

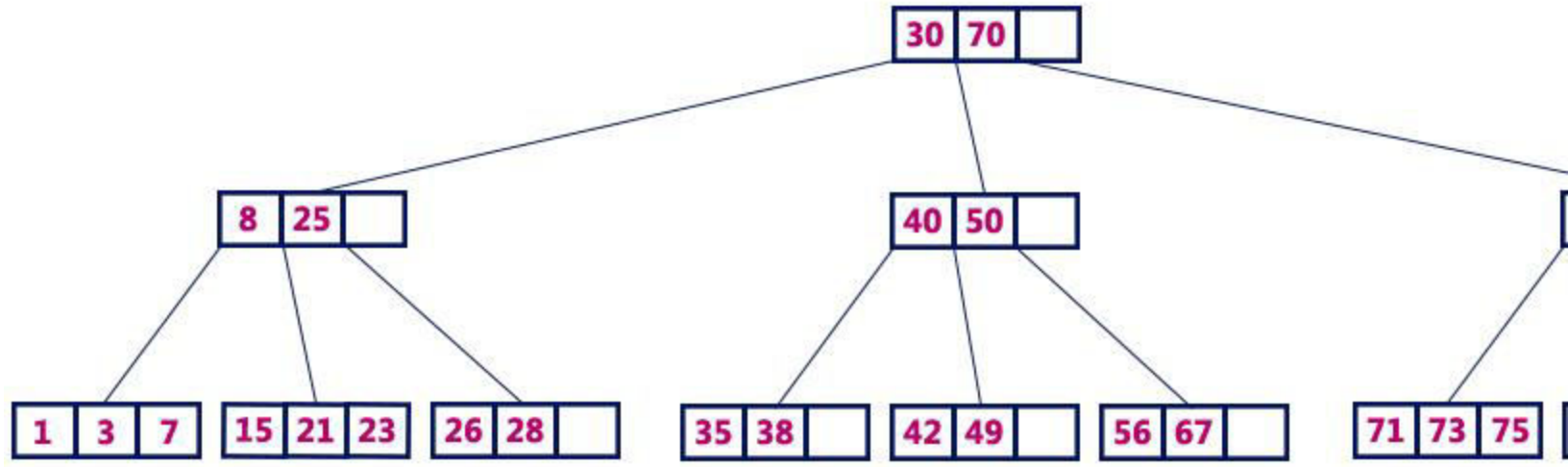
B-Tree of Order m has the following properties...

- Property #1 - All the **leaf nodes** must be **at same level**.
- Property #2 - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- Property #4 - If the root node is a non leaf node, then it must have **at least 2** children.
- Property #5 - A non leaf node with **n-1** keys must have **n** number of children.
- Property #6 - All the **key values within a node** must be in **Ascending Order**.

For example, B-Tree of Order 4 contains maximum 3 key values in a node and maximum 4 children for a node.

Example

B-Tree of Order 4



Operations on a B-Tree

The following operations are performed on a B-Tree...

1. **Search**
2. **Insertion**
3. **Deletion**

Search Operation in B-Tree

In a B-Ttree, the search operation is similar to that of Binary Search Tree. In a Binary search tree, the search process starts from the root node and every time we make a 2-way decision (we go to either left subtree or right subtree). In B-Tree also search process starts from the root node but every time we make n-way decision where n is the total number of children that node has. In a B-Ttree, the search operation is performed with **O(log n)** time complexity. The search operation is performed as follows...

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with first key value of root node in the tree.

- **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
- **Step 4:** If both are not matching, then check whether search element is smaller or larger than that key value.
- **Step 5:** If search element is smaller, then continue the search process in left subtree.
- **Step 6:** If search element is larger, then compare with next key value in the same node and repeate step 3, 4, 5 and 6 until we found exact match or comparision completed with last key value in a leaf node.
- **Step 7:** If we completed with last key value in a leaf node, then display "Element is not found" and terminate the function.

Insertion Operation in B-Tree

In a B-Tree, the new element must be added only at leaf node. That means, always the new keyValue is attached to leaf node only. The insertion operation is performed as follows...

- **Step 1:** Check whether tree is Empty.
- **Step 2:** If tree is **Empty**, then create a new node with new key value and insert into the tree as a root node.
- **Step 3:** If tree is **Not Empty**, then find a leaf node to which the new key value cab be added using Binary Search Tree logic.
- **Step 4:** If that leaf node has an empty position, then add the new key value to that leaf node by maintaining ascending order of key value within the node.
- **Step 5:** If that leaf node is already full, then **split** that leaf node by sending middle value to its parent node. Repeat tha same until sending value is fixed into a node.

- **Step 6:** If the spilting is occuring to the root node, then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a **B-Tree of Order 3** by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

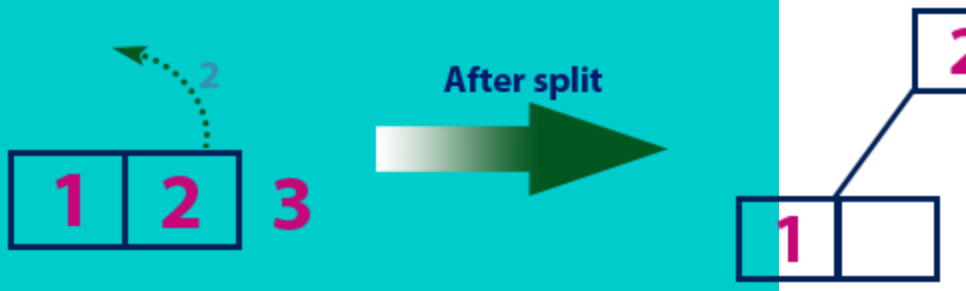
Since '1' is the first element into the tree that is inserted into a new node. It acts as th

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that noc
node has an empty position. So, new element (2) can be inserted at that empty posi

insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that noc
node doesn't has an empty position. So, we split that node by sending middle value
node doesn't has parent. So, this middle value becomes a new root node for the tre



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the ri
value '3' and it has an empty position. So, new element (4) can be inserted at that e

← Previous

Overview of Storage and Indexing

Data on External Storage

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can only read pages in sequence
  - Cheaper than disks; used for archival storage
- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - Indexes are data structures that allow us to find the record ids of records with given values in index search key fields
- Architecture: Buffer manager stages **pages** from external storage to main memory buffer pool. File and index layers make calls to the buffer manager. Page: typically 4 Kbytes.

Alternative File Organizations

Many alternatives exist, *each ideal for some situations, and not so good in others:*

- Heap (random order) files: Suitable when typical access is a file scan retrieving all records.
- Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.
- Indexes: Data structures to organize records via trees or hashing.
  - Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields
  - Updates are much faster than in sorted files.

Indexes

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

Index Classification

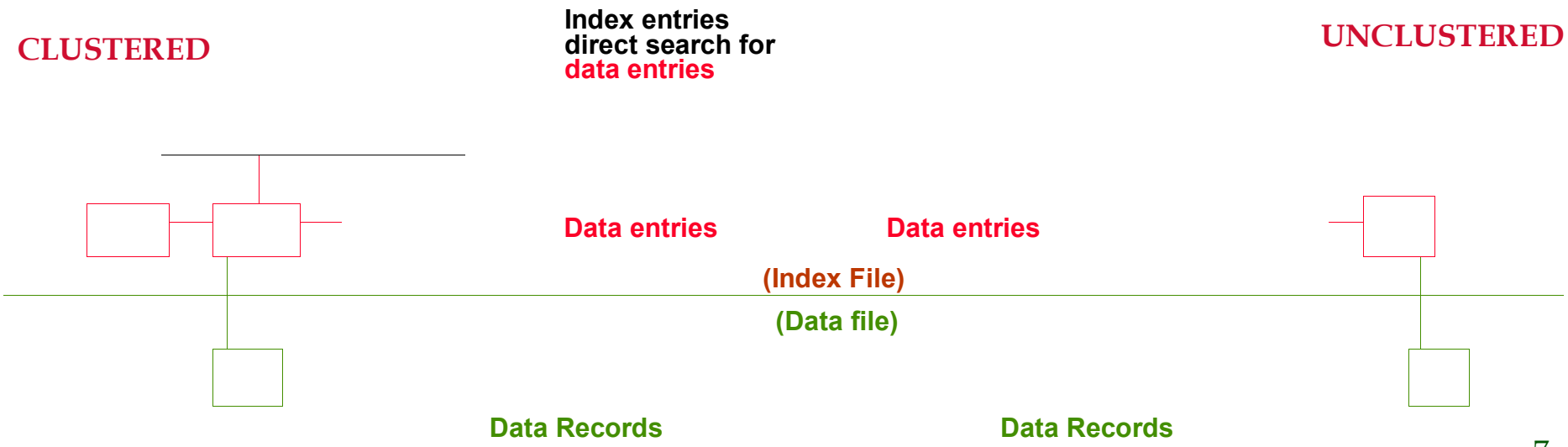
- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key contains a candidate key.
- *Clustered* vs. *unclustered*: If order of data records is the same as order of data entries, then called **clustered** index.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Index Classification

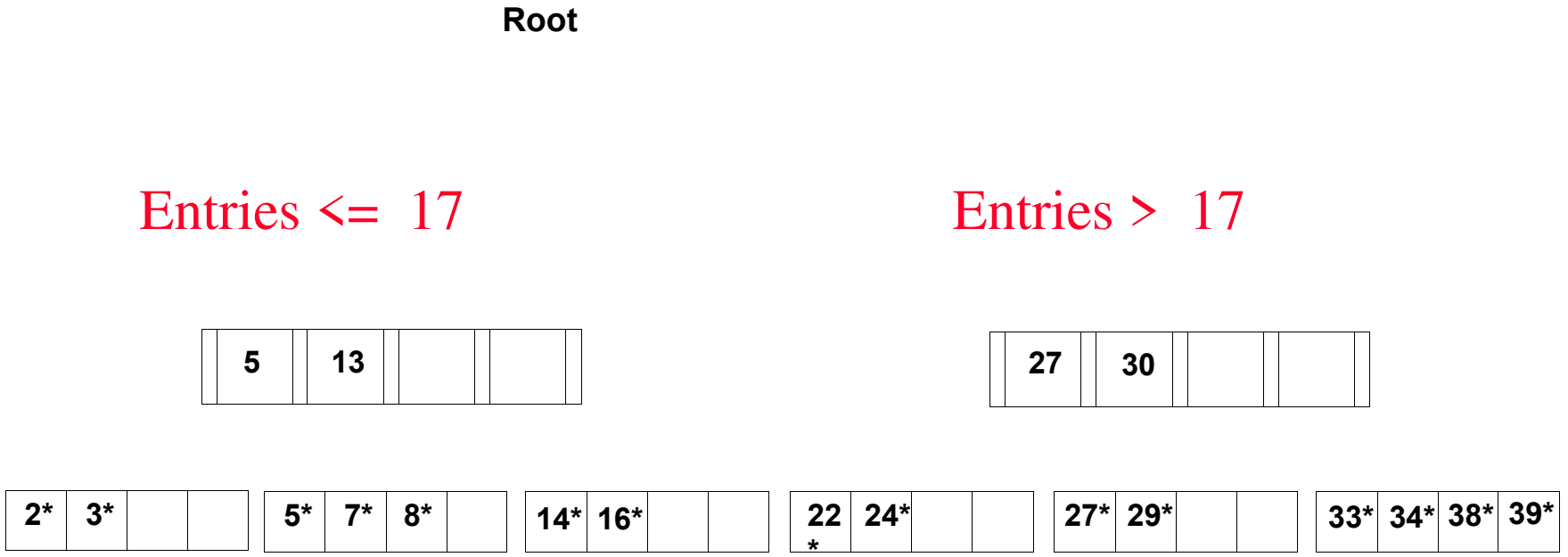
- *Dense* vs *Sparse*: If there is an entry in the index for each key value -> **dense index** (unclustered indices are dense). If there is an entry for each page -> **sparse index**.

Clustered vs. Unclustered Index

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
  - Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



Example B+ Tree



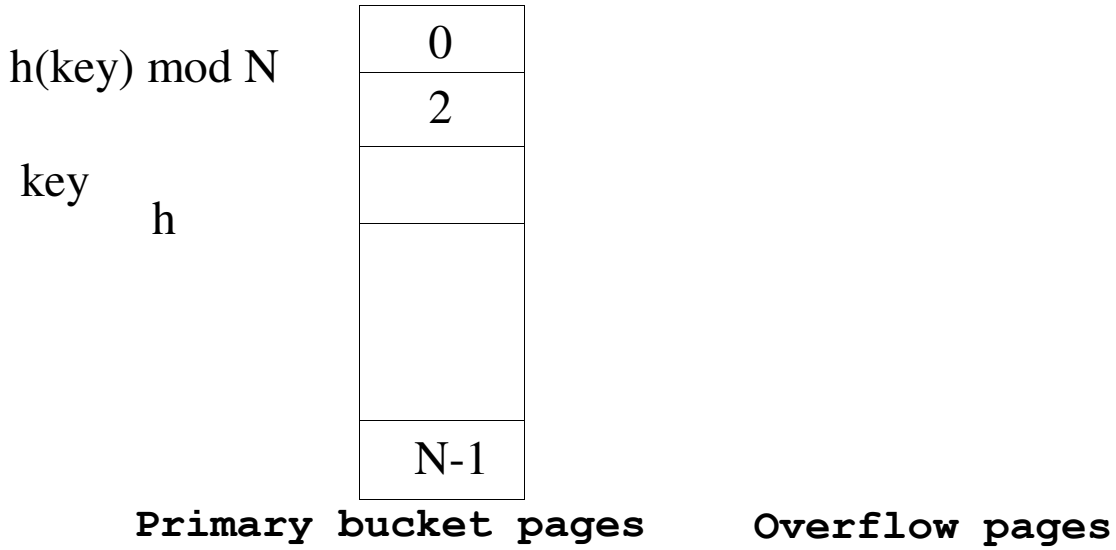
- Good for range queries.
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes. All leaves at he same height.

Hash-Based Indexes

- Good for equality selections.
  - Index is a collection of *buckets*. Bucket = *primary* page plus zero or more *overflow* pages.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r*.
- Buckets may contain the data records or just the rids.
- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(*k*) mod N = bucket to which data entry with key *k* belongs. (N = # of buckets)
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this.



Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) usually works well.
  - a and b are constants; lots known about how to tune **h**.

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

- **B:** The number of data pages
- **R:** Number of records per page
- **D:** (Average) time to read or write disk page
- Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.
- Average-case analysis; based on several simplistic assumptions.

✉ *Good enough to show the overall trends!*

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on <*age, sal*>
- Clustered B+ tree file, Alternative (1), search key <*age, sal*>
- Heap file with unclustered B + tree index on search key <*age, sal*>
- Heap file with unclustered hash index on search key <*age, sal*>

Choice of Indexes

- What indexes should we create?

- One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.
  - Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!
  - For now, we discuss simple 1-table queries.
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

- Attributes in WHERE clause are candidates for index keys.
  - Exact match condition suggests hash index.
  - Range query suggests tree index.
    - Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions.
- Try to choose indexes that benefit as many queries as possible. Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - **How selective is the condition?**
  - Is the index clustered?

```
SELECT  E.dno
FROM  Emp E
WHERE  E.age>40
```

- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!

```
FROM  Emp E
HERE  E.age>10
ROUP BY E.dno
```

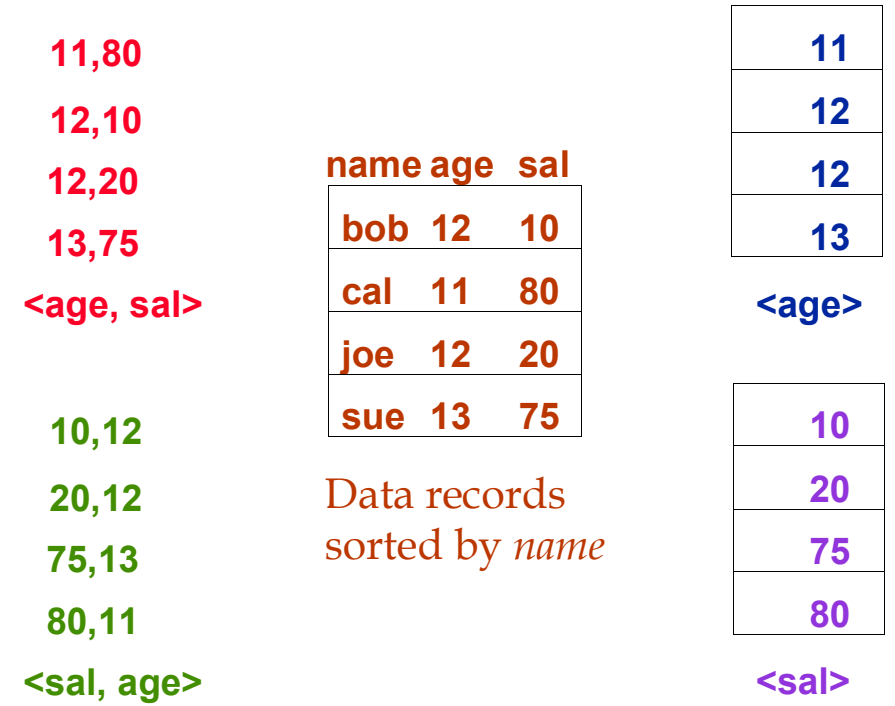
- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT  E.dno
FROM  Emp E
WHERE  E.hobby=Stamps
```

Indexes with Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10
- Data entries in index sorted by search key to support range queries.
- Order or attributes is relevant.

Examples of composite key



Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!
- Composite indexes are larger, updated more often.

Summary

- Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.
  - Choice orthogonal to *indexing technique* used to locate data entries with a given key value.
- Can have several indexes on a given file of data records, each with a different search key.
- Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision; only one index on a given relation can be clustered!
  - Order of fields in composite index key can be important.

Chapter 5 - Tree Indexes

Given a dynamic file (many insertions and deletions)
we would like to do frequent independent fetches, consider

- an unsorted file
- a sorted file
- having an index (look up table)

Inverted Files:

- A simplest index structure that is in the form of an ordered list where each each entry is a (key, ptr) pair.
- difficult to maintain
  - After insertion and deletions, whole file needs to be shifted.

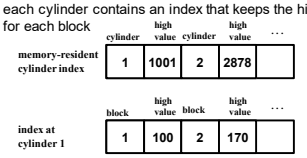
Most DBMSs use B+-trees and hash table utilities.

- we must learn how they work and what performance to expect.

ISAM (Indexed Sequential Access Method)

- the most extensively used indexing method in last decade.
- mostly promoted by IBM and INGRES DBMS, but obsolete today.

- ISAM is simple and efficient as long as no new records are added
  It contains
  - a memory-resident cylinder index that keeps the highest valued key for each cylinder
  - each cylinder contains an index that keeps the highest valued key for each block



$$T_F = \underbrace{r + s + btt}_{\text{Time to fetch the index on cylinder}} + \underbrace{r + btt}_{\text{Time to fetch correct block}}$$

$$T_x = \text{same as the sorted file}$$

(Actually a little bit longer since some space left on each cylinder for overflow.)

Disadvantages of ISAM:

- As new records are added, the ISAM file degrades in performance.
- It has to be reorganized at high cost.

Overflow Chains in ISAM

- We start with some empty tracks in each cylinder for overflow
- When a new record is added, old records are shifted to make place for the new one.
- The record which had the largest key in the block is moved to the overflow area.
- When the overflow area fills up, overflow is written to another cylinder
- Eventually the performance gets very slow.

Performance

- performance gets really poor when the distribution of new records could not be predicted in advance - very long overflow chains may occur
- With good prediction, enough space can be reserved in areas which are expected to grow

B+-Trees

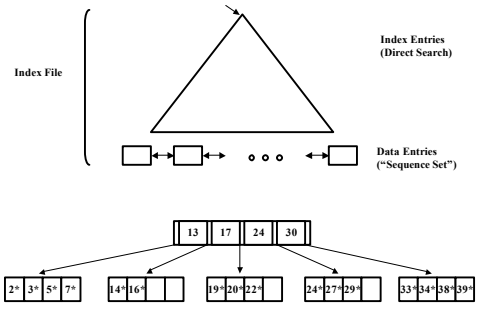
- Most used indexing method today.
- In B+-Trees:
  - nodes tend to have over 100 children
  - all leaves are on the same level
  - leaves contain the actual pointers to data on disk

Any indexing structure which supports an ordering on a large file is likely to be implemented by a B+-tree.

- we can make efficient range queries.

- We shall show how a B+-tree can be used as a secondary or primary indexing method.
- We will look at the costs of fetching, sequential operations, and insertion/deletion.

Structure of a B+-Tree



Example of a B+-Tree, Order v = 2

Definition of a B+-Tree of Order v

- The root has at least two children unless it is a leaf.
- No internal node has more than $2v$ keys.
  - Root may have less keys
  - Internal nodes contain only keys and addresses of nodes on the next lower level.
- All leaves are on the same level.
  - When B+-tree is used as a primary index, the leaves contain the data records.
  - When B+-tree is used as a secondary index, the leaves contain the keys and record addresses.
- An internal node with k keys has $k+1$ children.

Bucket factor (Bkfr) : the # records that can fit in a leaf node.
Fan-out: the average # children of an internal node.

- B+-trees are short and wide.
- The records take up more space than the keys and addresses.
  - Typically internal nodes carry on 100-200 keys, leaves carry on 15 records.
- A primary index determines the way the records are actually stored.
- Clustering index: records are stored together in buckets acc.to the values of the key.
  - The records in a given bucket will have nearby key values.
  - The index only note the lowest or the highest key in a given bucket.
    - For this reason, clustering index, is often called a ***sparse index*** (e.g., ISAM, a B+-tree with data in the leaves)

- A B+-tree can also be used for a secondary index.
  - The records in the file are not grouped in buckets according to the keys of secondary indexes.
  - A secondary index is also a dense index where an entry exists for each record in the file (e.g., a B+-tree where leaves contain keys and addresses of records)
- There may be many secondary indexes for the same file.
- Why not have a secondary index on each field in the file?
  - this would need repeating all the information in the file in the leaves of the trees.
  - with many indexes, update costs becomes high.