LECTURE NOTES ON

DATABASE MANAGEMENT SYSTEMS

Ms. K Laxmi Narayanamma

Assistant Professor

Mr. N Bhaswanth

Assistant Professor

DEPARTMENT OF INFORMATION TECHNOLOGY

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL – 500 043, HYDERABAD

SYLLBUS:

UNIT-I

CONCEPTUAL MODELING

Introduction to file and database systems: Database system structure, data models, introduction to network and hierarchical models, ER model, relational model.

UNIT-II

RELATIONAL APPROACH

Relational algebra and calculus: Relational algebra, selection and projection, set operations, renaming, joins, division, examples of algebra queries, relational calculus, tuple relational calculus, domain relational calculus, expressive power of algebra and calculus.

UNIT-III

BASIC SQL QUERY

SQL data definition; Queries in SQL: updates, views,integrity and security,relational database design. Functional dependencies and normalization for relational databases up to five normal forms.

UNIT-IV

TRANSACTION MANAGEMENT

Transaction processing: Introduction, need for concurrency control, desirable properties of transaction, schedule and recoverability, serializability and schedules, concurrency control; Types of locks: Two phases locking, deadlock, time stamp based concurrency control, recovery techniques , concepts, immediate update, deferred update, shadow paging.

UNIT-V

DATA STORAGE AND QUERY PROCESSING

Record storage and primary file organization, secondary storage devices, operations on files, heap File, sorted files, hashing techniques, and index structures for files; Different types of indexes, B tree, B+ tree, query processing.

UNIT-I

1. INTRODUCTION

- **Data:** data is a collection of raw facts and figures.
- **DataBase:** database is a collection of interrelated data.
- **DBMS:** database is a collection of interrelated data and set of programs can access that system.

1.1. DATABASE SYSTEM APPLICATIONS:

1. Enter Price Information:
   - *Sales:* customers, products, purchases
   - *Accounting:* payments, receipts, account balance, assets.
   - *Human Resources:* employee records, salaries, tax deductions
   - *Manufacturing:* production, inventory, orders, supply chain
   - *Online Retails:* order tracking, customized recommendations
2. *Banking and Finance:* all transactions
   - *Credit card Transaction:* generation of monthly statements.
   - *Finance:* storing information about holdings and sales,
3. *Universities:* registration, grades
4. *Airlines:* reservations, schedules
5. *Telecommunications*: keeping records of calls made, generating monthly bills

1.2. PURPOSE OF DATABASE SYSTEMS:

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

1.2.1. Drawbacks of using file systems to store data:

- Data redundancy and inconsistency: Multiple file formats, duplication of information in different files
- Difficulty in accessing data: Need to write a new program to carry out each new task
- Data isolation: multiple files and formats
- Integrity problems: Hard to add new constraints or change existing ones
- Atomicity of updates: Failures may leave database in an inconsistent state with partial updates carried out. Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent access anomalies: Example: Two people reading a balance and updating it at the same time

- Security problems: Database systems offer solutions to all the above problems

1.3. VIEW OF DATA:

A database is a collection of interrelated data and set of programs that allow users to accessand modify these data. A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data stored and maintained.

1.3.1. Data Abstraction:

Major purpose of DBMS is to provide users with abstract view of data i.e. the system hides certain details of how the data are stored and maintained. Since database system users are not computer trained, developers hide the complexity from users through 3 levels of abstraction, to simplify user's interaction with the system.

➢ Levels of Abstraction

- Physical level of data abstraction: How the data are actually stored. This s the lowest level of abstraction which describes how data are actually stored.

- Logical level of data abstraction: This level hides what data are actually stored in the database and what relations hip exists among them. Describes data stored in database, and the relationships among the data.

- View Level of data abstraction: View provides security mechanism to prevent user from accessing certain parts of database. application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

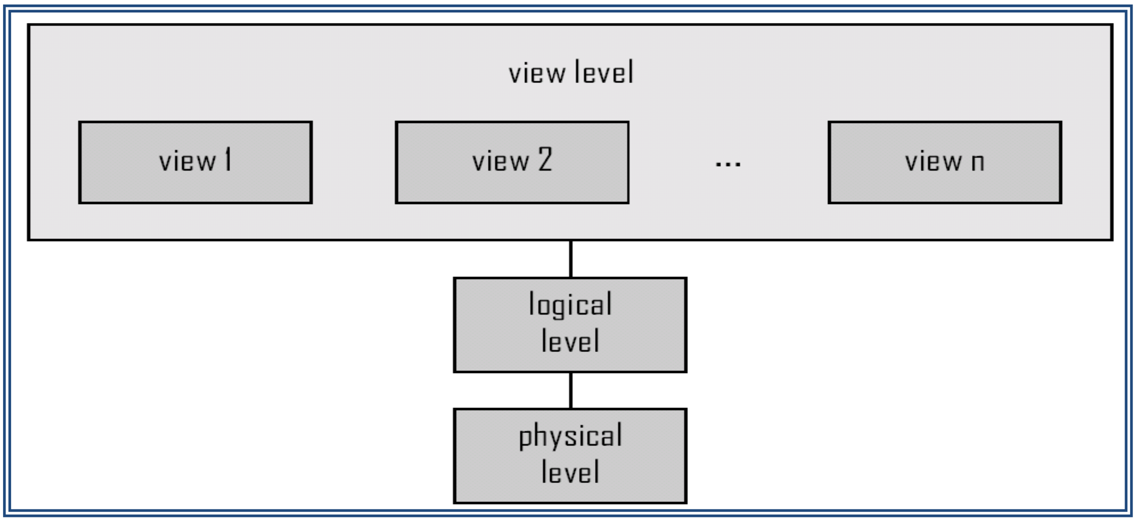


Figure 1.TheThree level of abstraction.

1.3.2. Instances and schemas:

Instance: The collection of information stored in the database at a particular movement is called an instance of the database.

- Similar to types and variables in programming languages

Schema: the overall design of the database is called the database schema.

- Example: 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
- – **Physical schema**: database design at the physical level.
- – **Logical schema**: database design at the logical level.

1.3.3. Data models:

Data Model: Underlying the structure of a database is the data model,
A collection of conceptual tools for describing data, data relationships, data semantics and consistency constraints.

- Relational model: The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name, Tables are also called known as relations.

- Entity-Relationship Model: The Entity –Relationship (E-R) data model uses a collection of basic objects, called entities, and relationships among these objects.

  An entity is a "thing" or "object" in the real world that is distinguishable from other object

- Object-Based Data Models: Object-oriented Programming (especially in Java, C++,or C#).

- Semi structured Data Model: The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes.

- Other older models:
  - – Network Model

– Hierarchical Model

1.4. DATA BASE LANGUAGES:

A Database provides a DDL to specify the database schema and a DML to express database queries and updates.

1.4.1 Data-Manipulation Language

A data-manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model.

The types of access are:

- Retrieval of information stored in the database
- Insertion of new information into the database
- Deletion of information from the database
- Medication of information stored in the database

There are basically two types:

- Procedural DMLs require a user to specify what data are needed and how to get those data.
- Declarative DMLs (also referred to as non procedural DMLs) require user to specify what data are needed without specifying how to get those data.

1.4.2. Data- Definition Language (DDL):

We specify a database schema by a set of definitions expressed by a special language called a data-definition language(DDL).The DDL is also used to specify additional properties of the data.

SQL provides a rich DDL that allows one to define tables, integrity constraints, assertions, etc…

Example:

```
create table account (
  account_number   char(10),
  branch_name      char(10),
  balance          integer)
```

In addition, the DDL statement updates the data dictionary, which contains metadata; the schema of a table is an example of metadata.

1.5. DATA BASE ACCESS FROM APPLICATION PROGRAMS:

Application programs are programs that are used to interact with the database.

To access the database, DML Statements need to be executed form the host language. ‘

There are two ways o do this.

- By Providing an Application Program interface (set of procedures) that can be used to send DML and DDL statement to the database and retrieve the results.(ODBC and JDBC).

- By extending the host language syntax to embed DML calls within the host language program. A special character prefaces DML calls and preprocessor called the DML pre complier ,converts the DML statements to normal procedure calls in the host language

SQL: widely used non-procedural language

Example 1: Find the name of the customer with customer-id 192-83-7465

```
select  customer.customer_name
from    customer
where   customer.customer_id = ‘192-83-7465’
```

Example 2: Find the balances of all accounts held by the customer with customer-Id 192-83-7465.

```
select  account.balance
from     depositor, account
where   depositor.customer_id = ‘192-83-7465’ and
        depositor.account_number = account.account_number
```

Example 3: Find the name of the customer with customer-id 192-83-7465

```
select  customer.customer_name
from    customer
where   customer.customer_id = ‘192-83-7465’
```

1.6. TRANSACTION MANAGEMENT:

- A transaction is a collection of operations that performs a single logical function in a Database application

- Transaction-management component ensures that the database remains in a

Consistent (correct) state despite system failures (e.g., power failures and operating System crashes) and transaction failures.

- Concurrency-control manager controls the interaction among the concurrent Transactions to ensure the consistency of the database.

1.7. THE QUERY PROCESSOR

The query processor components include:

- DDL interpreter, which interprets DDL statements and records the definitions in the data dictionary.
- DML compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs query optimization; that is, it picks the lowest cost evaluation plan from among the alternatives.

Query evaluation engine, which executes low-level instructions generated by the DML compiler.

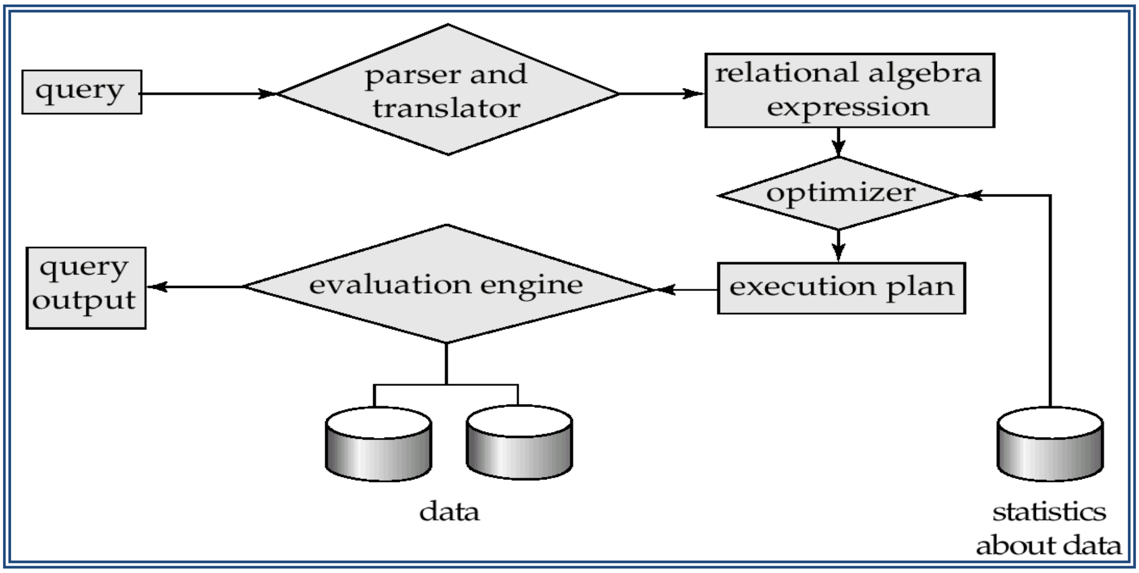


Figure 1.7.The Query Processor

1.8. DATABASE ARCHITECTURE:

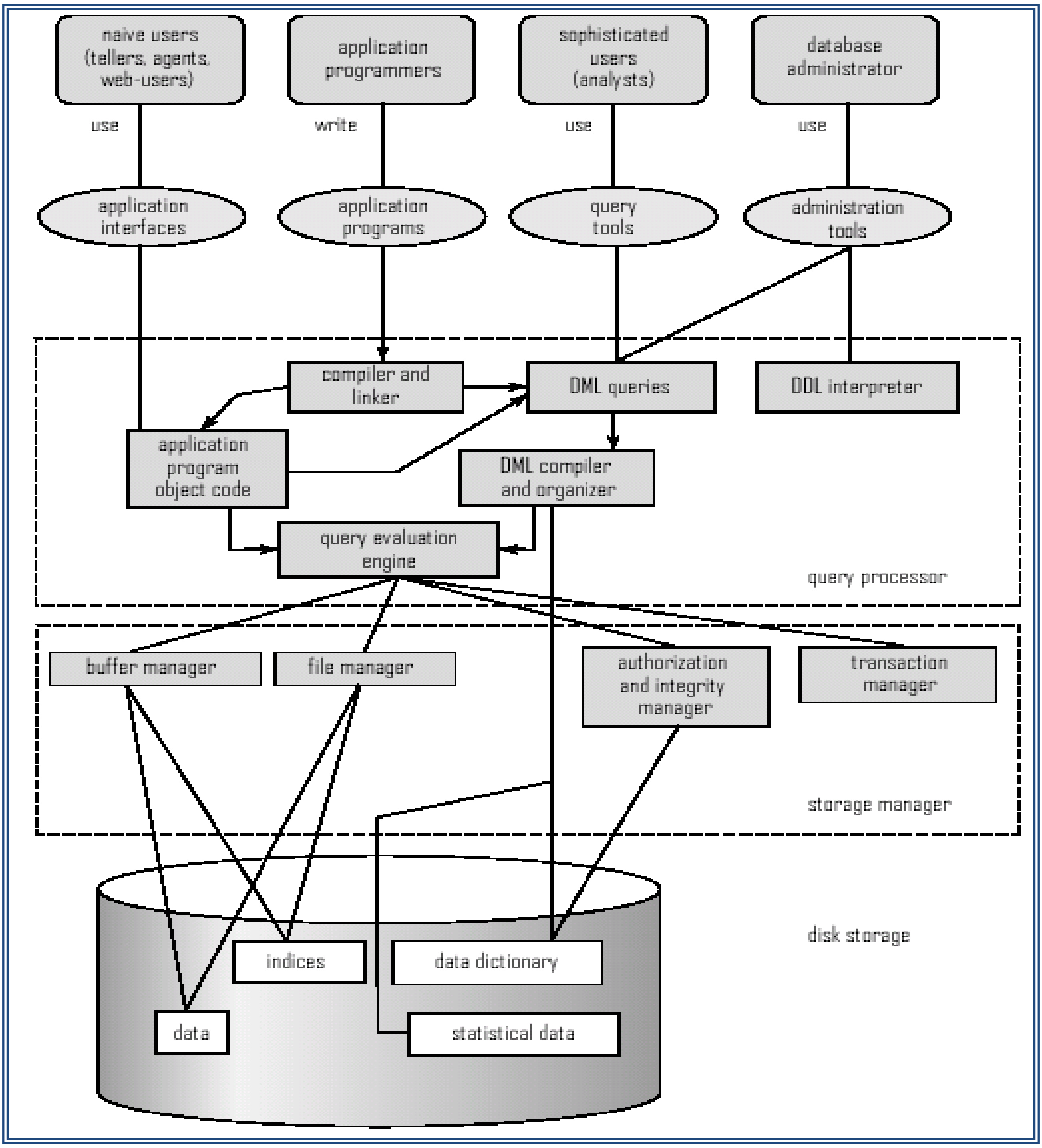
The architecture of database systems is greatly influenced by the underlying computer system on which the database is runs:

Database system can be.

- Centralized

- Client-server
- Parallel (multiple processors and disks)
- Distributed

Overall System Structure



1.8.1. Database Application Architectures:

- Database applications are usually partitioned into two or three parts, as in Figure1.8.1..In a two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements.

- Application program interface standards like ODBC and JDBC are used for interaction between the client and the server. In contrast,

- In a three-tier architecture, the client machine acts as merely a front end and does not contain any direct database calls.

- Instead, the client end communicates with an application server, usually through a forms interface.

- The application server in turn communicates with a database system to access data.

- The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.

- Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

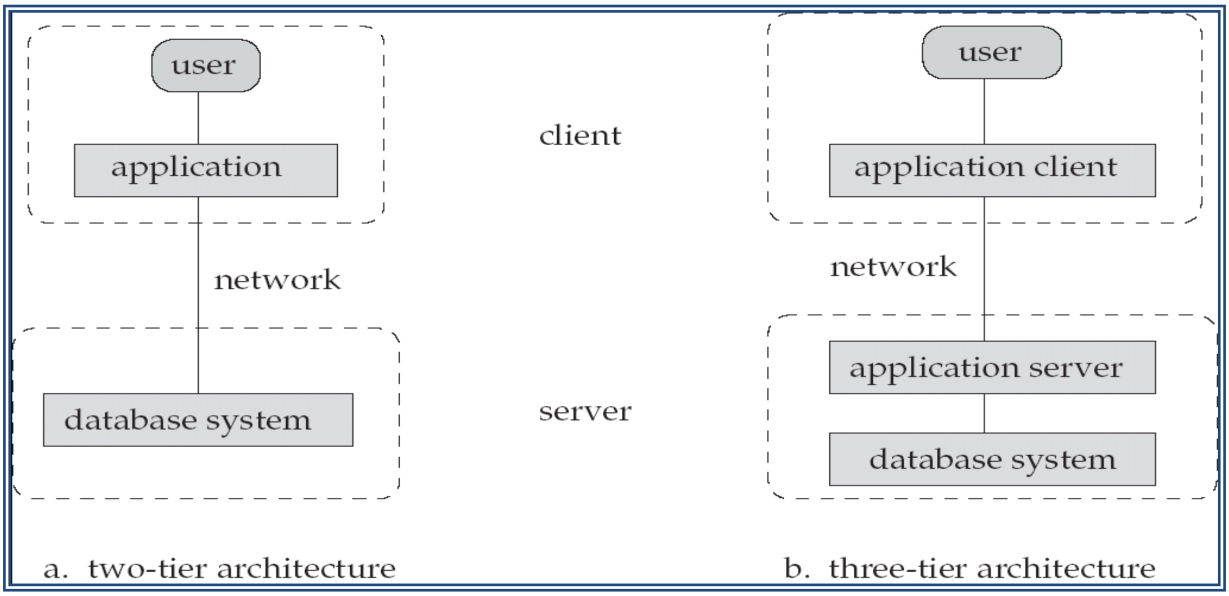


Figure1.8.1.Two-tier and Three –tier architecture.

1.9. DATABASE USERS AND ADMINISTRATORS:

- A primary goal of a database system is to retrieve information from and store new information into the database. People who work with a database can be categorized as database users or database administrators.

1.9.1.Data base Users and User Interfaces

- There are four different types of database system users, differentiated by the way they expect to interact with the system.

- Different types of user interfaces have been designed for the different types of users.

  - Na¨ıve users **are** unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a clerk in the university who needs to add a new instructor to Users are differentiated by the way they expect to interact with the system department A invokes a program called New - hire. This program asks the clerk for the name of the new instructor, her new ID, the name of the department (that is, A), and the salary

  - Application programmers: are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. Rapid application development (RAD) tools are tools that en- able an application programmer to construct forms and reports with minimal programming effort.

  - Sophisticated users: interact with the system with out writing programs. In- stead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

  - Specialized users: are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer aided design systems, knowledge- base and expert systems, systems that store data with complex data types (for example, graphics data and audio data),and environment-modeling systems.

1.9.2. Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a data base administrator (DBA).

The functions of a DBA include:

- Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

- Storage structure and access-method definition.

- Schema and physical-organization modification.
  - Routine maintenance.
  - Periodically backing up the database.
  - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  - Monitoring jobs running on the Data base.

1.10. HISTORY OF DATABASE SYSTEMS:

- 1950s and early 1960s: Data processing using magnetic tapes for storage Tapes provide only sequential access Punched cards for input

- Late 1960s and 1970s:Hard disks allow direct access to data Network and hierarchical data models in widespread use Ted Codd defines the relational data model Would win the ACM Turing Award for this work IBM Research begins System R prototype UC Berkeley begins Ingres prototype High-performance (for the era) transaction processing

- 1980s: Research relational prototypes evolve into commercial systems SQL becomes industry standard Parallel and distributed database systems Object-oriented database systems

- 1990s: Large decision support and data-mining applications Large multi-terabyte data warehouses Emergence of Web commerce

- 2000s: XML and XQuery standards Automated database administration Increasing use of highly parallel database systems Web-scale distributed data storage systems.

EXERCISES

1. List four applications you have used that most likely employed a database system to store persistent data.
2. List four significant differences between a file-processing system and a DBMS.
3. Explain the concept of physical data independence and its importance in database systems.
4. List five responsibilities of a database-management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.
5. What are the five main functions of a database administrator?
6. Explain the difference between two-tier and three-tier architectures. Which is better suited for Web applications? Why?

1.11. INTRODUCTION TO DATABASE DESIGN:

The entity-relationship (ER) data model allows us to describe the data involved in a real-world enterprise in terms of objects and their relationships and is widely used to develop an initial database design.

1.12. DATA BASE DESIGN

The database design process can be divided into six steps. The ER model is most relevant to the first three steps:

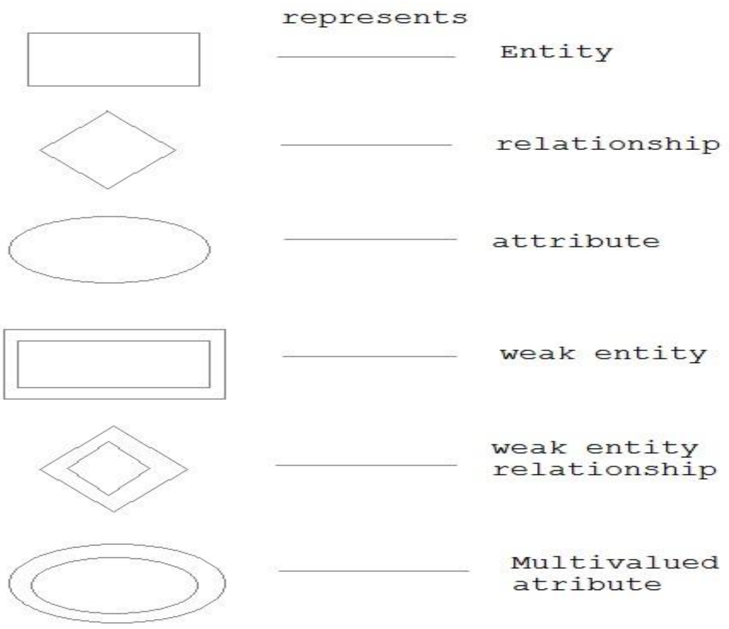
(1) Requirements Analysis: The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. In other words, we must find out what the users want from the database.

(2) Conceptual Database Design: The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.

(3) Logical Database Design: We must choose a DBMS to implement our database design, and convert the conceptual database design into a database schema in the data model of the chosen DBMS. We will only consider relational DBMSs, and therefore, the task in the logical design step is to convert an ER schema into a relational database schema.

The result is a conceptual schema, sometimes called the logical schema, in the relational data model.

1.13.E-R DIAGRAMS



1.14. BEYOND ER DESIGN

The ER diagram is just an approximate description of the data, constructed through a very subjective evaluation of the information collected during requirements analysis.

Once we have a good logical schema, we must consider performance criteria and design the physical schema. Finally, we must address security issues and ensure that users are able to access the data they need, but not data that we wish to hide from them. The remaining three steps of database design are briefly described below:

(4) Schema Refinement: The fourth step in database design is to analyze the collection of relations in our relational database schema to identify potential problems,

(5) Physical Database Design: In this step we must consider typical expected workloads that our database must support and further refine the database design to ensure that it meets desired performance criteria

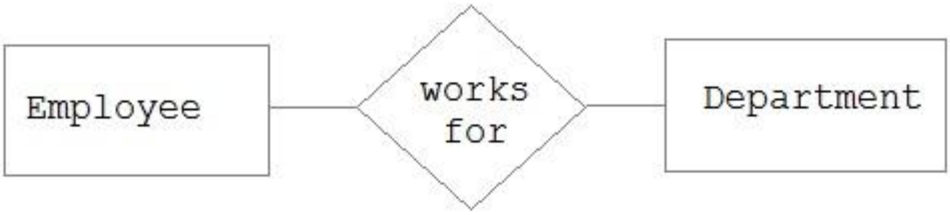
(6) Security Design: In this step, we identify different user groups and different roles played by various users (e.g., the development team for a product, the customer support representatives, the product manager).

For each role and user group, we must identify the parts of the database that they must be able to access and the parts of the database that they should not be allowed to access, and take steps to ensure that they can access only the necessary parts.

1.14.1. ENTITIES, ATTRIBUTES, AND ENTITY SETS

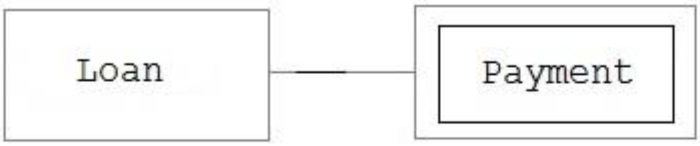
1) Entity

An **Entity** can be any object, place, person or class. In E-R Diagram, an **entity** is represented using rectangles. Consider an example of an Organisation. Employee, Manager, Department, Product and many more can be taken as entities from an Organisation.



Weak Entity

Weak entity is an entity that depends on another entity. Weak entity doen't have key attribute of their own. Double rectangle represents weak entity.



2.Attributes

Entities are represented by means of their properties, called attributes. All attributes have values. For example, a student entity may have name, class, age as attributes.

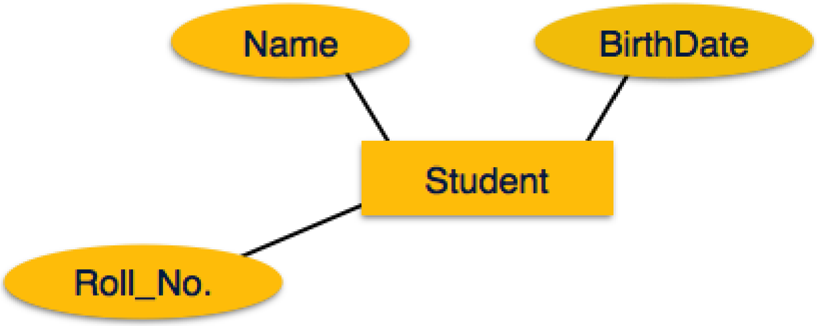
There exist a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

Types of attributes

Attributes are properties of entities. Attributes are represented by means of eclipses. Every eclipse represents one attribute and is directly connected to its entity (rectangle).

- **Simple attribute:**

  Simple attributes are atomic values, which cannot be divided further. For example, student's phone-number is an atomic value of 10 digits.

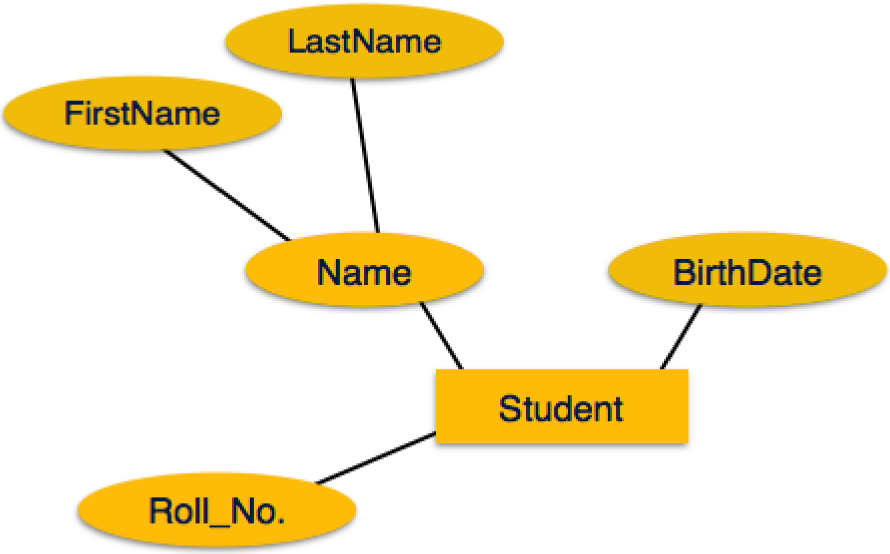


[*Image: Simple Attributes*]

- **Composite attribute:**

  Composite attributes are made of more than one simple attribute. For example, a student's complete name may have first_name and last_name.

If the attributes are ***composite***, they are further divided in a tree like structure. Every node is then connected to its attribute. That is composite attributes are represented by eclipses that are connected with an eclipse.



[*Image: Composite Attributes*]

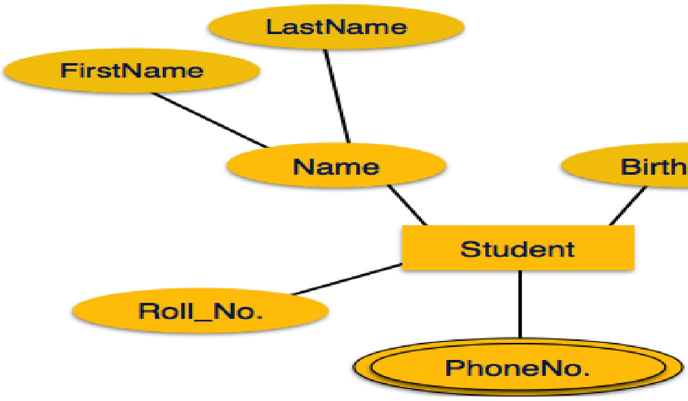
- **Single-valued attribute:**

  Single valued attributes contain on single value. For example: Social_Security_Number.

- **Multi-value attribute:**

  Multi-value attribute may contain more than one values. For example, a person can have more than one phone numbers, email_addresses etc.

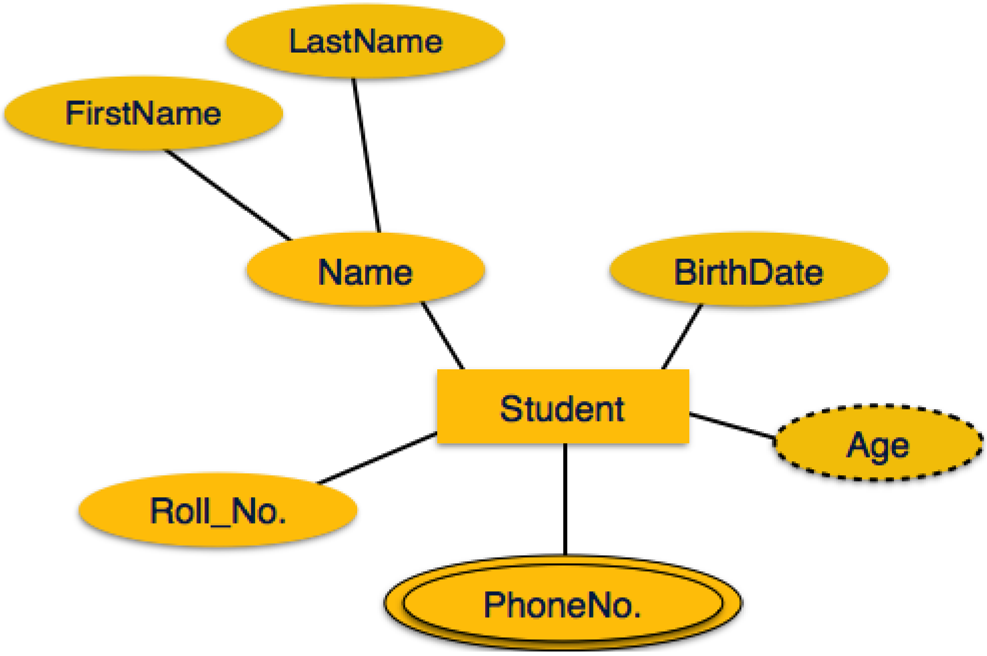
  Multivalued attributes are depicted by double eclipse.



- **Derived attribute:**

  Derived attributes are attributes, which do not exist physical in the database, but there values are derived from other attributes presented in the database. For example, average_salary in a department should be saved in database instead it can be derived. For another example, age can be derived from data_of_birth.

  Derived attributes are depicted by dashed eclipse.



[*Image: Derived Attributes*]

1.14.2. RELATIONSHIPS AND RELATIONSHIP SETS

The association among entities is called relationship. For example, employee entity has relation work

s_at with department. Another example is for student who enrolls in some course. Here, Works_at and Enrolls are called relationship.

Relationship Set:

Relationship of similar type is called relationship set. Like entities, a relationship too can have attributes. These attributes are called descriptive attributes.

Degree of relationship

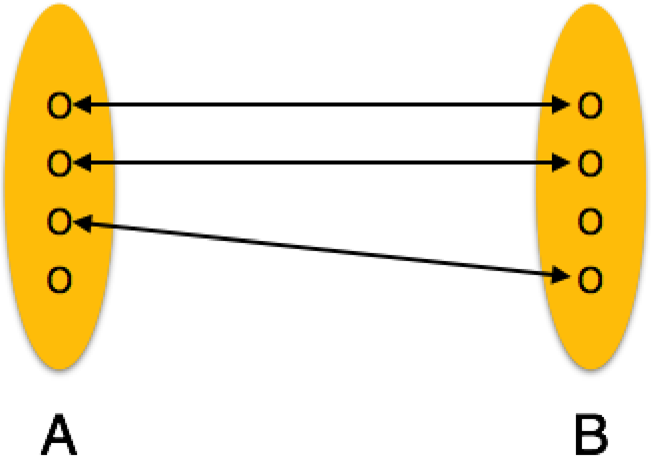
The number of participating entities in an relationship defines the degree of the relationship.

- Binary = degree 2
- Ternary = degree 3
- n-ary = degree

Mapping Cardinalities:

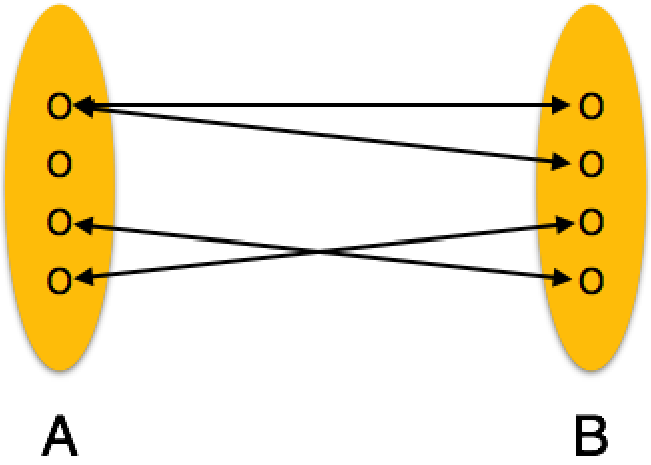
Cardinality defines the number of entities in one entity set which can be associated to the number of entities of other set via relationship set.

- **One-to-one:** one entity from entity set A can be associated with at most one entity of entity set B and vice versa.



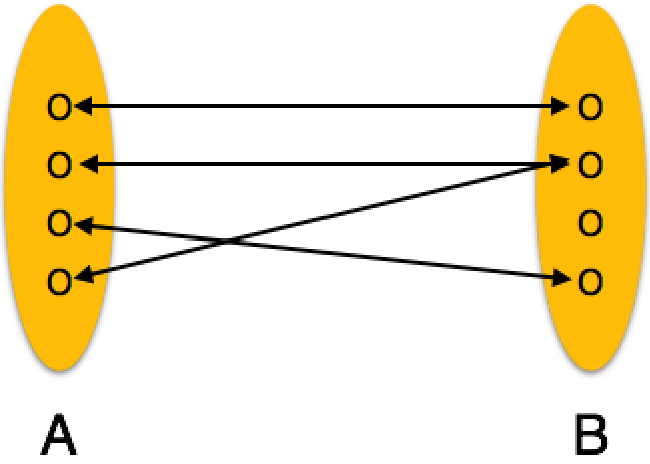
[*Image: One-to-one relation*]

- **One-to-many:** One entity from entity set A can be associated with more than one entities of entity set B but from entity set B one entity can be associated with at most one entity.



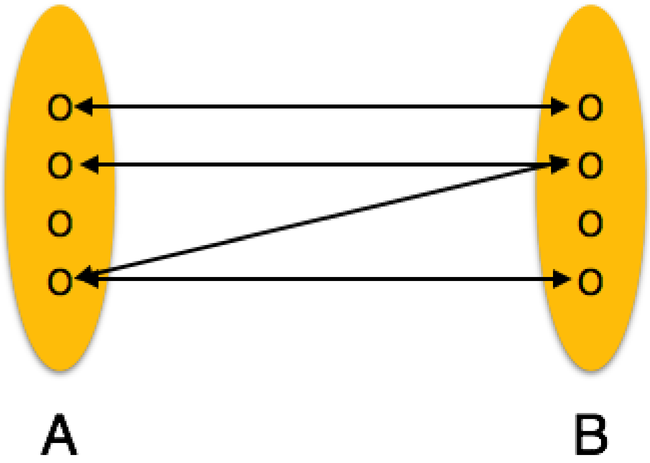
[*Image: One-to-many relation*]

- **Many-to-one:** More than one entities from entity set A can be associated with at most one entity of entity set B but one entity from entity set B can be associated with more than one entity from entity set A.



[*Image: Many-to-one relation*]

- **Many-to-many:** one entity from A can be associated with more than one entity from B and vice versa.



[*Image: Many-to-many relation*]

- **An entity is an object in the real world that is distinguishable from other objects**. Examples include the following: the Green Dragonzord toy, the toy department, the manager of the toy department, the home address of the manager of the toy department.

- **A collection of similar entities is called an entity set.**

- A attribute is an property of an entity.

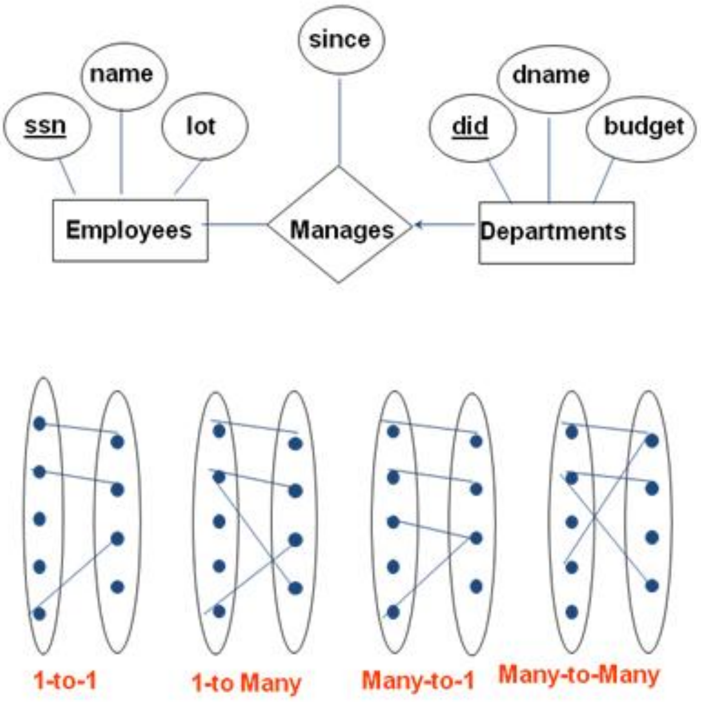
1.15. ADDITIONAL FEATURES OF ER MODEL

» Key Constraints
» Participation Constraints
» Weak Entities

» Class Hierarchies
» Aggregation

1.15.1.KEY CONSTRAINTS

There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there are more than one such minimal subsets, these are called ***candidate keys***.



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

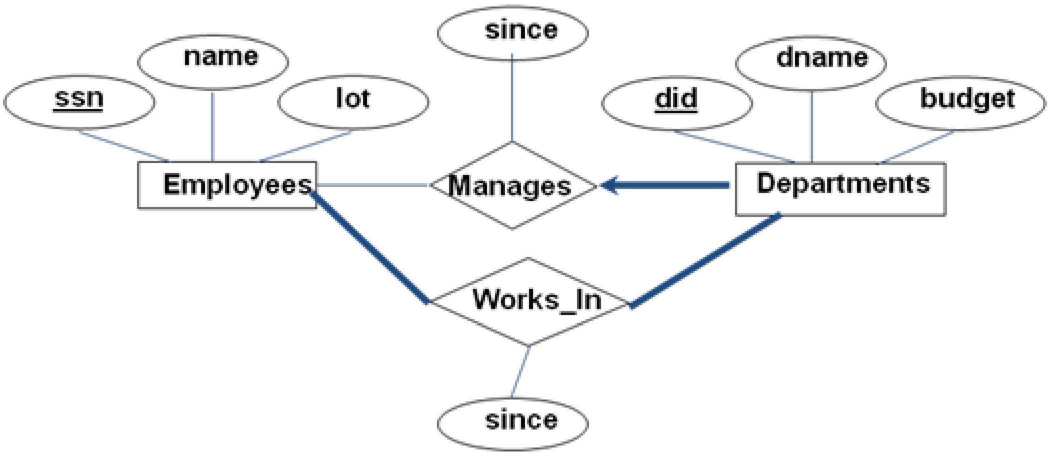
Key constraints forces that:

- in a relation with a key attribute, no two tuples can have identical value for key attributes.
- key attribute can not have NULL values.

Key constrains are also referred to as Entity Constraints.

1.15.2. PARTICIPATION CONSTRAINTS

- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every Departments entity must appear in an instance of the Manages relationship.



1.15.3.WEAK ENTITIES

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
- Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
- Weak entity set must have total participation in this *identifying* relationship set.

1.15.3.1.WEAK ENTITY SETS

➢ ***An entity set that does not have a primary key is referred to as a weak entity set.***

- ***The existence of a weak entity set depends on the existence of a identifying entity set***

it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set Identifying relationship depicted using a double diamond The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that

distinguishes among all the entities of a weak entity set. The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator depict a weak entity set by double rectangles.

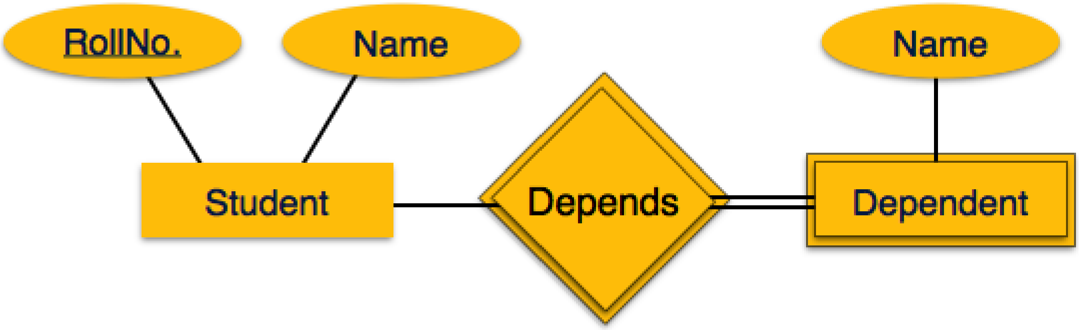
Under line the discriminator of a weak entity set with a dashed line.

1.15.3.2.More Weak Entity Set Examples

In a university, a *course* is a strong entity and a *course_offering* can be modeled as a weak entity The discriminator of *course_offering* would be *semester* (including year) and *section_number* (if there is more than one section). If we model *course_offering* as a strong entity we would model *course_number* as an attribute. Then the relationship with *course* would be implicit in the *course_number* attribute

A weak entity sets is one which does not have any primary key associated with it.

Mapping process (Algorithm):



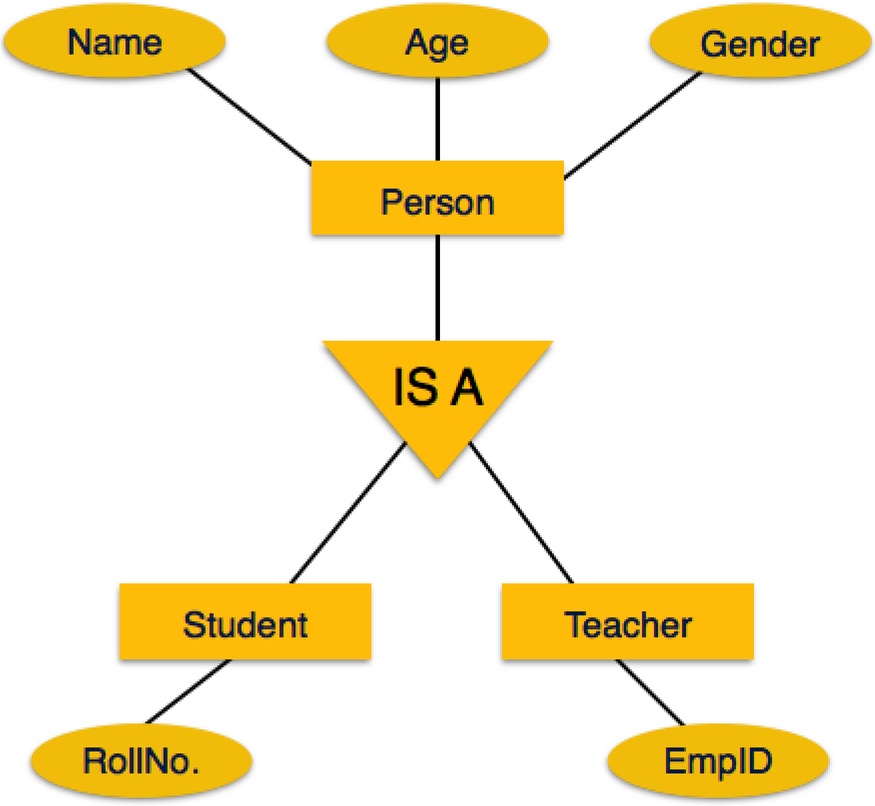
[*Image: Mapping Weak Entity Sets*]

- Create table for weak entity set
- Add all its attributes to table as field
- Add the primary key of identifying entity set
- Declare all foreign key constraints

1.15.4. CLASS HIERARCHIES

- ***Classifying the entities in an entity set into sub classes***
- ER specialization or generalization comes in the form of hierarchical entity sets.

Mapping process (Algorithm):

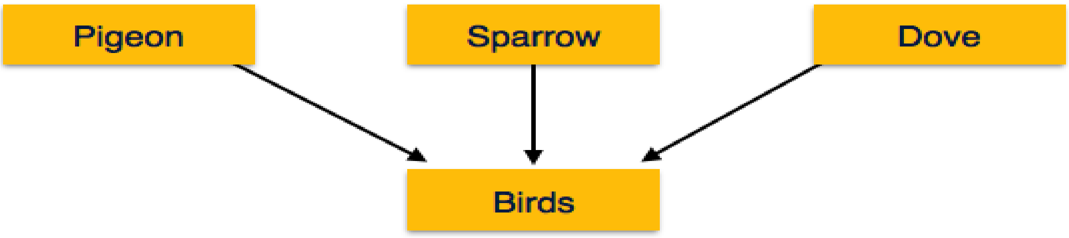


[*Image: Mapping hierarchical entities*]

- Create tables for all higher level entities
- Create tables for lower level entities
- Add primary keys of higher level entities in the table of lower level entities
- In lower level tables, add all other attributes of lower entities.
- Declare primary key of higher level table the primary key for lower level table
- Declare foreign key constraints.

1.15.4.1.Generalization

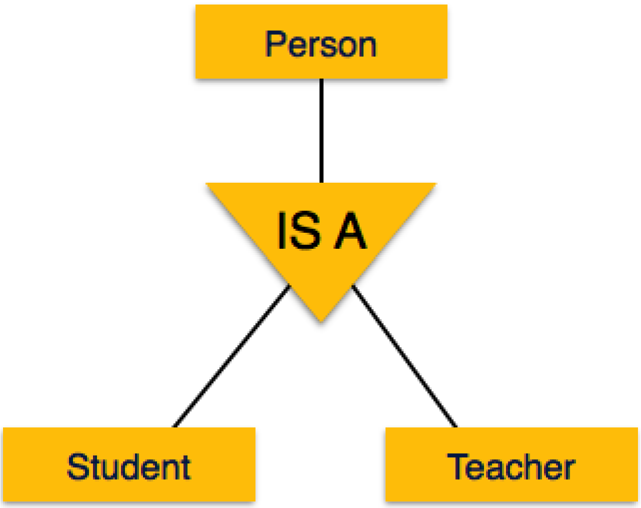
As mentioned above, the process of generalizing entities, where the generalized entities contain the properties of all the generalized entities is called Generalization. In generalization, a number of entities are brought together into one generalized entity based on their similar characteristics. For an example, pigeon, house sparrow, crow and dove all can be generalized as Birds.



[*Image: Generalization*]

1.15.4.2.Specialization

Specialization is a process, which is opposite to generalization, as mentioned above. In specialization, a group of entities is divided into sub-groups based on their characteristics. Take a group Person for example. A person has name, date of birth, gender etc. These properties are common in all persons, human beings. But in a company, a person can be identified as employee, employer, customer or vendor based on what role do they play in company.

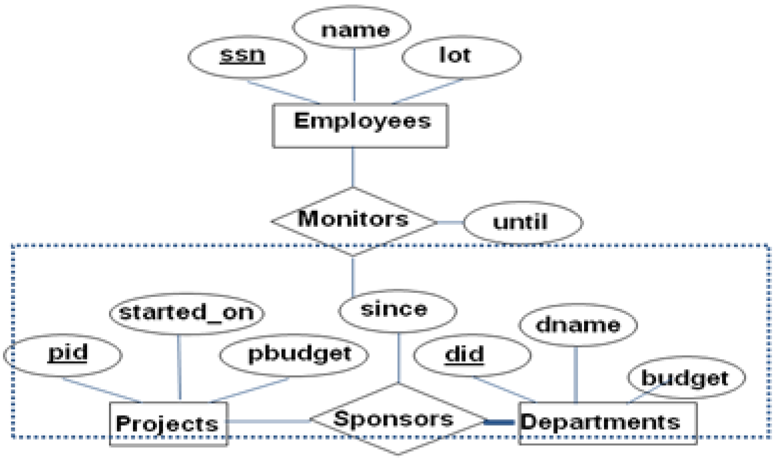


[*Image: Specialization*]

Similarly, in a school database, a person can be specialized as teacher, student or staff; based on what role do they play in school as entities.

1.15.4.3. Aggregation

- Used when we have to model a relationship involving (entitity sets and) a *relationship set.*
  - *Aggregation* allows us to treat a relationship set as an entity set for purposes of participation in (other) relationships.



✉ *Aggregation vs. ternary relationship:*

❖ Monitors is a distinct relationship, with a descriptive attribute.

❖ Also, can say that each sponsorship is monitored by at most one employee.

1.16. CONCEPTUAL DESIGN WITH ER MODEL

Developing an ER diagram presents several choices, including the following:

- Should a concept be modeled as an entity or an attribute?
- Should a concept be modeled as an entity or a relationship?
- What are the relationship sets and their participating entity sets?
- Should we use binary or ternary relationships?
- Should we use aggregation?
-

Issues involved in making these choices.

- Entity versus Attribute
- Entity Versus Relationship
- Binary versus Ternary Relationship
- Aggregation versus Ternary Relationships

1.18.1. CONCEPTUAL DESIGN FOR LARGE ENTERPRISES

- The process of conceptual design consists of more than just describing small fragments of the application in terms of ER diagrams.

- For a large enterprise, the design may require the efforts of more than one designer and span data and application code used by a number of user groups.

- Using a high-level, semantic data model such as ER diagrams for conceptual design in such an environment offers the additional advantage that the high-level design can be diagrammatically represented and is easily understood by the many people who must provide input to the design process.

- An important aspect of the design process is the methodology used to structure the development of the overall design and to ensure that the design takes into account all user requirements and is consistent.

- The usual approach is that the requirements of various user groups are considered, any conflicting requirements are somehow resolved, and a single set of global requirements is generated at the end of the requirements analysis phase. Generating a single set of global requirements is a difficult task, but it allows the conceptual design phase to proceed with the development of a logical schema that spans all the data and applications throughout the enterprise.

1.17. RELATIONAL MODEL

Codd proposed the relational data model in 1970. At that time most database systems were based on one of two older data models (the hierarchical model and the network model); the relational model revolutionized the database field and largely supplanted these earlier models.

Today, the relational model is by far the dominant data model and is the foundation for the leading DBMS products, including IBM's DB2 family, Microsoft's Access and SQL-Server, FoxBase, and Paradox.

The relational model is very simple and elegant; a database is a collection of one or more relations, where each relation is a table with rows and columns. This simple tabular representation enables even novice users to understand the contents of a database, and it permits the use of simple, high-level languages to query the data. The major advantages of the relational model over the older data models are its simple data representation and the ease with which even complex queries can be expressed.

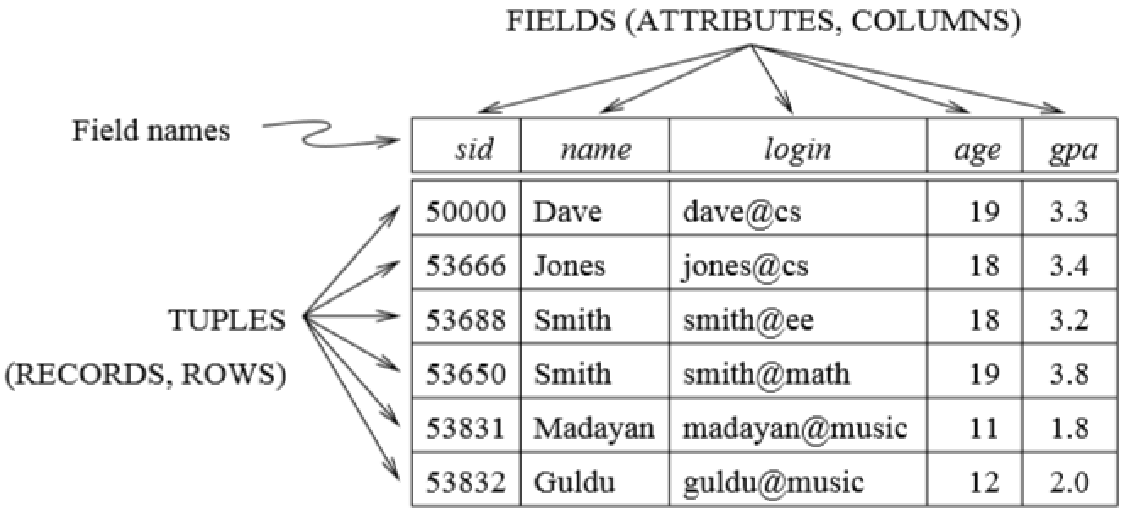
1.17.1. Introduction to the relational model

The main construct for representing data in the relational model is a relation. A relation consists of a relation schema and a relation instance. The relation instance is a table, and the relation schema describes the column heads for the table. We first describe the relation schema and then the relation instance.

The schema specifies the relation's name, the name of each field (or column, orattribute), and the domain of each field. A domain is referred to in a relation schema by the domain name and has a set of associated values.

We use the example of student information in a university database from Chapter 1 to illustrate the parts of a relation schema:
Students(sid: string, name: string, login: string, age: integer, gpa: real)

This says, for instance, that the field named sid has a domain named string. The set of values associated with domain string is the set of all character strings.We now turn to the instances of a relation. An instance of a relation is a set of tuples, also called records, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields. (The term relation instance is often abbreviated to just relation, when there is no confusion with other aspects of a relation such as its schema.)



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance of the S1 of the Students Relation

- **Creating and modifying relations using SQL**
    - Create
    - Insert
    - Update
    - Delete

1.17.2. INTEGRITY CONSTRAINTS OVER RELATIONS

- IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*
- ICs are specified when schema is defined.
- ICs are checked when relations are modified.
- A *legal* instance of a relation is one that satisfies all specified ICs.
- DBMS should not allow illegal instances.
- If the DBMS checks ICs, stored data is more faithful to real-world meaning.

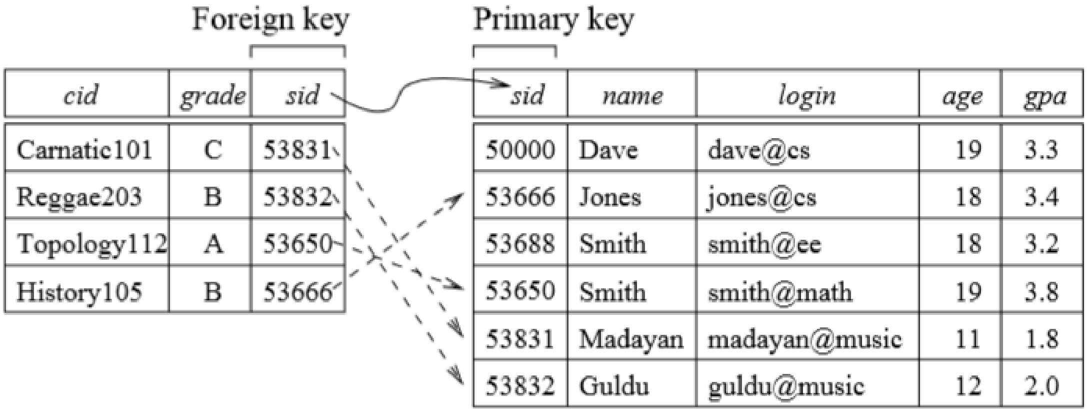
1.17.2.1Key constraints

- Candidate Key
- Primary Key
- Super Key

1.17.2.2. Foreign Key Constraints

- Specifying Key constraints in SQL

1.17.2.3. General Constraints



cid	*grade*	*sid*
Carnatic101	C	53831
Reggae203	B	53832
Topology112	A	53650
History105	B	53666

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Enrolled (Referencing relation)

Students (Referenced relation)

1.18. ENFORCING INTEGRITY CONSTRAINTS:

- ICs are specified when a relation is created and enforced when a relation is modified.
- The impact of domain, PRIMARY KEY, and UNIQUE constraints is straightforward: if an insert, delete, or update command causes a violation, it is rejected.
- Potential IC violation is generally checked at the end of each SQL statement execution, although it can be deferred until the end of the transaction executing the statement.

- Consider the instance S1 of Students shown in Figure 3.1. The following insertion violates the primary key constraint because there is already a tuple with the sid 53688, and it will be rejected by the DBMS:

INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, 'Mike', 'mike@ee', 17, 3.4)

The following insertion violates the constraint that the primary key cannot contain null:

INSERT INTO Students (sid, name, login, age, gpa) VALUES (null, 'Mike', 'mike@ee', 17, 3.4)

Of course, a similar problem arises whenever we try to insert a tuple with a value in a field that is not in the domain associated with that field, i.e., whenever we violate a domain constraint. Deletion does not cause a violation of domain, primary key or unique constraints. However, an update can cause violations, similar to an insertion:

UPDATE Students S SET S.sid = 50000 WHERE S.sid = 53688

This update violates the primary key constraint because there is already a tuple with sid 50000.

The impact of foreign key constraints is more complex because SQL sometimes tries to rectifya foreign key constraint violation instead of simply rejecting the change.

1. What should we do if an Enrolled row is inserted, with a sid column value that does not appear in any row of the Students table? In this case the INSERT command is simply rejected.

2. What should we do if a Students row is deleted? The options are: Delete all Enrolled rows that refer to the deleted Students row. Disallow the deletion of the Students row if an Enrolled row refers to it. Set the sid column to the sid of some (existing) 'default' sudent, for every Enrolled row that refers to the deleted Students row. For every Enrolled row that refers to it, set the sid column to null. In our example, this option conflicts with the fact that sid is part of the primary key of Enrolled and therefore cannot be set to null. Thus, we are limited to the first three options in our example, although this fourth option (setting the foreign key to null) is available in the general case.

3. What should we do if the primary key value of a Students row is updated? The options here are similar to the previous case.

```
CREATE TABLE Enrolled (
sid CHAR(20),
 cid CHAR(20),
grade CHAR(10), PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students ON DELETE CASCADE ON UPDATE NO ACTION )
```

If a Students row is deleted, we can switch the enrollment to a ‘default’ student by using ON DELETE SET DEFAULT. The default student is specified as part of the definition of the sid field in Enrolled;

For example, sid CHAR(20) DEFAULT ‘53666’.

Although the specification of a default value is appropriate in some situations (e.g., a default parts supplier if a particular supplier goes out of business), it is really not appropriate to switch enrollments to a default student. The correct solution in this example is to also delete all enrollment tuples for the deleted student (that is, CASCADE), or to reject the update.

SQL also allows the use of null as the default value by specifying ON DELETE SET NULL.

1.19. QUERYING RELATIONAL DATA:

Relational data base Query is a Question about the Data and the answer consists of a new relation containing the result.

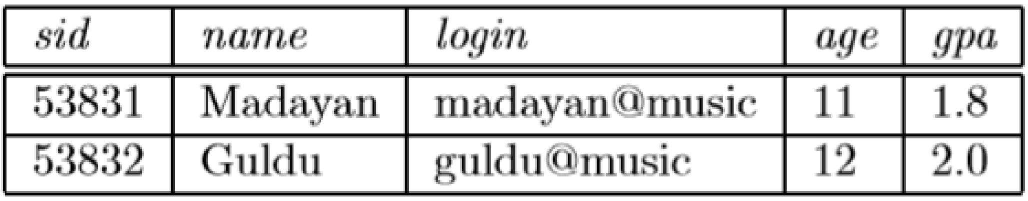
We can retrieve rows corresponding to students who are younger than 18 with the following SQL query:

```
SELECT * FROM Students S WHERE S.age < 18
```

The symbol * means that we retain all fields of selected tuples in the result.

To understand this query, think of S as a variable that takes on the value of each tuple in Students, one tuple after the other.

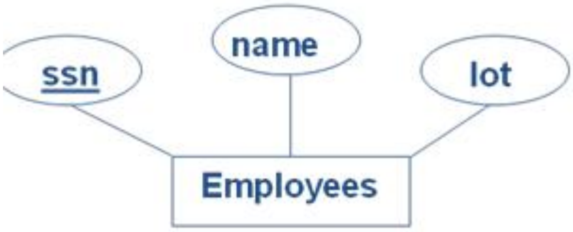
The condition S.age < 18 in the WHERE clause specifies that we want to select only tuples in which the age field has a value less than 18.



sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

1.20.LOGICAL DB DESIGN:ER TO RELATIONAL

- Entity sets to tables:



```
CREATE TABLE Employees
        (ssn CHAR(11),
        name CHAR(20),
        lot  INTEGER,
        PRIMARY KEY  (ssn))
```

Relationship Sets to Tables

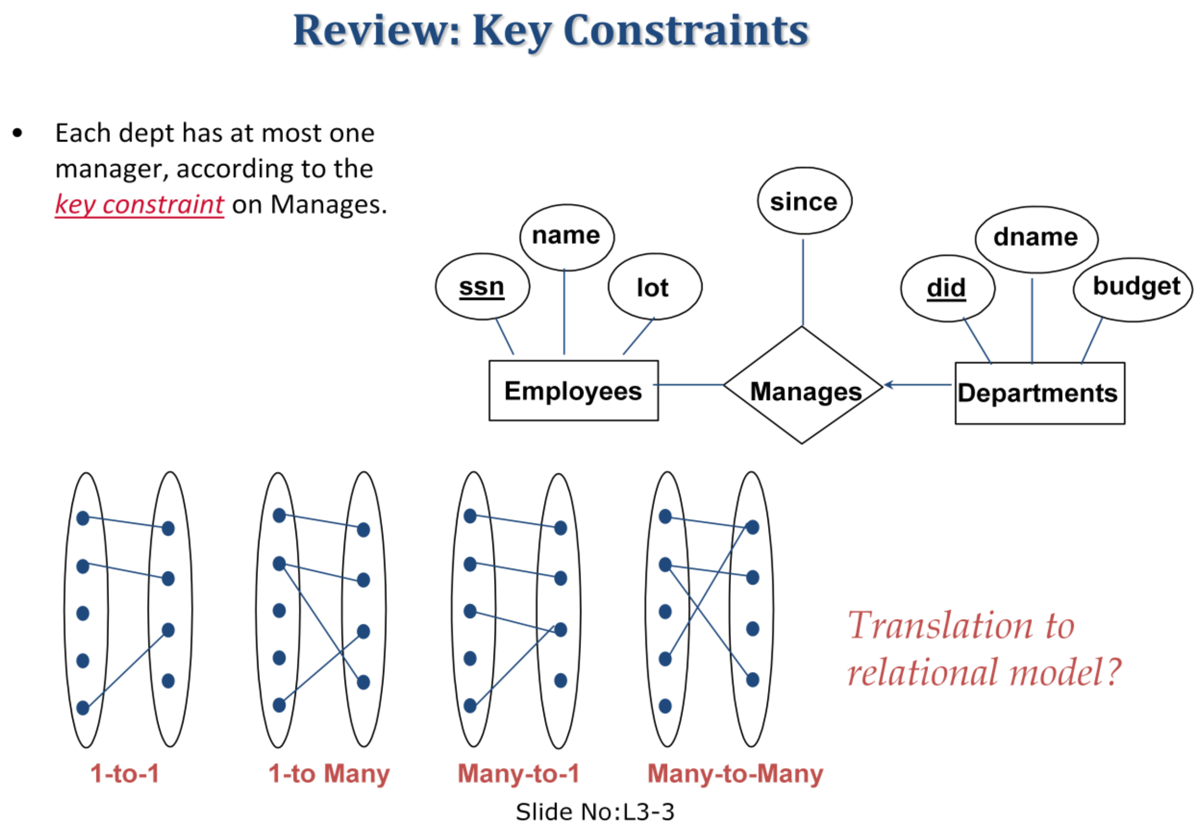
- In translating a relationship set to a relation, attributes of the relation must include:

– Keys for each participating entity set (as foreign keys).

- This set of attributes forms a *superkey* for the relation.

– All descriptive attributes.

Review: Key Constraints



Review: Key Constraints

- Each dept has at most one manager, according to the *key constraint* on Manages.

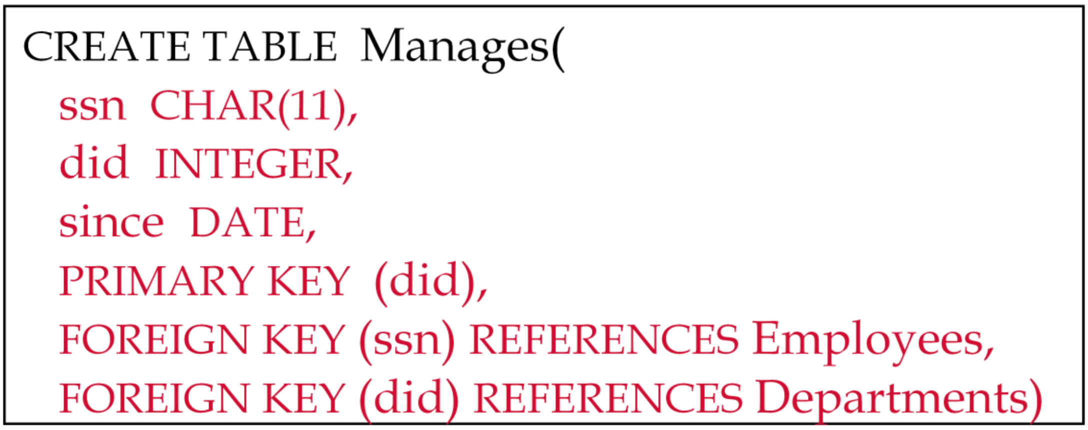
Translation to relational model?

Slide No:L3-3

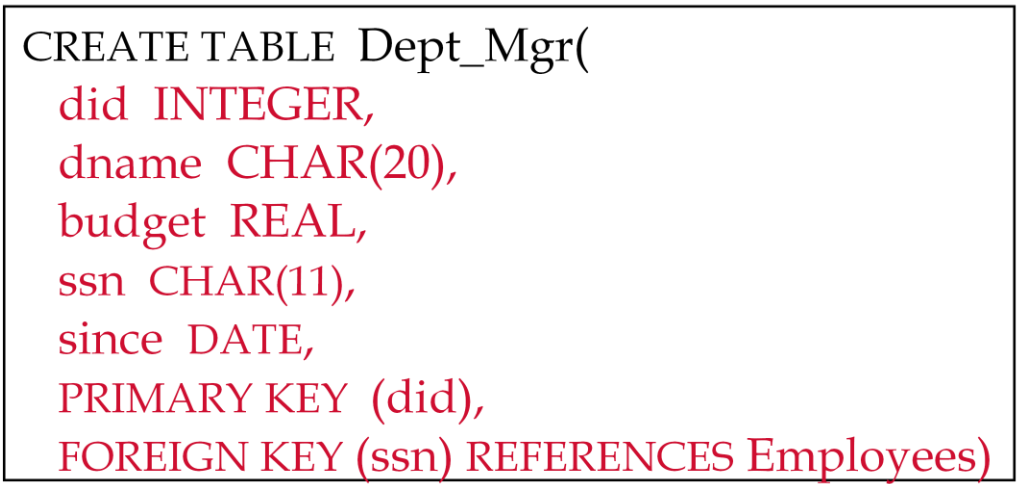
- Each dept has at most one manager, according to the *key constraint* on Manages.

Translating ER Diagrams with Key Constraints

- Map relationship to a table:
  - Note that did is the key now!
  - Separate tables for Employees and Departments.
- Since each department has a unique manager, we could instead combine Manages and Departments.

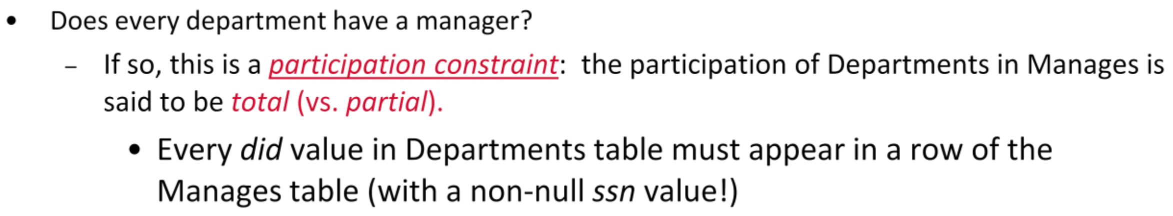


```
CREATE TABLE  Manages(
   ssn  CHAR(11),
   did  INTEGER,
   since  DATE,
   PRIMARY KEY  (did),
   FOREIGN KEY (ssn) REFERENCES Employees,
   FOREIGN KEY (did) REFERENCES Departments)
```



```
CREATE TABLE  Dept_Mgr(
   did  INTEGER,
   dname  CHAR(20),
   budget  REAL,
   ssn  CHAR(11),
   since  DATE,
   PRIMARY KEY  (did),
   FOREIGN KEY (ssn) REFERENCES Employees)
```

Review: Participation Constraints



- Does every department have a manager?
  - If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).
    - Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)

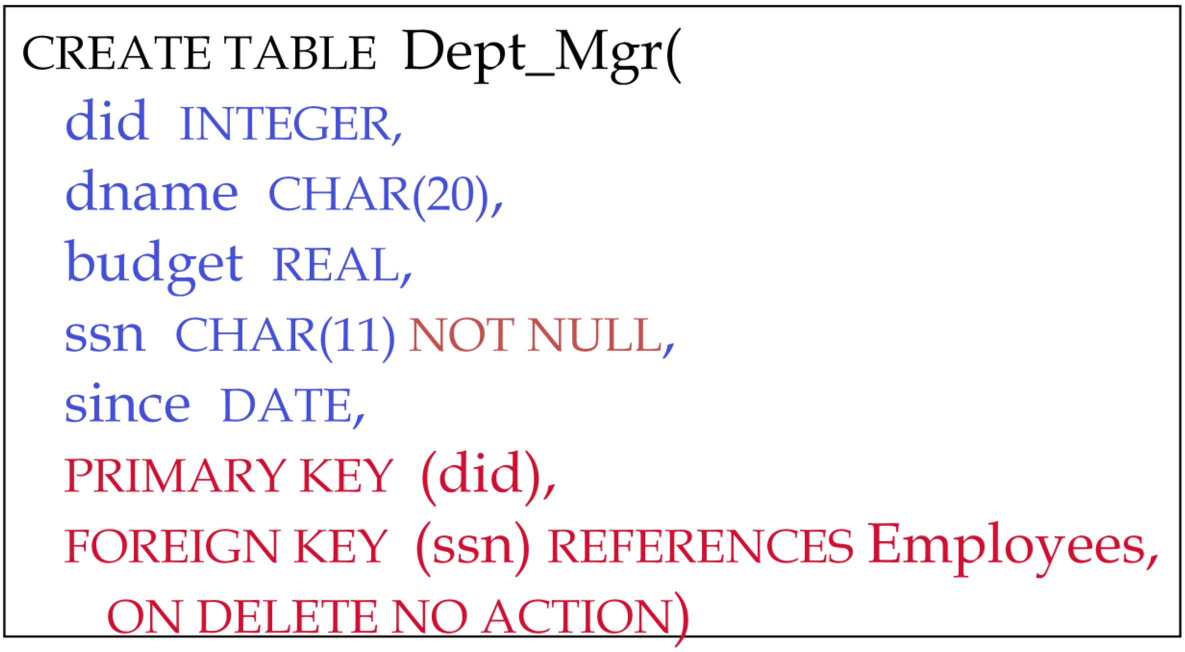
- Does every department have a manager?

– If so, this is a *participation constraint*: the participation of Departments in Manages is said to be *total* (vs. *partial*).

- Every *did* value in Departments table must appear in a row of the Manages table (with a non-null *ssn* value!)

Participation Constraints in SQL

- We can capture participation constraints involving one entity set in a binary relationship, but little else (without resorting to CHECK constraints).



```
CREATE TABLE Dept_Mgr(
   did INTEGER,
   dname CHAR(20),
   budget REAL,
   ssn CHAR(11) NOT NULL,
   since DATE,
   PRIMARY KEY (did),
   FOREIGN KEY (ssn) REFERENCES Employees,
      ON DELETE NO ACTION)
```

Review: Weak Entities

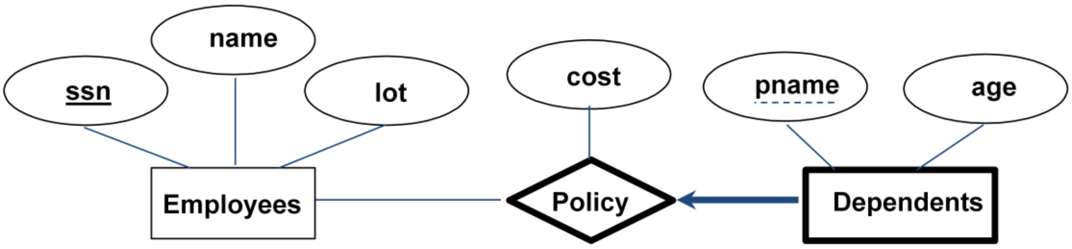
- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.

– Owner entity set and weak entity set must participate in a one-to-many relationship set (1 owner, many weak entities).

– Weak entity set must have total participation in this *identifying* relationship set.

Review: Weak Entities

- A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.
  - Owner entity set and weak entity set must participate in a one-to-many relationship set (1 owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.

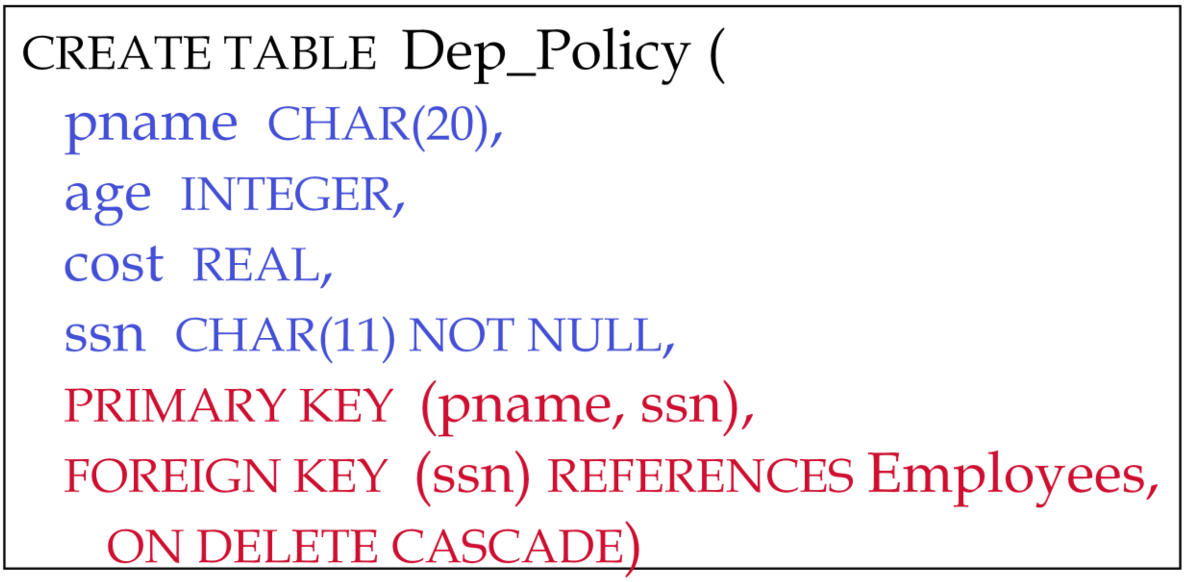


Slide No:L4-1

Translating Weak Entity Sets

- Weak entity set and identifying relationship set are translated into a single table.

– When the owner entity is deleted, all owned weak entities must also be deleted.

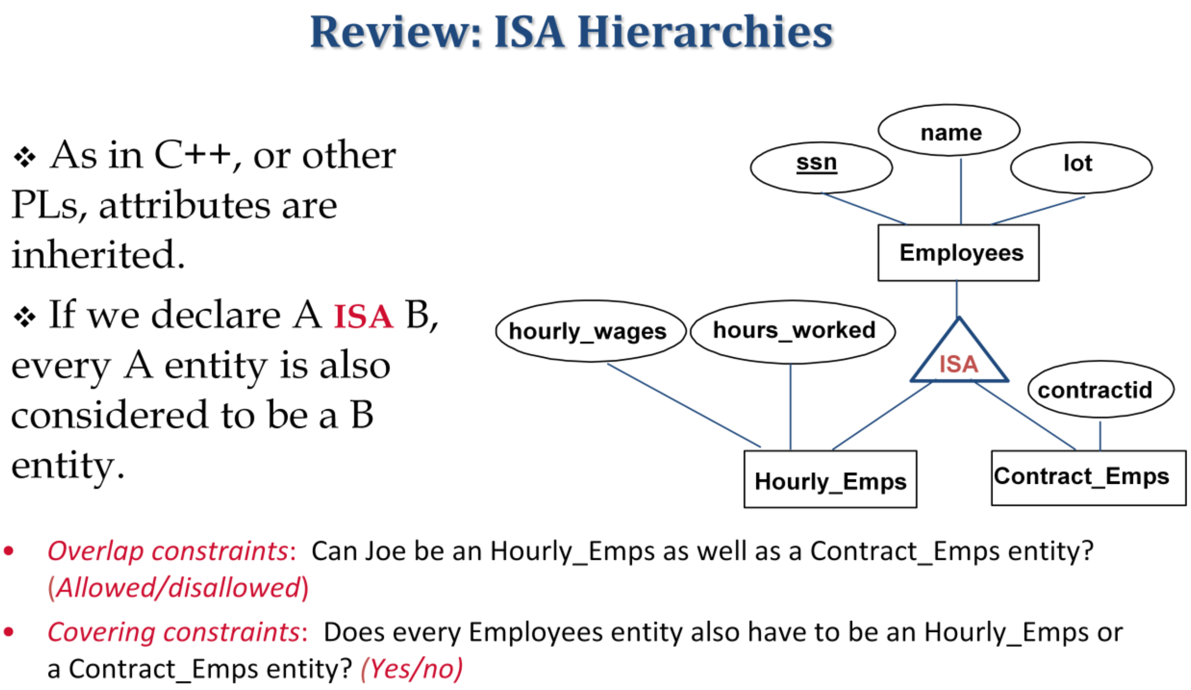


```
CREATE TABLE  Dep_Policy (
   pname  CHAR(20),
   age  INTEGER,
   cost  REAL,
   ssn  CHAR(11) NOT NULL,
   PRIMARY KEY  (pname, ssn),
   FOREIGN KEY  (ssn) REFERENCES Employees,
      ON DELETE CASCADE)
```

Review: ISA Hierarchies

- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)

- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*



Review: ISA Hierarchies

❖ As in C++, or other PLs, attributes are inherited.

❖ If we declare A **ISA** B, every A entity is also considered to be a B entity.

- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? *(Allowed/disallowed)*
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*

Slide No:L4-3

Translating ISA Hierarchies to Relations

- ***General approach:***

– 3 relations: Employees, Hourly_Emps and Contract_Emps.

Hourly_Emps: Every employee is recorded in Employees. For hourly emps, extra info recorded in Hourly_Emps (*hourly_wages*, *hours_worked*, *ssn*); must delete Hourly_Emps tuple if referenced Employees tuple is deleted).

- Queries involving all employees easy, those involving just Hourly_Emps require a join to get some attributes.

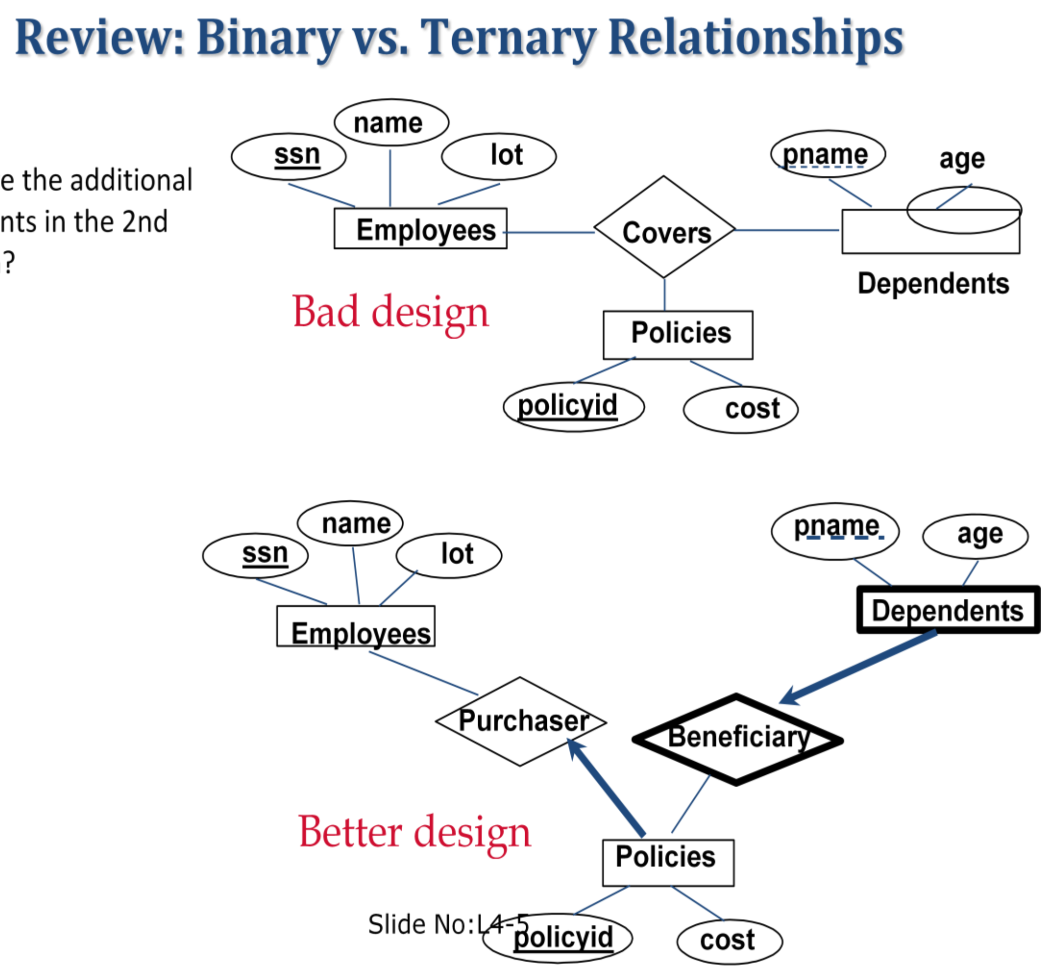
- Alternative: Just Hourly_Emps and Contract_Emps.

– *Hourly_Emps*: *ssn, name, lot, hourly_wages, hours_worked.*

– Each employee must be in one of these two subclasses.

Review: Binary vs. Ternary Relationships

- What are the additional constraints in the 2nd diagram?



Review: Binary vs. Ternary Relationships

- What are the additional constraints in the 2nd diagram?

Bad design

Better design

Slide No:L4-5

Binary vs. Ternary Relationships (Contd.)

- The key constraints allow us to combine Purchaser with Policies and Beneficiary with Dependents.

- Participation constraints lead to NOT NULL constraints.

```
CREATE TABLE Policies (
   policyid INTEGER,
   cost REAL,
   ssn CHAR(11) NOT NULL,
   PRIMARY KEY (policyid).
   FOREIGN KEY (ssn) REFERENCES Employees,
      ON DELETE CASCADE)
```

```
 CREATE TABLE Dependents (
  pname CHAR(20),
  age INTEGER,
  policyid INTEGER,
  PRIMARY KEY (pname, policyid).
  FOREIGN KEY (policyid) REFERENCES Policies,
     ON DELETE CASCADE)
```

1.21.1. INTRODUCTION TO VIEWS:

- A relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

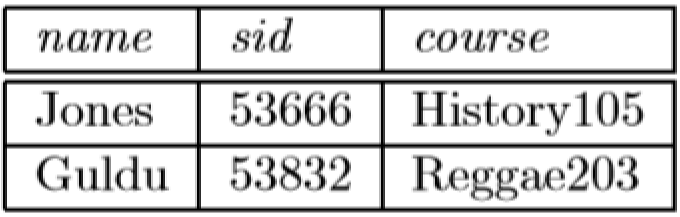
- A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition.

- Consider the Students and Enrolled relations. Suppose that we are often interested in finding the names and student identifiers of students who got a grade of B in some course, together with the cid for the course. We can define a view for this purpose.

- CREATE VIEW B-Students (name, sid, course) AS SELECT S.sname, S.sid, E.cid FROM Students S, Enrolled E WHERE S.sid = E.sid AND E.grade = ‘B’

- The view B-Students has three fields called name, sid, andcourse with the same domains as the fields sname and sid in Students and cid in Enrolled. (If the optional arguments name, sid, andcourse are omitted from the CREATE VIEW statement, the column names sname, sid, andcid are inherited.)

- This view can be used just like a base table, or explicitly stored table, in defining new queries or views.



name	*sid*	*course*
Jones	53666	History105
Guldu	53832	Reggae203

1.21.1. Destroying and Altering Tables and Views:

- If we decide that we no longer need a base table and want to destroy it (i.e., delete all the rows and remove the table definition information),

- we can use the DROP TABLE command. For example, DROP TABLE Students RESTRICT destroys the Students table unless some view or integrity constraint refers to Students; if so, the command fails. If the keyword RESTRICT is replaced by CASCADE,

- Students is dropped and any ref- erencing views or integrity constraints are (recursively) dropped as well; one of these two keywords must always be specified. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

- ALTER TABLE modifies the structure of an existing table. To add a column called maiden-name to Students, for example, we would use the following command:

```
ALTER TABLE Students ADD COLUMN maiden-name CHAR(10)
```

The definition of Students is modified to add this column, and all existing rows are padded with null values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table; we will not discuss these aspects of the command beyond remarking that dropping columns is treated very similarly to dropping tables or views.

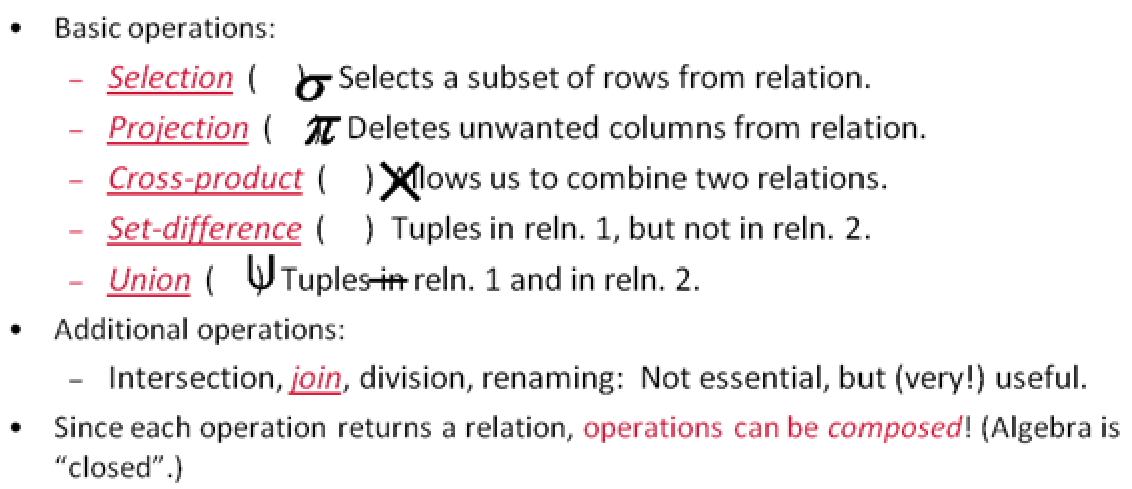
UNIT-II

2. RELATIONAL ALGEBRA AND RELATIONAL CALCULUS

2.1. RELATIONAL ALGEBRA

Relational algebra is one of the two formal query languages associated with the re- lational model. Queries in algebra are composed using a collection of operators. A fundamental property is that every operator in the algebra accepts (one or two) rela- tion instances as arguments and returns a relation instance as the result.

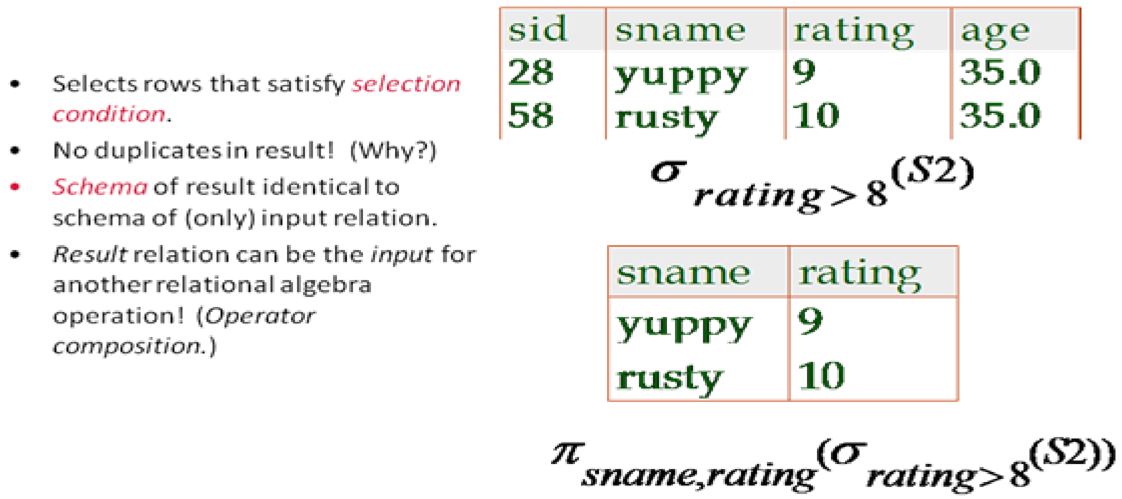
a relational algebra expression is recursively defined to be a relation, a unary algebra operator applied to a single expression, or a binary algebra operator applied to two expressions.



- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ( ) ×llows 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”.)

2.1.1. SELECTION:

Relational algebra includes operator to select rows from a relation (σ).



- 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.*)

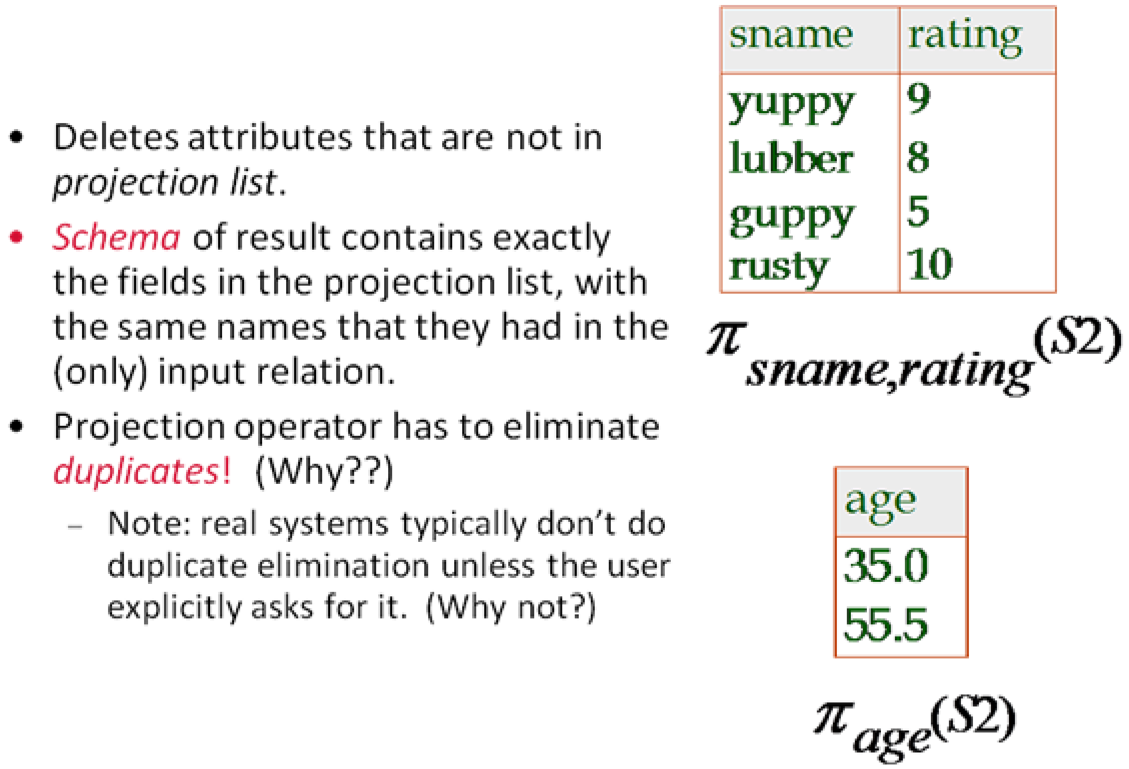
sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

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

sname	rating
yuppy	9
rusty	10

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

2.1.2. 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

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

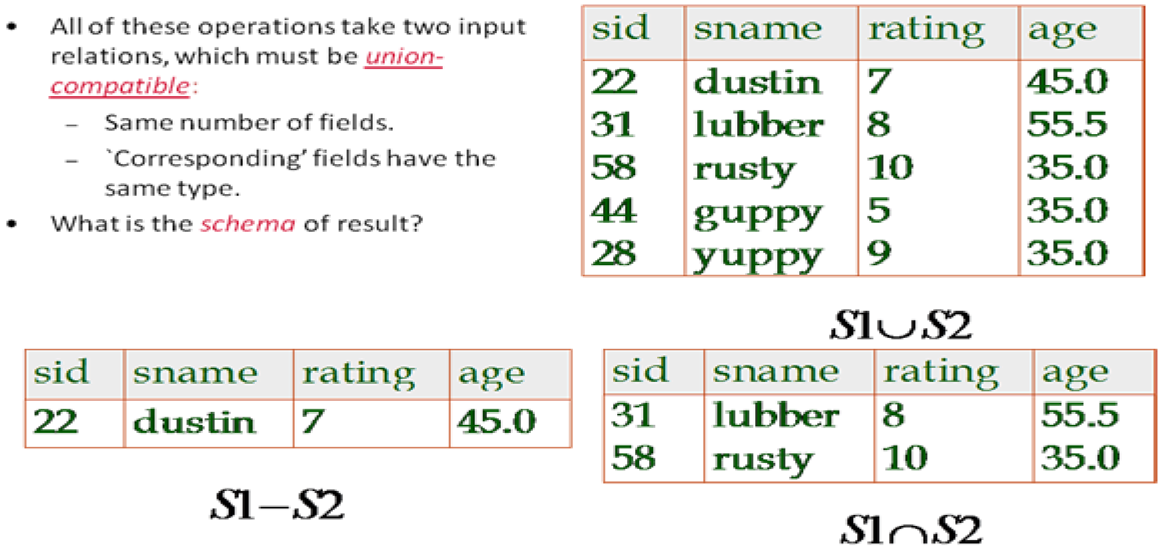
age
35.0
55.5

$\pi_{age}(S2)$

2.1.3. SET OPERATIONS

The following standard operations on sets are also available in relational algebra: union (∪), intersection (∩), set-difference (−), and cross-product (×).

Cross-Product



- 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

$S1 \cup S2$

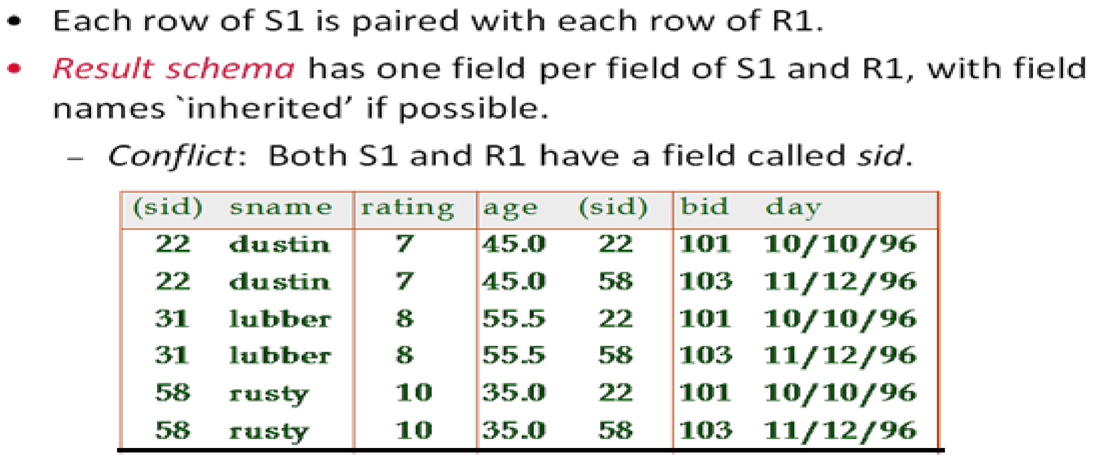
sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

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

$S1 \cap S2$

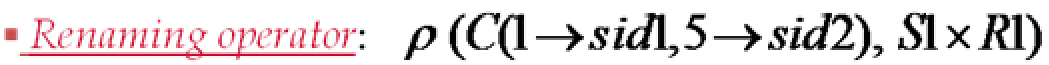
- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
- *Conflict*: Both S1 and R1 have a field called *sid*.



- 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

2.1.4. RENAMING



- Renaming operator: $\rho\,(C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$

2.2.4 JOINS

The Join operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations.

The most general version of the join operation accepts a join condition c and a pair of relation instances as arguments, and returns a relation instance.

2.2.4.1. CONDITION JOINS

The most general version of the join operation accepts a join condition c and a pair of relation instances as arguments, and returns a relation instance. The join condition is identical to a selection condition in form.The operation is defined as follows:

$$R \bowtie_c S = \sigma_c (R \times S)$$

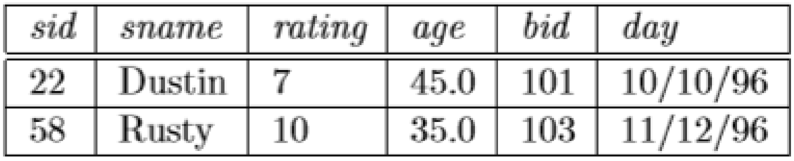
2.2.4.2. EQUIJOIN

A common special case of the join operation $R \bowtie S$ is when the join condition consists solely of equalities (connected by ∧) of the form R.name1=S.name2, that is, equalities between two fields in R and S.

In this case, obviously, there is some redundancy in retaining both attributes in the result. For join conditions that contain only such equalities, the join operation is refined by doing an additional projection in which S.name2 is dropped. The join operation with this refinement is called equijoin.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} R1$ Notice that only one field called *sid* appears in the result.



sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

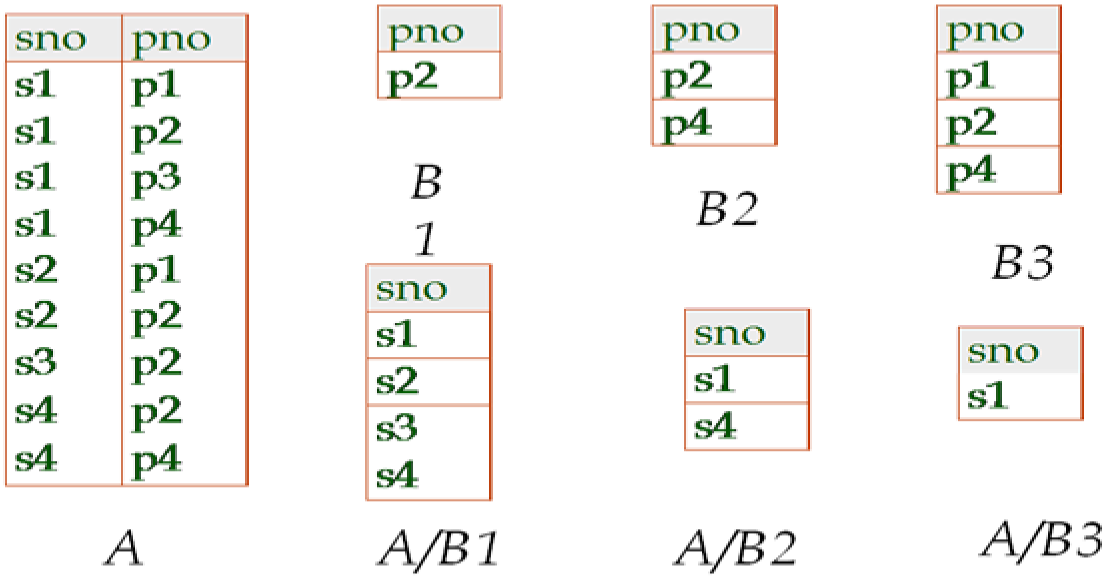
$S1 \bowtie_{R.sid=S.sid} R1$

2.2.4.3. NATURAL JOIN

A further special case of the join operation R ./ Sis an equijoin in which equalities are specified on all fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. We call this special case a natural join, and it has the nice property that the result is guaranteed not to have two fields with the same name.

The equijoin expression S1 ./R.sid=S.sid R1 is actually a natural join and can simply be denoted as S1 ./ R1, since the only common field is sid. If the two relations have no attributes in common, S1 ./ R1 is simply the cross-product.

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

B 1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

2.1.7. EXAMPLES OF ALGEBRA QUERIES

1.Find names of sailors who've reserved boat #103

Find names of sailors who’ve reserved a red boat

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

$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

2. 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:

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

2.2. RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative.

Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).

2.2.1. TUPLE RELATIONAL CALCULUS

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields.

A tuple relational calculus query has the form { T | p(T) },

where T is a tuple variable and p(T) denotes a formula that describes T;

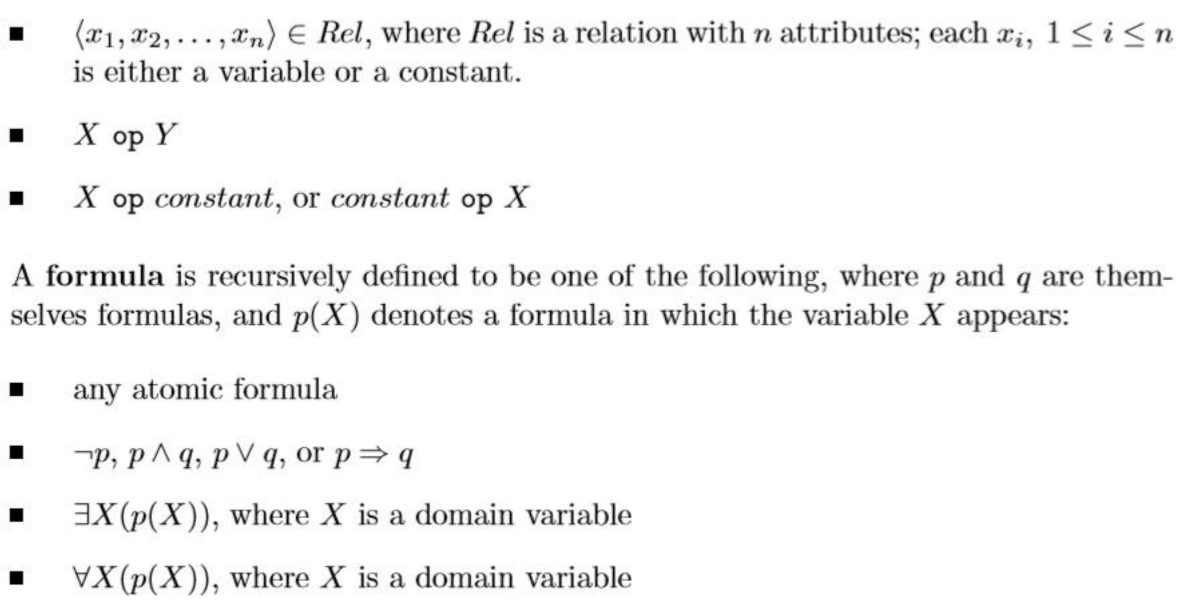
The result of this query is the set of all tuples t for which the formula p(T) evaluates totrue with T = t. The language for writing formulas

p(T) is thus at the heart of TRC and is essentially a simple subset of first-order logic.

2.2.2. DOMAIN RELATIONAL CALCULUS

A domain variable is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers).

A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set{<,>,=,≤,≥,6=}and let X and Y be domain variables. An atomic formula in DRC is one of the following:



- $\langle x_1, x_2, \ldots, x_n \rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \leq i \leq n$ is either a variable or a constant.
- X op Y
- X op $constant$, or $constant$ op X

A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and $p(X)$ denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

2.3. EXPRESSIVE POWER OF RELATIONAL ALGEBRA AND CALCULUS

It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called *unsafe*.

Example.

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.

2.4. THE FORM OF A BASIC SQL QUERIES

- The basic form of an SQL query is as follows:

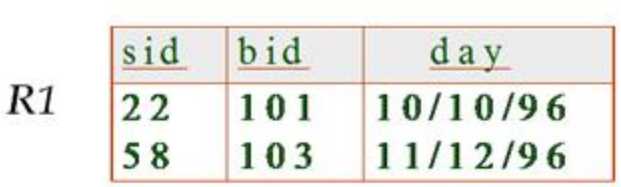
  SELECT [DISTINCT] select-list
  FROM from-list
  WHERE qualification

Such a query intuitively corresponds to a relational algebra expression involving selec- tions, projections, and cross-products.

- SELECT clause, which specifies columns to be retained in the result, and a
- FROM clause, which specifies a cross-product of tables.

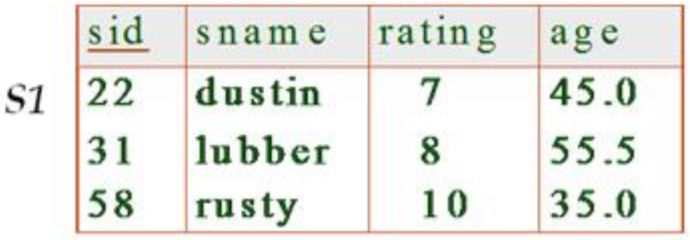
- The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

2.4.1. EXAMPLES OF BASIC SQL QUERIES



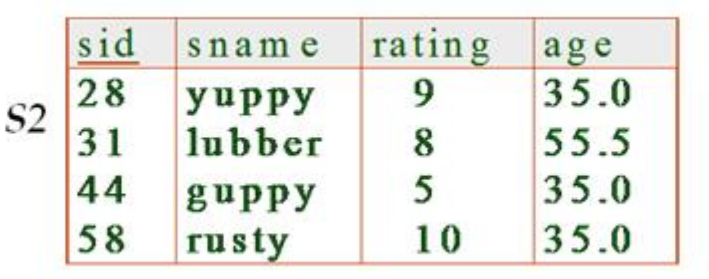
R1

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

Q1.Find the names and ages of all sailors.
SELECT DISTINCT S.sname, S.age FROM Sailors S

Q2.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

Q3.Find the names of sailors who have reserved boat number 103.
It can be expressed in SQL as follows.
SELECT S.sname FROM Sailors S, Reserves R WHERE S.sid = R.sid AND R.bid=103

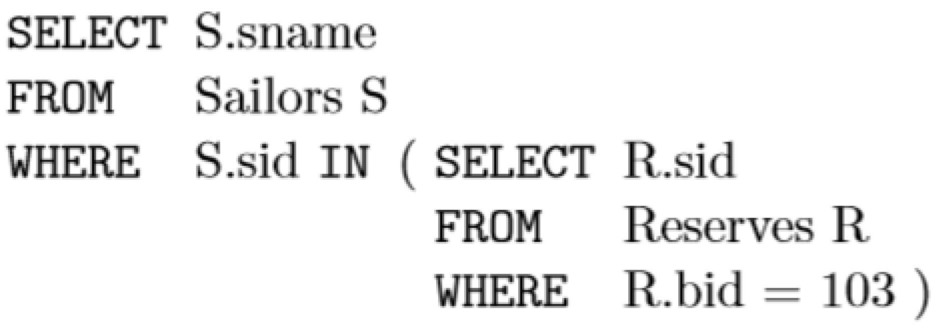
2.5. NESTED QUERIES

- A nested query is a query that has another query embedded within it;
- The embedded query is called a subquery.

- When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed.

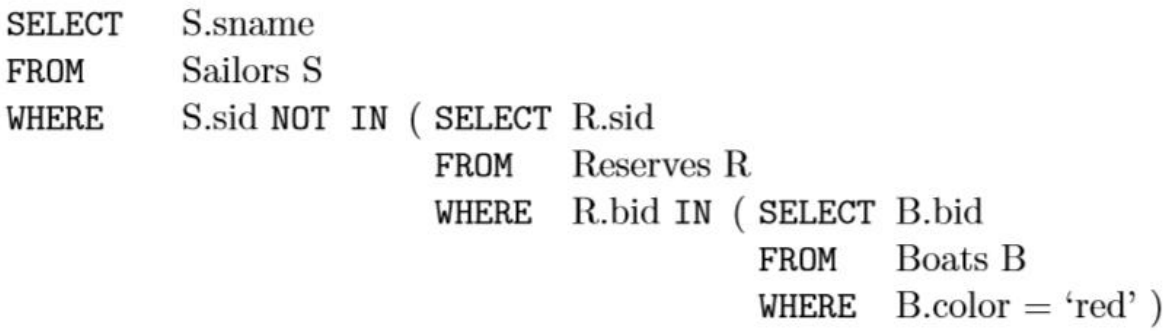
2.5.1. INTRODUCTION TO NESTED QUERIES

(Q1) 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 )
```

(Q2)Find the names of sailors who have reserved a red boat.

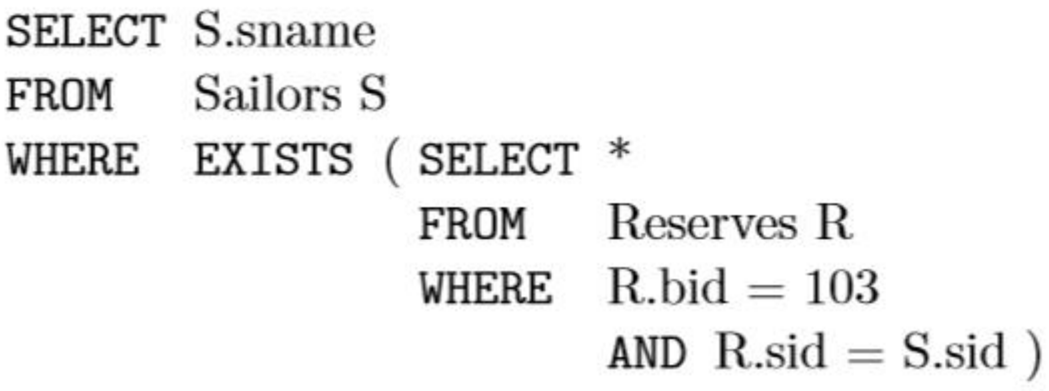


```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid NOT IN ( SELECT  R.sid
                       FROM    Reserves R
                       WHERE   R.bid IN ( SELECT  B.bid
                                          FROM    Boats B
                                          WHERE   B.color = 'red' )
```

2.5.2. CORRELATED NESTED QUERIES

In general the inner subquery could depend on the row that is currently being examined in the outer query (in terms of our conceptual evaluation strategy).

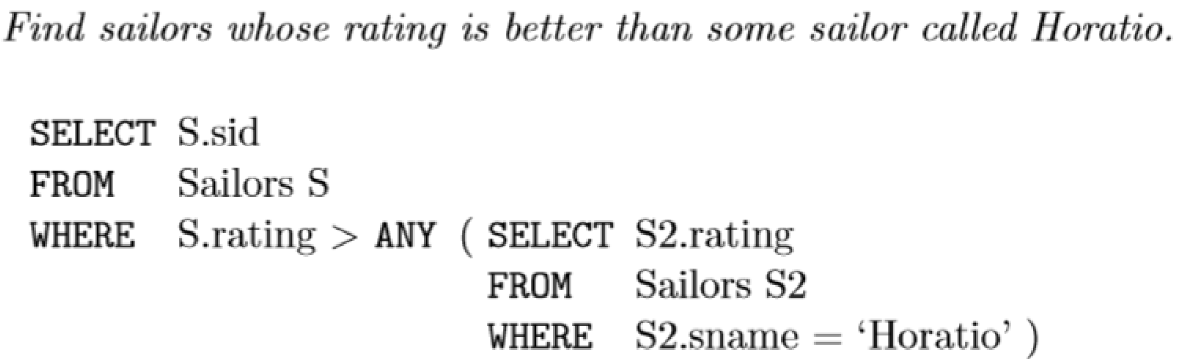
(Q1) Find the names of sailors who have reserved boat number 103.



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

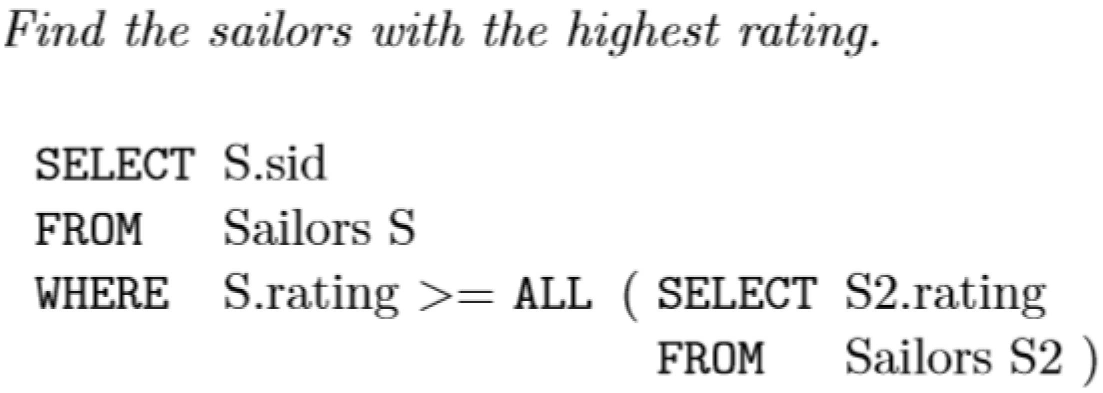
2.5.3. SET-COMPARISON OPERATORS

SQL also supports op ANY and op ALL, whereop is one of the arithmetic comparison operators {<,<=,=,<>,>=,>}.(SOME is also available, but it is just a synonym for ANY.)



Find sailors whose rating is better than some sailor called Horatio.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating > ANY ( SELECT  S2.rating
                         FROM    Sailors S2
                         WHERE   S2.sname = 'Horatio' )
```



Find the sailors with the highest rating.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating >= ALL ( SELECT  S2.rating
                          FROM    Sailors S2 )
```

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, −.

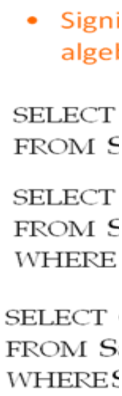
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

2.6. AGGREGATE OPERATORS

These functions operate on the multiset of values of a column of a relation, and return a value

- **avg:** average value,
- **min:** minimum value
- **max:** maximum value
- **sum:** sum of values
- **count:** number of values

Aggregate Operators examples



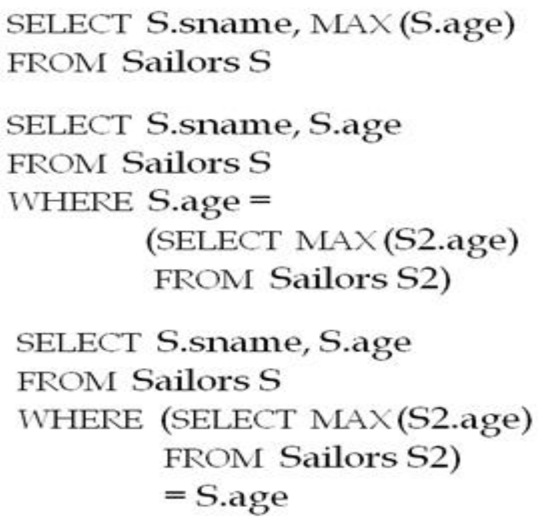
- Signi
  algel

```
SELECT
FROM S

SELECT
FROM S
WHERE

SELECT
FROM S.
WHERE S
```

Find name and age of the oldest sailor



```
SELECT S.sname, MAX (S.age)
FROM Sailors S

SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
          (SELECT MAX (S2.age)
           FROM Sailors S2)

SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
           FROM Sailors S2)
           = S.age
```

2.6.1. THE GROUP BY AND HAVING CLAUSES

– **Motivation for Grouping**

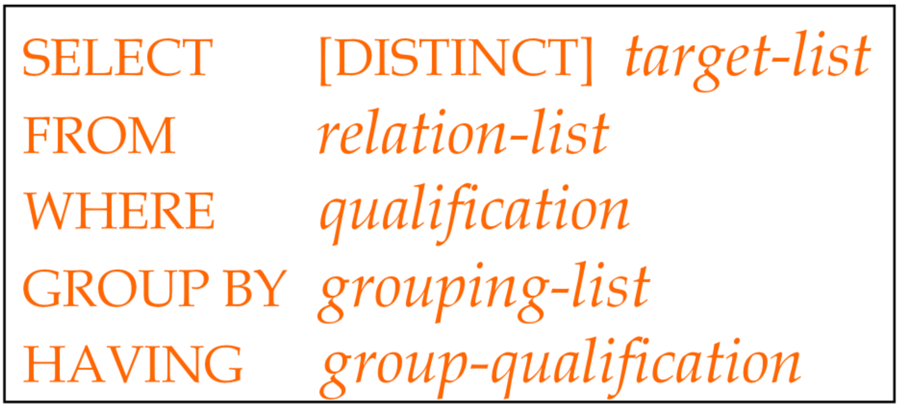
So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.

- consider: *Find the age of the youngest sailor for each rating level.*
  In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

– **Queries With GROUP BY and HAVING**

The *target-list* contain

(i) attribute names
(ii) (ii) terms with aggregate operations (e.g., MIN (*S.age*)).



```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

The attribute list (i) must be a subset of *grouping-list*.
Intuitively, each answer tuple corresponds to a *group*,
and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

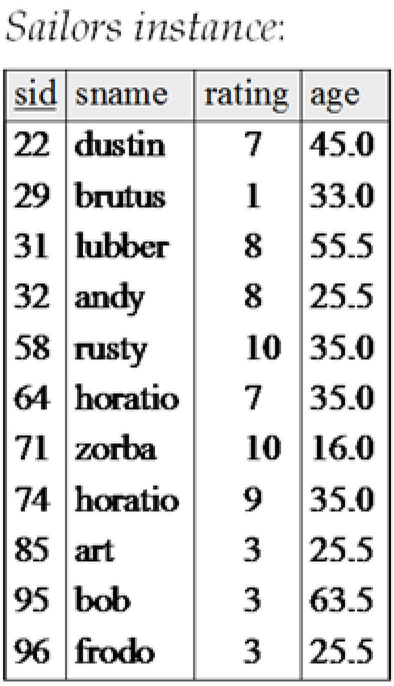
- ✓ The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary'* fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.

- ✓ The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a single value per group!

- ✓ In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)

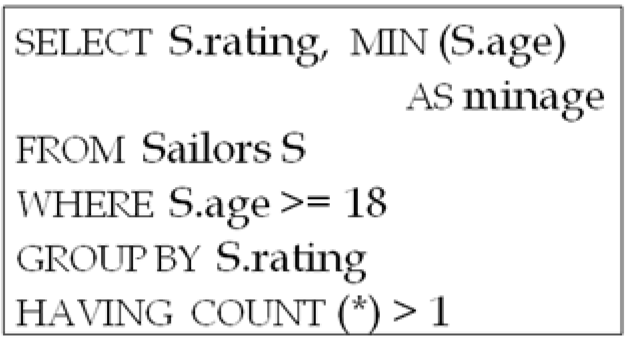
- ✓ One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age 18, for each rating with at least 2 such sailors

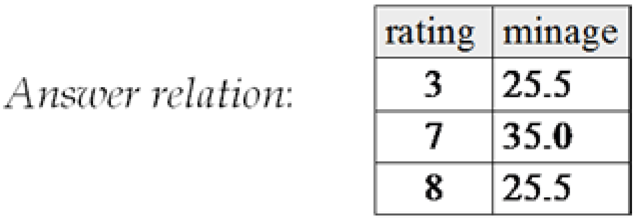


Sailors instance:

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
96	frodo	3	25.5



```
SELECT S.rating, MIN (S.age)
                      AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```



Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

1. HAVING clause can also contain a subquery.
2. Aggregate operations cannot be nested! WRONG:

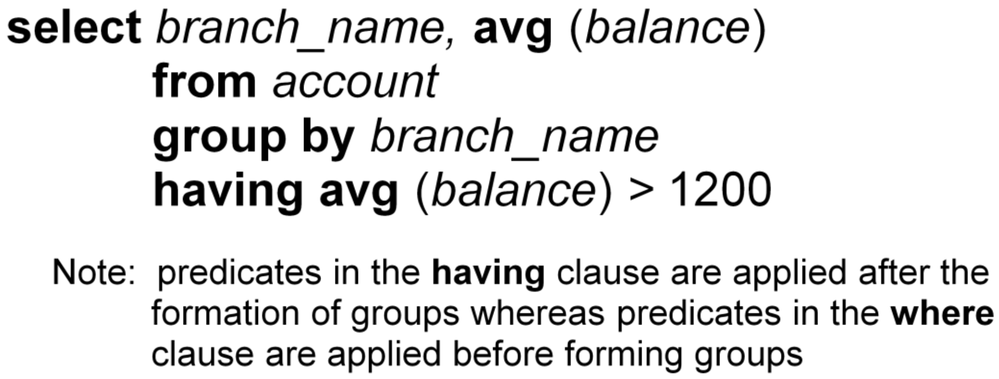
Examples:

(Q1).Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
        from depositor, account
        where depositor.account_number = account.account_number
        group by branch_name
```

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

(Q2). Find the names of all branches where the average account balance is more than $1,200.



```
select branch_name, avg (balance)
          from account
          group by branch_name
          having avg (balance) > 1200
```

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

2.7. NULL VALUES

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
- SQL provides a special value *null* for such situations.The presence of *null* complicates many issues. E.g.:Special operators needed to check if value is/is not *null*. Is *rating>8* true or false when *rating* is equal to *null*?

2.7.1. COMPARISON USING NULL VALUES

It is possible for tuples to have a null value, denoted by *null*, for some of their attributes *null* signifies an unknown value or that a value does not exist.The predicate **is null** can be used to check for null values.

if we compare two null values using <,>,=, and so on, the result is always unknown. For example, if we have null in two distinct rows of the sailor relation, any comparison returns unknown.

Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

```
select loan_number
from loan
where amount is null
```

The result of any arithmetic expression involving *null* is *null*
Example: 5 + *null* returns null However, aggregate functions simply ignore nulls

Any comparison with *null* returns *unknown*
Example*:*
5 < null or null <> null or null = null

2.7.2. LOGICAL CONNECTIVES-AND, OR AND NOT

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

OR:

(*unknown* **or** *true*) = *true*,
(*unknown* **or** *false*) = *unknown*
(*unknown* **or** *unknown*) = *unknown*

AND:

(true **and** *unknown)* = *unknown,*
(false **and** *unknown)* = *false,*
(unknown **and** *unknown)* = *unknown*

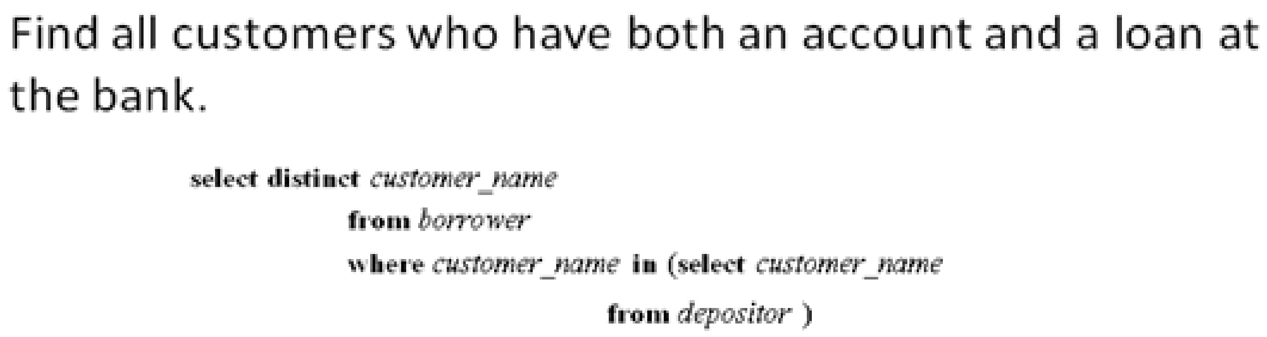
NOT:

*(***not** *unknown)* = *unknown*
"*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*

- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

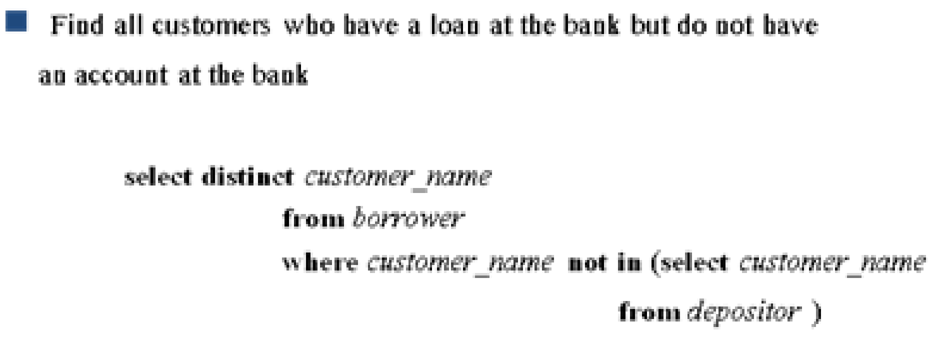
2.7.3. IMPACT ON SQL CONSTRUCTS

"In" Construct:



Find all customers who have both an account and a loan at the bank.

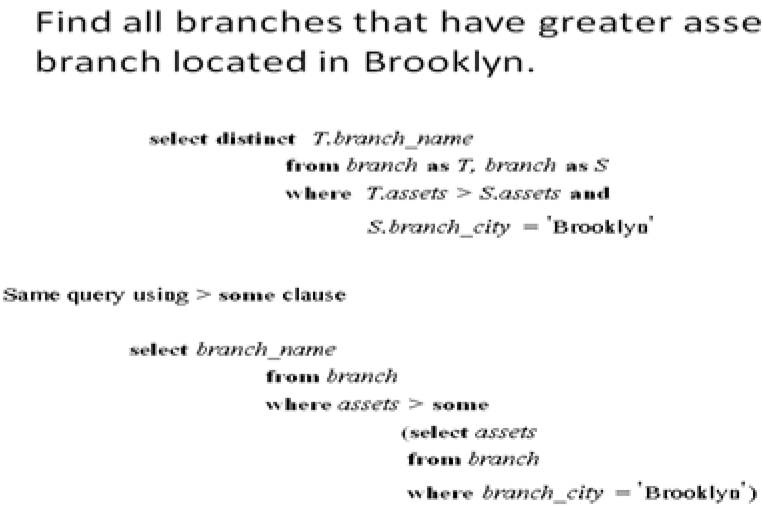
```
select distinct customer_name
       from borrower
       where customer_name in (select customer_name
                                  from depositor )
```



■ Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
       from borrower
       where customer_name not in (select customer_name
                                      from depositor )
```

“Some” Construct



Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
                 from branch as T, branch as S
                 where  T.assets > S.assets and
                          S.branch_city = 'Brooklyn'
```

- Same query using > some clause

```
select branch_name
             from branch
             where assets > some
                         (select assets
                          from branch
                          where branch_city = 'Brooklyn')
```

“All” Construct

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
            from branch
            where assets > all
                        (select assets
                        from branch
                        where branch_city = 'Brooklyn')
```

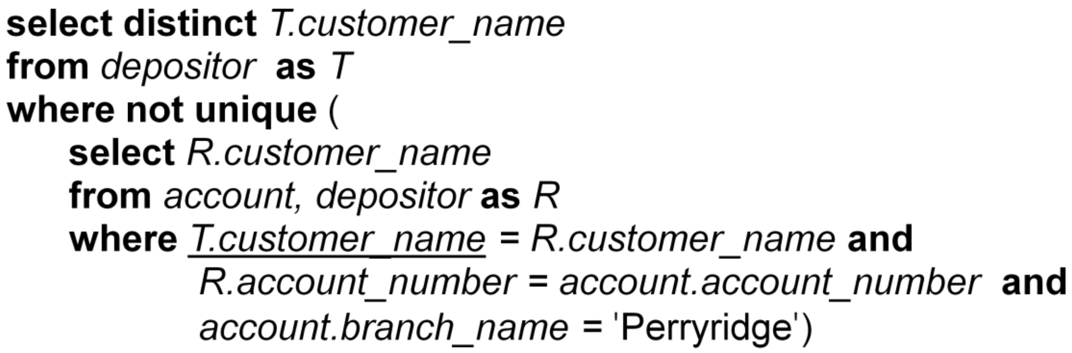
“Exists” Construct

Find all customers who have an account at all branches located in Brooklyn.
absence of duplicate tuples the **unique** construct tests whether a sub query has any duplicate tuples in its result.

Find all customers who have at most one account at the Perryridge branch.

```
select distinct S.customer_name
          from depositor as S
          where not exists (
                    (select branch_name
                    from branch
                    where branch_city = 'Brooklyn')
                    except
                    (select R.branch_name
                    from depositor as T, account as R
                    where T.account_number = R.account_number and
                              S.customer_name = T.customer_name ))
```

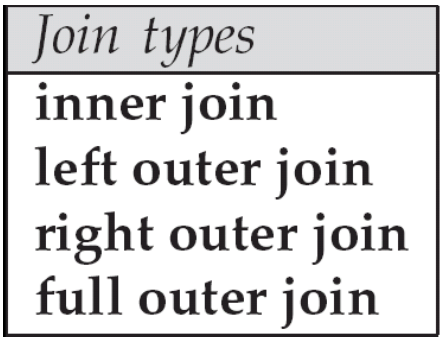
- Find all customers who have at least two accounts at the Perryridge branch.



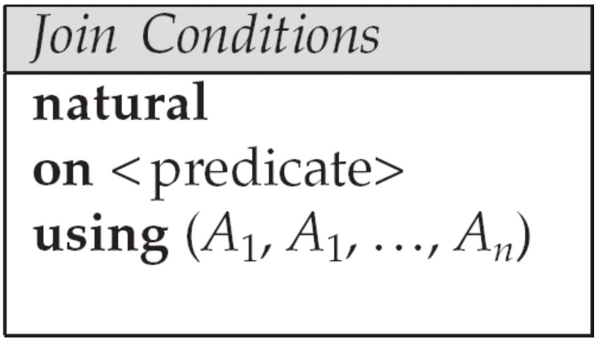
```
select distinct T.customer_name
from depositor  as  T
where not unique (
        select R.customer_name
        from account, depositor as R
        where T.customer_name = R.customer_name and
                  R.account_number = account.account_number  and
                  account.branch_name = 'Perryridge')
```

2.7.4.. OUTER JOINS

Join operations take two relations and return as a result another relation.These additional operations are typically used as subquery expressions in the **from** clause



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



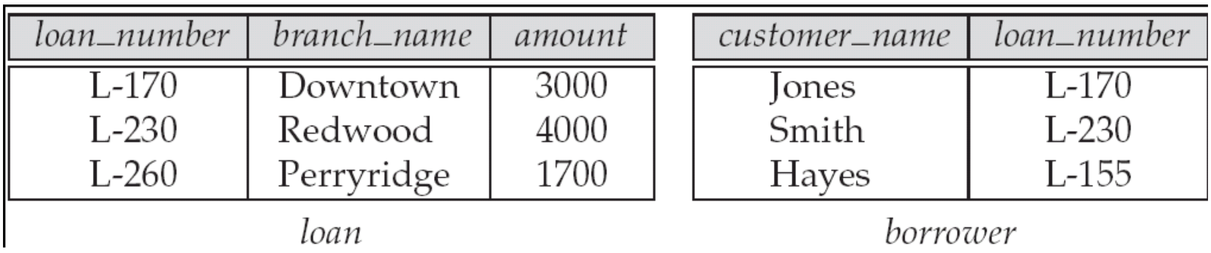
Join Conditions
natural
on < predicate>
using $(A_1, A_1, \ldots, A_n)$

Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.

Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Joined Relations – Datasets for Examples

Relation *loan*



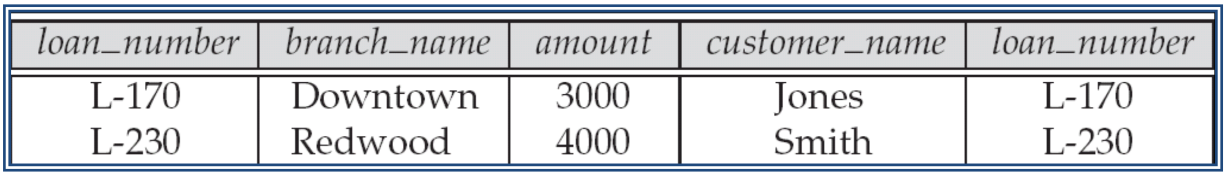
loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

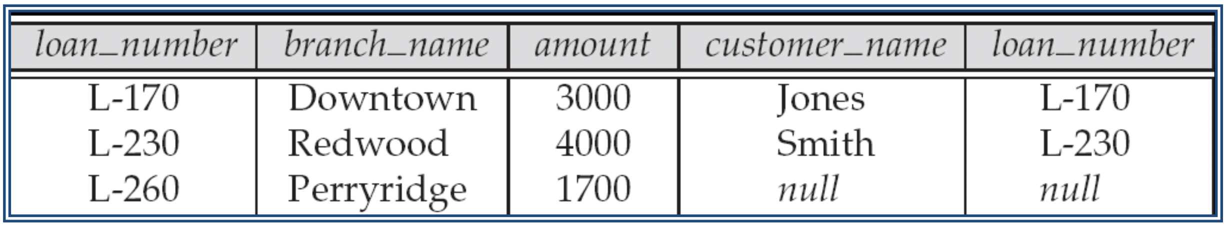
borrower

- Joined Relations – Examples



loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

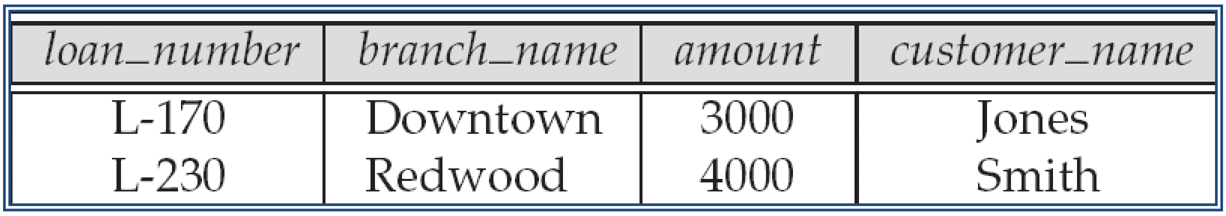
loan **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*



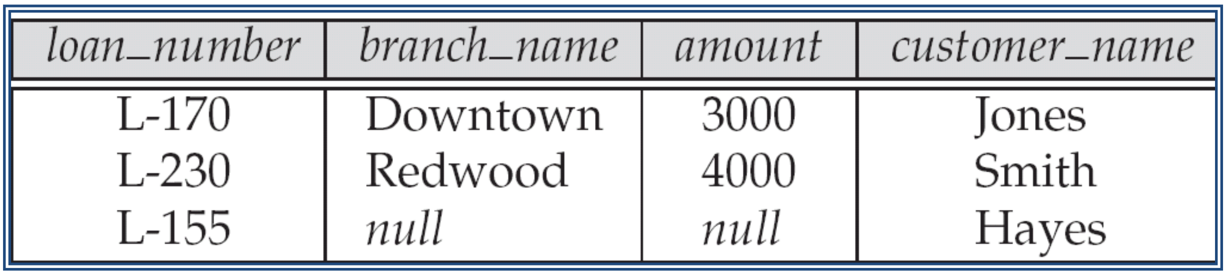
loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

loan **natural inner join** *borrower*

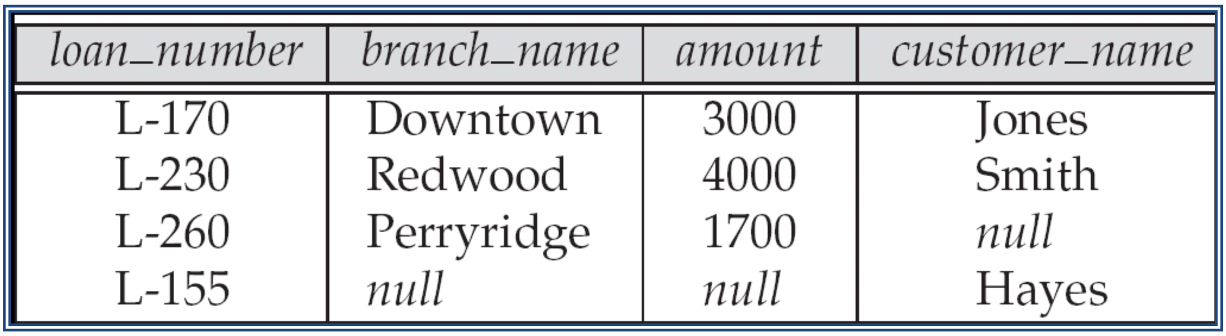


loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith



loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Joined Relations – Examples



loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition

2.7.5. 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.

2.8. COMPLEX INTEGRITY CONSTRAINTS IN SQL

1. Constraints over a Single Table

We can specify complex constraints over a single table using table constraints,

which have the form CHECK conditional-expression.
example, to ensure that rating must be an integer in the range 1 to 10, use:

```
CREATE TABLE Sailors (
 sid INTEGER,
 sname CHAR(10),
rating INTEGER, age REAL,
 PRIMARY KEY (sid),
CHECK ( rating >=1AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:
CREATE TABLE Reserves (sid INTEGER, bid INTEGER, day DATE, FOREIGN KEY (sid) REFERENCES Sailors FOREIGN KEY (bid) REFERENCES Boats CONSTRAINT noInterlakeRes CHECK (‘Interlake’ <> (SELECT B.bname FROM Boats B WHERE B.bid = Reserves.bid)))

When a row is inserted into Reserves or an existing row is modified, the conditional expression in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

3. Domain Constraints

A user can define a new domain using the CREATE DOMAIN statement, which makes use of CHECK constraints.

CREATE DOMAIN ratingval INTEGER DEFAULT 0 CHECK (VALUE >=1AND VALUE <= 10)
INTEGER is the base type for the domain ratingval, and every ratingval value must be of this type.The optional DEFAULT keyword is used to associate a default value with a domain.

3. Assertions: ICs over Several Tables

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold only if the associated table is nonempty. SQL supports the creation of assertions, which are constraints not associated with any one table.

Example: enforce the constraint that the number of boats plus the number of sailors should be less than 100.

```
CREATE TABLE Sailors (
sid INTEGER,
sname CHAR(10),
rating INTEGER,
age REAL,
PRIMARY KEY (sid),
CHECK ( rating >=1AND rating <= 10) CHECK ((SELECT COUNT (S.sid)
FROM Sailors S ) +(SELECT COUNT (B.bid) FROM Boats B ) < 100 ))
```

The best solution is to create an assertion, as follows:

```
CREATE ASSERTION smallClub CHECK ((SELECT COUNT (S.sid) FROM Sailors S )
+(SELECT COUNT (B.bid) FROM Boats B) < 100 )
```

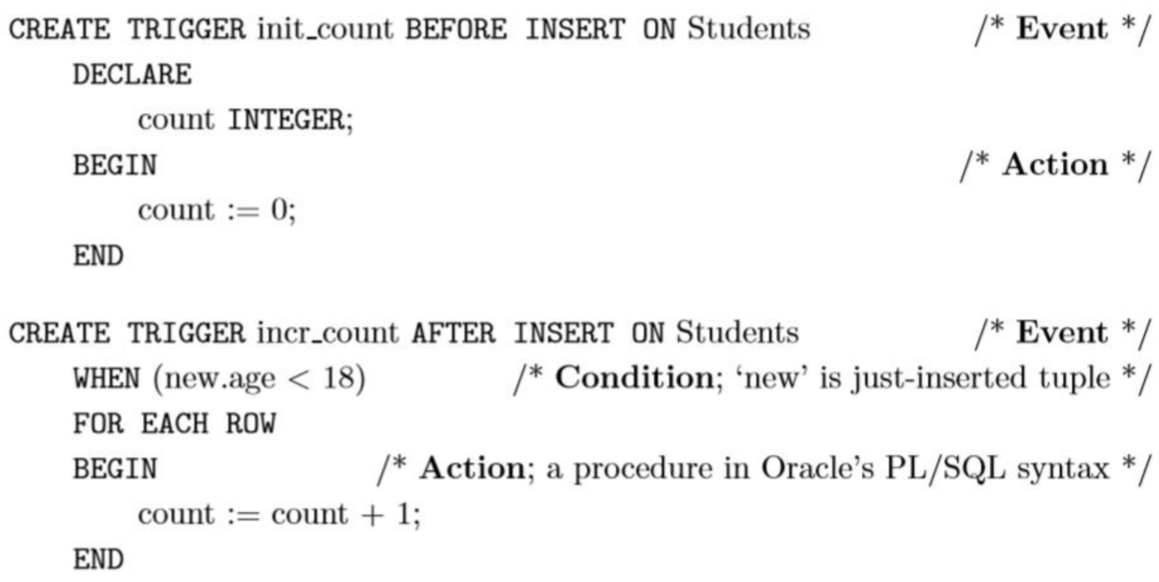
2.9. TRIGGERS AND ACTIVE DATA BASES

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA.

A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

- Event: A change to the database that activates the trigger.
- Condition: A query or test that is run when the trigger is activated.
- Action: A procedure that is executed when the trigger is activated and its con- dition is true.

Example: The trigger called incr count increments the counter for each inserted tuple that satisfies the condition age < 18.



```
CREATE TRIGGER init_count BEFORE INSERT ON Students              /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                                        /* Action */
        count := 0;
    END

CREATE TRIGGER incr_count AFTER INSERT ON Students               /* Event */
    WHEN (new.age < 18)          /* Condition; 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN                 /* Action; a procedure in Oracle's PL/SQL syntax */
        count := count + 1;
    END
```

```
CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
                INSERT
                 INTO YoungSailors(sid, name, age, rating)
                 SELECT sid, name, age, rating
                 FROM NewSailors N
                 WHERE N.age <= 18
```

2.9.10. ACTIVE DATABASES

- Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution.

- The effect of a collection of triggers can be very complex, and maintaining an active database can come very difficult. Often, a judicious use of integrity constraints can replace the use of triggers.

UNIT-III

3. INTRODUCTION TO SCHEMA REFINEMENT

Redundant storage of information is the root cause of these problems. Although decomposition can eliminate redundancy, it can lead to problems of its own and should be used with caution.

3.1. PROBLEMS CAUSED BY REDUNDANCY

Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

- Redundant storage: Some information is stored repeatedly.
- Update anomalies: If one copy of such repeated data is updated, an inconsis- tency is created unless all copies are similarly updated.
- Insertion anomalies: It may not be possible to store some information unless some other information is stored as well.
- Deletion anomalies: It may not be possible to delete some information without losing some other information as well.

3.2. DECOMPOSITIONS

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural.

Functional dependencies (ICs) can be used to identify such situations and to suggest refinetments to the schema.

rating	***hourly wages***
8	10
5	7

ssn	***Name***	***lot***	***rating***	***hours worked***
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32

612-67-4134	Madayan	35	8	40

The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of smaller relations.

- Each of the smaller relations contains a subset of the attributes of the original relation.
- We refer to this process as decomposition of the larger relation into the Smaller relations.We can deal with the redundancy in Hourly Emps by decomposing it into two relations:
- Hourly Emps2(*ssn*, *name*, *lot*, *rating*, *hours worked*)
- Wages(*rating*, *hourly wages*)

3.3. PROBLEM RELATED TO DECOMPOSITION

Unless we are careful, decomposing a relation schema can create more problems than it Solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

3.4. FUNCTIONAL DEPENDENCIES

Functional dependency (FD) is set of constraints between two attributes in a relation. Functional dependency says that if two tuples have same values for attributes A1, A2,..., An then those two tuples must have to have same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by arrow sign (→), that is X→Y, where X functionally determines Y. The left hand side attributes determines the values of attributes at right hand side.

3.4.1.Armstrong's Axioms

If F is set of functional dependencies then the closure of F, denoted as F^+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are set of rules, when applied repeatedly generates closure of functional dependencies.

- **Reflexive rule:** If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule:** if a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule:** Same as transitive rule in algebra, if a → b holds and b → c holds then a → c also hold. a → b is called as a functionally determines b.

3.4.2. TRIVIAL FUNCTIONAL DEPENDENCY

- **Trivial:** If an FD X → Y holds where Y subset of X, then it is called a trivial FD. Trivial FDs are always hold.
- **Non-trivial:** If an FD X → Y holds where Y is not subset of X, then it is called non-trivial FD.
- **Completely non-trivial:** If an FD X → Y holds where x intersect Y = Φ, is said to be completely non-trivial FD.

3.5. REASONING ABOUT FDS

Given a set of FDs over a relation schema R, there are typically several additional FDs that hold over R whenever all of the given FDs hold. As an example, consider:

> Workers(ssn, name, lot, did, since)
> We know that ssn → did holds, since ssn is the key,
> and FD did→ lot is given to hold.

Therefore, in any legal instance of Workers, if two tuples have the same ssn value, they must have the same did value (from the first FD), and because they have the same did value, they must also have the same lot value (from the second FD).

Thus, the FD ssn → lot also holds on Workers. We say that an FD f is implied by a given setFof FDs if f holds on every relation instance that satisfies all dependencies in F, that is,f holds whenever all FDs in F hold. Note that it is not sufficient for f to hold on some instance that satisfies all dependencies in F; rather, f must hold on every instance that satisfies all dependencies in F.

3.6. NORMAL FORMS:

- ***Definition :****Normalization* is the process of organizing the fields and tables of a relational database to minimize redundancy and dependency.

- The normal forms based on FDs are first *normal form (1NF), second normal form (2NF), third normal form (3NF),* and *Boyce-Codd normal form (BCNF).*

- These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF,every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF.

- A relation is in **first normal form** if every field contains only atomic values, that is, not lists or sets.

- This requirement is implicit in our definition of the relational model.

- Although some of the newer database systems are relaxing this requirement 2NF is mainly of historical interest. 3NF and BCNF are important from a database design standpoint.

3.6.1. FIRST NORMAL FORM

- A relation schema is said to be in first normal form if the attributes values in the relation are atomic, i.e there should be no repeated values in a particular column

- A attribute is said to be value atomic value if it contains only a single.

Example First Normal Form

Emp_id	**Emp_section_id**	**Emp_name**	**Emp_address**	**dependents**
0012	575	Manideep	Hyderabad	Father, Mother,Brother
0013	572	Bhaskar reddy	Delhi	Wife, Mother, Son
0014	5A0	Priyanka	Bangalore	Brother, Sister
0015	5B8	Anusha reddy	Hyderabad	Sister, Mother

Here,The column dependents have non atomic values, In order to convert this relation in INF,we have to convert these non atomic values to atomic values

Emp_id	**Emp_section_id**	**Emp_name**	**Emp_address**	**Dependents**
0012	575	Manideep	Hyderabad	Father,
0012	575	Manideep	Hyderabad	Mother
0012	575	Manideep	Hyderabad	Brother
0013	572	Bhaskar reddy	Delhi	Wife
0013	572	Bhaskar reddy	Delhi	Mother
0013	572	Bhaskar reddy	Delhi	Son
0014	5A0	Priyanka	Bangalore	Brother
0014	5A0	Priyanka	Bangalore	Sister
0015	5B8	Anusha reddy	Hyderabad	Sister
0015	5B8	Anusha reddy	Hyderabad	Mother

The relation employee is in 1NF since the column dependents have atomic value But the other attributes i.e. emp_id, emp_section_id, emp_name, emp_address are all repeating and forming a group called repeated groups.

3.6.2. SECOND NORMAL FORM

- A relation is said to be in 1NF and every non Key attribute is fully functionally dependent on primary key attribute

- If any one of the following conditions are satisfied then a relation(which is in 1NF) is in 2NF

  Rules:

  1. There should be only one attribute associated with primary key
  2. There must be no non key attributes in the relation

Example:

- Student(student_id,class_id,name,cource,time)
- (student_id,class_id,)is the primary key,
- A student can attend different course in different classes at different times.

Consider a simple example of student relation

Student_id	Class_id	Name	Cource_id	time
0123	502	Ravi	312	10/10
0124	503	Kumar	313	10/07
0125	502	Mahesh	312	10/15
0126	504	mehta	460	10/08

The above relation is not in2NF,as the name of the student can be determined by student_id. there ,a non key attribute(name) is functionally depend on a part of key (student_id)

3.6.3. THIRD NORMAL FORM

- A relation R in 3NF if and only if it is in 2NF and every non key column does not depend on another non key column

    - All nonprime attributes of R must be non-transitively functionally dependent on a key of the relation

- Relation R with FDs *F* is in 3NF if, for all X   A in
    - A   X (called a *trivial* FD), or
    - X contains a key for R, or
    - A is part of some key for R.
- *Minimality* of a key is crucial in third condition above!
- If R is in BCNF, obviously in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).
    - *Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.*

SUPPLIER (SNAME, STREET, CITY, STATE, TAX)
SNAME → STREET, CITY, STATE
STATE → TAX (non key → non key)
SNAME → STATE → TAX (transitive FD)

    - solution: decompose the relation

SUPPLIER2 (SNAME, STREET, CITY, STATE)
TAXINFO (STATE, TAX)

3.6.4.Boyce-Codd NORMAL FORM (BCNF)

- Reln R with FDs *F* is in BCNF if, for all X   A in
    - A   X (called a *trivial* FD), or
    - X contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.

    - No dependency in R that can be predicted using FDs alone.

    - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.

    - If example relation is in BCNF, the 2 tuples must be identical (since x is a key).

3.7. PROPERTIES OF DECOMPOSITIONS

3.7.1. DECOMPOSITION OF A RELATION SCHEME

- Suppose that relation R contains attributes *A1* ... *An.* A *decomposition* of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

Decompositions should be used only when needed.

- SNLRWH has FDs S   SNLRWH and R   W
- Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:i.e., we decompose SNLRWH into SNLRH and RW

The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

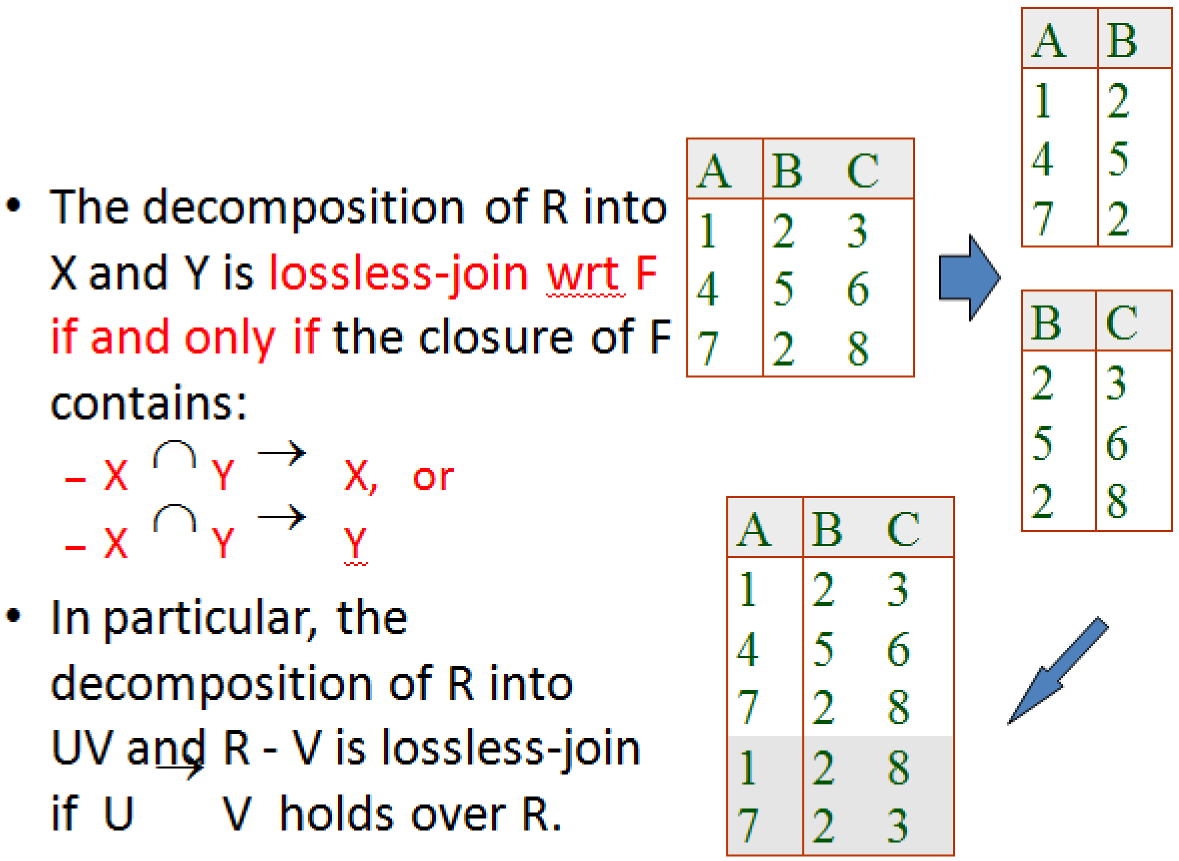
3.7.2. LOSSLESS JOIN DECOMPOSITIONS:

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

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

- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold! If it does, the decomposition is lossless-join.
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- *It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem (2).)*

More on Lossless Join



- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - $X \cap Y \to X$, or
  - $X \cap Y \to Y$
- In particular, the decomposition of R into UV and R - V is lossless-join if U $\to$ V holds over R.

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

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

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

- Dependency Preserving Decomposition
- Consider CSJDPQV, C is key, JP C and SD P.

  BCNF decomposition: CSJDQV and SDP

  Problem: Checking JP C requires a join!

3.8. DEPENDENCY PRESERVING DECOMPOSITION (INTUITIVE)

If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3).)*

Projection of set of FDs F: If R is decomposed into X, ... projection of F onto X enoted F_X) is the set of FDs UV in F^+ (*closure of F*) such that U, V are in X.

Decomposition of R into X and Y is *dependency preserving* if $(F_X$ union $F_Y)^+ = F^+$

i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.

Important to consider F^+, not F, in this definition:

ABC, A B, B C, C A, decomposed into AB and BC.Is this dependency preserving? Is C A preserved????? Dependency preserving does not imply lossless join:ABC, A B, decomposed into AB and BC.And vice-versa!

Decomposition into BCNF

Consider relation R with FDs F. If X Y violates BCNF, decompose R into R - Y and XY. Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

– e.g., CSJDPQV, key C, JP C, SD P, J S

– To deal with SD P, decompose into SDP, CSJDQV.

– To deal with J S, decompose CSJDQV into JS and CJDQV

In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.

– e.g., CSZ, CS Z, Z C

– Can't decompose while preserving 1st FD; not in BCNF.

– Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP C, SD P and J S).However, it is a lossless join decomposition.In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition. JPC tuples stored only for checking FD! (*Redundancy!*)

Decomposition into 3NF

Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).

To ensure dependency preservation, one idea:

If X Y is not preserved, add relation XY.

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

3.9. SCHEMA REFINEMENT IN DATA BASE DESIGN:

Constraints on an Entity Set

Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:*{ ssn }-> { ssn, name, lot, rating, hourly wages, hours worked}*

For brevity, we will write this FD as *S -> SNLRWH*, using a single letter to denote each attribute In addition, the constraint that the *hourly wages* attribute is determined by the *rating* attribute is an FD: *R -> W*.

Constraints on a Relationship Set

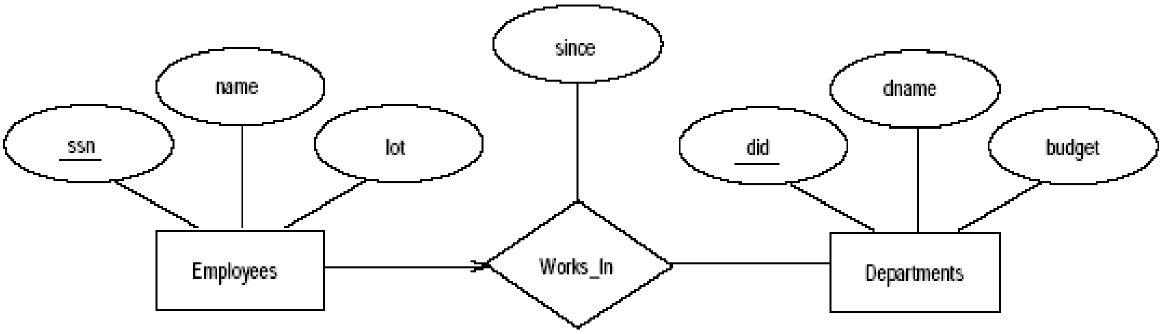
The previous example illustrated how FDs can help to rene the subjective decisions made during ER design, but one could argue that the best possible ER diagram would have led to the same nall set of relations.

Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

Identifying Attributes of Entities

in particular, it shows that attributes can easily be associated with the `wrong' entity set during ER design. The ER diagram shows a relationship set called Works In that is similar to the Works In relationship set Using the key constraint, we can translate this ER diagram into two relations:

Workers(*ssn*, *name*, *lot*, *did*, *since*)



Identifying Entity Sets

Let Reserves contain attributes *S*, *B*, and *D* as before, indicating that sailor *S* has a reservation for boat *B* on day *D*.

In addition, let there be an attribute *C* denoting the credit card to which the reservation is charged.

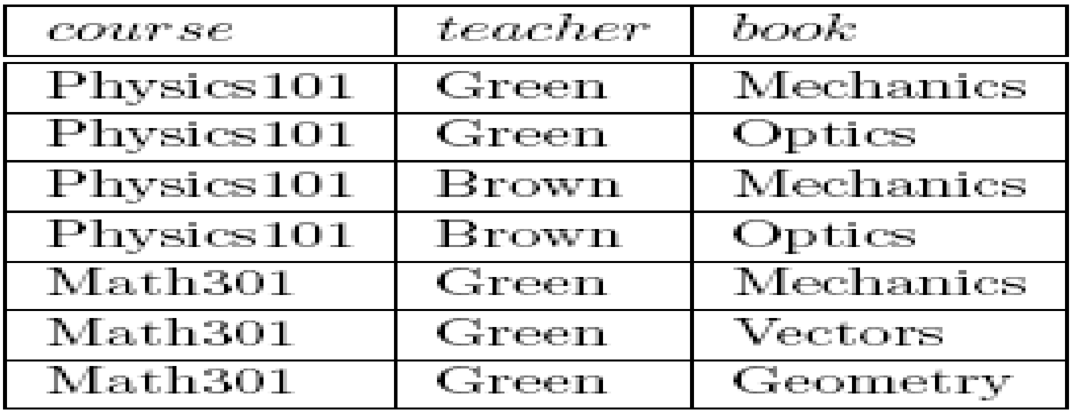
Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD

S -> *C*. This constraint indicates that in relation Reserves, we store the credit card number for a sailor as often as we have reservations for that sailor, and we have redundancy and potential update anomalies.

3.9. MULTIVALUED DEPENDENCIES:

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*.The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course.

There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor.



course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

There are three points to note here:

The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.

There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics 101 is recorded once per potential teacher.

The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.Let *R* be a relation schema and let *X* and *Y* be subsets of the attributes of *R*. Intuitively, the multivalued dependency *X !! Y* is said to hold over *R* if, in every legal The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be epressed in terms of FDs. This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*.Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate.Three of the additional rules involve only MVDs:

MVD Complementation: If $X \to\to Y$, then $X \to\to R - XY$
MVD Augmentation: If $X \to\to Y$ and $W >\ \ Z$, then $WX \to\to YZ$.
MVD Transitivity: If $X \to\to Y$ and $Y \to\to Z$, then $X \to\to (Z - Y)$.

3.9.1. FOURTH NORMAL FORM

R is said to be in **fourth normal form (4NF)** if for every MVD $X \to\to Y$ that holds over *R*, one of the following statements is true:

- *Y subset of X* or $XY = R$, or
- *X* is a super key.

3.10. JOIN DEPENDENCIES

A join dependency is a further generalization of MVDs. A **join dependency** (JD) ∞{ *R*1,….. *Rn }* is said to hold over a relation *R* if *R*1*,….* *Rn* is a lossless-join decomposition of *R*.

An MVD *X* ->-> *Y* over a relation *R* can be expressed as the join dependency ∞{ XY,X(R−Y)*}*

As an example, in the *CTB* relation, the MVD *C* ->->*T* can be expressed as the join dependency ∞{ CT, CB*}* Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

3.11. FIFTH NORMAL FORM

A relation schema *R* is said to be in **fth normal form (5NF)** if for every JD ∞{ *R*1,…. *Rn }* that holds over *R*, one of the following statements is true:

Ri = *R* for some *i*, or The JD is implied by the set of those FDs over *R* in which the left side is a key for *R*.

The following result, also due to Date and Fagin, identies conditions again, detected using only FD information under which we can safely ignore JD information.

If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also
in 5NF.

3.12. INCLUSION DEPENDENCIES

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about. In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design the main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency. Most inclusion dependencies in practice are *key-based*, that is, involve only keys.

UNIT - IV

4.1. TRANSACTION CONCEPT

A **Transaction** is a *unit* of program execution that accesses and possibly updates various data items.

Example transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

Two main issues to deal with:
Failures of various kinds, such as hardware failures and system crashes
Concurrent execution of multiple transactions

Example of Fund Transfer Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B)*

Atomicity requirement
if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

Failure could be due to software or hardware the system should ensure that updates of a partially executed transaction are not reflected in the database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)

```
5.    $B := B + 50$
6.    write($B$)
```

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction In general, consistency requirements include Explicitly specified integrity constraints such as primary keys and foreign keys Implicit integrity constraints Example sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand A transaction must see a consistent database. During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully the database must be consistent Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

```
            T1                              T2
1.          read($A$)
2.          $A := A – 50$
3.          write($A$)
            read(A), read(B), print(A+B)
4.          read($B$)
5.          $B := B + 50$
6.          write($B$ )
```

Isolation can be ensured trivially by running transactions **serially** that is, one after the other. However, executing multiple transactions concurrently has significant benefits.

ACID Properties

Atomicity. Either all operations of the transaction are properly reflected in the database or none are.

Consistency. Execution of a transaction in isolation preserves the consistency of the database.

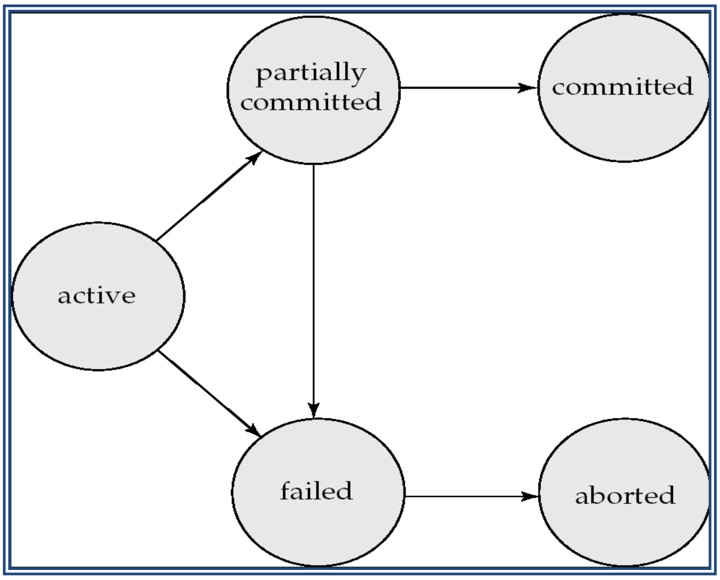
Isolation. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

4.2. TRANSACTION STATE

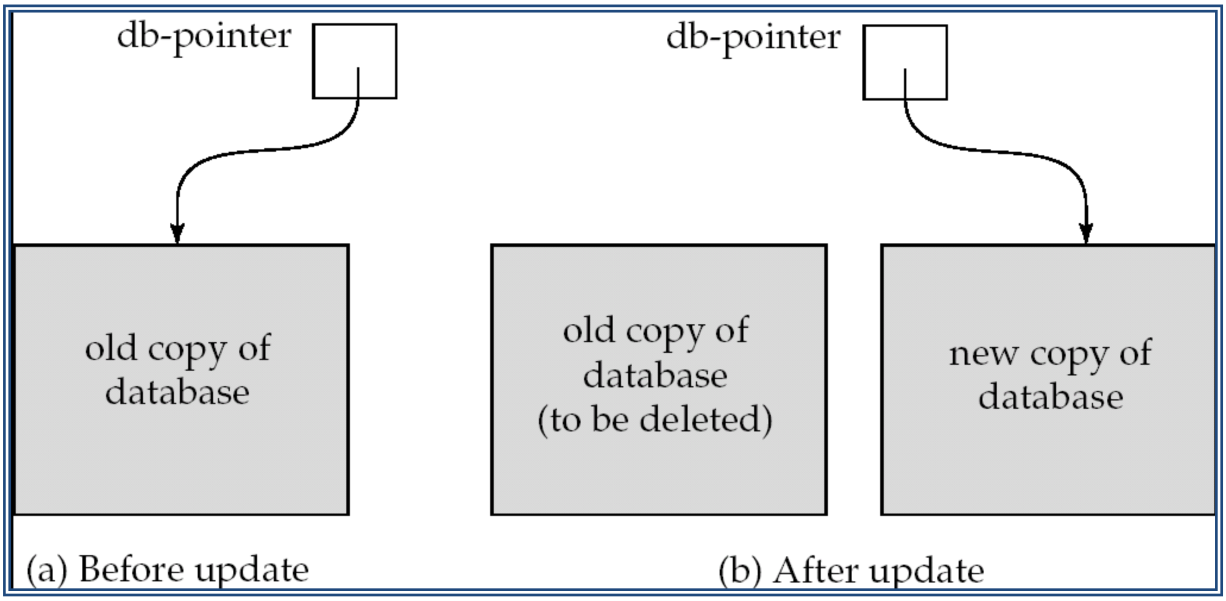
- **Active** – the initial state; the transaction stays in this state while it is executing

- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its State prior to the start of the transaction. Two options after it has been aborted: restart the transaction can be done only if no internal logical error kill the transaction
- **Committed** – after successful completion.



4.3. IMPLEMENTATION OF ATOMICITY AND DURABILITY

The **recovery-management** component of a database system implements the support for atomicity and durability. Example of the ***shadow-database*** scheme:all updates are made on a *shadow copy* of the database **db_pointer** is made to point to the updated shadow copy after the transaction reaches partial commit and all updated pages have been flushed to disk.
db_pointer always points to the current consistent copy of the database.In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.



The shadow-database scheme: Assumes that only one transaction is active at a time. Assumes disks do not fail Useful for text editors, but extremely inefficient for large databases (why?) Variant called shadow paging reduces copying of data, but is still not practical for large databases does not handle concurrent transactions

4.4. CONCURRENT EXECUTIONS

Multiple transactions are allowed to run concurrently in the system. Advantages are: increased processor and disk utilization, leading to better transaction throughput

Example one transaction can be using the CPU while another is reading from or writing to the disk reduced average response time for transactions: short transactions need not wait behind long ones Concurrency control schemes – mechanisms to achieve isolation that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

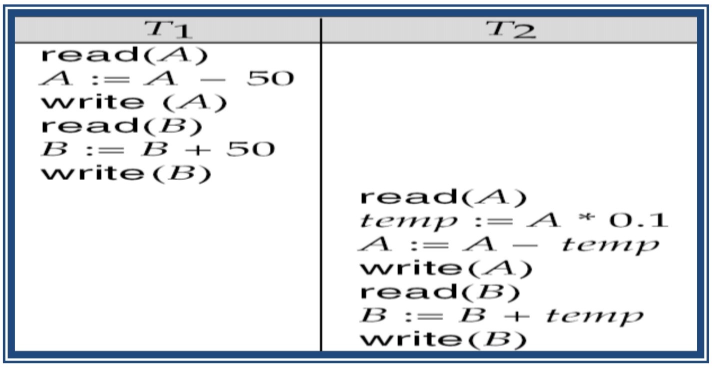
Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed a schedule for a set of transactions must consist of all instructions of those transactions must preserve the order in which the instructions appear in each individual ransaction.

A transaction that successfully completes its execution will have commit instructions as the last statement by default transaction assumed to execute commit instruction as its last step
A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedule 1

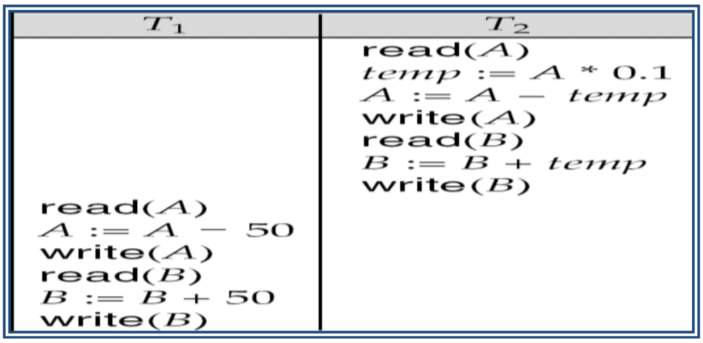
- Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B*.

- A serial schedule in which T_1 is followed by T_2 :



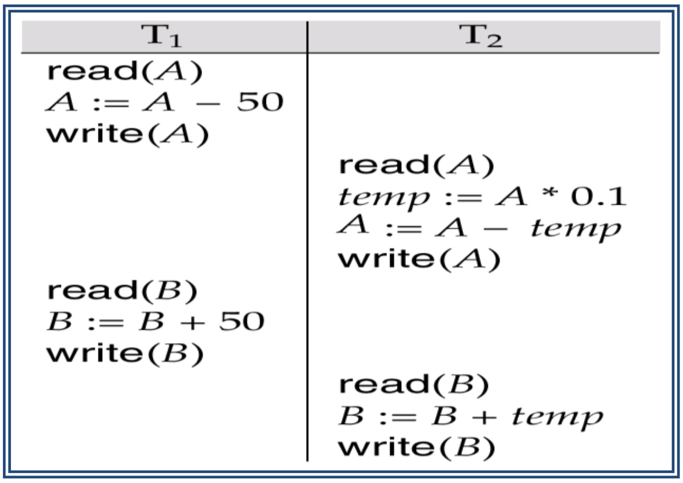
T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2



T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 3



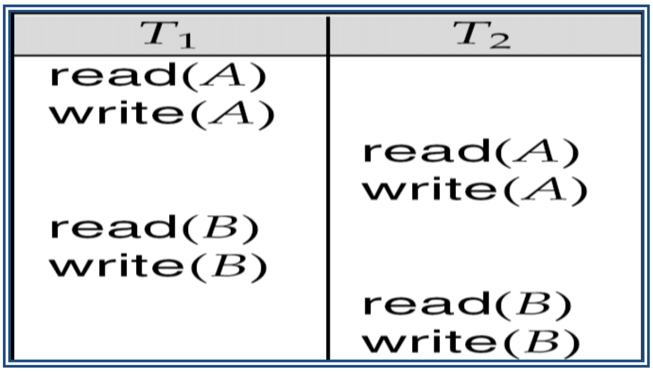
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Let T_1 and T_2 be the transactions defined previously.The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

4.5. SERIALIZABILITY

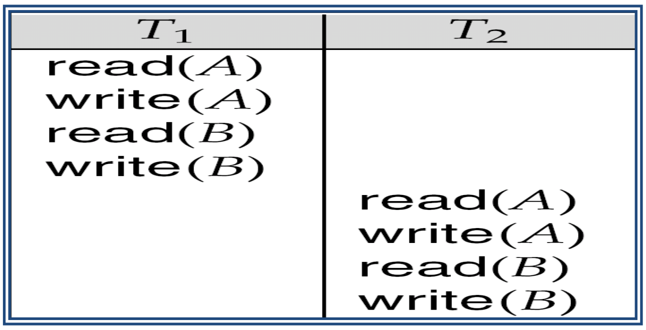
Basic Assumption – Each transaction preserves database consistency.Thus serial execution of a set of transactions preserves database consistency.A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

2.**view serializability**



T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Simplified view of transactions We ignore operations other than **read** and **write** instructions; We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions. Conflicting Instructions Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

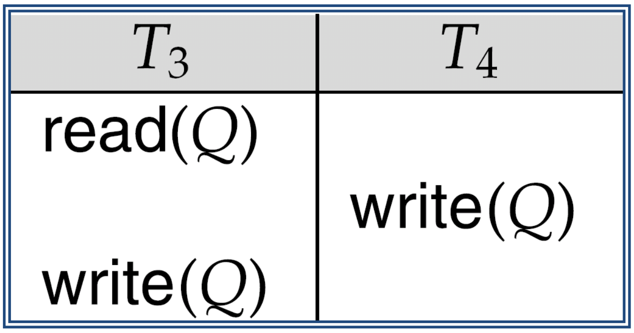
1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

If a schedule S can be transformed into a schedule $S´$ by a series of swaps of non-conflicting instructions, we say that S and $S´$ are **conflict equivalent**.

We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule



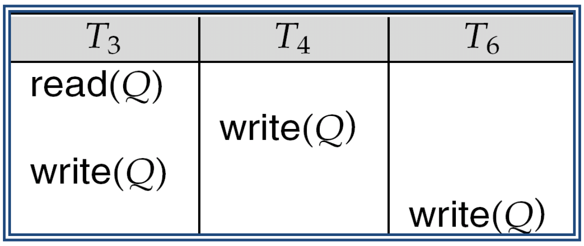
T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.Therefore Schedule 3 is conflict serializable. Example of a schedule that is not conflict serializable:We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q*,If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.

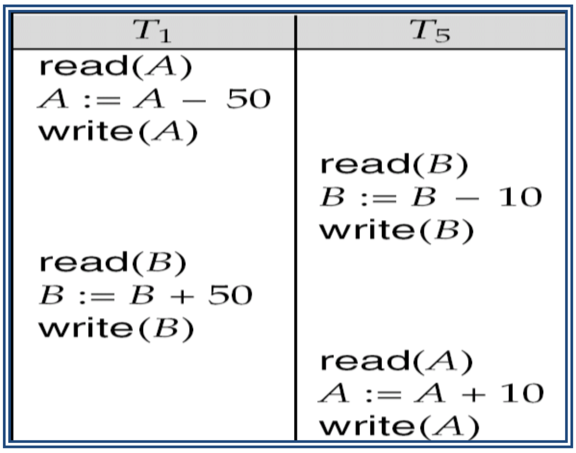
If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*. As can be seen, view equivalence is also based purely on **reads** and **writes** alone.



T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

A schedule *S* is view serializable if it is view equivalent to a serial schedule.Every conflict serializable schedule is also view serializable.Below is a schedule which is view-serializable but *not* conflict serializable.

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has blind writes. Other Notions of Serializability

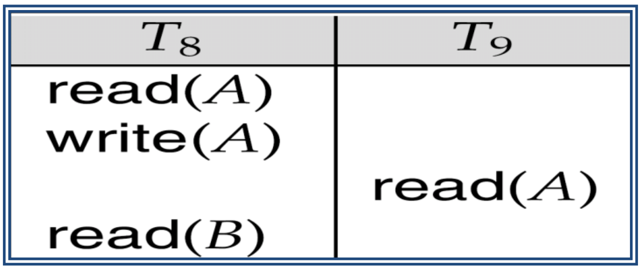


T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it. Determining such equivalence requires analysis of operations other than read and write.

4.6. RECOVERABILITY

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

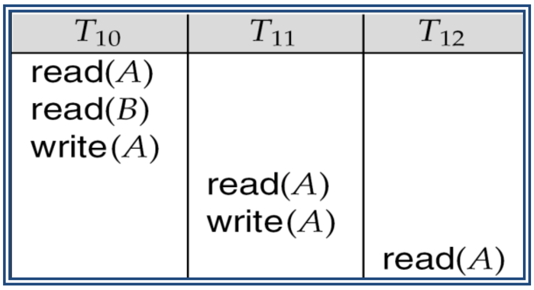


T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)



T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

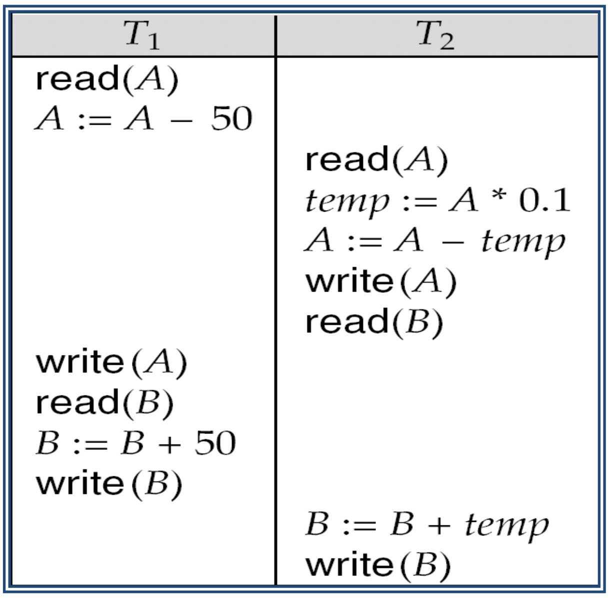
If T_{10} fails, T_{11} and T_{12} must also be rolled back.Can lead to the undoing of a significant amount of work **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.Every cascadeless schedule is also recoverable It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

A database must provide a mechanism that will ensure that all possible schedules are either conflict or view serializable, and are recoverable and preferably cascadeless A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency Are serial schedules recoverable/cascadeless? Testing a schedule for serializability *after* it has executed is a little too late! **Goal** – to develop concurrency control protocols that will assure serializability

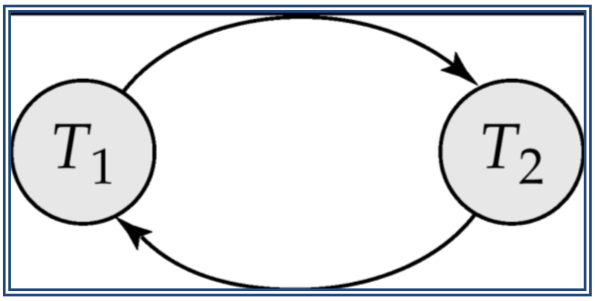
4.7. IMPLEMENTATION OF ISOLATION

Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.A policy in which only one transaction can execute at a time generates serial schedules,but provides a poor degree of concurrency.Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

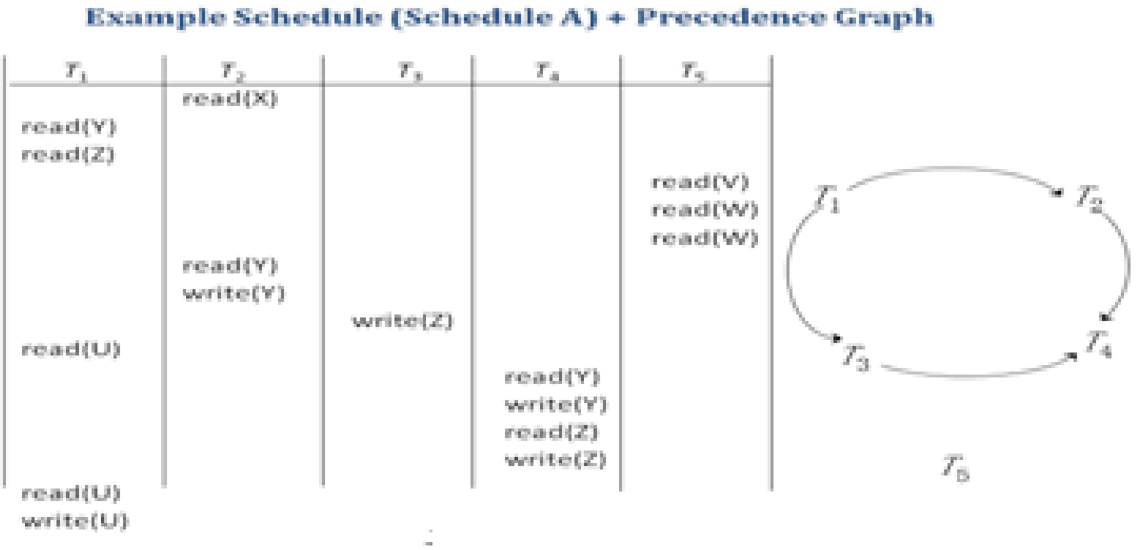


T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

4.8. TESTING FOR SERIALIZABILITY



- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.



Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			

read(Y)
read(Z)				

				read(V)
				read(W)
				read(W)

	read(Y)
	write(Y)			

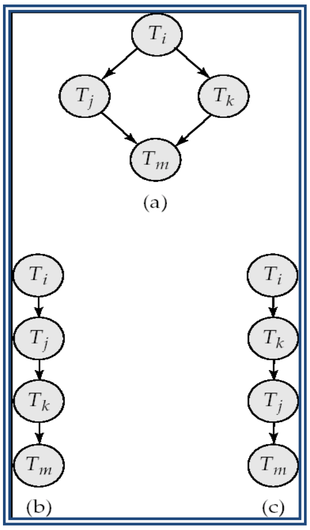
		write(Z)		
read(U)				

			read(Y)
			write(Y)
			read(Z)
			write(Z)	

read(U)
write(U)				

T_1
T_2
T_3
T_4
T_5

Test for Conflict Serializability A schedule is conflict serializable if and only if its precedence graph is acyclic.Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)



If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph.
For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$ Are there others?

Test for View Serializability

The precedence graph test for conflict serializability cannot be used directly to test for view serializability.Extension to test for view serializability has cost exponential in the size of the precedence graph.The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus existence of an efficient algorithm is *extremely* unlikely.
However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

4.9. CONCURRENCY CONTROL

Concurrency Control vs. Serializability Tests

Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .Concurrency control protocols generally do not examine the precedence graph as it is being created Instead a protocol imposes a discipline that avoids nonseralizable schedules.Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable E.g. a read-only transaction that wants to get an approximate total balance of all Accounts. Example. database statistics computed for query optimization can be approximate (why?) Such transactions need not be serializable with respect to other transactions Tradeoff accuracy for performance Levels of Consistency in SQL-92 **Serializable** — default **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable it may find some records inserted by a transaction but not find others.

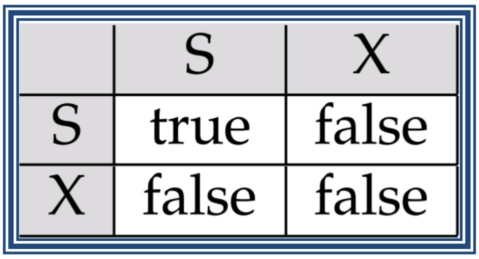
Read committed — only committed records can be read, but successive reads of recor may return different (but committed) values.

Read uncommitted — even uncommitted records may be read.Transaction Definition in SQL Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.In SQL, a transaction begins implicitly.A transaction in SQL ends by:**Commit work** commits current transaction and begins a new one.

Rollback work causes current transaction to abort In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully Implicit commit can be turned off by a database directive E.g. in JDBC,connection. setAutoCommit(false);

4.9.1. LOCK BASED PROTOCOLS

A lock is a mechanism to control concurrent access to a data item



	S	X
S	true	false
X	false	false

Fig:Lock-compatibility matrix

Data items can be locked in two modes :

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.

2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions

- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.
  Example :if a transaction performing locking:

```
T2: lock-S(A);
    read (A);
    unlock(A);
    lock-S(B);
    read (B);
    unlock(B);
    display(A+B)
```

Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

- A **locking protocol** is a set of rules followed by all transactions while requesting and

releasing locks. Locking protocols restrict the set of possible schedules.Pitfalls of Lock-Based Protocols Consider the partial schedule Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.Such a situation is called a **deadlock**. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

Starvation is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.The same transaction is repeatedly rolled back due to deadlocks.Concurrency control manager can be designed to prevent starvation.

Two-Phase Locking Protocol

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock). All locks are released after commit or abort

Implementation of Locking

A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).The requesting transaction waits until its request is answered The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.

Two-phase locking *does not* ensure freedom from deadlocks

- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

4.9.2. TIMESTAMP BASED PROTOCOLS

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

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

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

write(Q) successfully.

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

read(Q) successfully.

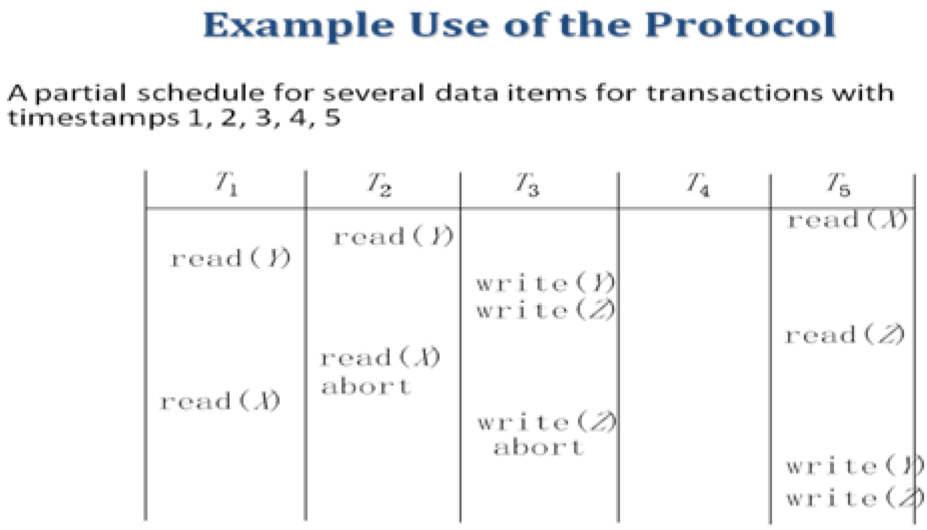
The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

Suppose a transaction T_i issues a **read**(Q)
If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was
already overwritten.Hence, the **read** operation is rejected, and T_i is rolled back.

If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).

Suppose that transaction T_i issues **write**(Q).
If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
Hence, the **write** operation is rejected, and T_i is rolled back.

If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
Hence, this **write** operation is rejected, and T_i is rolled back.Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).



Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

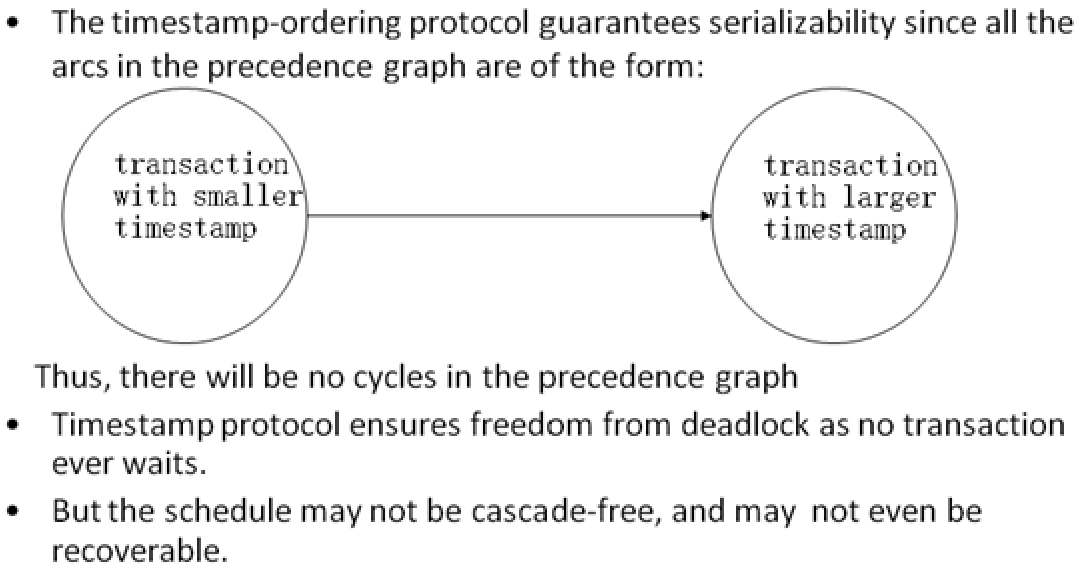
T_1	T_2	T_3	T_4	T_5
				read(X)
	read(Y)			
read(Y)				

		write(Y)
		write(Z)		

				read(Z)
	read(X) abort			
read(X)				
		write(Z) abort		

				write(Y)
				write(Z)

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5
Correctness of Timestamp-Ordering Protocol



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

transaction with smaller timestamp
transaction with larger timestamp

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.

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.

Thomas' Write Rule Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances. When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}. Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.Otherwise this protocol is the same as the timestamp ordering protocol.

- ✓ Thomas' Write Rule allows greater potential concurrency.
- ✓ Allows some view-serializable schedules that are not conflict-serializable.

4.9.3. VALIDATION BASED PROTOCOL

Execution of transaction T_i is done in three phases.

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

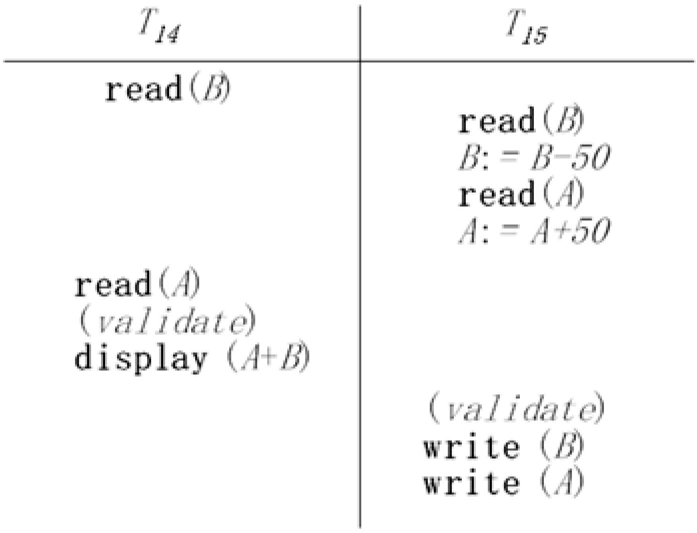
2. **Validation phase**: Transaction T_i performs a ``validation test" to determine if local variables can be written without violating serializability.

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

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.Assume for simplicity that the validation and write phase occur together,atomically and serially i.e., only one transaction executes validation/write at a time.Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation.
Each transaction T_i has 3 timestamps

- Start(T_i) : the time when T_i started its execution
- Validation(T_i): the time when T_i entered its validation phase
- Finish(T_i) : the time when T_i finished its write phase Serializability order is determined by timestamp given at validation time, to increase concurrency.

- Example of schedule produced using validation



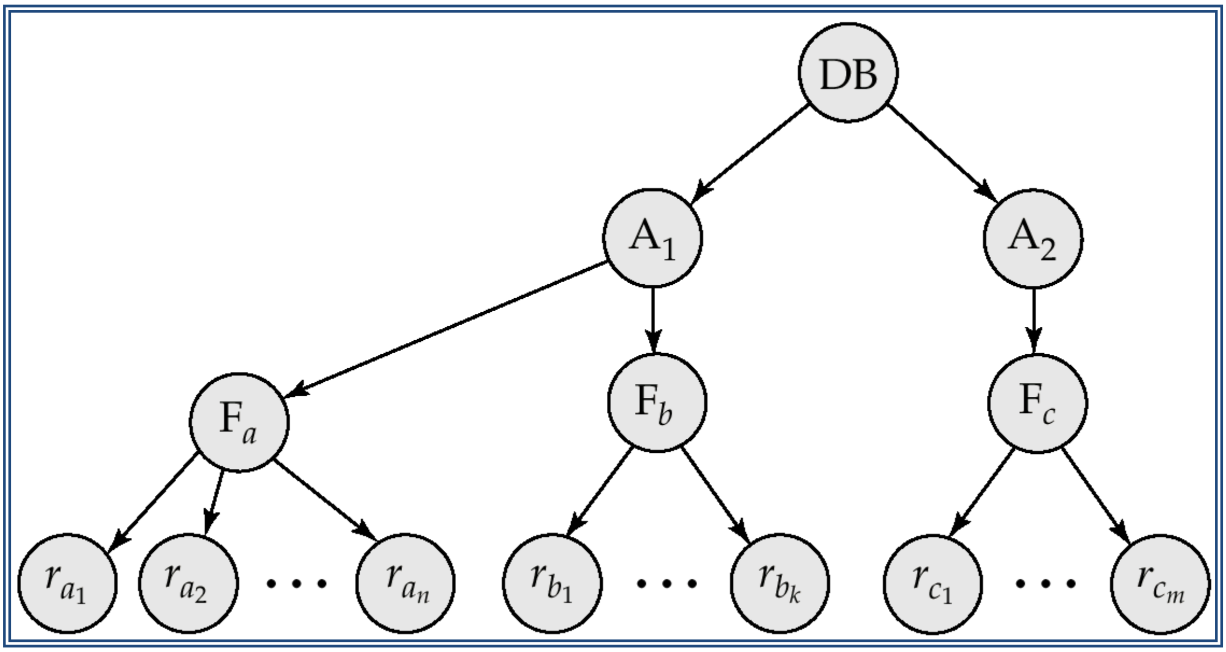
T_{14}	T_{15}
read(*B*)	
	read(*B*)
	B: = *B*−*50*
	read(*A*)
	A: = *A*+*50*
read(*A*)	
(*validate*)	
display (*A*+*B*)	
	(*validate*)
	write (*B*)
	write (*A*)

4.9.4. MULTIPLE GRANULARITIES

Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones Can be represented graphically as a tree (but don't confuse with tree-locking protocol) When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

Granularity of locking (level in tree where locking is done):**ine granularity** (lower in tree): high concurrency, high locking overhead **coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy



The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

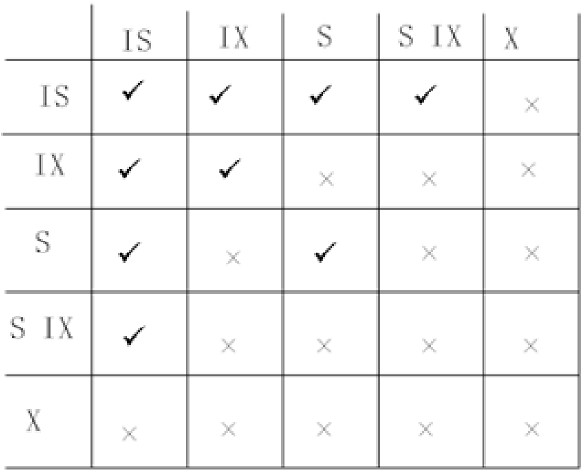
In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

intention-shared (IS): indicates explicit locking at a lower level of the tree but only with shared locks.

intention-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks

shared and intention-exclusive (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

- The compatibility matrix for all lock modes is:



	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
S IX	✓	×	×	×	×
X	×	×	×	×	×

4.10. RECOVERY SYSTEM

4.10.1. Failure classification

To see where the problem has occurred we generalize the failure into various categories, as follows:

Transaction failure

When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Reason for transaction failure could be:

- **Logical errors:** where a transaction cannot complete because of it has some code error or any internal error condition
- **System errors:** where the database system itself terminates an active transaction because DBMS is not able to execute it or it has to stop because of some system condition. For example, in case of deadlock or resource unavailability systems aborts an active transaction.

System crash

There are problems, which are external to the system, which may cause the system to stop abruptly and cause the system to crash. For example interruption in power supply, failure of underlying hardware or software failure.

Examples may include operating system errors.

Disk failure:

In early days of technology evolution, it was a common problem where hard disk drives or storage drives used to fail frequently.

Disk failures include formation of bad sectors, unreachability to the disk, disk head crash or any other failure, which destroys all or part of disk storage

4.11.RECOVERY AND ATOMICITY

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all database modifications made by T_i or none at all. Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications have been made but before all of them are made.To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.We study two approaches:

log-based recovery, and **shadow-paging** .
We assume (initially) that transactions run serially, that is, one after the other.

Recovery Algorithms

Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures.

Recovery algorithms have two parts

Actions taken during normal transaction processing to ensure enough information exists to recover from failures

Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

4.11.1.Log-based recovery

A **log** is kept on stable storage. The log is a sequence of **log records**, and maintains a record of update activities on the database.

When transaction T_i starts, it registers itself by writing a <T_i **start**>log record

Before T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written,

where V_1 is the value of X before the write, and V_2 is the value to be written to X.

Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.

When T_i finishes it last statement, the log record <T_i **commit**> is written.

We assume for now that log records are written directly to stable storage (that is, they are not buffered)

Two approaches using logs

– Deferred database modification

– Immediate database modification

Deferred Database Modification

The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.

Immediate Database Modification

The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued since undoing may be needed, update logs must have both old value and new value Update log record must be written *before* database item is written We assume that the log record is output directly to stable storage Can be extended to postpone log record output, so long as prior to execution of an **output**(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage

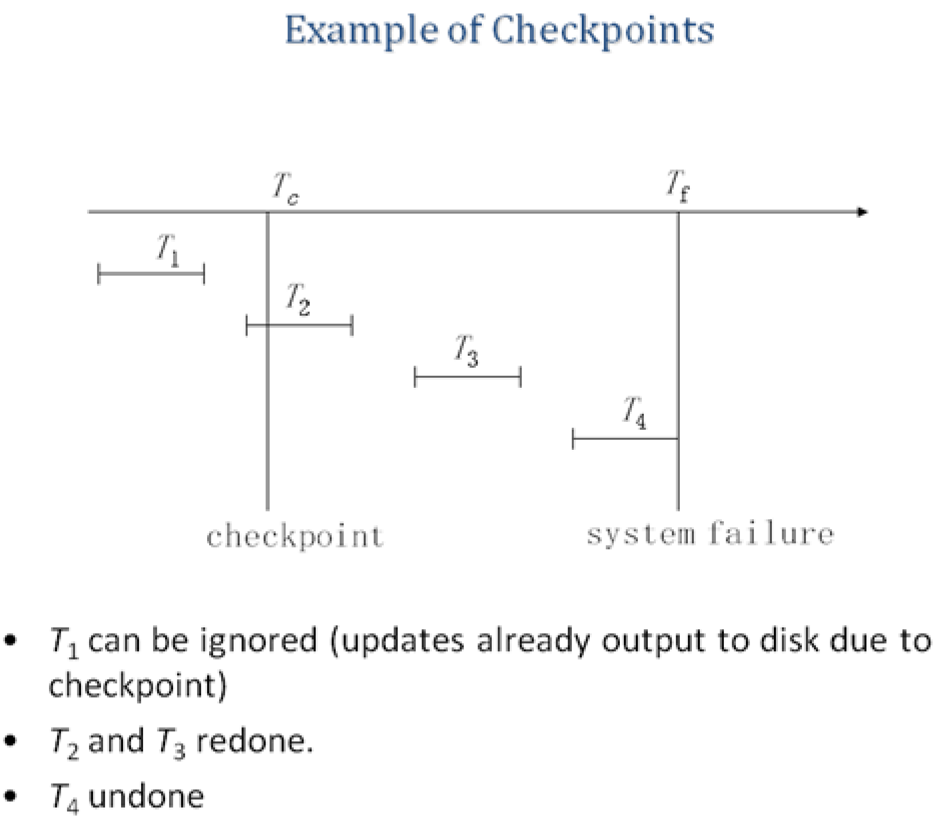
Checkpoints

Problems in recovery procedure :

searching the entire log is time-consuming

we might unnecessarily redo transactions which have already output their updates to the database.

Streamline recovery procedure by periodically performing **checkpointing** Output all log records currently residing in main memory onto stable storage.Output all modified buffer blocks to the disk.Write a log record < **checkpoint**> onto stable storage.During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i. Scan backwards from end of log to find the most recent <**checkpoint**> record Continue scanning backwards till a record <T_i **start**> is found. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(**T_i**)**. (Done only in case of immediate modification.) Scanning forward in the log, for all transactions starting from T_i or later w a <T_i **commit**>, execute **redo(**T_i**)**.



Example of Checkpoints

- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

4.11.2 Recovery with concurrent transactions

We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.

All transactions share a single disk buffer and a single log.A buffer block can have data items updated by one or more transactions .We assume concurrency control using strict two-phase locking; i.e. the updates of uncommitted transactions should not be visible to other transactions .

Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?Logging is done as described earlier. Log records of different transactions may be interspersed in the log.

- The checkpointing technique and actions taken on recovery have to be changed since several transactions may be active when a checkpoint is performed.

Log Record Buffering

Log record buffering: log records are buffered in main memory, instead of of being output directly to stable storage.Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.

4.12. BUFFER MANAGEMENT

Database maintains an in-memory buffer of data blocks.When a new block is needed, if buffer is full an existing block needs to be removed from buffer If the block chosen for removal has been updated, it must be output to disk If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first (Write ahead logging).No updates should be in progress on a block when it is output to disk. Can be ensured as follows.

Before writing a data item, transaction acquires exclusive lock on block containing the data item

Lock can be released once the write is completed. Such locks held for short duration are called **latches**.Before a block is output to disk, the system acquires an exclusive latch on the block.Ensures no update can be in progress on the block .Database buffer can be implemented either in an area of real main-memory reserved for the database, or in virtual memory.Implementing buffer in reserved main-memory has drawbacks:Memory is partitioned before-hand between database buffer and applications, limiting flexibility. Database buffers are generally implemented in virtual memory in spite of some drawbacks: When operating system needs to evict a page that has been modified, the page is written to swap space on disk.When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O! Known as **dual paging** problem.

4.13. FAILURE WITH LOSS OF NONVOLATILE STORAGE

Technique similar to checkpointing used to deal with loss of non-volatile storage Periodically **dump** the entire content of the database to stable storage No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place

Output all log records currently residing in main memory onto stable storage.Output all buffer blocks onto the disk.Copy the contents of the database to stable storage.output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage

- To recover from disk failure restore database from most recent dump.
- Consult the log and redo all transactions that committed after the dump Can be extended to allow transactions to be active during dump; known as **fuzzy dump**

4.14. ADVANCED RECOVERY TECHNIQUES

Advanced Recovery: Key Features

Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early .Supports "logical undo" Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing including redo of log records of incomplete transactions, followed by subsequent undo Key benefits supports logical undo easier to understand/show correctness

Advanced Recovery: Logical Undo Logging

Operations like B^+-tree insertions and deletions release locks early. They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+-tree.Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).For such operations, undo log records should contain the undo operation to be executed Such logging is called **logical undo logging**, in contrast to **physical undo Logging** Operations are called **logical operations.**

Advanced Recovery: Physical Redo

Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts Physical redo logging does not conflict with early lock release.

Advanced Recovery: Operation Logging

Operation logging is done as follows:When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.While operation is executing, normal log records with physical redo and physical undo information are logged.

When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Advanced Recovery: Crash Recovery

The following actions are taken when recovering from system crash (**Redo phase**): Scan log forward from last < **checkpoint** *L*> record till end of log **Repeat history** by physically redoing all updates of all transactions,Create an undo-list during the scan as follows *undo-list* is set to *L* initially

Whenever <T_i **start**> is found T_i is added to *undo-list* Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo- list* This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.(**Undo phase**): Scan log backwards, performing undo on log records of transactions found in *undo-list*. Log records of transactions being rolled back are processed as re found.

Advanced Recovery: Check pointing

Checkpointing is done as follows:

- Output all log records in memory to stable storage
- Output to disk all modified buffer blocks
- Output to log on stable storage a < **checkpoint** *L*> record.

Transactions are not allowed to perform any actions while checkpointing is in progress.

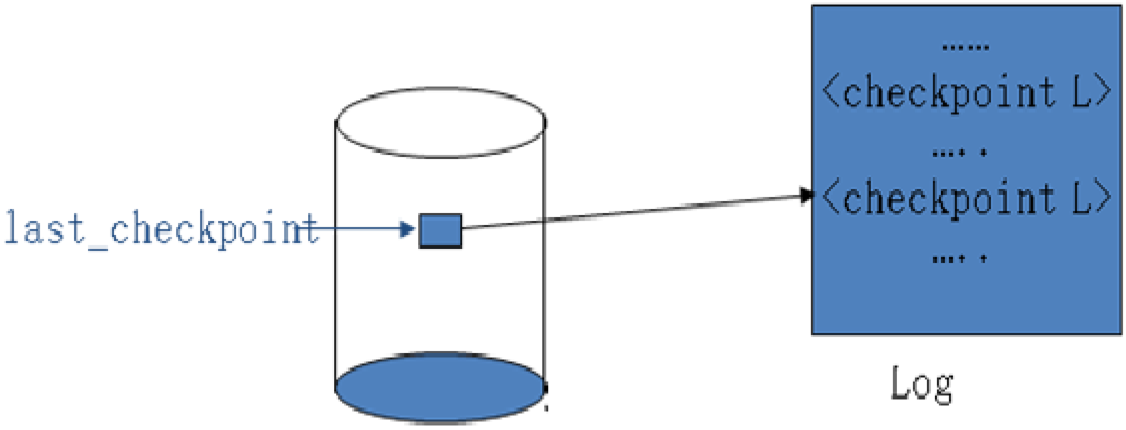
Advanced Recovery: Fuzzy Checkpointing

Fuzzy checkpointing is done as follows:

- Temporarily stop all updates by transactions
- Write a <**checkpoint** *L*> log record and force log to stable storage
- Note list *M* of modified buffer blocks
- Now permit transactions to proceed with their actions
- Output to disk all modified buffer blocks in list *M* blocks should not be updated while being output

Follow WAL: all log records pertaining to a block must be output before

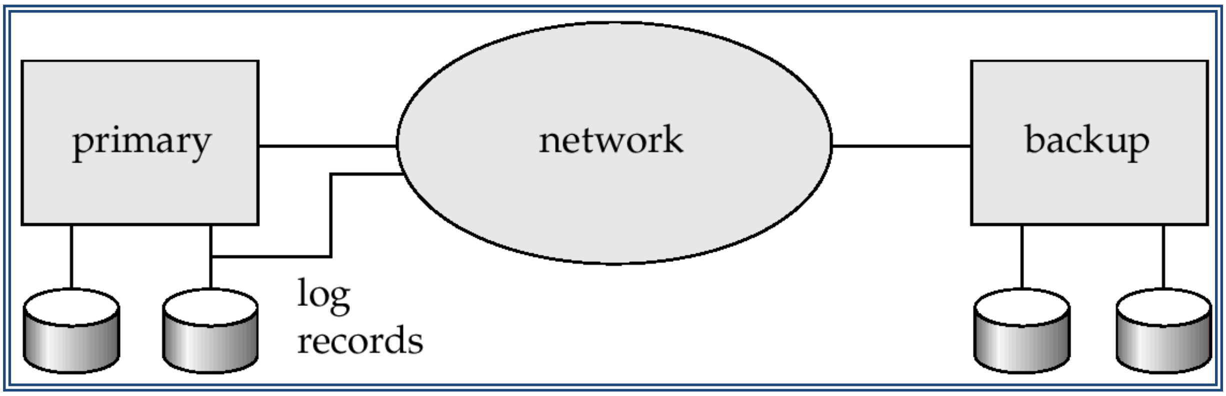
the block is output Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk.



When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint** Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely ARIES ARIES is a state of the art recovery method Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery .The "advanced recovery algorithm" we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations Unlike the advanced recovery algorithm, ARIES Uses **log sequence number (LSN)** to identify log records.Stores LSNs in pages to identify what updates have already been applied to a database page.

4.15. REMOTE BACKUP SYSTEMS

Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.**Detection of failure**: Backup site must detect when primary site has failed .



To distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.Heart-beat messages

Transfer of control:
To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary. Thus, completed transactions are redone and incomplete transactions are rolled back.

When the backup site takes over processing it becomes the new primary to transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Time to recover: To reduce delay in takeover, backup site periodically proceses the
redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.

Hot-Spare configuration permits very fast takeover:Backup continually processes redo log record as they arrive, applying the updates locally.When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.Alternative to remote backup: distributed database with replicated data .Remote backup is faster and cheaper, but less tolerant to failure .

Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.**One-safe:** commit as soon as transaction's commit log record is written at primary Problem: updates may not arrive at backup before it takes over. **Two-very-safe:** commit when transaction's commit log record is written at primary and **b**ackup Reduces availability since transactions cannot commit if either site fails. **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary. Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

.

UNIT-V

5.1. 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.

5.2. FILE ORGANIZATION AND INDEXING

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

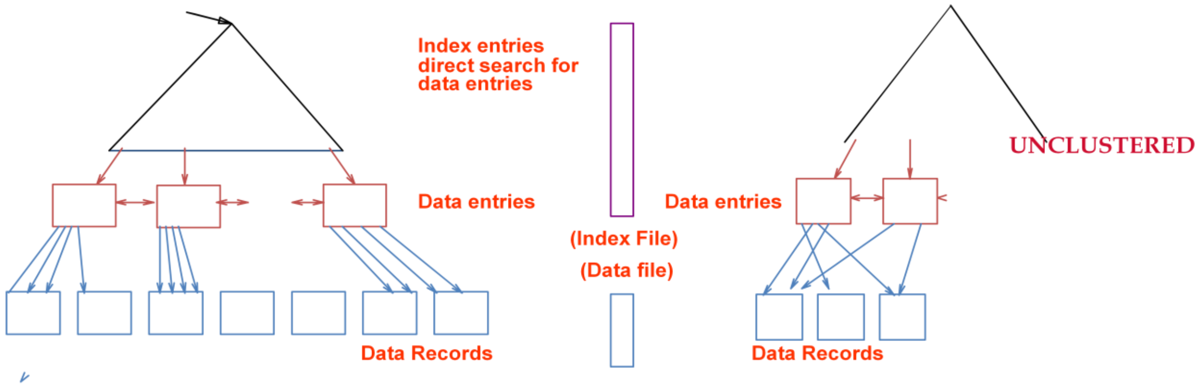
Heap (random order) files: Suitable when typical access is a file scan retrieving all records.

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.

5.2.1.Clustered Indexes

Clustered vs. unclustered: If order of data records is the same as, or `close to', order of data entries, then called clustered index.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.To build clustered index, first sort the Heap file (with some free space on each for future inserts).



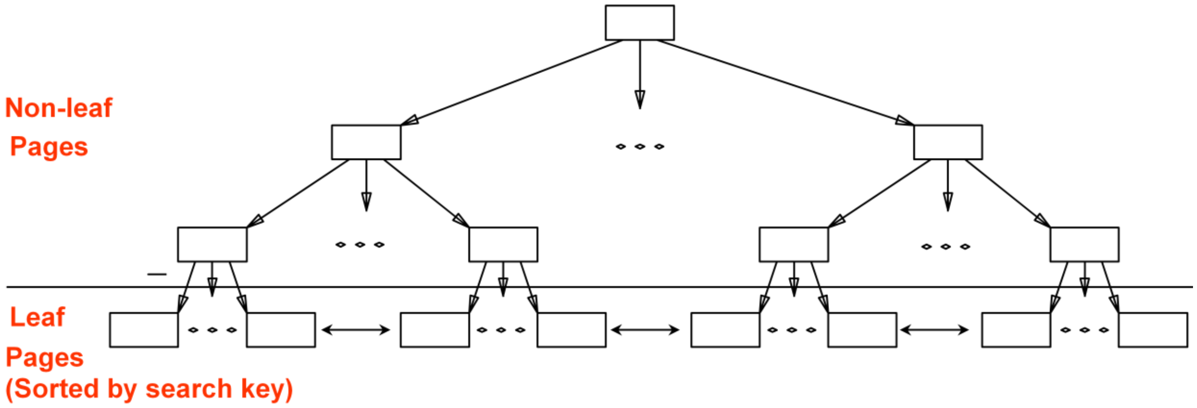
Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)

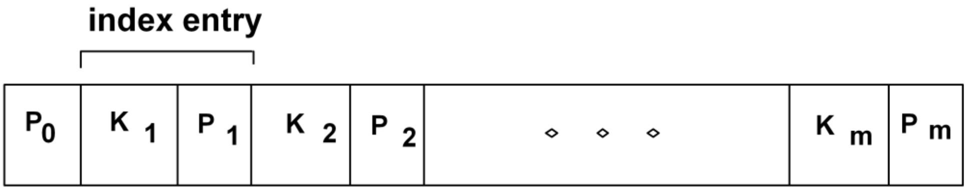
5.2.2. PRIMARY AND SECONDARY INDEXES

Primary vs. secondary: If search key contains primary key, then called primary index.*Unique* index: Search key contains a candidate key.

5.3 INDEX DATA STRUCTURES

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).

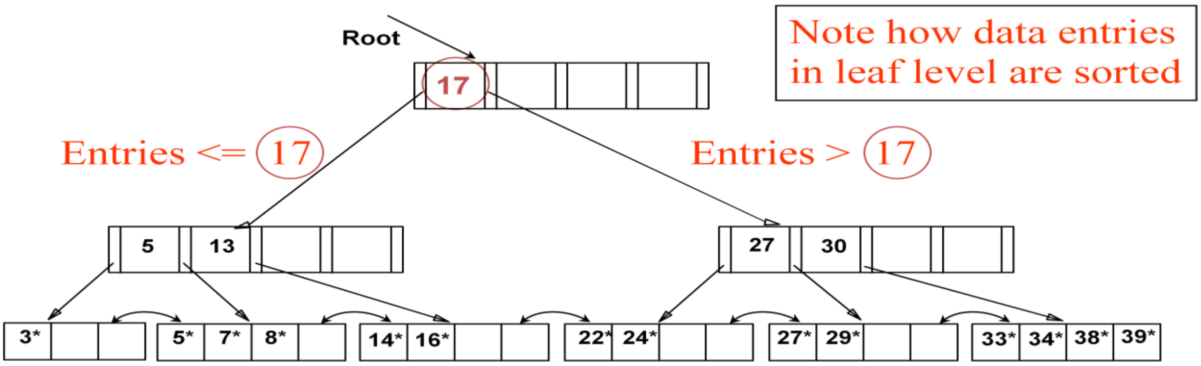




An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.Given data entry k*, we can find record with key k in at most one disk I/O.

B+ Tree Indexes

Example B+ Tree



- Find 28*? 29*? All > 15* and < 30*
- Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes. And change sometimes bubbles up the tree

5.3.1. HASH BASED INDEXING

Hash-Based Indexes Good for equality selections. Index is a collection of *buckets*.

Bucket = *primary* page plus zero or more *overflow* pages.Buckets contain data entries.

- *Hashing function* **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs. **h** looks

  at the *search key* fields of *r*.*No need for "index entries" in this scheme.*

Alternatives for Data Entry **k*** in Index. In a data entry k* we can store:

– Data record with key value **k,** or
– <**k**, rid of data record with search key value **k**>, or
– <**k**, list of rids of data records with search key **k**>

Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.

5.3.2. TREE BASED INDEXING

Examples of indexing techniques: B+ trees, hash-based structures.

Typically, index contains auxiliary information that directs searches to the desired data entries.

Alternative 1:

If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).

At most one index on a given collection of data records can use Alternative 1.(Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)

If data records are very large, # of pages containing data entries is high.Implies size of auxiliary information in the index is also large, typically.

Alternatives 2 and 3:

Data entries typically much smaller than data records. So, better than Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)

Alternative 3 more compact than Alternative 2, but leads to variable sized data . entries even if search keys are of fixed length.

5.4. COMPARISON OF 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*>

Operations to Compare

- Scan: Fetch all records from disk
- Equality search
- Range selection
- Insert a record
- Delete a record

Assumptions in Our Analysis

Heap Files:

Equality selection on key; exactly one match.

Sorted Files:

Files compacted after deletions.

Indexes:

Alt (2), (3): data entry size = 10% size of record

Hash: No overflow buckets.

80% page occupancy => File size = 1.25 data size

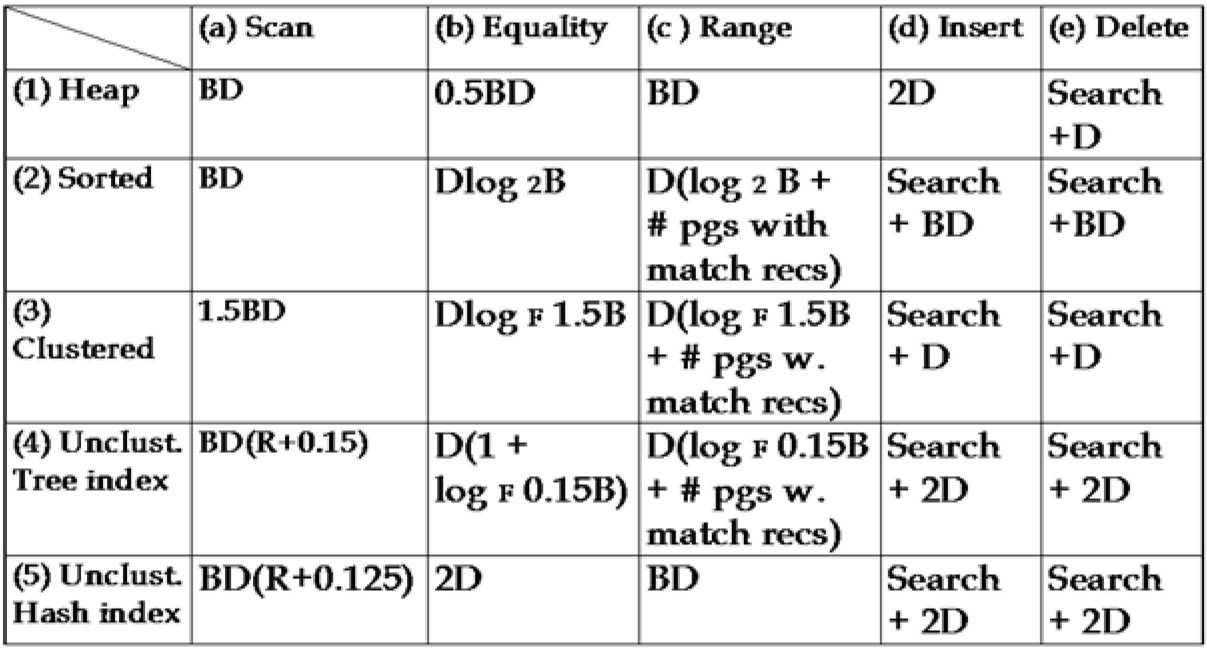
Tree: 67% occupancy (this is typical).

Implies file size = 1.5 data size

Scans:

Leaf levels of a tree-index are chained.Index data-entries plus actual file scanned for unclustered indexes.Range searches:We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

Cost of Operations



	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	$D\log_2 B$	$D(\log_2 B$ + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	$D\log_F 1.5B$	$D(\log_F 1.5B$ + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	$D(1 + \log_F 0.15B)$	$D(\log_F 0.15B$ + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Understanding the Workload.For each query in the workload.

Which relations does it access?

Which attributes are retrieved?

Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

For each update in the workload:

Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choice of Indexes

What indexes should we create?

Which relations should have indexes? What field(s) should be the search key?

Should we build several indexes?

For each index, what kind of an index should it be?

Clustered? Hash/tree?

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!

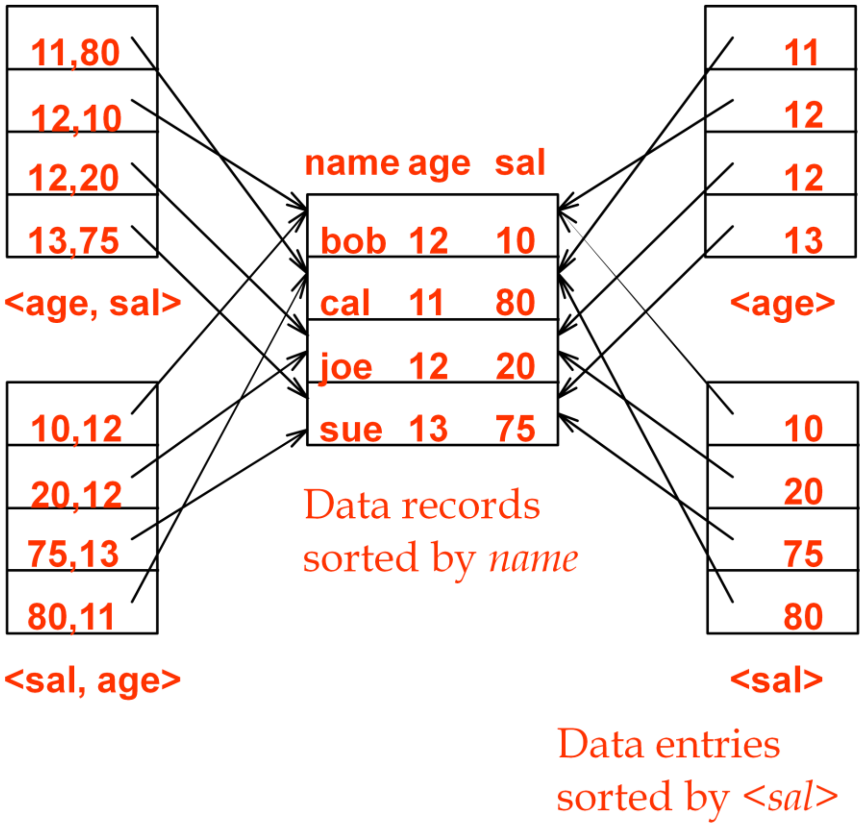
simple 1-table queries:
Before creating an index, must also consider the impact on updates in the workload!
Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

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 contain several conditions.Order of attributes is important for range queries.Such indexes can sometimes enable index-only strategies for important queries.For index-only strategies, clustering is not important!

Indexes with Composite Search Keys

Examples of composite key indexes using lexicographic order.



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. Lexicographic order, or Spatial order.

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.Index-Only Plans. A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

5.5. TREE STRUCTURED INDEXING

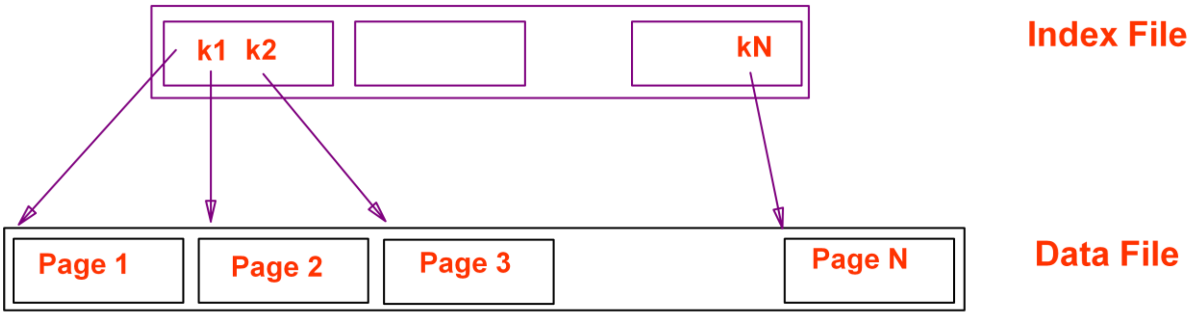
Tree-structured indexing techniques support both *range searches* and *equality searches*.

5.5.1. Intuition for Tree Indexes

ISAM: static structure; *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

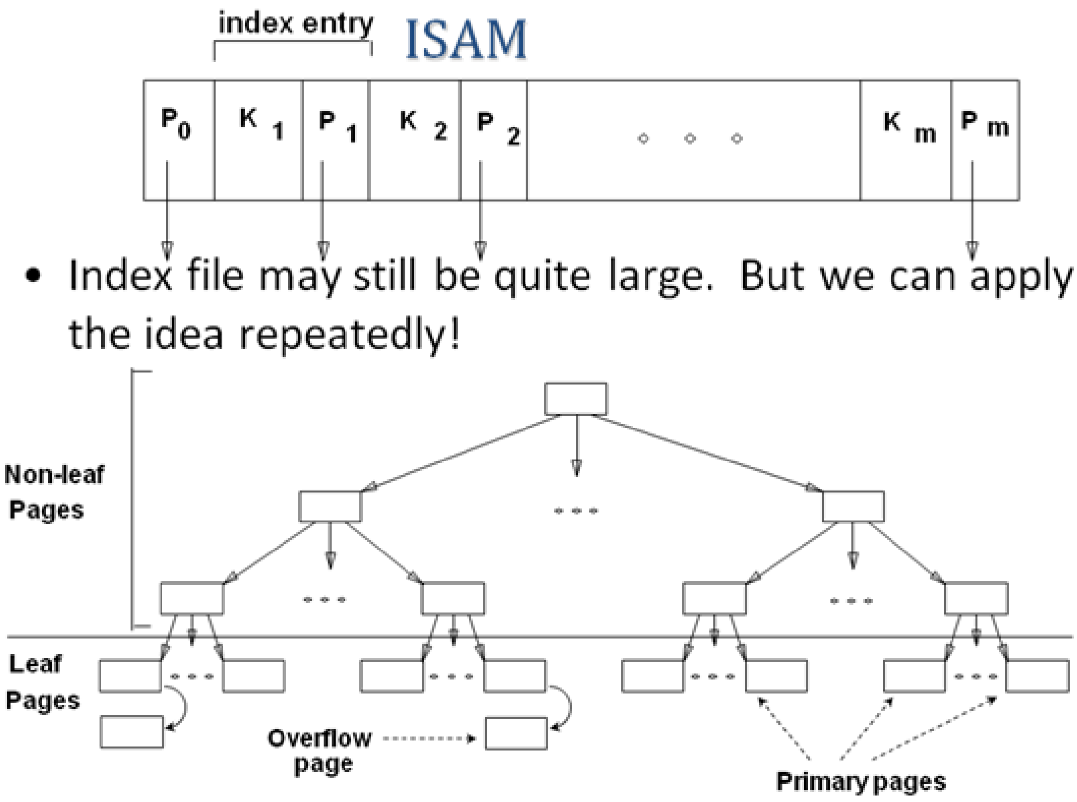
Range Searches

- *``Find all students with gpa > 3.0''*



If data is in sorted file, do binary search to find first such student, then scan to find others.
Cost of binary search can be quite high.
Simple idea: Create an `index' file.

5.5.2. INDEXED SEQUENTIAL ACCESS METHODS (ISAM)



- Index file may still be quite large. But we can apply the idea repeatedly!

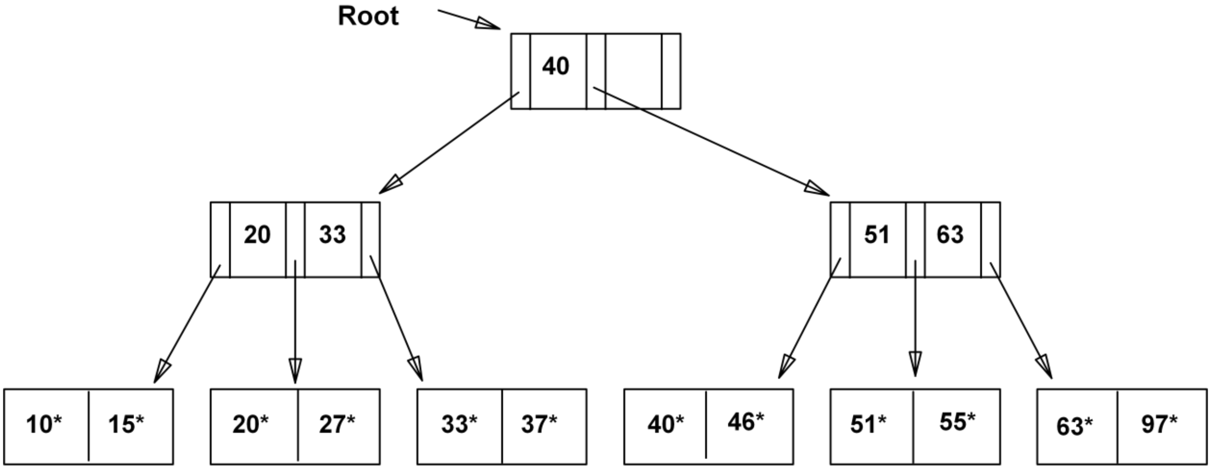
File creation: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
Index entries: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.

Search: Start at root; use key comparisons to go to leaf. Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs

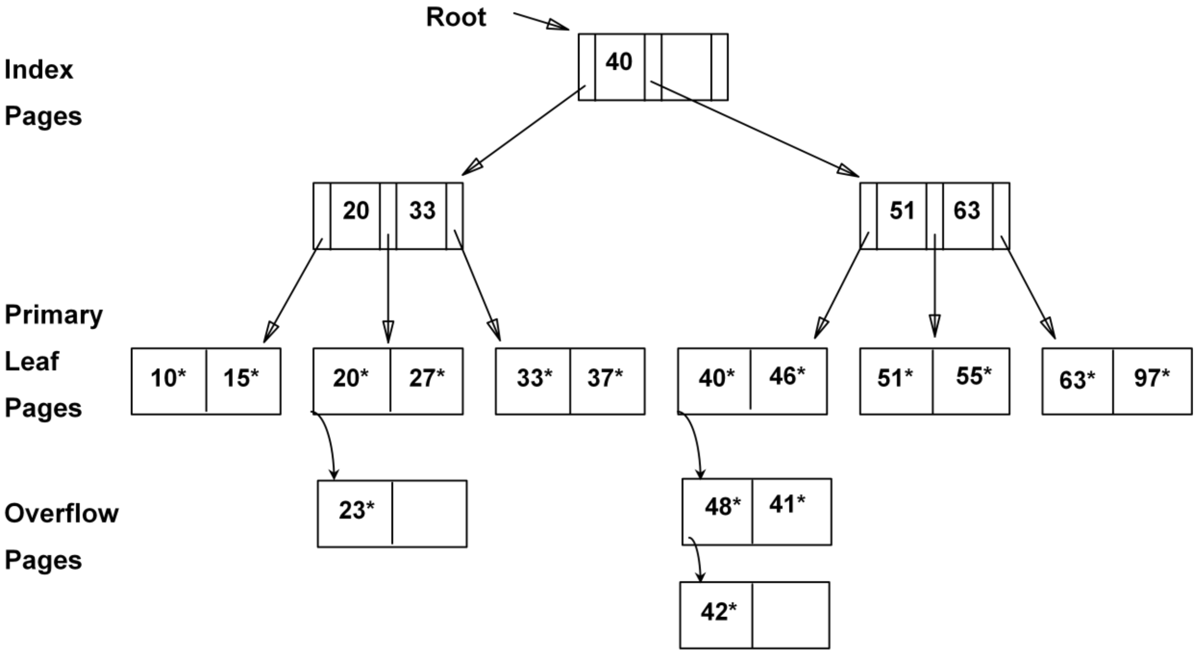
Insert: Find leaf data entry belongs to, and put it there.

Delete: Find and remove from leaf; if empty overflow page, de-allocate.

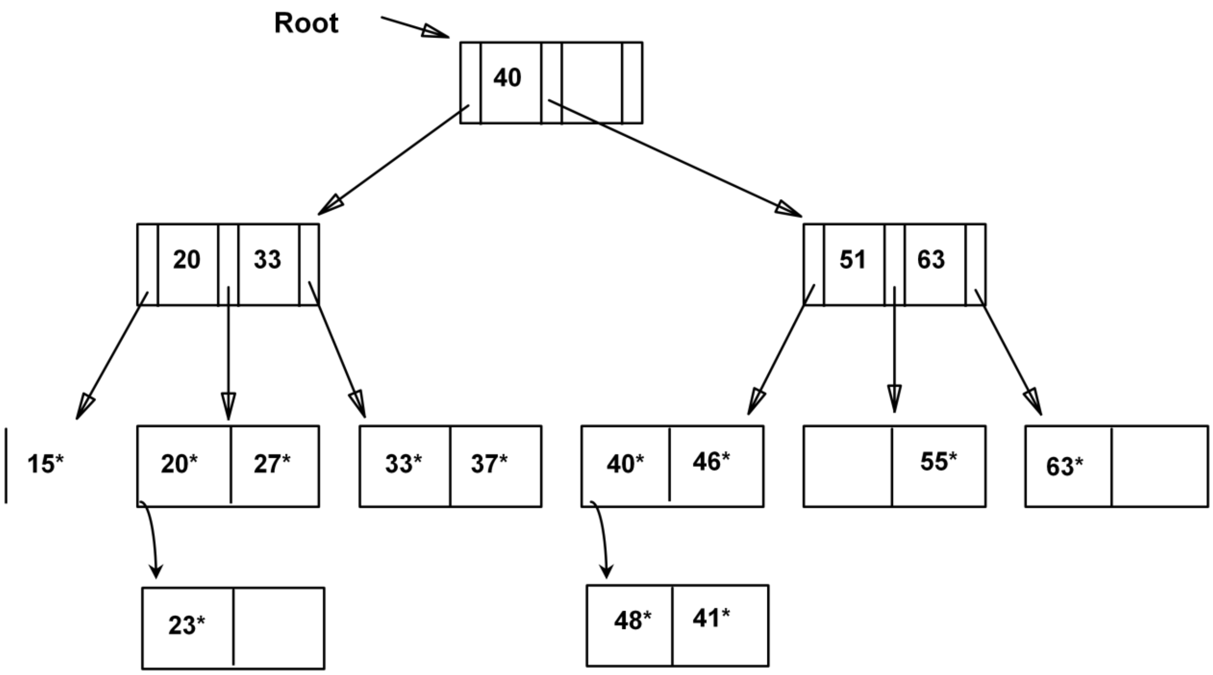
Example ISAM Tree



- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)
- After Inserting 23*, 48*, 41*, 42* ...



- ... Then Deleting 42*, 51*, 97*



5.5.3. B + TREES-A DYNAMIC INDEX STRUCTURE

B+ Tree: Most Widely Used Index Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages) Minimum 50% occupancy (except for root). Each node contains **d** <= <u>*m*</u> <= 2**d** entries. The parameter **d** is called the *order* of the tree.
Supports equality and range-searches efficiently.

Example B+ Tree

5.5.3.1. Search

Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
Search for 5*, 15*, all data entries >= 24* ...

B+ Trees in Practice

Typical order: 100. Typical fill-factor: 67%.
average fanout = 133
Typical capacities:
Height 4: 133^4 = 312,900,700 records
Height 3: 133^3 = 2,352,637 records
Can often hold top levels in buffer pool:
Level 1 = 1 page = 8 Kbytes
Level 2 = 133 pages = 1 Mbyte
Level 3 = 17,689 pages = 133 MBytes

5.5.3.2. Insert

Inserting a Data Entry into a B+ Tree
Find correct leaf *L*.
Put data entry onto *L*.
If *L* has enough space, *done*!

Else, must *split* *L (into L and a new node L2)*

Redistribute entries evenly, **copy up** middle key.

Insert index entry pointing to *L2* into parent of *L*.

This can happen recursively

To split index node, redistribute entries evenly, but **push up** middle key.

(Contrast with leaf splits.)

Splits "grow" tree; root split increases height.

Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into Example B+ Tree

Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

Example B+ Tree After Inserting 8*

5.5.3.3. DELETE

Deleting a Data Entry from a B+ Tree

Start at root, find leaf *L* where entry belongs.Remove the entry.

If L is at least half-full, *done!*

If L has only **d-1** entries,

Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L).*

If re-distribution fails, *merge* *L* and sibling.

If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.

Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...

Deleting 19* is easy.

Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.And Then Deleting 24*

Must merge.

Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).

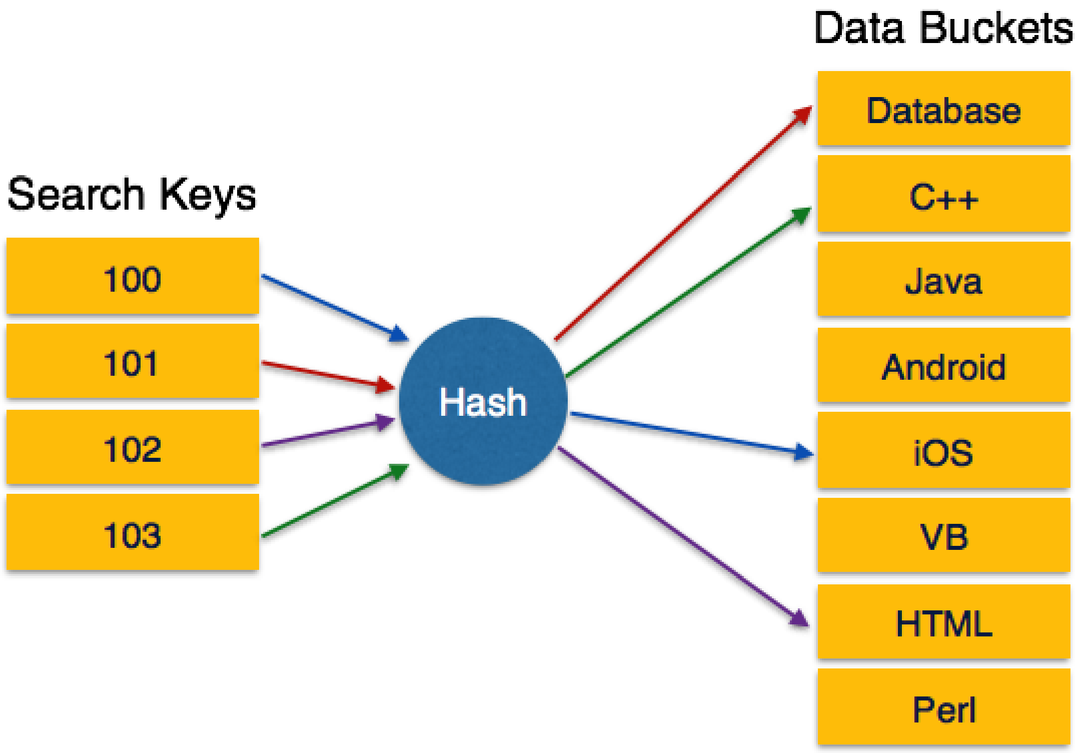
5.6. HASH BASED IDEXING

Bucket: Hash file stores data in bucket format. Bucket is considered a unit of storage. 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 set of search-keys K to the address where actual records are placed. It is a function from search keyto bucket addresses.

5.6.1 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 numbers of buckets provided remain same at all times.



[*Image: Static Hashing*]

Operations:

- **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.

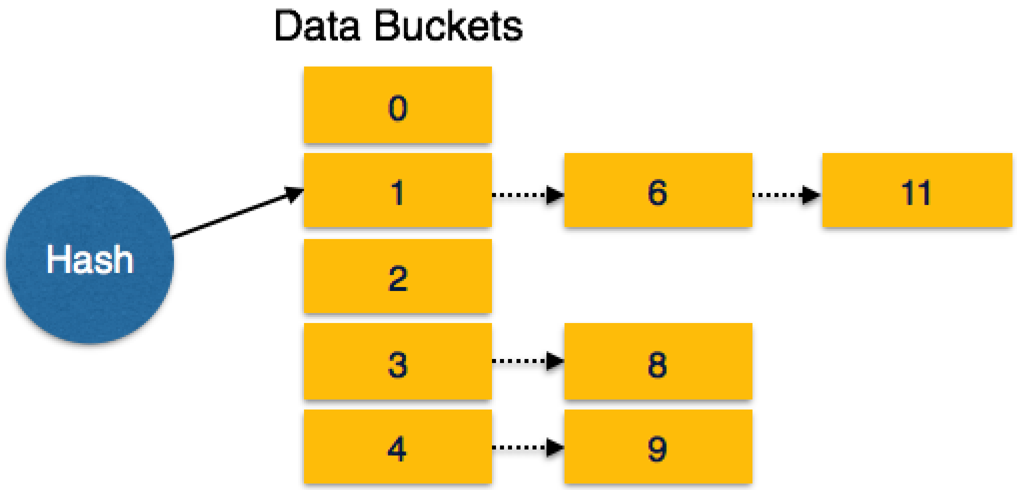
  Bucket address = h(K)

- **Search:** When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
- **Delete:** This is simply search followed by 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.

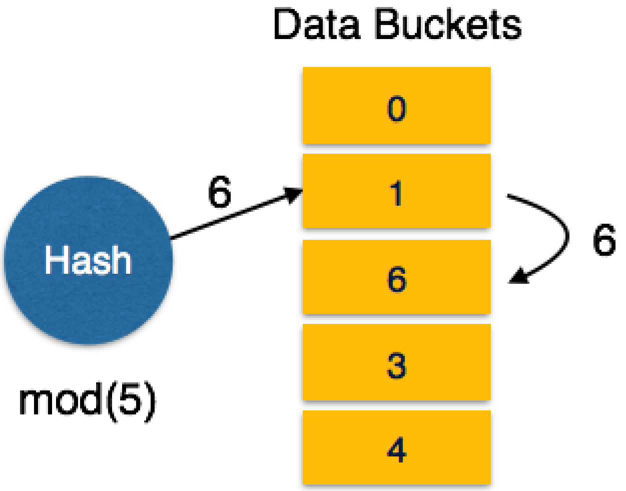
- **Overflow Chaining:** When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.



[*Image: Overflow chaining*]

5.6.2. Linear Hashing

- **Linear Probing:** When 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.



[*Image: Linear Probing*]

For a hash function to work efficiently and effectively the following must match:

Distribution of records should be uniform

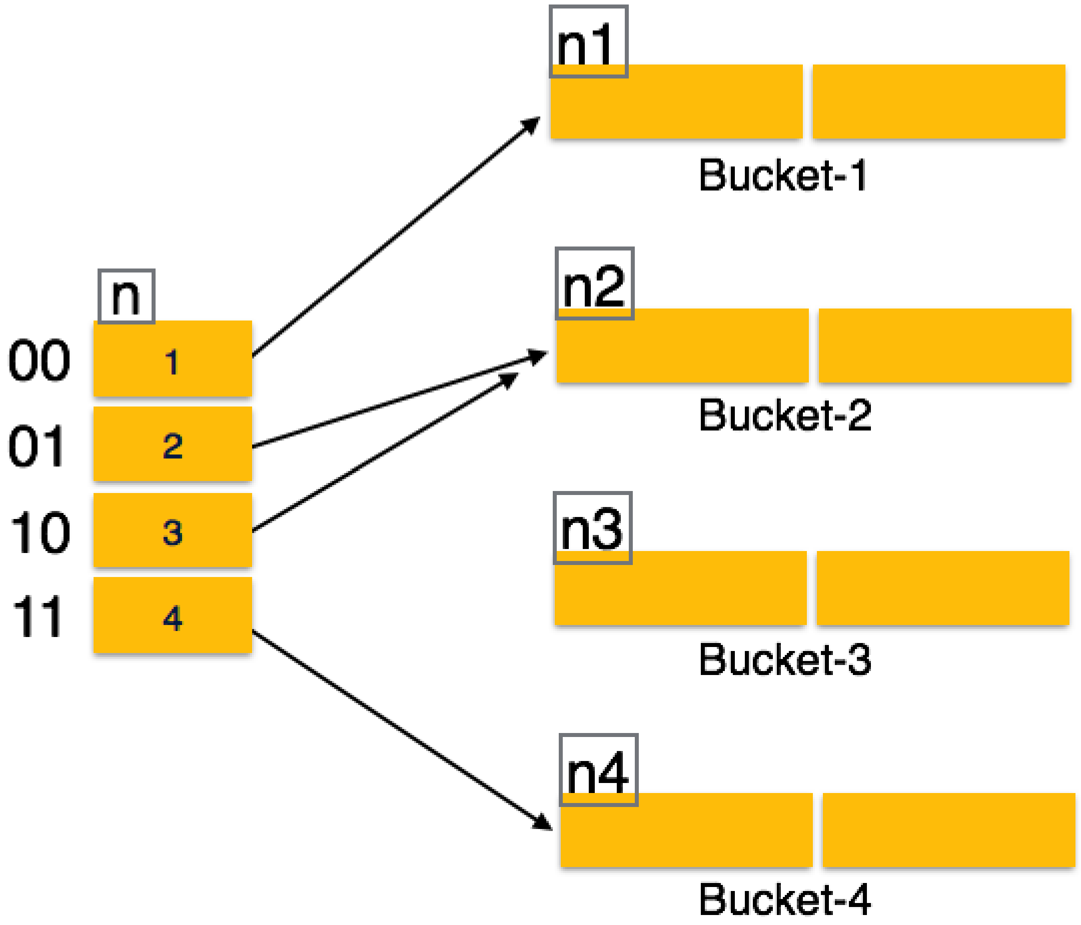
- Distribution should be random instead of any ordering

5.6.3. Extendable Hashing

Dynamic Hashing

Problem with static hashing is that it does not expand or shrink dynamically as the size of 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 large number of values and only a few are used initially.



[*Image: Dynamic Hashing*]

Organization

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

Operation

- **Querying:** Look at the depth value of hash index and use those bits to compute the bucket address.
- **Update:** Perform a query as above and update data.
- **Deletion:** Perform a query to locate desired data and delete data.

- **Insertion:** compute the address of bucket
    - If the bucket is already full
        - Add more buckets
        - Add additional bit to hash value
        - Re-compute the hash function
    - Else
        - Add data to the bucket
    - If all buckets are full, perform the remedies of static hashing.

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

Hashing algorithm and implementation have high complexity than indexing. All hash operations are done in constant time.

5.6.4. EXTENDABLE VS. LINEAR HASHING

Benefits of extendable hashing:

- hash performance doesn’t degrade with growth of file
- minimal space overhead

Disadvantages of extendable hashing:

- extra level of indirection (bucket address table) to find desired record
- bucket address table may itself become very big (larger than memory)
  need a tree structure to locate desired record in the structure!
- Changing size of bucket address table is an expensive operation

Linear hashing: is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows.