Module1

MODULE 1: RELATIONAL DATABASES

Introduction

- A **database-management system (DBMS)** is a collection of interrelated data and a set of programs to access those data.
- The collection of data, usually referred to as the **database**, contains information relevant to an enterprise.
- The primary goal of a DBMS is to **provide a way to store and retrieve database information** that is both convenient and efficient.

Purpose of Database System

- Before database management systems (DBMSs) were introduced, organizations usually stored information in file-processing system.
- Keeping organizational information in a file-processing system has a number of major disadvantages:
  - Data redundancy and inconsistency
    - In addition, it may lead to data inconsistency;
  - Difficulty in accessing data.
  - Data isolation
  - Integrity problems. The data values stored in the database must satisfy certain types of consistency constraints

- • Atomicity problems.
- Concurrent-access anomalies.
- Security problems

View of Data

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

Data Abstraction

- developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:
- **Physical level.**
- The lowest level of abstraction describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

- **Logical level.**
- The next-higher level of abstraction describes what data are stored in the database, and what relationships exist among those data.
- The user of the logical level does not need to be aware of the complexity of physical-level structures. This is referred to as **physical data independence.**

- **View level.**
- The highest level of abstraction describes only part of the entire database.
- Many users of the database system do not need all information; instead, they need to access only a part of the database.
- The system may provide many views for the same database.

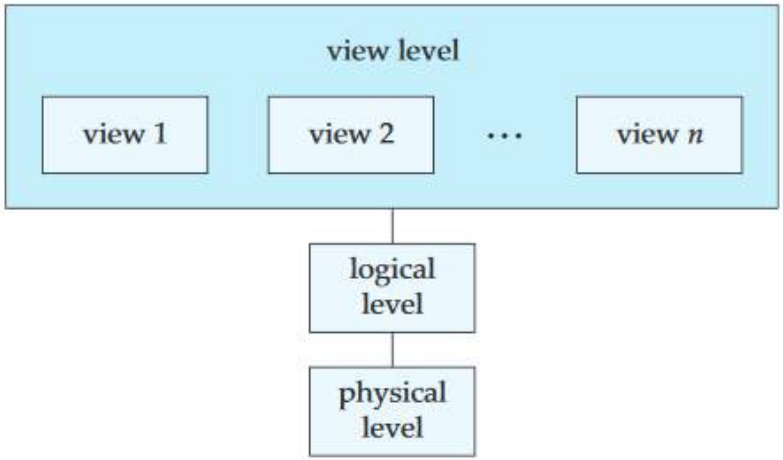


Figure 1.1 The three levels of data abstraction.

Instances and Schemas

- *Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an* ***instance*** *of the database.*
- *The overall design of the database is called the* ***database schema***

- *Database systems have several schemas, partitioned according to the levels of abstraction.*
- *The* ***physical schema*** *describes the database design at the physical level, while the* ***logical schema*** *describes the database design at the logical level.*
- *A database may also have several schemas at the view level, sometimes called* ***sub schemas****, that describe different views of the database.*

Data Models

- *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.*
- *A data model provides a way to describe the design of a database at the physical, logical, and view levels.*
- *The data models can be classified into four different categories:*

1. 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 known as **relations.**
- The relational model is an example of a **record-based model.**

2. 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 objects.
- The entity-relationship model is widely used in data base design

3. Object-Based Data Model.

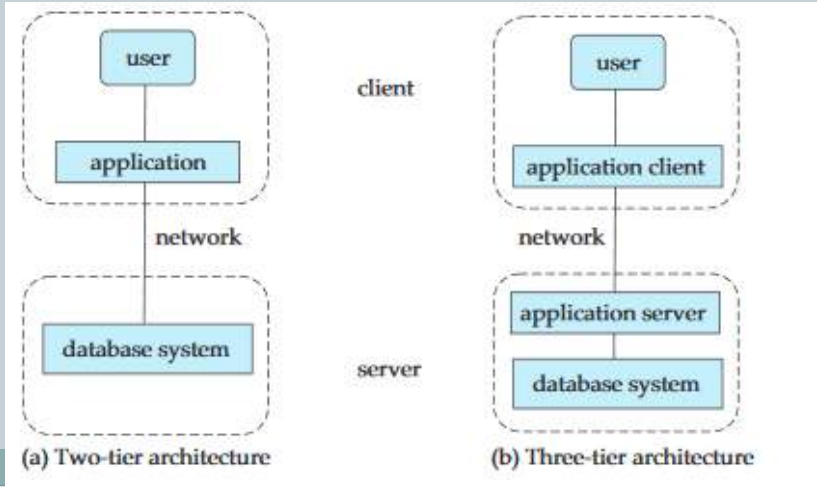
- Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology .
- This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods(functions), and object identity.
- The object-relational data model combines features of the object-oriented data model and relational data model.

4. 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.
- This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The Extensible Markup Language (XML)is widely used to represent semi structured data.

Database Architecture

- Database applications are usually partitioned into two or three parts.



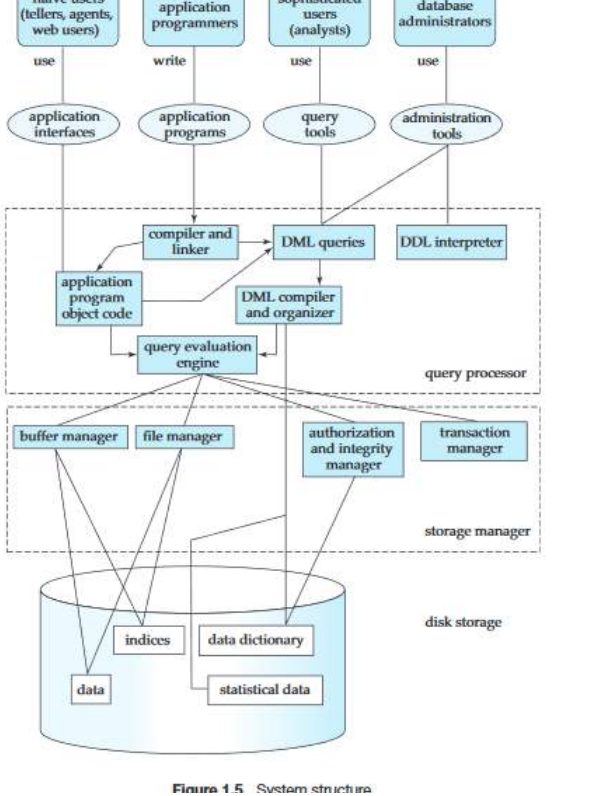


Figure 1.5 System structure

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**

Database 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.
- 1. **Naive 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
- The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form.

- 2. **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.
- 3. **Sophisticated users** interact with the system without writing programs. In-stead, they form their requests either using a database query language or by using tools such as data analysis software.

- 4. **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing frame work.
- 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.

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 database 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.** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

- **Granting of authorization for data access.** By granting different types of authorization, the database administrator can regulate which parts of the data base various users can access. The authorization information is kept in a special system structure that the database system consults whenever some one attempts to access the data in the system.

- **Routine maintenance.**
- Examples of the database administrator's routine maintenance activities are:
- Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
- Ensuring that enough free disk space is available for normal operations , and upgrading disk space as required.
- Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

The Entity-Relationship Model

- The entity-relationship(E-R) data model was developed to facilitate data base design.
- The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes.

Entity Sets

- An **entity** is a "thing" or "object" in the real world that is distinguishable from all other objects.
- For example, each person in a university is an entity.
- An entity has a set of properties, and the values for some set of properties may uniquely identify an entity.
- For instance, a person may have a person id property whose value uniquely identifies that person.

- An **entity set** is a set of entities of the same type that share the same properties , or attributes.
- The set of all people who are instructors at a given university, for example, can be defined as the entity set instructor.
- Similarly, the entity set student might represent the set of all students in the university

- Entity sets do not need to be disjoint.
- For example, it is possible to define the entity set of all people in a university (person). A person entity may be an instructor entity, a student entity, both, or neither.

Attributes

- An entity is represented by a set of attributes. Attributes are descriptive properties possessed by each member of an entity set.
- Possible attributes of the instructor entity set are ID , name , deptname , and salary.
- Possible attributes of the course entity set are courseid , title , deptname , and credits.
- Each entity has a value for each of its attributes.

Relationship Sets

- A relationship is an association among several entities.
- For example, we can define a relationship advisor that associates instructor James with student Shankar. This relationship specifies that James is an advisor to student Shankar.

- A relationship set is a set of relationships of the same type.
- Formally, it is a mathematical relation on n≥2 (possibly non distinct) entity sets. If E1, E2, . . . , En are entity sets, then a relationship set Ris a subset of

  {(e1, e2, . . . , en)|e1∈E1, e2∈E2, . . . , en∈En} where (e1, e2, . . . , en) is a relationship

- The association between entity sets is referred to as participation;
- that is, the entity setsE1, E2, . . . , En participate in relationship set R.
- The function that an entity plays in a relationship is called that entity's role.

- A relationship may also have attributes called descriptive attributes .
- Consider a relationship set advisor with entity sets instructor and student.
- We could associate the attribute date with that relationship to specify the date when an instructor became the advisor of a student.

Attributes

- For each attribute, there is a set of permitted values, called the domain, or value set, of that attribute.
- The domain of attribute courseid might be the set of all text strings of a certain length.
- An attribute, as used in the E-R model, can be characterized by the following attribute types
  - Simple and composite attributes.
  - Single-valued and multi valued attributes.
  - Derived attribute.

Simple and composite attributes.

- Simple- that is, they have not been divided into subparts.
- Composite attributes- can be divided into subparts (that is, other attributes).
- For example, an attribute name could be structured as a composite attribute consisting of first name , middle initial , and last name.

Single-valued and multi valued attributes.

- single value for a particular entity.
- For instance, the studentID attributefor a specific student entity refers to only one studentID. Such attributes are said to be single valued.
- multi valued attributes -an attribute has a set of values for a specific entity.
- Eg: a phone number attribute.

Derived attribute.

- The value for this type of attribute can be derived from the values of other related attributes or entities.
- Eg: age

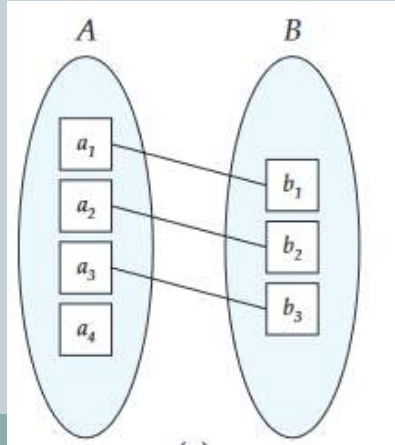
Constraints

- An E-R enterprise schema may define certain constraints to which the contents of a database must conform.

Mapping Cardinalities

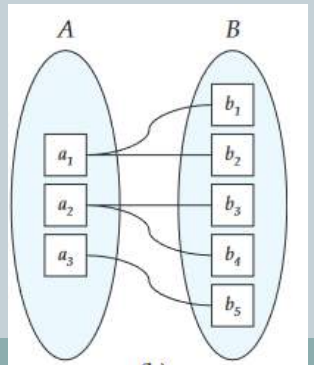
- **Mapping cardinalities, or cardinality ratios**, express the number of entities to which another entity can be associated via a relationship set.
- For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:
- One-to-one.
- One-to-many.
- Many-to-one.
- Many-to-many.

- One-to-one.
- An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A



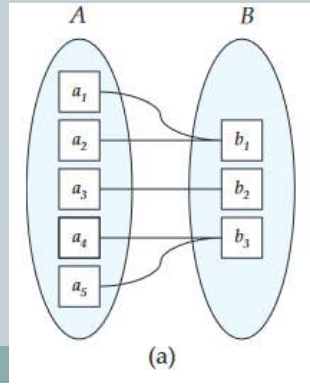
(a)

- One-to-many.
- An entity in A is associated with any number (zero or more)of entities in B. An entity in B, however, can be associated with at most one entity in A.



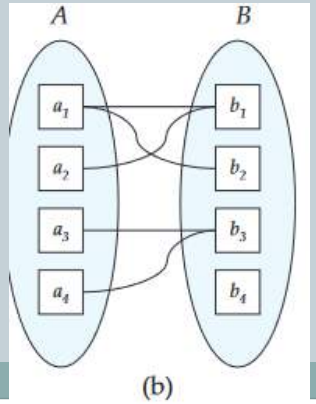
(b)

- Many-to-one.
- An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.



(a)

- Many-to-many.
- An entity in A is associated with any number (zero or more)of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



Participation Constraints

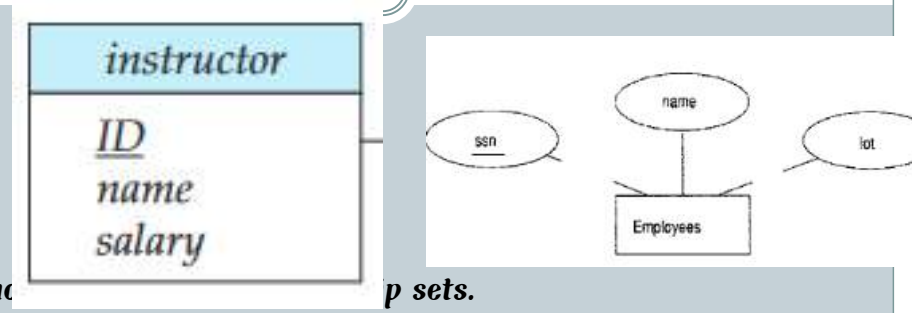
- The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R.
- If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be **partial.**

Keys

- The values of the attribute values of an entity must be such that they can uniquely identify the entity.

Entity-Relationship Diagrams

- An E-R diagram can express the overall logical structure of a database graphically.
- Basic Structure
- An E-R diagram consists of the following major components
- **Rectangles divided into two parts** :represent entity sets. The first part, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.



- Diamo p sets.

- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
- **Dashed lines** link attributes of a relationship set to the relationship set.
- **Double lines** indicate total participation of an entity in a relationship set
- **Double diamonds** represent identifying relationship sets linked to weak entity sets

Figure 7.7 E-R diagram corresponding to instructors and students.

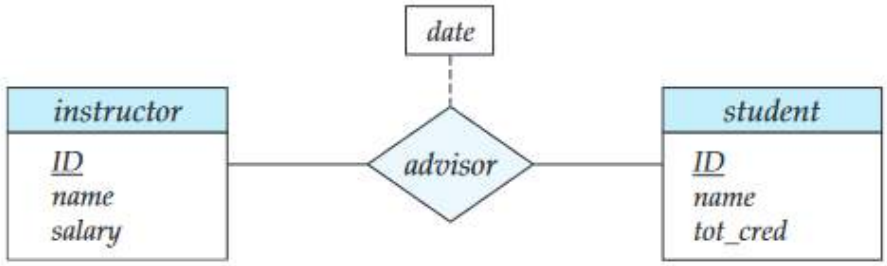
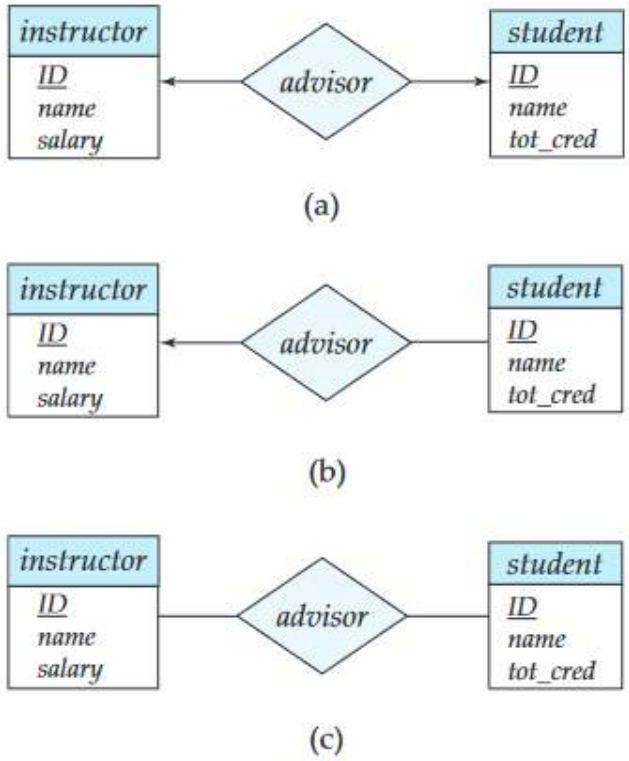
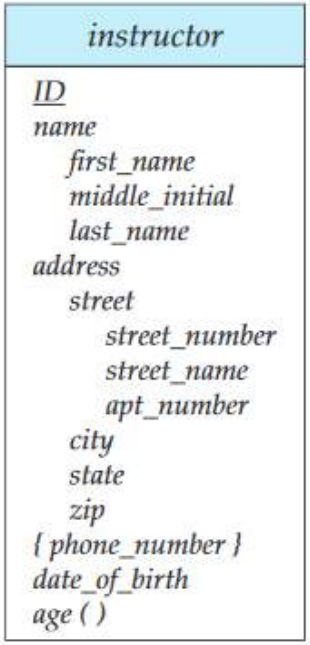


Figure 7.8 E-R diagram with an attribute attached to a relationship set.





E-R diagram with composite, multivalued, and derived attributes.

Roles

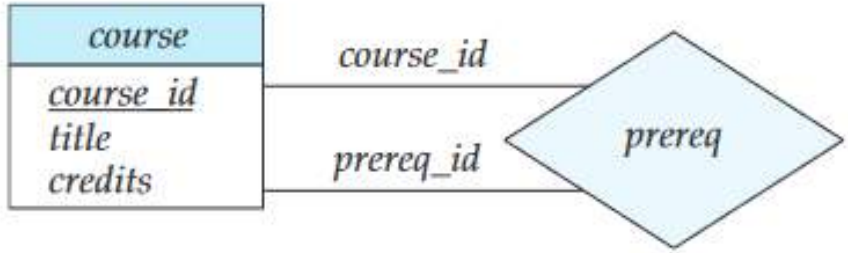


Figure 7.12 E-R diagram with role indicators.

Weak Entity Sets

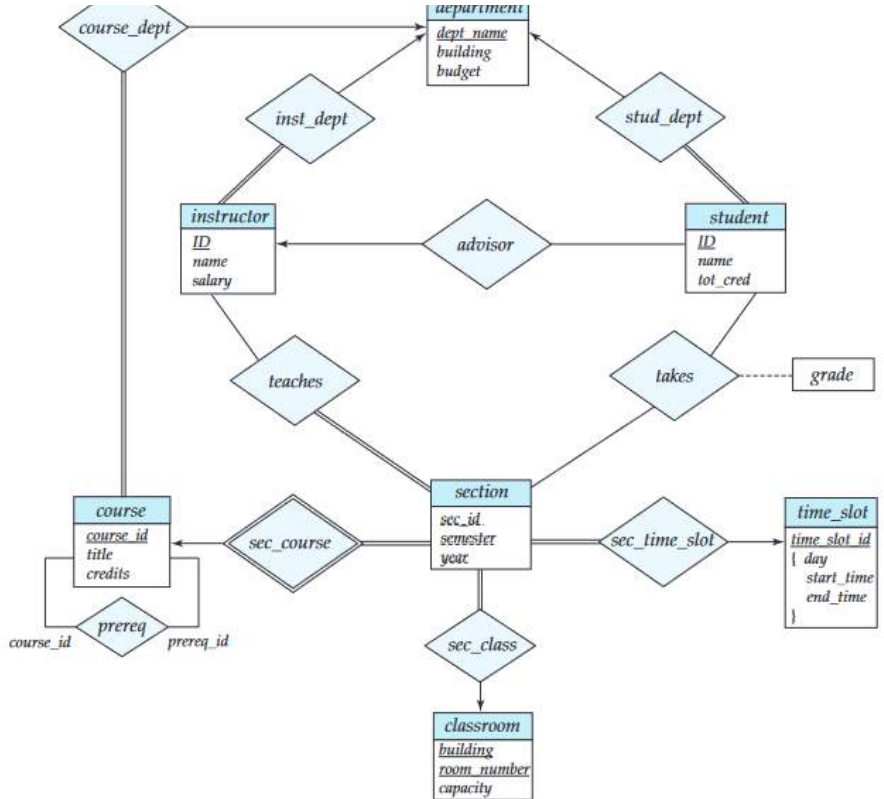
- An entity set that does **not have sufficient attributes to form a primary key** is termed a **weak entity set**.
- An entity set that has a primary key is termed a strong entity set.
- For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying or owner entity set**.

- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be existence dependent on the identifying entity set.
- The identifying entity set is said to own the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**

- The **identifying relationship is many-to-one** from the weak entity set to the identifying entity set, and the **participation of the weak entity set in the relationship is total**.
- The identifying relationship set should not have any descriptive attributes, since any such attributes can instead be associated with the weak entity set.
- The discriminator of a weak entity set is a set of attributes that allows this distinction to be made.

Figure 7.14 E-R diagram with a weak entity set.

- The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator.
- In E-R diagrams, a weak entity set is depicted via a rectangle, like a strong entity set, but there are two main differences:
- The **discriminator of a weak entity is underlined with a dashed**, rather than a solid, line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a **double diamond**.



Problem

- A company database needs to store information about employees (identified by *ssn*, with *salary* and *phone* as attributes), *departments* (identified by *dno*, with *dname* and *budget* as attributes), and *children* of employees (with *name* and *age* as attributes).

Problem

- Employees *work in departments*; each department is managed by an employee; a child must be identified uniquely by *name* when the parent (who is an employee; assume that only one parent works for the company) is known. We are not interested in information about a child once the parent leaves the company.
- Draw an ER diagram that captures this information.

Introduction to the Relational Model

- **Structure of Relational Databases**
- A relational database consists of a collection of tables, each of which is assigned a unique name.
- For example, consider the instructor table, which stores information about instructors. The table has four column headers: ID, name, deptname, and salary.
- Each row of this table records information about an instructor.

- in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.
- the term relation instance to refer to a specific instance of a relation, i. e. , containing a specific set of rows.
- For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute.
- The domain of the name attribute is the set of all possible instructor names.

- *for all relations r, the domains of all attributes of r be atomic.*
- *A domain is atomic if elements of the domain are considered to be indivisible units.*

Database Schema

- *The database schema, which is the logical design of the database, and the database instance, which is a snapshot of the data in the database at a given instant in time.*
- *In general, a relation schema consists of a list of attributes and their corresponding domains.*
- *department(deptname, building, budget)*

Keys

- ***Super key** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.*
- *Such minimal super keys are called **candidate keys***
- *The term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.*
- *A relation, say r1, may include among its attributes the primary key of an other relation, say r2. This attribute is called a **foreign key** from r1, referencing r2.*

Relational Query Languages

- *A **query language** is a language in which a user requests information from the database.*
- *These languages are usually on a level higher than that of a standard programming language.*
- *Query languages can be categorized as either **procedural or nonprocedural.***
- *In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.*
- *In a non procedural language, the user describes the desired information with out giving a specific procedure for obtaining that information*

The Relational Algebra

- The relational algebra is **a procedural query language.**
- It consists of a set of operations that take one or two relations as input and produce a new relation as their result.
- The fundamental operations in the relational algebra are **select, project, union , set difference , Cartesian product , and rename.**
- In addition to the fundamental operations, there are several other operations—namely, **set intersection , naturaljoin , and assignment.**

Fundamental Operations

- The **select, project, and rename operations are called unary operations**, because they operate on one relation.
- The other three operations operate on pairs of relations and are, therefore, called **binary operations**

The Select Operation

- The select operation selects tuples that satisfy a given predicate.
- We use the lower case Greek letter sigma (σ) to denote selection.
- The predicate appears as a subscript to σ .
- The argument relation is in parentheses after the σ
- Thus, to select those tuples of the instructor relation where the instructor is in the "Physics" department, we write:
- $\sigma_{deptname="Physics"}(instructor)$

- We can find all instructors with salary greater than $90,000 by writing:
- $\sigma_{salary>90000}(instructor)$
- we allow comparisons using=, ≠, <, ≤,>, and ≥ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives and (∧), or(∨), and not(¬).
- Thus, to find the instructors in Physics with a salary greater than $90,000, we write:
- $\sigma_{deptname="Physics" \wedge salary>90000}(instructor)$

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

Figure 6.1 The *instructor* relation.

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

Figure 6.2 Result of $\sigma_{dept\_name = \text{"Physics"}}(instructor)$.

The Project Operation

- *Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π .*
- *The argument relation follows in parentheses.*
- $\Pi_{ID, name, salary}(instructor)$

ID	name	salary
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Composition of Relational Operations

- *"Find the name of all instructors in the Physics department."*
- $\Pi_{name}(\sigma_{deptname = \text{"Physics"}}(instructor))$

The Union Operation

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Figure 6.4 The *section* relation.

- To find the set of all courses taught in the Fall 2009 semester, we write:
- $\Pi_{courseid}(\sigma_{semester="Fall" \wedge year=2009}(section))$
- To find the set of all courses taught in the Spring 2010 semester, we write:
- $\Pi_{courseid}(\sigma_{semester="Spring" \wedge year=2010}(section))$
- $\Pi_{courseid}(\sigma_{semester="Fall" \wedge year=2009}(section)) \cup \Pi_{courseid}(\sigma_{semester="Spring" \wedge year=2010}(section))$

The Set-Difference Operation

- The set-difference operation, denoted by –, allows us to find **tuples that are in one relation but are not in another.** The expression **r–s** produces a relation containing those **tuples in r but not in s.**
- Courses offered in the Fall 2009 semester but not in Spring 2010 semester.
- $\Pi_{courseid}(\sigma_{semester="Fall" \wedge year=2009}(section)) - \Pi_{courseid}(\sigma_{semester="Spring" \wedge year=2010}(section))$

The Cartesian-Product Operation

- The Cartesian-product operation, denoted by a cross (x), allows us to combine information from any two relations. We write the Cartesian product of relations r1 and r2 as r1×r2.

R

A	1
B	2
D	3
F	4
E	5

S

A	1
C	2
D	3
E	4

R CROSS S

A	1	A	1
A	1	C	2
A	1	D	3
A	1	E	4
B	2	A	1
B	2	C	2
B	2	D	3
B	2	E	4
D	3	A	1
D	3	C	2
D	3	D	3
D	3	E	4
F	4	A	1
F	4	C	2
F	4	D	3
F	4	E	4
E	5	A	1
E	5	C	2
E	5	D	3
E	5	E	4

The Rename Operation

- The rename operator, denoted by the lowercase Greek letter rho (ρ),
- $\rho_x(E)$

Formal Definition of the Relational Algebra

- Let E1 and E2 be relational-algebra expressions. Then, the following are all relational-algebra expressions:
- •E1∪E2
- •E1-E2
- •E1×E2
- •$\sigma_p(E1)$, where P is a predicate on attributes inE1
- •$\Pi S(E1)$, where S is a list consisting of some of the attributes inE1
- $\rho x(E1)$, where x is the new name for the result of E1

Additional Relational-Algebra Operations

- The Set-Intersection Operation
- Suppose that we wish to find the set of all courses taught in both theFall 2009 and the Spring 2010 semesters. Using set intersection, we can write
- Π courseid ($\sigma_{semester= \text{"Fall"} \wedge year=2009}(section)$) $\cap$ Π courseid ($\sigma_{semester= \text{"Spring"} \wedge year=2010}(section)$)
- Note that we can rewrite any relational-algebra expression that uses set intersection by replacing the intersection operation with a pair of set-difference operations as:
- $r \cap s = r - (r - s)$

The Natural-Join Operation

- The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
- It is denoted by the join symbol ⋈.
- The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both rela-tion schemas, and finally removes duplicate attributes.

- "Find the names of all instructors together with the courseid of all courses they taught.

$$\Pi_{name,\ course\_id}\ (instructor \bowtie teaches)$$

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation, denoted by ←, works like assignment in a programming language.

$$temp1 \leftarrow R \times S$$
$$temp2 \leftarrow \sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \ldots \wedge r.A_n = s.A_n}(temp1)$$
$$result = \Pi_{R \cup S}(temp2)$$

OUTER JOINs

- Notice that much of the data is lost when applying a join to two relations. In some cases this lost data might hold useful information. An outer join retains the information that would have been lost from the tables, replacing missing data with nulls.
- There are three forms of the outer join, depending on which data is to be kept.
- LEFT OUTER JOIN - keep data from the left-hand table
- RIGHT OUTER JOIN - keep data from the right-hand table
- FULL OUTER JOIN - keep data from both tables

R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

R LEFT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-

R RIGHT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
-	-	C	2

R

ColA	ColB
A	1
B	2
D	3
F	4
E	5

S

SColA	SColB
A	1
C	2
D	3
E	4

R FULL OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-
-	-	C	2

recipe

name	inventor	kitchen
ta and Meatballs	Le cook	Italian
Cheese Soup	The french	French
Burger	Cowboys	American

ingredient

recipe	foodItem
Pasta and Meatballs	Pasta
Pasta and Meatballs	Meatballs
Pasta and Meatballs	Tomato Sauce
Pasta and Meatballs	Onions
Cheese Soup	Onions
Cheese Soup	Cheese
Cheese Soup	Bread
Burger	Bread
Burger	Ground Beef

foodItem

item	type	calories
Pasta	Wheat product	20
Meatballs	Meat	40
omato Sauce	Sauce	5
Onions	Vegetables	1
Cheese	Diary	30
Bread	Wheat product	25
Ground Beef	Meat	45

stock

foodItem	shop	price
Pasta	Aldi	5
Meatballs	Aldi	10
Tomato Sauce	Aldi	3
Tomato Sauce	Walmart	3
Cheese	Treasury Island	15

- *1. Write a relational algebra expression that returns the food items required to cook the recipe "Pasta and Meat-balls". For each such food item return the item paired with the number of ounces required by the recipe.*

$$dItem,ounces(\sigma_{recipe='Pasta\ and\ Meatballs'}(ingredien$$

- *2. Write a relational algebra expression that returns food items that are sold at "Aldi" and their price*

$$foodItem,price(\sigma_{shop='Aldi'}(stock))$$

- *3. Write a relational algebra expression that returns food items (item) that are of type "Wheat product" or oftype "Meat" and have at least 20 calories per ounce (attribute calories)*

$$m(\sigma_{(type='\text{Wheat product}'\vee type='Meat')\wedge calories\geq 20}(foodIte$$

Module II

DATA BASE DESIGN

NORMALIZATION

- Normalization is the process of efficiently organizing data in a database.
- E. F Codd proposed the concept of normalization.
- Normalization removes redundant data from the tables to improve the storage efficiency , data integrity and scalability.

Need for normalization

- Normalization is the process of converting a relation into a standard form.
- The problem in an unnormalized relation are as follows:-
  - Data redundancy
  - Update anomalies
  - Deletion anomalies
  - Insertion anomalies

Need for normalization

Data redundancy:-

- In an unnormalizaed table design some information may be stored repeatedly.
- In the below example , **student table** the branch information , hod, office telephone number is repeated.
- This information is known as redundant data.

STUDENTS TABLE

rollno	name	branch	hod	office_tel
1	Akon	CSE	Mr. X	53337
2	Bkon	CSE	Mr. X	53337
3	Ckon	CSE	Mr. X	53337
4	Dkon	CSE	Mr. X	53337

What is an Anomaly?

- Definition
  - Problems that can occur in poorly planned, un-normalized databases where all the data is store in one table (a flat-file database).

Types of Anomalies

- Insert
- Delete
- Update

Insert Anomaly

- An **Insert Anomaly** occurs when certain attributes cannot be inserted into the database without the presence of other attributes.

Insert Anomaly

Course _no	Tutor	Room	Room_size	En_limit
353	Smith	A532	45	40
351	Smith	C320	100	60
355	Clark	H940	400	300
456	Turner	H940	400	45

e.g. we have built a new room (e.g. B123) but it has not yet been timetabled for any courses or members of staff.

Delete Anomaly

- A **Delete Anomaly** exists when certain attributes are lost because of the deletion of other attributes.

Delete Anomaly

Course_no	Tutor	Room	Room_size	En_limit
353	Smith	A532	45	40
351	Smith	C320	100	60
355	Clark	H940	400	300
456	Turner	H940	400	45

e.g. if we remove the entity, course_no:351 from the above table, the details of room C320 get deleted. Which implies the corresponding course will also get deleted.

Update Anomaly

- An **Update Anomaly** exists when one or more instances of duplicated data is updated, but not all.

Update Anomaly

Course_no	Tutor	Room	Room_size	En_limit
353	Smith	A532	45	40
351	Smith	C320	100	60
355	Clark	H940	400	300
456	Turner	H940	400	45

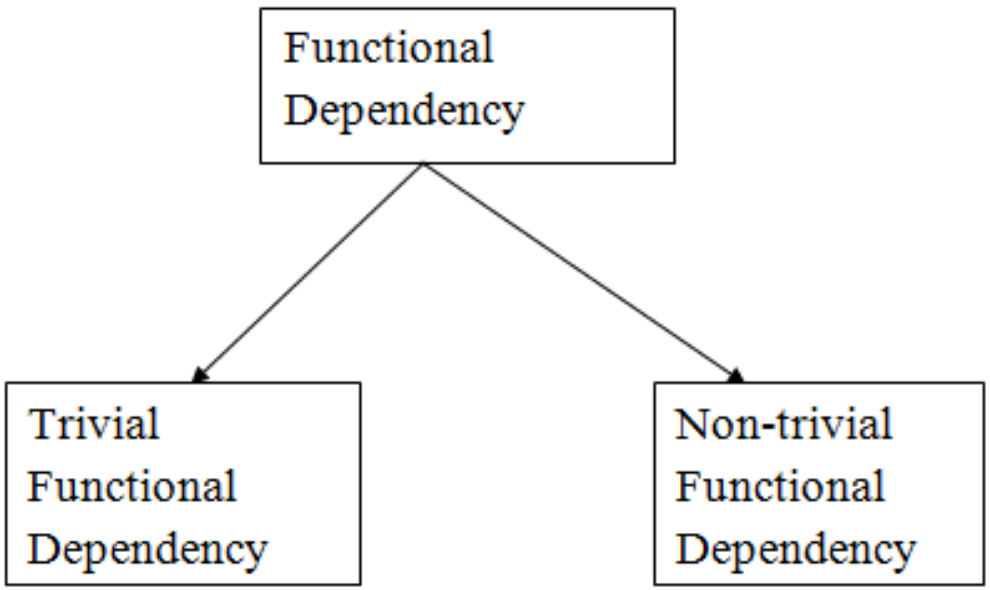
e.g. Room H940 has been improved, it is now of RSize = 500. For updating a single entity, we have to update all other columns where room=H940.

Functional Dependency

- A ***functional dependency*** (FD) is a relationship between two attributes, typically between the PK and other non-key attributes within a table.
- For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y.
- This relationship is indicated by the representation below :
- **X ----> Y**

- The left side of the above FD diagram is called the ***determinant***, and the right side is the ***dependent.***
- **SIN ----> Name, Address, Birthdate**
- SIN determines Name, Address and Birthdate.
- **SIN, Course ----> DateCompleted**
  - SIN and Course determine the date completed (DateCompleted). This must also work for a composite PK.

Types of Functional dependency



1. Trivial functional dependency

- A → B has trivial functional dependency if B is a subset of A.
- Consider a table with two columns Employee_Id and Employee _Name.
- {Employee_id, Employee_Name} → Employee_Id is a trivial functional de pendency as
- Employee_Id is a subset of {Employee_Id, Employee_Name}.

2. Non-trivial functional dependency

- A → B has a non-trivial functional dependency if B is not a subset of A.
- When A intersection B is NULL, then A → B is called as complete non-trivial.
- ID → Name

Inference Rules

- *Armstrong's axioms* are a set of inference rules used to infer all the functional dependencies on a relational database.
- They were developed by William W. Armstrong.
- **Axiom of reflexivity**
- This axiom says, if Y is a subset of X, then X determines Y

$$\text{If } Y \subseteq X, \text{ then } X \rightarrow Y$$

- **Axiom of augmentation**
- The axiom of augmentation, also known as a partial dependency, says if X determines Y, then XZ determines YZ for any Z

$$\text{If } X \rightarrow Y, \text{ then } XZ \rightarrow YZ \text{ for any } Z$$

- pr
- attributes of candidate key, are called prime attributes. And rest of the attributes of the relation are non prime.

- **Axiom of transitivity**
- The axiom of transitivity says if X determines Y, and Y determines Z, then X must also determine Z

$$\text{If } X \rightarrow Y \text{ and } Y \rightarrow Z, \text{ then } X \rightarrow Z$$

- **Secondary Rules –**
- These rules can be derived from the axioms.

1. **Union** – If $A \rightarrow B$ holds and $A \rightarrow C$ holds, then $A \rightarrow BC$ holds. If $X \rightarrow Y$ and $X \rightarrow Z$ then $X \rightarrow YZ$
2. **Composition** – If $A \rightarrow B$ and $X \rightarrow Y$ holds, then $AX \rightarrow BY$ holds.
3. **Decomposition** – If $A \rightarrow BC$ holds then $A \rightarrow B$ and $A \rightarrow C$ hold. If $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$
4. **Pseudo Transitivity** – If $A \rightarrow B$ holds and $BC \rightarrow D$ holds, then $AC \rightarrow L$ holds. If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$.

Functional Dependency Set

- Functional Dependency set or FD set of a relation is the set of all FDs present in the relation.
- { STUD_NO->STUD_NAME, STUD_NO->STUD_PHONE, STUD_NO->STUD_STATE, STUD_NO->STUD_COUNTRY, STUD_NO -> STUD_AGE, STUD_STATE->STUD_COUNTRY }

Attribute Closure:

- Attribute closure of an attribute set can be defined as set of attributes which can be functionally determined from it.
- To find attribute closure of an attribute set:
- Add elements of attribute set to the result set.
- Recursively add elements to the result set which can be functionally determined from the elements of the result set

- If attribute closure of an attribute set contains all attributes of relation, the attribute set will be super key of the relation.
- Question 1:
- Given relational schema R(P Q R S T) having following attributes P Q R S and T, also there is a set of functional dependency denoted by FD = { P->QR, RS->T, Q->S, T-> P }.
- Determine Closure of (T)$^+$

- FD = { P->QR, RS->T, Q->S, T-> P }.
- T+={ T, P, Q, R, S, T}

- **Consider the relation scheme R = {E, F, G, H, I, J, K, L, M, N} and the set of functional dependencies {{E, F} -> {G}, {F} -> {I, J}, {E, H} -> {K, L}, K -> {M}, L -> {N} on R. What is the key for R?**
- A. {E, F}
  B. {E, F, H}
  C. {E}

- **{{E, F} -> {G}, {F} -> {I, J}, {E, H} -> {K, L}, K -> {M}, L -> {N}**
- {E, F}+={ E, F, G, I, J}

{E, F, H}+={E, F, H, G, I, J, K, L, M, N }

- {E}+={ E}

Canonical Cover of Functional Dependencies/Minimal set of Functional dependency

- A canonical cover of a set of functional dependencies F is a simplified set of functional dependencies that has the same closure as the original set F.
- **Extraneous attributes:** An attribute of a functional dependency is said to be extraneous if we can remove it without changing the closure of the set of functional dependencies.

- A canonical cover F_c of a set of functional dependencies F such that ALL the following properties are satisfied:
- F logically implies all dependencies in F_c .
- F_c logically implies all dependencies in F.
- No functional dependency in contains an extraneous attribute.
- Each left side of a functional dependency in Fc is unique.

- **Finding Canonical Cover**

repeat

1. Use the union rule to replace any dependencies in $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$.
2. Find a functional dependency $\alpha \rightarrow \beta$ with an extraneous attribute either in α or in β.
3. If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$.

until F does not change

Example1:

Consider the following set *F* of functional dependencies:

F= {
A → BC
B → C
A → B
AB → C
}

Steps to find canonical cover:

1. There are two functional dependencies with the same set of attributes on the left:
   A → BC
   A → B
   These two can be combined to get
   A → BC.

   Now, the revised set F becomes:

   F= {
   A → BC
   B → C
   AB → C
   }

here is an extraneous attribute in AB → C because even after removing AB → C
om the set F, we get the same closures. This is because B → C is already a part

ow, the revised set F becomes:

= {
→ BC
→ C

is an extraneous attribute in A → BC, also A → B is logically implied by A → B
nd B → C (by transitivity).
= {
→ B
→ C

4. After this step, F does not change anymore. So,

Hence the required canonical cover is,

F_c = {

A $\rightarrow$ B

B $\rightarrow$ C

}

- Let F = {A → B, A → C, BC → D}. Can A determine D uniquely?

Consider the given functional dependencies-

AB → CD

AF → D

DE → F

C → G

F → E

G → A

Which of the following options is false?

(A) { CF }$^+$ = { A , C , D , E , F , G }

(B) { BG }$^+$ = { A , B , C , D , G }

(C) { AF }$^+$ = { A , C , D , E , F , G }

- Consider a relation scheme R = (A, B, C, D, E, H) on which the following functional dependencies hold: {A->B, BC-> D, E->C, D->A}. What are the candidate keys of R? [GATE 2005]

(a) AE, BE

(b) AE, BE, DE

(c) AEH, BEH, BCH

(d) AEH, BEH, DEH

Module III

TRANSACTION MANAGEMENT & CONCURRENCY CONTROL

Transactions

- Collections of operations that form a single logical unit of work are called **transactions**.
- A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.
- **Transaction** is a unit of program execution that accesses and possibly updates various data items.

What is a Transaction?

- Any action that reads from and/or writes to a database may consist of
  - Simple SELECT statement to generate a list of table contents
  - A series of related UPDATE statements to change the values of attributes in various tables
  - A series of INSERT statements to add rows to one or more tables
  - A combination of SELECT, UPDATE, and INSERT statements

What is a Transaction?

- A *logical* unit of work that must be either entirely completed or aborted
- Successful transaction changes the database from one *consistent* state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
  - The equivalent of a single SQL statement in an application program or transaction

Evaluating Transaction Results

- Not all transactions update the database
- SQL code represents a transaction because database was accessed
- Improper or incomplete transactions can have a devastating effect on database integrity
  - Some DBMSs provide means by which user can define enforceable constraints based on business rules
  - Other integrity rules are enforced automatically by the DBMS when table structures are properly defined, thereby letting the DBMS validate some transactions

Transaction Properties(ACID properties)

- **Atomicity**
  - Requires that *all* operations (SQL requests) of a transaction be completed; if not, then the transaction is aborted
  - A transaction is treated as a single, indivisible, logical unit of work
  - This **"all-or-none"** property is referred to as atomicity.

Consistency

- Consistency property ensures that the database must remain in **the consistent state before the start of transaction and after the transaction is over.**
- Consistency states that only valid data will be written to the database.
- If for some reason a transaction is executed that violates the database consistency rules the entire transaction will be rolled back.

- **Isolation**
  - Data used during execution of a transaction cannot be used by second transaction until first one is completed
  - Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions *Ti and Tj*, it appears to *Ti* that either *Tj* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.

Durability

- After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- Durability can be implemented by writing all transaction into a **transaction log** that can be used to crate a system state right before failure.
- A transaction can only regard as committed after it is written safely in the log.
- For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed.

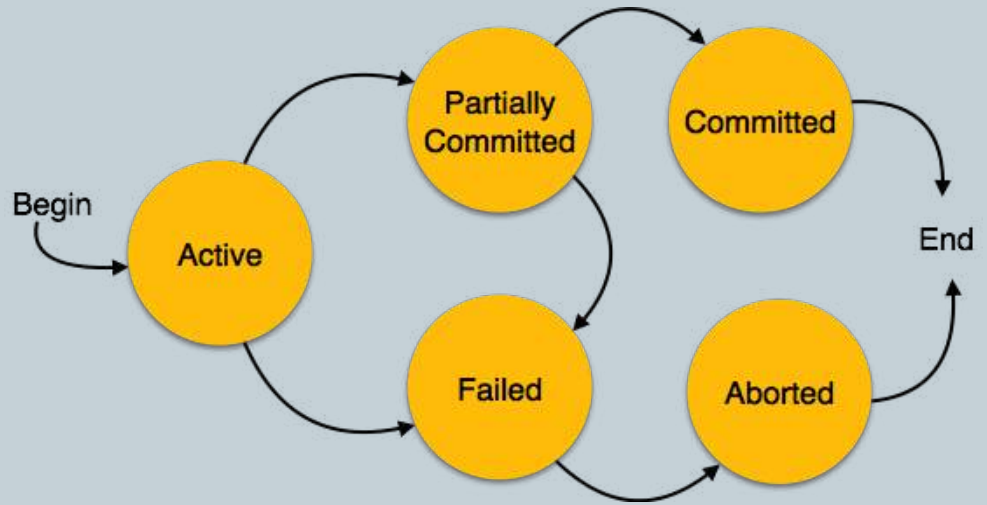
- These properties are called the **ACID properties.**

Transaction State

- A transaction must be in one of the following states:
  - **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.

Transaction State

- **Aborted:-**
  After the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- **Committed:-**
  After successful completion



Transaction Management with SQL

- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK
- ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of four events occurs

Transaction Management with SQL

1. A COMMIT statement is reached- all changes are permanently recorded within the database
2. A ROLLBACK is reached – all changes are aborted and the database is restored to a previous consistent state
3. The end of the program is successfully reached – equivalent to a COMMIT
4. The program abnormally terminates and a rollback occurs

The Transaction Log

- Keeps track of all transactions that update the database. It contains:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement)
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by the transaction (the name of the table)
    - "Before" and "after" values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - The ending (COMMIT) of the transaction
- Increases processing overhead but the ability to restore a corrupted database is worth the price

The Transaction Log

- Increases processing overhead but the ability to restore a corrupted database is worth the price
- If a system failure occurs, the DBMS will examine the log for all uncommitted or incomplete transactions and it will restore the database to a previous state
- The log it itself a database and to maintain its integrity many DBMSs will implement it on several different disks to reduce the risk of system failure

A Transaction Log

TABLE 9.1 A TRANSACTION LOG

TRL ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start Transaction				
352	101	341	363	UPDATE	PRODUCT	1558-QW1	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	525.75	615.73
365	101	363	Null	COMMIT	**** End of Transaction				

TRL_ID = Transaction log record ID **PTR** = Pointer to a transaction log record ID

TRX_NUM = Transaction number

(Note: The transaction number is automatically assigned by the DBMS.)

Transactions and schedules

- A transaction is seen by the dbms as a series or list of actions.
- Actions include read and writes of database object.
- Assume that an object O is always read into a program variable that is also named O
- Denote transaction T reading an object O as $R_T(O)$
- Similarly writing as $W_T(O)$

- Each transaction must specify as its final action either commit or abort
- $Abort_T$ and $Commit_T$
- Schedule is a list of actions from a set of transactions,
- Schedule represents an actual or potential execution sequence.

- T1 T2

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- A schedule that contains either abort or commit for each transactions is called **complete schedule.**
- If transactions are executed from start to finish, one by one---**-serial schedule**

Concurrent execution of Transactions

- Transaction processing system usually allow multiple transaction to run concurrently.
- Allowing multiple transaction to run concurrently and allowing multiple transaction to update data concurrently causes several complications with consistency of data.
- Ensuring consistency with concurrency require an extra work.

Concurrent execution of Transactions

- Two reasons to allow concurrency are:-
  - Improve throughput and resource utilization:-(Throughput - Number of transactions that can be executed in a given amount of time.)
  - Reduced waiting time.

- There may be a mix of transactions running on a system, some short and some long.
- If transactions are run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction.
- But concurrent execution reduces the unpredictable delays in running transactions.

TYPES OF SCHEDULE

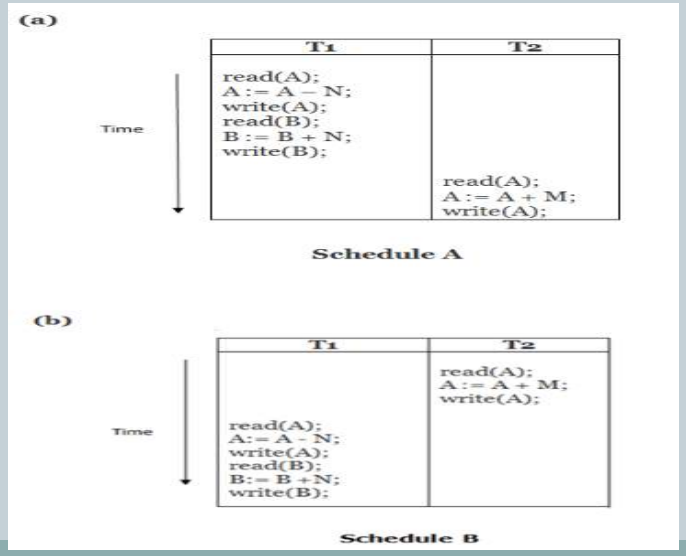
1. **Serial Schedule**
2. **Non-serial Schedule**
3. **Serializable schedule**

1. Serial Schedule

- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction.
- In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.
- For example: Suppose there are two transactions T1 and T2 which have some operations.

- Execute all the operations of T1 which was followed by all the operations of T2.
- Execute all the operations of T2 which was followed by all the operations of T1.
- In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
- In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

- If it has no interleaving of operations, then there are the following two possible outcomes:

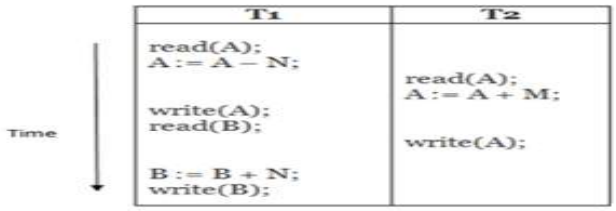


2. Non-serial Schedule/ Concurrent Execution

- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

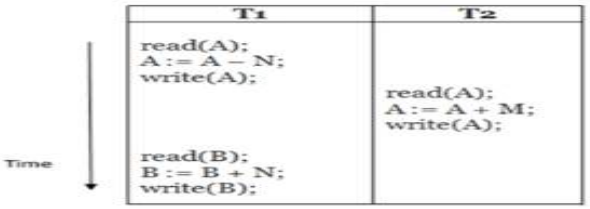
Non-serial Schedule

(c)



Schedule C

(d)



Schedule D

Problems with Concurrent Execution

- In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions.
- following problems occur with the Concurrent Execution of the operations:
- **Problem 1: Lost Update Problems (W - W Conflict)**
- **Dirty Read Problems (W-R Conflict)**
- **Unrepeatable Read Problem (W-R Conflict)/** *Inconsistent Retrievals Problem*

Problem 1: Lost Update Problems (W - W Conflict)

- The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i. e. , concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.*
- **Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is $300.**

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_χ reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_χ deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_y reads the value of account A that will be $300 only because T_χ didn't update the value yet.
- At time t4, transaction T_y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_χ writes the value of account A that will be updated as $250 only, as T_y didn't update the value yet.
- Similarly, at time t7, transaction T_y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_χ is lost, i.e., $250 is lost.
- Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict) / Uncommitted Data

- The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_χ reads the value of account A, i.e., $300.
- At time t2, transaction T_χ adds $50 to account A that becomes $350.
- At time t3, transaction T_χ writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_y reads account A that will be read as $350.
- Then at time t5, transaction T_χ rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict) / Inconsistent Retrievals Problem

- Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

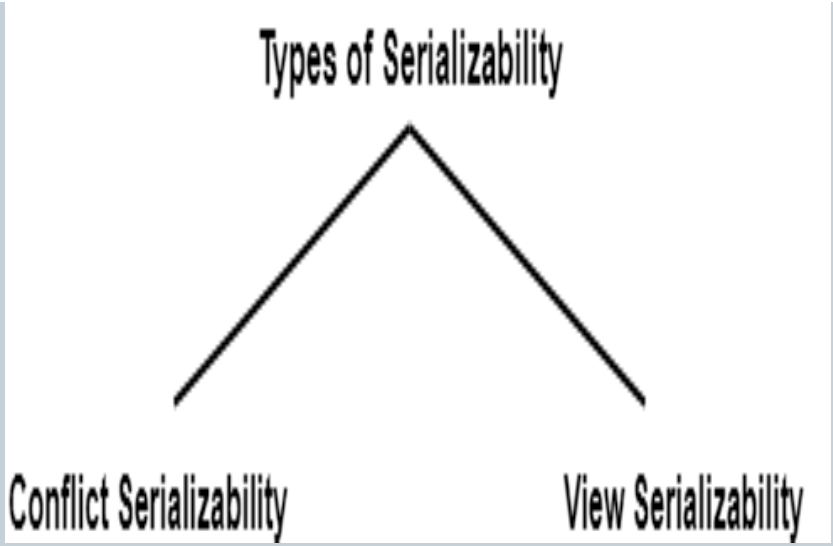
Serializability

- When multiple transactions run concurrently, then it may give rise to inconsistency of the database.
- **Serializability** is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.
- If a given schedule of 'n' transactions is found to be equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule.**

Difference between Serial Schedules and Serializable Schedules

- The only difference between serial schedules and serializable schedules is that-
- In serial schedules, only one transaction is allowed to execute at a time i. e. no concurrency is allowed.
- Whereas in serializable schedules, multiple transactions can execute simultaneously i. e. concurrency is allowed.

Types of Serializability



Conflict Serializability

- A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
- Let us consider a schedule S in which there are two consecutive instructions , I_i and I_j of transactions T_i and T_j , respectively(i!=j).

- If I_i and I_j refer to different data items , then we can swap I_i and Ij , without affecting the results of any instruction in the schedule.
- However , if Ii and Ij refer to the same data item Q, then the order of the two steps may matter.

- There are four cases we need to consider
- Ii=read(Q) , Ij =read(Q), the order of Ii and Ij does not matter
- Ii=read(Q) , Ij =Write(Q),
  - If Ii comes before Ij , then Ti doesnot read the value of Q that is written by Tj in instruction Ij. thus the order of Ii and Ij matters
- Ii=Write(Q) , Ij =read(Q), the order of Ii and Ij matter
- Ii=write(Q) , Ij =write(Q), the order of Ii and Ij does not matter, how ever the value obtained by the next read(Q)instn is affected.

- 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 conflit equivalent.
- A Schedule S is conflict serializable if it is conflict equivalent

- A schedule is called **conflict serializable** if it can be transformed into a serial schedule by swapping non-conflicting operations.
- Two operations are said to be conflicting if all conditions satisfy:
  - They belong to different transactions
  - They operate on the same data item
  - At Least one of them is a write operation

Precedence Graph

- **Precedence Graph** or **Serialization Graph** is used commonly to test Conflict Serializability of a schedule.
- It is a directed Graph (V, E) consisting of a set of nodes $V = \{T_1, T_2, T_3, \ldots T_n\}$ and a set of directed edges $E = \{e_1, e_2, e_3, \ldots\ldots e_m\}$.

- The graph contains **one node for each Transaction T_i.**
- An edge e_i is of the form $T_j \rightarrow T_k$.
- where T_j is the starting node of e_i and T_k is the ending node of e_i.
- An edge e_i is constructed between nodes T_j to T_k if one of the operations in T_j appears in the schedule before some conflicting operation in T_k .

ALGORITHM

- Create a node T in the graph for each participating transaction in the schedule.
- Check for conflicting instructions in the schedule:-
  - For the conflicting operation read_item(X) and write_item(X)(RW Conflict) - If a Transaction Ti executes a read_item (X) after Tj executes a write_item (X), draw an edge from Ti to Tj in the graph.
  - For the conflicting operation write_item(X) and read_item(X)(i. e WR conflict) - If a Transaction Ti executes a write_item (X) after Tj executes a read_item (X), draw an edge from Ti to Tj in the graph.

ALGORITHM

- For the conflicting operation write_item(X) and write_item(X) (i. e WW conflict)- If a Transaction Tj executes a write_item (X) after Ti executes a write_item (X), draw an edge from Ti to Tj in the graph.
- The Schedule S is serializable if there is no cycle in the precedence graph.

- If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to the schedule S.

PROBLEM 1

Check whether the given schedule S is conflict serializable or not-

S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$

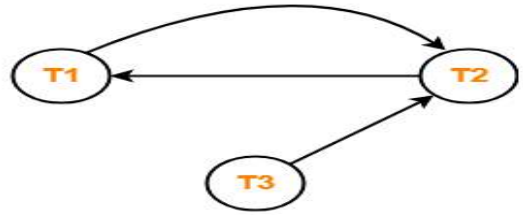
Step-01:

List all the conflicting operations and determine the dependency between the transactions-

- $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
- $R_3(B)$, $W_2(B)$ $(T_3 \rightarrow T_2)$

Step-02:

Draw the precedence graph-



- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

Example: conflict serializable and conflict equivalent

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

Schedule 3 — showing only the read and write instructions.

CONFLICT EQUIVALENT

- Using precedence graph we found that the schedule 3 is conflict serializable since no cycles formed in graph.
- If a schedule S can be transformed into a schedule S' by a series of swapping of non conflicting instruction , then we can say S and S' are **conflict equivalent.**
- Adjacent non conflicting pairs are swapped by position

Example :CONFLICT EQUIVALENT

- Consider schedule 3 which is conflict serializable.

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

Schedule 3 — showing only the read and write instructions.

Example :conflict equivalent(conti. .)

- To find the conflict equivalent of the schedule 3 we need to perform certain swapping, i. e **swapping of positions of non conflicting adjacent instructions** in T1 and T2

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

Schedule 5 — schedule 3 after swapping of a pair of instructions.

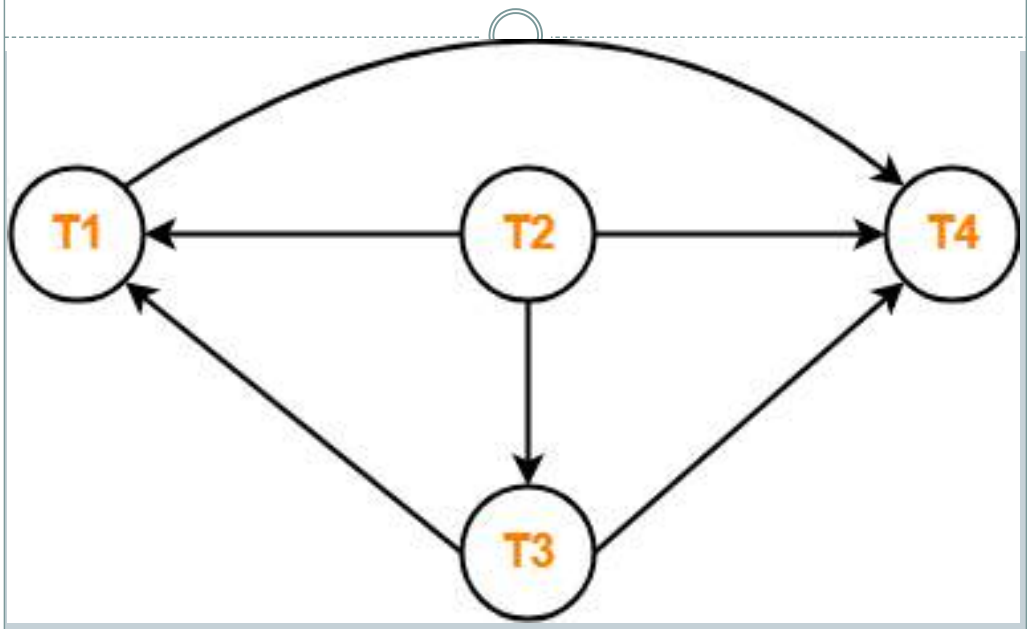
Example : conflict equivalent(conti. .)

- After a series of swapping we will get a serial schedule which is conflict equivalent of schedule 3.

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

Schedule 6 — a serial schedule that is equivalent to schedule 3.

T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit



- Question: Consider the following schedules involving two transactions. Which one of the following statement is true?
- S1: $R_1(X)\ R_1(y)\ R_2(X)\ R_2(y)\ W_2(y)\ W_1(X)$
  S2: $R_1(X)\ R_2(X)\ R_2(y)\ W_2(y)\ R_1(y)\ W_1(X)$
- Both S1 and S2 are conflict serializable
- Only S1 is conflict serializable
- Only S2 is conflict serializable
- None

- Only S2 is conflict serializable.

View Serializability

- If a given schedule is found to be view equivalent to some serial schedule, then it is called as a **view serializable schedule.**

View Equivalent Schedules-

- Consider two schedules S1 and S2 each consisting of two transactions T1 and T2.
- Two schedules S1 and S2 are said to be **view equivalent** if below conditions are satisfied .
  - 1. Initial Read
  - 2. Updated Read
  - 3. Final Write

1. Initial Read

- An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

- In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

3. Final Write

- A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

T1	T2	T3
Write(A)		
	Read(A)	
		Write(A)

Schedule S1

T1	T2	T3
	Read(A)	
Write(A)		
		Write(A)

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

- Condition 1 and 2 ensure that each transaction reads the same value in both schedules S1 and S2(so perform same computation) .
- Condition 3 together with conditions 1 and 2 ensures both schedules result in same final state.

- Every conflict serializable schedule is also view serializable .
- But all view serializable are not conflict serializable.
- Blind writes appear in any view serializable schedule that is not conflict serializable.

View serializability

- If a given schedule is found to be view equivalent to some serial schedule, then it is called as a **view serializable schedule**. .
- Consider two schedules S1 and S2 each consisting of two transactions T1 and T2. Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

- 1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must in schedule S'' also read the initial value of Q.
  - "Initial reads must be same for all data items"
- If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S1, then in schedule S2 also, transaction T_i must read the same data item that has been updated by transaction T_j.
  - "Write-read sequence must be same.".
-

- For each data item Q , the transaction that perform the final write(Q) operation in schedule S must perform the final Write(Q) operation in schedule in S" .
  - "Final writers must be same for all data items".

How to check whether a given schedule is view serializable or not?

- **Method-01:**

- Check whether the given schedule is conflict serializable or not.
- If the given schedule is conflict serializable, then it is surely view serializable.
- If the given schedule is not conflict serializable, then it may or may not be view serializable. Go and check using other methods.

- **Method-02:**
- Check if there exists any blind write operation (writing without reading a value is known as a blind write).
- If there does not exist any blind write, then the schedule is surely not view serializable. Stop and report your answer.
- If there exists any blind write, then the schedule may or may not be view serializable. Go and check using other methods.

- **Method-03:**
- In this method, try finding a view equivalent serial schedule.

EXAMPLE :

- To check whether S is view serializable:-

Example:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Schedule S

EXAMPLE : SOLUTION

With 3 transactions, the total number of possible schedule

= 3! = 6
S1 = <T1 T2 T3>
S2 = <T1 T3 T2>
S3 = <T2 T3 T1>
S4 = <T2 T1 T3>
S5 = <T3 T1 T2>
S6 = <T3 T2 T1>

Taking first schedule S1:

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

EXAMPLE:- solution

- Step 1: final updation on data items
- In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.
- Step 2: Initial Read
- The initial read operation in S is done by T1 and in S1, it is also done by T1.
- Step 3: Final Write
- The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

EXAMPLE:-SOLUTION (conti. .)

- The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.
- Hence, view equivalent serial schedule of S is S1:

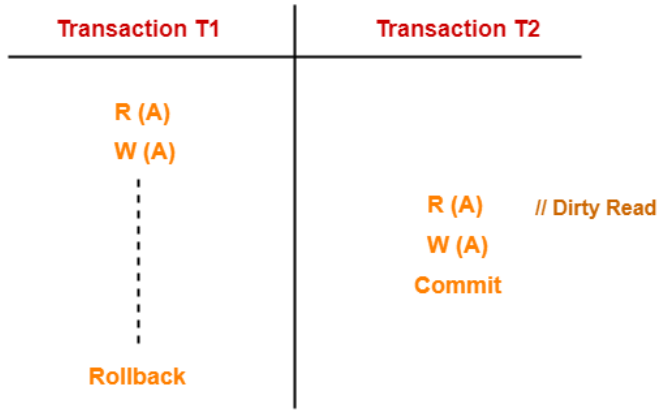
$$T1 \rightarrow T2 \rightarrow T3$$

Irrecoverable Schedules-

- If in a schedule,
  - A transaction performs a ***dirty read* operation** from an uncommitted transaction
  - And commits before the transaction from which it has read the value then such a schedule is known as an **Irrecoverable Schedule.**

Example : Irrecoverable schedule

Consider the following schedule-



Irrecoverable Schedule

Example: Irrecoverable schedule

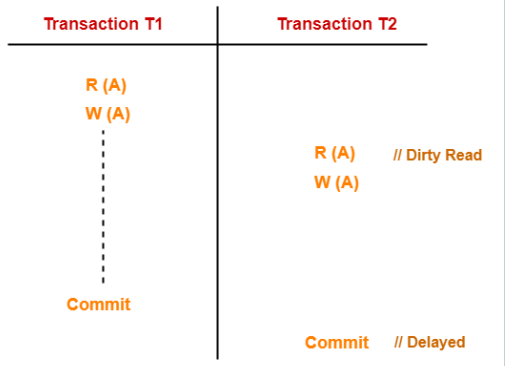
- In the above example
  - T2 performs a dirty read operation.
  - T2 commits before T1.
  - T1 fails later and roll backs.
  - The value that T2 read now stands to be incorrect.
  - T2 can not recover since it has already committed.
  - So the above schedule is an irrecoverable schedule.

Recoverable Schedules-

- If in a schedule,
  - A transaction performs a ***dirty read operation*** from an uncommitted transaction.
  - And its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is known as a **Recoverable Schedule.**

EXAMPLE: Recoverable Schedules-

Consider the following schedule-



Recoverable Schedule

- In the above example T2 performs a dirty read operation.
- The commit operation of T2 is delayed till T1 commits or roll backs.
- T1 commits later.
- T2 is now allowed to commit.

- In case, T1 would have failed, T2 has a chance to recover by rolling back.
- Since the commit operation of the transaction that performs the dirty read is delayed.
- This ensures that it still has a chance to recover if the uncommitted transaction fails later.

Recoverable schedule

- Two types:
  - Cascadeless schedule
  - Cascading schedule

CASCADING SCHEDULE

- Even if a schedule is recoverable , to recover correctly from failure of transaction Ti, we may have to roll back several transaction.
- Such situations occur if transactions have read data written by Ti.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

Figure 14.15 Schedule 10.

- In the above example , transaction T8 has been aborted.
- T8 must be rolled back.
- Since T9 is dependent on T8, T9 must be rolled back. Since T10 is dependent on T9, T10 must be rolled back.
- The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback.**

- *Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.*

Cascadeless schedule

- *A cascadeless schedule is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the read operation of Tj.*
- *This type of schedule is called* ***cascadeless schedule.***

Cascadeless Schedule

T10	T11
read(A)	
write(A)	
	read(B)
commit	
	read(A)

Module iv

DATA STORAGE AND QUERYING

RAID(redundant arrays of independent disks

- The data-storage requirements of some applications (in particular Web, database,and multimedia applications) have been growing so fast that a large number of disks are needed to store their data, even though disk-drive capacities have been growing very fast.
- Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel

- A variety of disk-organization techniques, collectively called redundant arrays of independent disks (RAID), have been proposed to achieve improved performance and reliability.
- RAID (redundant array of independent disks) is a way of storing the same data in different places on multiple hard disks or solid-state drives to protect data in the case of a drive failure.

- Store extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information
- Effective mean time to failure is increased
- Simplest (but expensive) approach to redundancy is to duplicate every disk.
- This technique is called ***mirroring*** (**shadowing**).

- A logical disk then consists of two physical disks, and every write is carried out on both disks.
- If one of the disks fails, the data can be read from the other.Data will be lost only if the second disk fails before the first failed disk is repaired.
- With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled.

- With multiple disks, we can improve the transfer rate as well (or instead) by **striping** data across multiple disks. In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

Different type of data striping :

1. **Bit level striping** : Splitting the bits of each byte across multiple disks

   : No of disks either is a multiple of 8 or a factor of 8

   : These disks are considered as single disk.

   E.g. : Array of eight disks, write bit i of each byte to disk I

2. **Block-level striping** : Stripes blocks across multiple disks

   : Fetches n blocks in parallel from the n disks

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- **Block-level striping** stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers; we assume the block numbers start from 0. With an array of n disks, block-level striping assigns logical block i of the disk array to disk (i mod n) + 1; it uses the (i /n)th physical block of the disk to store logical block i . For example, logical block 11 is stored in physical block 1 of disk 4.

- When reading a large file, block-level striping fetches n blocks at a time in parallel from the n disks, giving a high data-transfer rate for large reads.
  When a single block is read, the data-transfer rate is the same as on one disk, but the remaining n − 1 disks are free to perform other actions.

- In summary, there are two main goals of parallelism in a disk system:
  1. Load-balance multiple small accesses (block accesses), so that the throughput of such accesses increases.
  2. Parallelize large accesses so that the response time of large accesses is reduced.

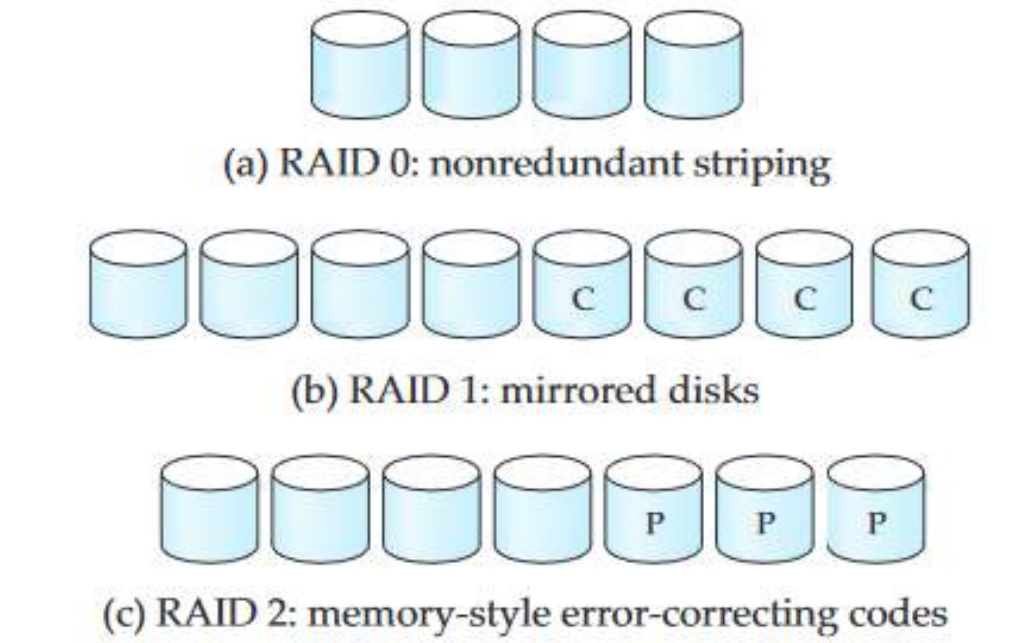
RAID Levels

- Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but does not improve reliability.
- Various alternative schemes aim to provide redundancy at lower cost by combining disk striping with “parity” bits
- These schemes have different cost—performance trade-offs. The schemes are classified into RAID levels
- (For all levels, the figure depicts four disks’ worth of data, and the extra disks depicted are used to store redundant information for failure recovery.)

- **RAID level 0** refers to disk arrays with striping at the level of blocks, but without any redundancy (such as mirroring or parity bits).
- **RAID level 1** refers to disk mirroring with block striping.
- **RAID level 2**, known as memory-style error-correcting-code (ECC) organization, employs parity bits. Memory systems have long used parity bits for error detection and correction

- Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte that are set to 1 is even (parity = 0) or odd (parity = 1).
- If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity.
- Similarly,if the stored parity bit gets damaged, it will not match the computed parity.Thus, all 1-bit errors will be detected by the memory system. Error-correcting schemes store 2 or more extra bits, and can reconstruct the data if a single bitgets damaged.

- The idea of error-correcting codes can be used directly in disk arrays by striping bytes across disks. For example, the first bit of each byte could be
stored in disk 0, the second bit in disk 1, and so on until the eighth bit is stored in disk 7, and the error-correction bits are stored in further disks.



(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks

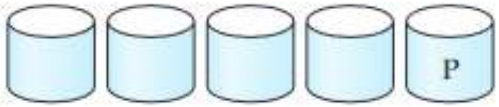
(c) RAID 2: memory-style error-correcting codes

- The disks labeled P store the error correction bits. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks, and can be used to reconstruct the damaged data.

- **RAID level 3**, bit-interleaved parity organization, improves on level 2 by exploiting the fact that disk controllers, unlike memory systems, can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection.
- If one of the sectors gets damaged, the system knows exactly which sector it is, and, for each bit in the sector, the system can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1

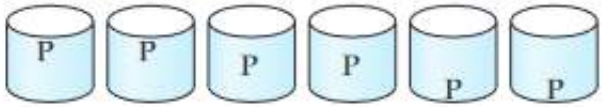
- RAID level 3 is as good as level 2, but is less expensive in the number of extra disks (it has only a one-disk overhead), so level 2 is not used in practice.
- RAID level 3 has two benefits over level 1. It needs only one parity disk for several regular disks, whereas level 1 needs one mirror disk for every
disk, and thus level 3 reduces the storage overhead.

(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity

(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

- **RAID level 4**, block-interleaved parity organization, uses block-level striping, like RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks.
- If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk

- **RAID level 5**, block-interleaved distributed parity, improves on level 4 by partitioning data and parity among all N + 1 disks, instead of storing data in
N disks and parity in one disk.

- **RAID level 6**, the P + Q redundancy scheme, is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures.
Instead of using parity, level 6 uses error-correcting codes.

File Organization

- A database is mapped into a number of different files that are maintained by the underlying operating system. These files reside permanently on disks.
- A file is organized logically as a sequence of records. These records are mapped onto disk blocks.
- Each file is also logically partitioned into fixed-length storage units called **blocks,** which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default

- A block may contain several records; the exact set of records that a block contains is determined by the form of physical data organization being used.
- In a relational database, tuples of distinct relations are generally of different sizes. One approach to mapping the database to files is to use several files, and to store records of only one fixed length in any given file. An alternative is to structure our files so that we can accommodate multiple lengths for records;

Fixed-Length Records

- As an example, let us consider a file of instructor records for our university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
                    ID varchar (5);
                    name varchar(20);
                    dept_name varchar (20);
                    salary numeric (8,2);
                  end
```

- Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes.
- Suppose that instead of allocating a variable amount of bytes for the attributes ID, name, and dept name, we allocate the maximum number of bytes that each attribute can hold.
- Then, the instructor record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on

- However, there are two problems with this simple approach:
- 1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
  2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored

- To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block.
- When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead .
- Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

e 10.5 File of Figure 10.4, with record 3 deleted and all records moved.

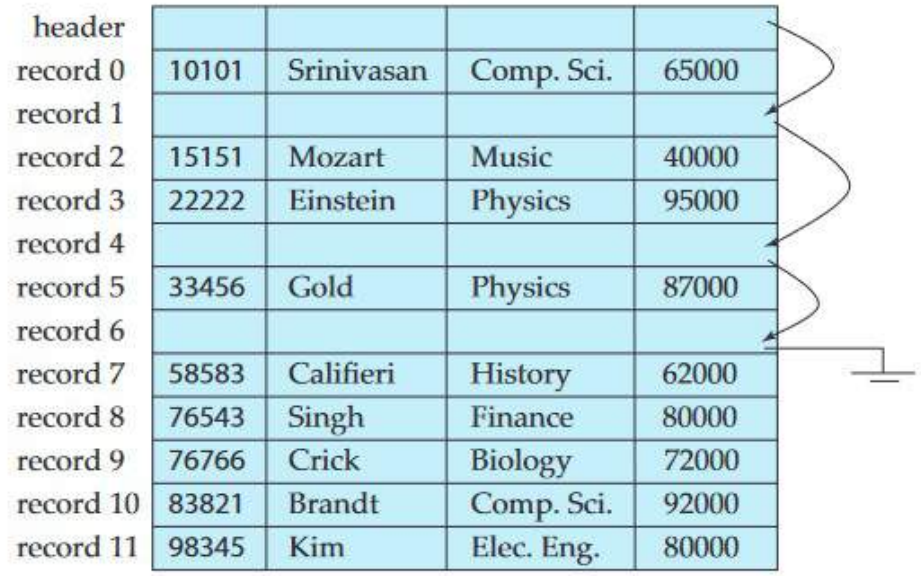
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000

10.6 File of Figure 10.4, with record 3 deleted and final record moved.

- It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

- At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted.
- We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as pointers, since they point to the location of a record.
- The deleted records thus form a linked list, which is often referred to as a **free list**.

- On insertion of a new record, we use the record pointed to by the header.
- We change the header pointer to point to the next available record. If no space is available, we add the new record to the end of the file.
- Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record.



header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Figure 10.7 File of Figure 10.4, with free list after deletion of records 1, 4, and 6.

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields, such as arrays or multisets.

- Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:
  - How to represent a single record in such a way that individual attributes can be extracted easily.
  - How to store variable-length records within a block, such that records in a block can be extracted easily

- The representation of a record with variable-length attributes typically has two parts: **an initial part** with fixed length attributes, followed **by data for variable-length attributes**.
- Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value.
- Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (offset, length), where offset denotes where the data for that attribute begins within the record, and length is the length in bytes of the variable-sized attribute.

- The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

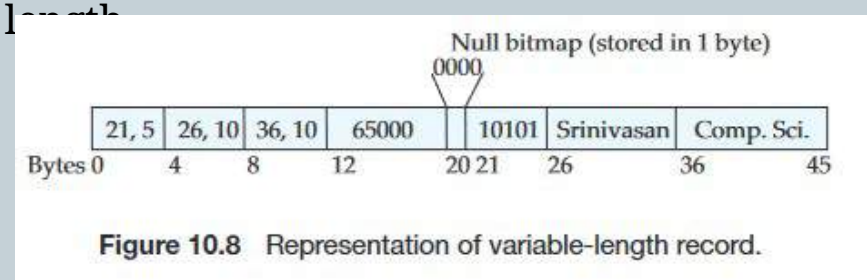


Figure 10.8 Representation of variable-length record.

- **null bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored
  in bytes 12 through 19 would be ignored.
- The slotted-page structure is commonly used for organizing variable length records within a block

- There is a header at the beginning of each block, containing the following information:
  1. The number of record entries in the header.
  2. The end of free space in the block.
  3. An array whose entries contain the location and size of each record

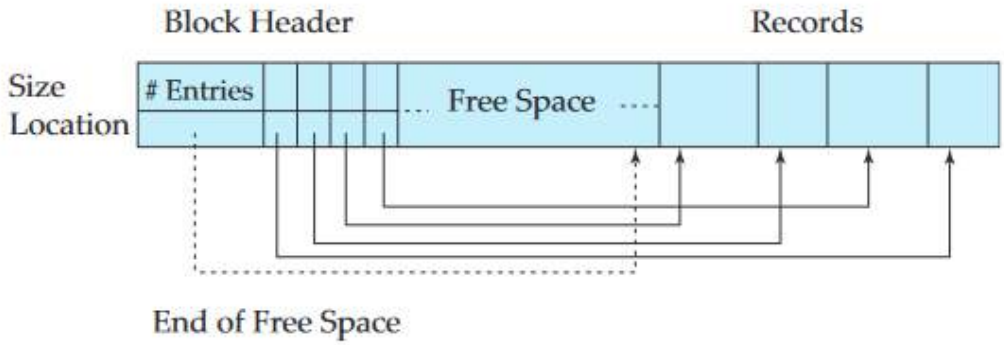


Figure 10.9 Slotted-page structure.

- The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array, and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to deleted (its size is set to −1, for example).

Organization of Records in Files

- Several of the possible ways of organizing records in files are:
- **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization.** Records are stored in sequential order, according to the value of a “search key”of each record

- **Hashing file organization**. A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the file the record should be placed.
- Generally, a separate file is used to store the records of each relation. However, in a multitable clustering file organization, records of several different relations are stored in the same file

Sequential File Organization

- A sequential file is designed for efficient processing of records in sorted order based on some search key.
- A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- To permit fast retrieval of records in search-key order, we chain together records by pointers.
- The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

- The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms.
- For insertion, we apply the following rules:
  1. Locate the record in the file that comes before the record to be inserted in search-key order.
  2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

multitable clustering file organization

- A multitable clustering file organization is a file organization, such as that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently

Indexing and Hashing

- **Basic Concepts**
- An index for a file in a database system works in much the same way as the index of textbook.
- Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

- Keeping a sorted list of students' ID would not work well on very large databases with thousands of students, since the index would itself be very big;
  further, even though keeping the index sorted reduces the search time, finding a student can still be rather time-consuming. Instead, more sophisticated indexing techniques may be used.

- There are two basic kinds of indices:
- **Ordered indices**. Based on a sorted ordering of the values.
- **Hash indices**. Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function,
called a hash function.

- several techniques for both ordered indexing and hashing.
- Each technique must be evaluated on the basis of these factors:
- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
• **Access time**: The time it takes to find a particular data item, or set of items, using the technique in question.

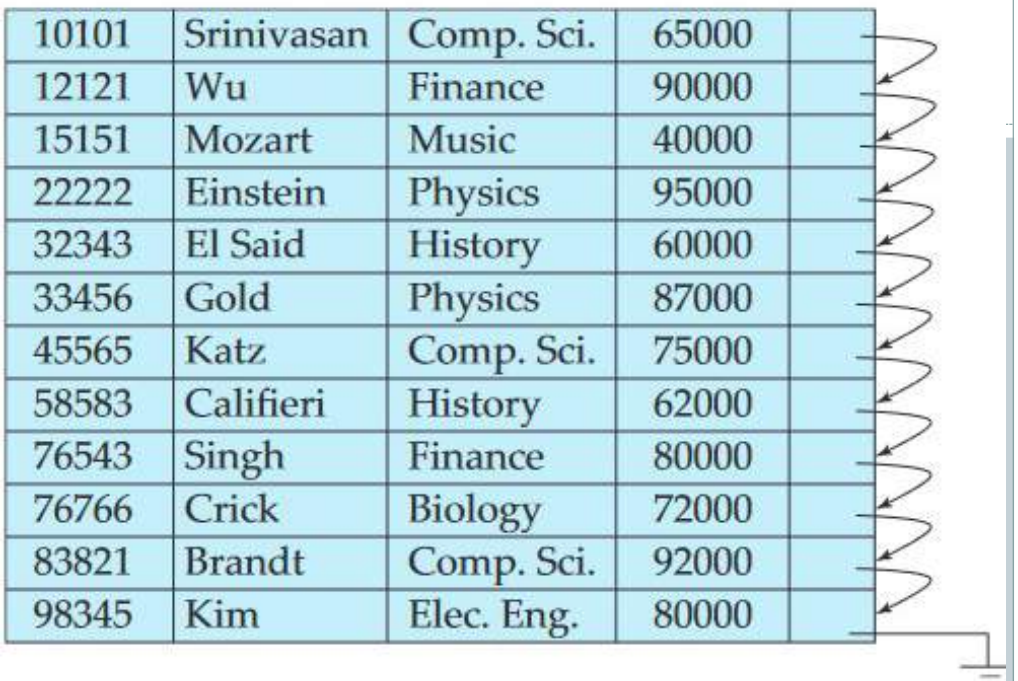
- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
• **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worth while to sacrifice the space to achieve improved performance.

- We often want to have more than one index for a file.
- An attribute or set of attributes used to look up records in a file is called a **search key**.

1.Ordered Indices

- To gain fast random access to records in a file, we can use an index structure.
- Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.
- The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute.

- file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.
- Clustering indices are also called **primary indices**
- Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices, or secondary indices**



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

Figure 11.1 Sequential file for *instructor* records.

1.1 Dense and Sparse Indices

- An index entry, or index record, consists of a search-key value and pointers to one or more records with that value as their search-key value.
- The pointer to a record consists of the identifier of a disk block and an offset within the disk block
  to identify the record within the block.
- There are two types of ordered indices that we can use:

- **Dense index:** In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value.
- The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value

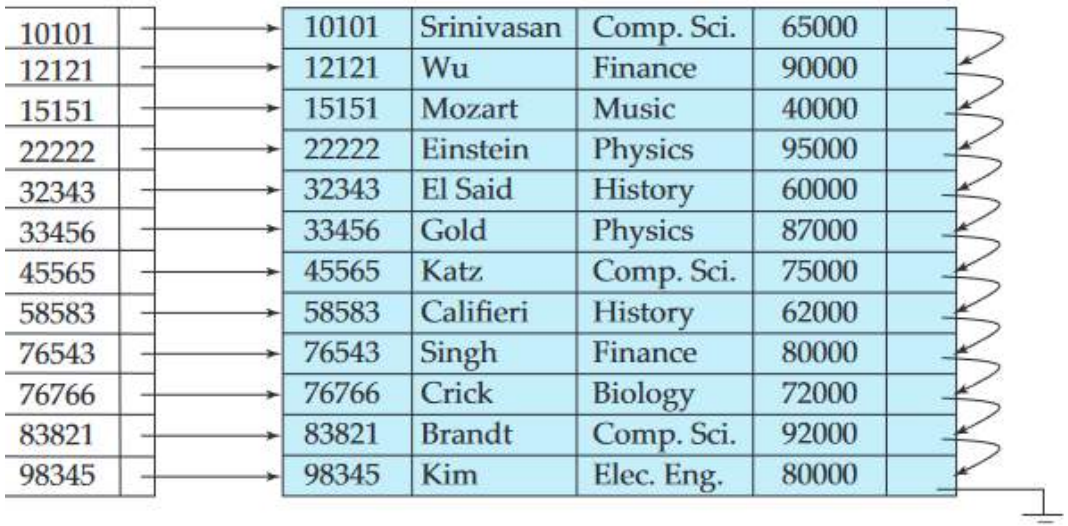


Figure 11.2 Dense index.

- **Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index.

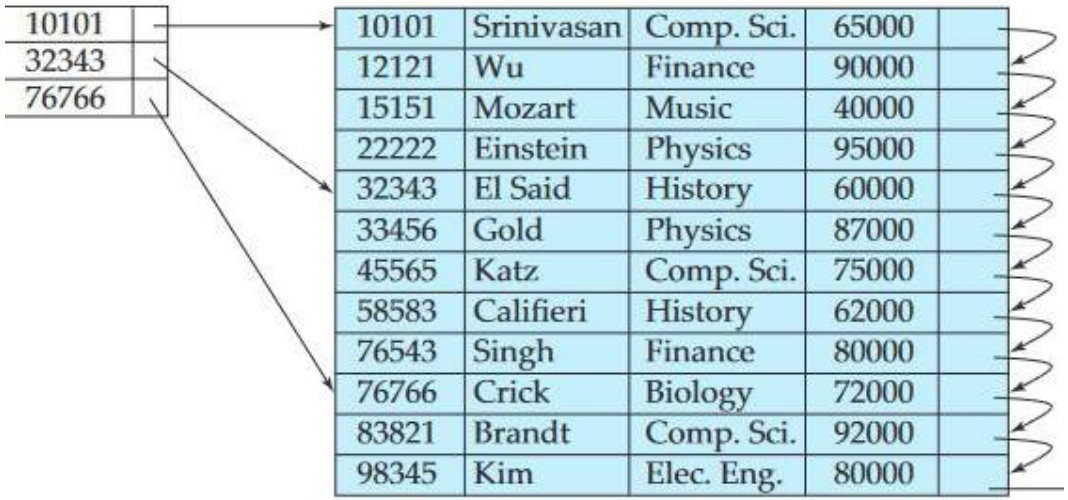
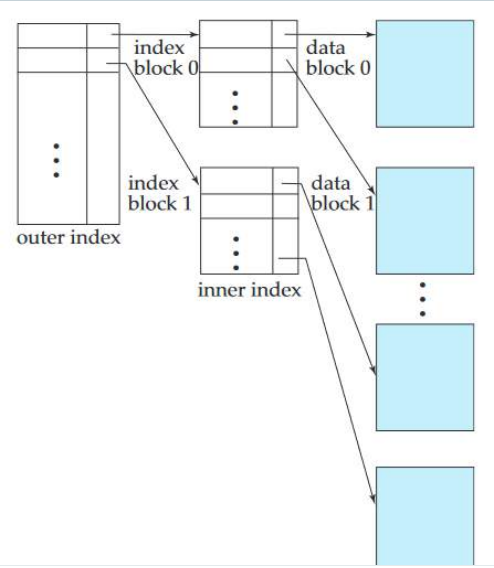


Figure 11.3 Sparse index.

multilevel indices

- Indices with two or more levels are called multilev



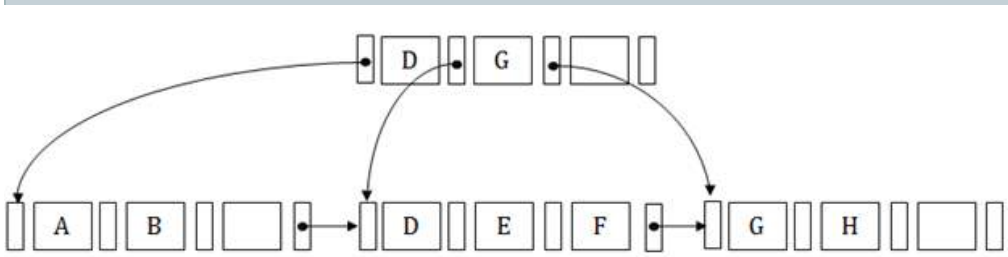
B+-Tree Index Files

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

- The main disadvantage of the index-sequential file organization is that performance degrades as the file grows.
- The **B+-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.
- A **B+-tree** index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each nonleaf node in the tree has between [n/2] and n children, where n is fixed for a particular tree.

Structure of a B+-Tree

- In the B+ tree, every leaf node is at equal distance from the root node. The B+ tree is of the order n where n is fixed for every B+ tree.
- It contains an internal node and leaf node.



- A B +-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file.

Figure 11.7 Typical node of a B+-tree.

- It contains up to n − 1 search-key values K1,K2,...,Kn − 1, and n pointers P1,P2,...,Pn. The search-key values within a node are kept in sorted order; thus, if i <j , then Ki <K j

- We consider first the structure of the leaf nodes.
- For i = 1,2,...,n−1, pointer Pi points to a file record with search-key value Ki . Pointer Pn has a special purpose
- Since there is a linear order on the leaves based on the search-key values that they contain, we use Pn to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.

Figure 11.7 Typical node of a B+-tree.

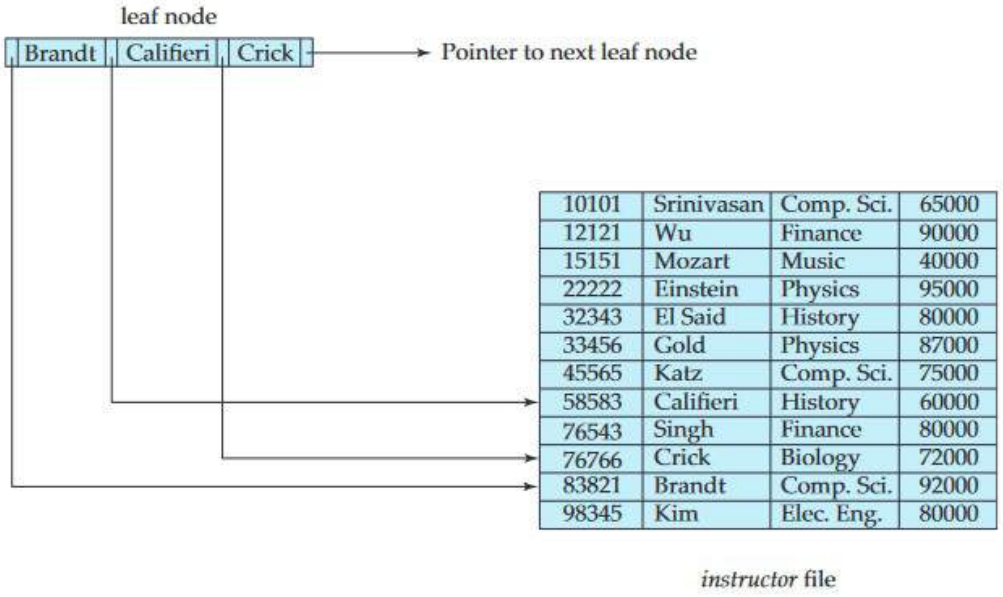


Figure 11.8 A leaf node for *instructor* B+-tree index ($n = 4$).

- The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes.
- The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.
- A nonleaf node may hold up to n pointers, and must hold at least [n/2] pointers.
- The number of pointers in a node is called the fanout of the node.
- Nonleaf nodes are also referred to as internal nodes.

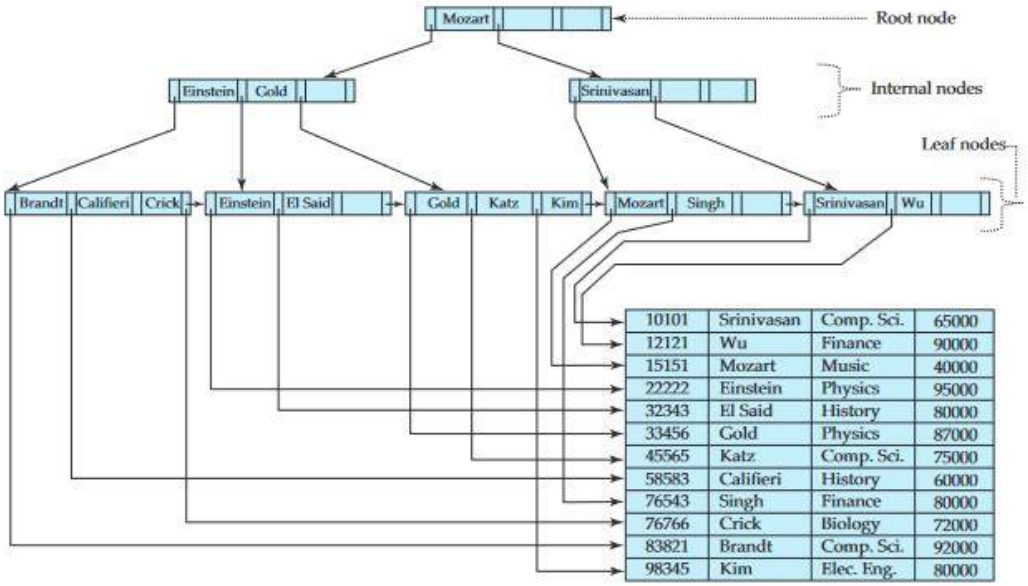


Figure 11.9 B+-tree for *instructor* file ($n = 4$).

- These examples of B+-trees are all balanced. That is, the length of every path from the root to a leaf node is the same.
- This property is a requirement for a B+tree. Indeed, the "B"in B+-tree stands for "balanced."
- It is the balance property of B+-trees that ensures good performance for lookup, insertion, and deletion.

Module4

PART-2

Hashing –Statuc hashing

- One disadvantage of sequential file organization is that we must access an index structure to locate data.
- File organizations based on the technique of hashing allow us to avoid accessing an index structure.
- In our description of hashing, we shall use the term bucket to denote a unit of storage that can store one or more records.
- A **bucket** is typically a disk block, but could be chosen to be smaller or larger than a disk block

- let K denote the set of all search-key values, and let B denote the set of all bucket addresses. A hash function h is a function from K to B. Let h denote
a hash function.
- To insert a record with search key Ki , we compute h(Ki), which gives the address of the bucket for that record.
- To perform a lookup on a search-key value Ki , we simply compute h(Ki), then search the bucket with that address.

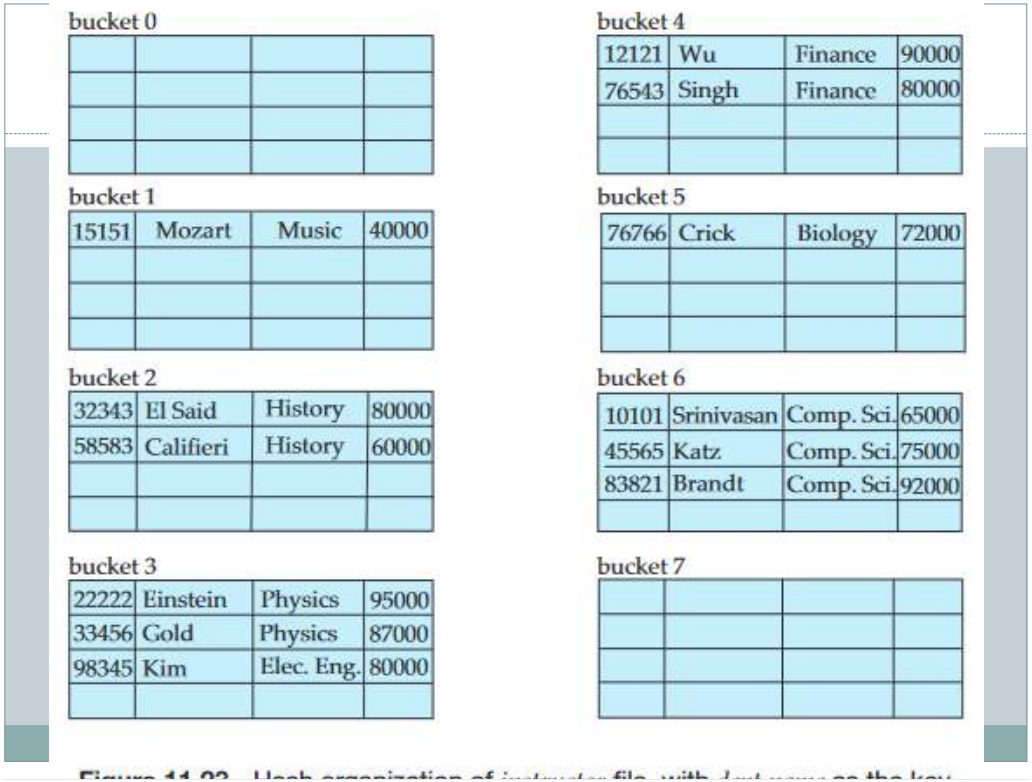
- Suppose that two search keys, K5 and K7, have the same hash value; that is, h(K5) = h(K7).
- If we perform a lookup on K5, the bucket h(K5) contains records with search-key values K5 and records with search-key values K7.
- Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

- Deletion is equally straightforward. If the search-key value of the record to be deleted is Ki , we compute h(Ki), then search the corresponding bucket for that
record, and delete the record from the bucket.

- Hashing can be used for two different purposes. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by
computing a function on the search-key value of the record.
- In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

Hash Functions

- An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.
- Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key
  values to buckets in such a way that the distribution has these qualities:
  - The distribution is uniform.
  - The distribution is random



bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

- Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file

Handling of Bucket Overflows

- If the bucket does not have enough space, a bucket overflow is said to occur. Bucket overflow can occur for several reasons:
- **Insufficient buckets**.
- The number of buckets, which we denote nB , must be chosen such that nB >nr /fr , where nr denotes the total number of records that will be stored and fr denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.

- **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space.
- This situation is called bucket skew.
- Skew can occur for two reasons:
  1. Multiple records may have the same search key.
  2. The chosen hash function may result in nonuniform distribution of search keys

- So that the probability of bucket overflow is reduced, the number of buckets is chosen to be (nr /fr) ∗ (1 + d), where d is a fudge factor, typically around 0.2.
  Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.

- Despite allocation of a few more buckets than required, bucket overflow can still occur.
- We handle bucket overflow by using **overflow buckets**. If a record must be inserted into a bucket b, and b is already full, the system provides an overflow bucket for b, and inserts the record into the overflow bucket.
- If the overflow bucket is also full, the system provides another overflow bucket, and so on. All the overflow buckets of a given bucket are chained together in a linked list,

- Overflow handling using such a linked list is called **overflow chaining.**

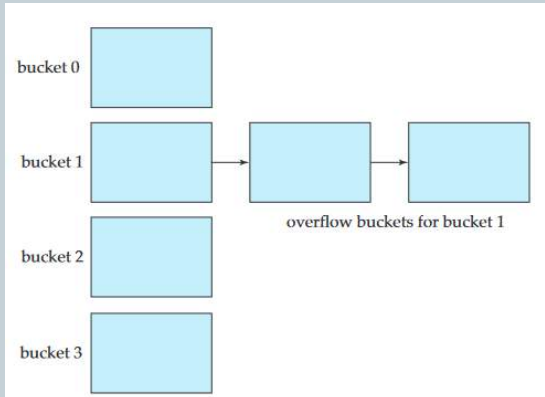


Figure 11.24 Overflow chaining in a hash structure.

- We must change the lookup algorithm slightly to handle overflow chaining.
  As before, the system uses the hash function on the search key to identify a bucket b. The system must examine all the records in bucket b to see whether they match the search key, as before. In addition, if bucket b has overflow buckets, the system
  must examine the records in all the overflow buckets also.
- The form of hash structure is closed hashing.

- Under an alternative approach, called open hashing, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets B.
  One policy is to use the next bucket (in cyclic order) that has space; this policy is called **linear probing**.

Dynamic Hashing

- the need to fix the set B of bucket addresses presents a serious problem with the static hashing technique.
- Most databases grow larger over time. If we are to use static hashing for such a database, we have
  three classes of options:
- 1. Choose a hash function based on the current file size
- 2. Choose a hash function based on the anticipated size of the file at some point
  in the future
- 3. Periodically reorganize the hash structure in response to file growth.

- Several dynamic hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database.
- Extendable hashing(dynamic hashing) copes with changes in database size by splitting and combining buckets as the database grows and shrinks. As a result, space efficiency is retained.
- With extendable hashing, we choose a hash function h with the desirable properties of uniformity and randomness. However, this hash function generates values over a relatively large range—namely, b-bit binary integers. A typical value for b is 32.

Module v

Distributed Databases

- A distributed database is basically a database that is not limited to one system, it is spread over different sites, i. e, on multiple computers or over a network of computers.
- A distributed database system is located on various sites that don't share physical components. This may be required when a particular database needs to be accessed by various users globally.
- It needs to be managed such that for the users it looks like one single database.

Homogeneous and Heterogeneous Databases

- In a homogeneous distributed database system, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests.
- In contrast, in a heterogeneous distributed database, different sites may use different schemas, and different database-management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing.

Distributed Data Storage

- Consider a relation r that is to be stored in the database. There are two approaches
  to storing this relation in the distributed database:
  • **Replication.** The system maintains several identical replicas (copies) of the relation, and stores each replica at a different site. The alternative to replication is to store only one copy of relation r.
  • **Fragmentation.** The system partitions the relation into several fragments, and stores each fragment at a different site.

- Fragmentation and replication can be combined: A relation can be partitioned into several fragments and there may be several replicas of each fragment

Data Replication

- If relation r is replicated, a copy of relation r is stored in two or more sites. In the most extreme case, we have full replication, in which a copy is stored in every site in the system. There are a number of advantages and disadvantages to replication.
- **Availability:** If one of the sites containing relation r fails, then the relation r can be found in another site. Thus, the system can continue to process queries involving r, despite the failure of one site

- **Increased parallelism.**
- In the case where the majority of accesses to the relation r result in only the reading of the relation, then several sites can process queries involving r in parallel. The more replicas of r there are, the greater the chance that the needed data will be found in the site where the transaction is executing. Hence, data replication minimizes movement of data between sites.

- **Increased overhead on update.**
- The system must ensure that all replicas of a relation r are consistent; otherwise, erroneous computations may result. Thus, whenever r is updated, the update must be propagated to all sites containing replicas. The result is increased overhead. For example, in a banking system, where account information is replicated in various sites, it is necessary to ensure that the balance in a particular account agrees in all sites.

- We can simplify the management of replicas of relation r by choosing one of them as the primary copy of r.
- For example, in a banking system, an account can be associated with the site in which the account has been opened.

Data Fragmentation

- If relation r is fragmented, r is divided into a number of fragments r1, r2, ... , rn. These fragments contain sufficient information to allow reconstruction of the original relation r.
- There are two different schemes for fragmenting a relation: **horizontal fragmentation** and **vertical fragmentation**.
- Horizontal fragmentation splits the relation by assigning each tuple of r to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme R of relation r.

- In horizontal fragmentation, a relation r is partitioned into a number of subsets, r1, r2, ... , rn. Each tuple of relation r must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.
- the account relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.

$$account_1 = \sigma_{branch\_name = \text{“Hillside”}}(account)$$
$$account_2 = \sigma_{branch\_name = \text{“Valleyview”}}(account)$$

- a horizontal fragment can be defined as a selection on the global relation r. That is, we use a predicate P_i to construct fragment r_i :

$$r_i = \sigma_{P_i}(r)$$

- We reconstruct the relation r by taking the union of all fragments; that is:

$$r = r_1 \cup r_2 \cup \cdots \cup r_n$$

- vertical fragmentation is the same as decomposition. Vertical fragmentation of r(R) involves the definition of several subsets of attributes R1, R2, . . . , Rn of the schema R so that:
  $R = R1 \cup R2 \cup \cdots \cup R$
- Each fragment r_i of r is defined by

$$r_i = \Pi_{R_i}(r)$$

- we can r
  r from t natural join:

$$r = r_1 \bowtie r_2 \bowtie r_3 \bowtie \cdots \bowtie r_n$$

- One way of ensuring that the relation r can be reconstructed is to include the primary-key attributes of R in each Ri . More generally, any superkey can be used.
  It is often convenient to add a special attribute, called a tuple-id, to the schema R

Transparency

- The user of a distributed database system should not be required to know where the data are physically located nor how the data can be accessed at the specific local site. This characteristic, called data transparency, can take several forms:
- Fragmentation transparency. Users are not required to know how a relation has been fragmented.

- **Replication transparency.** Users view each data object as logically unique. The distributed system may replicate an object to increase either system performance or data availability.
  Users do not have to be concerned with what data objects have been replicated, or where replicas have been placed.
- **Location transparency.** Users are not required to know the physical location of the data. The distributed database system should be able to find any data
  as long as the data identifier is supplied by the user transaction.

- Data items—such as relations, fragments, and replicas—must have unique names. This property is easy to ensure in a centralized database. In a distributed database, however, we must take care to ensure that two sites do not use the same name for distinct data items.
- One solution to this problem is to require all names to be registered in a central **name server**. The name server helps to ensure that the same name does not get used for different data items.

- The database system can create a set of alternative names, or aliases, for data items. A user may thus refer to data items by simple names that are translated by the system to complete names.

Distributed Transactions

- Access to the various data items in a distributed system is usually accomplished through transactions, which must preserve the ACID properties .
- There are two types of transaction that we need to consider. **The local transactions** are those that access and update data in only one local database; the **global transactions** are those that access and update data in several local databases.

System Structure

- Each site has its own local transaction manager, whose function is to ensure the ACID properties of those transactions that execute at that site. The various transaction managers cooperate to execute global transactions.
- each site contains two subsystems
- The **transaction manager** manages the execution of those transactions (or subtransactions) that access data stored in a local site.
- The transaction coordinator coordinates the execution of the various transactions (both local and global) initiated at that site.

- Each transaction manager is responsible for:
  - Maintaining a log for recovery purposes.
  - Participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site.

- the coordinator is responsible for:
  - Starting the execution of the transaction.
  - Breaking the transaction into a number of sub transactions and distributing these subtransactions to the appropriate sites for execution.
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

System Failure Modes

- A distributed system may suffer from the same types of failure that a centralized system does
- • Failure of a site.
  - Loss of messages.
  - Failure of a communication link.
  - Network partition.

Object-BasedDatabases

- complex application domains require correspondingly complex data types, such as nested record structures, multivalued attributes, and inheritance, which are supported by traditional programming languages.
- The object-relational data model extends the relational data model by providing a richer type system including complex data types and object orientation.

- Object-relational database systems, that is, database systems based on the object-relation model, provide a convenient migration path for users of relational databases who wish to use object-oriented features.

- Two approaches are used
  1. Build an object-oriented database system, that is, a database system that natively supports an object-oriented type system, and allows direct access to data from an object-oriented programming language using the native type system of the language.
  2. Automatically convert data from the native type system of the programming language to a relational representation, and vice versa. Data conversion is specified using an object-relational mapping.

Complex Data Types

- Traditional database applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic.
- Consider, for example, addresses.
- While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries.

- On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field.
- A better alternative is to allow structured data types that allow a type address with subparts street address, city, state, and postal code.
- With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization, and specialization directly, without a complex translation to the relational model.

- Rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities).

Structured Types and Inheritance in SQL

- Structured Types
- Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute name with component attribute firstname and lastname:

```
create type Name as
        (firstname varchar(20),
        lastname varchar(20))
        final;
```

- the following structured type can be used to represent a composite attribute address:

```
create type Address as
        (street varchar(20),
        city varchar(20),
        zipcode varchar(9))
        not final;
```

- Such types are called user-defined types in SQL. The final and not final specifications are related to subtyping.

- We can now use these types to create composite attributes in a relation, by simply declaring an attribute to be of one of these types. For example, we could create a table person as follows:

```
create table person (
        name Name,
        address Address,
        dateOfBirth date);
```

- The components of a composite attribute can be accessed using a "dot" notation; for instance name. firstname returns the firstname component of the name attribute. An access to attribute name would return a value of the structured type Name.
- We can also create a table whose rows are of a user-defined type. For example, we could define a type PersonType and create the table person as follows:

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
create table person of PersonType;
```

- An alternative way of defining composite attributes in SQL is to use unnamed row types.

```
create table person_r (
        name row (firstname varchar(20),
                  lastname varchar(20)),
        address row (street varchar(20),
                     city varchar(20),
                     zipcode varchar(9)),
        dateOfBirth date);
```

- This definition is equivalent to the preceding table definition, except that the attributes name and address have unnamed types, and the rows of the table also have an unnamed type

- The following query illustrates how to access component attributes of a composite attribute. The query finds the last name and city of each person.

```
select name.lastname, address.city
from person;
```

- A structured type can have methods defined on it. We declare methods as part of the type definition of a structured type:

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
    method ageOnDate(onDate date)
        returns interval year;
```

We create the method body separately:

```
create instance method ageOnDate (onDate date)
        returns interval year
        for PersonType
    begin
        return onDate – self.dateOfBirth;
    end
```

- *Note that the for clause indicates which type this method is for, while the keyword instance indicates that this method executes on an instance of the Person type. The variable self refers to the Person instance on which the method is invoked.*
- *constructor functions are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type.*

- *we could declare a constructor for the type Name like this*

```
ate function Name (firstname varchar(20), lastname varchar(20
urns Name
gin
   set self.firstname = firstname;
   set self.lastname = lastname;
d
```

- *We can then use new Name('John', 'Smith') to create a value of the type Name.*
  We can construct a row value by listing its attributes within parentheses.
- *By default every structured type has a constructor with no arguments, which sets the attributes to their default values. Any other constructors have to be created explicitly. There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments*

- The following statement illustrates how we can create a new tuple in the Person relation

```
insert into Person
        values
                (new Name('John', 'Smith'),
                new Address('20 Main St', 'New York', '11001'),
                date '1960-8-22');
```

Type Inheritance

- Suppose that we have the following type definition for people:

```
create type Person
        (name varchar(20),
         address varchar(20));
```

- We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are alsopeople, we can use inheritance to define the student and teacher types in SQL:

```
create type Student
        under Person
        (degree varchar(20),
         department varchar(20));

create type Teacher
        under Person
        (salary integer,
         department varchar(20));
```

- Both Student and Teacher inherit the attributes of Person —namely, name and address. Student and Teacher are said to be subtypes of Person, and Person is a supertype of Student, as well as of Teacher .
- Multiple inheritance

```
create type TeachingAssistant
        under Student, Teacher;
```

- The SQL standard requires an extra field at the end of the type definition, whose value is either **final** or **not final**. The keyword final says that subtypes may not be created from the given type, while not final says that subtypes may be created.

Table Inheritance

- create table people of Person;
- We can then define tables students and teachers as subtables of people,

```
create table students of Student
    under people;

create table teachers of Teacher
    under people;
```

- Further, when we declare students and teachers as subtables of people, everytuple present in students or teachers becomes implicitly present in people. Thus, if a query uses the table people, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely students and teachers. However, only those attributes that are present in people can be accessed by that query.

- SQL permits us to find tuples that are in people but not in its subtables by using "only people" in place of people in a query. The only keyword can also be used in delete and update statements. Without the only keyword, a delete statement on a supertable, such as people, also deletes tuples that were originally inserted in subtables (such as students);
- delete from people where P;
- would delete all tuples from the table people, as well as its subtables students and teachers, that satisfy P

- If the only keyword is added to the above statement, tuples that were inserted in subtables are not affected, even if they satisfy the where clause conditions
- multiple inheritance is possible with tables,

```
create table teaching_assistants
of TeachingAssistant
      under students, teachers;
```

Array and Multiset Types in SQL

- SQL supports two collection types: arrays and multisets;
- a multiset is an unordered collection, where an element may occur multiple times.

```
create type Publisher as
      (name varchar(20),
       branch varchar(20));

create type Book as
      (title varchar(20),
       author_array varchar(20) array [10],
       pub_date date,
       publisher Publisher,
       keyword_set varchar(20) multiset);

create table books of Book;
```

Creating and Accessing Collection Values

- An array of values can be created in SQL:

```
array['Silberschatz', 'Korth', 'Sudarshan']
```

- Similarly, a multiset of keywords can be constructed as

```
multiset['computer', 'database', 'SQL']
```

is, we can create a tuple of the type defined by the *books* relation as:

```
ompilers', array['Smith', 'Jones'], new Publisher('McGraw-Hill', 'New York'),
                                    multiset['parsing', 'analysis'])
```

```
nsert into books
alues ('Compilers', array['Smith', 'Jones'],
                    new Publisher('McGraw-Hill', 'New York'),
                    multiset['parsing', 'analysis']);
```

Object-Identity and Reference Types in SQL

- Object-oriented languages provide the ability to refer to objects. An attribute of a type can be a reference to an object of a specified type.
- For example, in SQL we can define a type Department with a field name and a field head that is a reference to the type Person, and a table departments of type Department, as follows:

```
create type Department (
        name varchar(20),
        head ref(Person) scope people);

create table departments of Department;
```

- We can omit the declaration scope people from the type declaration and instead
  make an addition to the create table statement

```
create table departments of Department
        (head with options scope people);
```

- The referenced table must have an attribute that stores the identifier of the tuple. We declare this attribute, called the self-referential attribute, by adding a ref is clause to the create table statement:

Next generation databases

- CAP theorem
- The CAP Theorem is comprised of three components (hence its name) as they relate to distributed data stores:
- **Consistency.** All reads receive the most recent write or an error.
- **Availability.** All reads contain data, but it might not be the most recent.
- **Partition tolerance.** The system continues to operate despite network failures (ie; dropped partitions, slow network connections, or unavailable network connections between nodes.)

- In normal operations, your data store provides all three functions. But the CAP theorem maintains that when a distributed database experiences a network failure, you can provide either consistency or availability.
- In the theorem, partition tolerance is a must. The assumption is that the system operates on a distributed data store so the system, by nature, operates with network partitions. Network failures will happen, so to offer any kind of reliable service, partition tolerance is necessary—the P of CAP.

- if Partition means a break in communication then Partition tolerance would mean that the system should still be able to work even if there is a partition in the system. Meaning if a node fails to communicate, then one of the replicas of the node should be able to retrieve the data required by the user.
- The CAP theorem states that a distributed database system has to make a tradeoff between Consistency and Availability when a Partition occurs.

- That leaves a decision between the other two, C and A. When a network failure happens, one can choose to guarantee consistency or availability:
- High consistency comes at the cost of lower availability.
- High availability comes at the cost of lower consistency.

Non-relational database

- Non-relational databases (often called NoSQL databases) are different from traditional relational databases in that they store their data in a non-tabular form.
- Instead, non-relational databases might be based on data structures like documents. A document can be highly detailed while containing a range of different types of information in different formats.

- There are several advantages to using non-relational databases, including:
- **Massive dataset organization**
  In the age of Big Data, non-relational databases can not only store massive quantities of information, but they can also query these datasets with ease. Scale and speed are crucial advantages of non-relational databases.

- **Flexible database expansion**
  Data is not static. As more information is collected, a non-relational database can absorb these new data points, enriching the existing database with new levels of granular value even if they don't fit the data types of previously existing information.
- **Multiple data structures**
- **Built for the cloud**

MongoDB

- MongoDB is an open-source document database and leading NoSQL database.
- MongoDB works on concept of collection and document.
- Rather than using the tables and fixed schemas of a relational database management system (RDBMS), MongoDB uses key-value storage in the collection of documents. It also supports a number of options for horizontal scaling in large, production environments.

- MongoDB is a NoSQL document database system that scales well horizontally and implements data storage through a key-value system.

MongoDB Sharding

- MongoDB achieves scaling through a technique known as "sharding". It is the process of writing data across different servers to distribute the read and write load and data storage requirements
- Sharding is the process of storing data records across multiple machines and it is MongoDB's approach to meeting the demands of data growth. As the size of the data increases, a single machine may not be sufficient to store the data nor provide an acceptable read and write throughput

- Sharding solves the problem with horizontal scaling. With sharding, you add more machines to support data growth and the demands of read and write operations.

MongoDB Replication

- Replica Sets are a great way to replicate MongoDB data across multiple servers and have the database automatically failover in case of server failure.

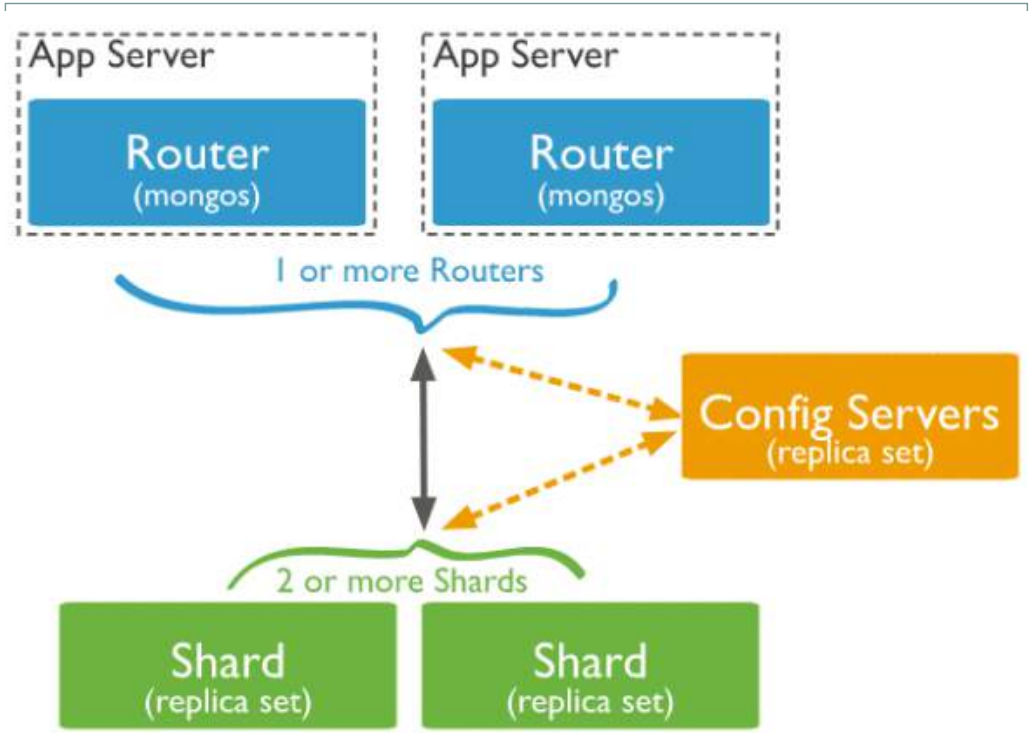
MongoDB sharding basics

- MongoDB sharding works by creating a cluster of MongoDB instances consisting of at least three servers. That means sharded clusters consist of three main components:
- The shard
- Mongos
- Config servers

Shard

- A shard is a single MongoDB instance that holds a subset of the sharded data. Shards can be deployed as replica sets to increase availability and provide redundancy. The combination of multiple shards creates a complete data set. For example, a 2 TB data set can be broken down into four shards, each containing 500 GB of data from the original data set.

- **Mongos**
- Mongos act as the query router providing a stable interface between the application and the sharded cluster. This MongoDB instance is responsible for routing the client requests to the correct shard.



- **Config Servers**
- Configuration servers store the metadata and the configuration settings for the whole cluster.

- The application communicates with the routers (mongos) about the query to be executed.
- The mongos instance consults the config servers to check which shard contains the required data set to send the query to that shard.
- Finally, the result of the query will be returned to the application.

HBase

- HBase is a column-oriented non-relational database management system that runs on top of Hadoop Distributed File System (HDFS). HBase provides a fault-tolerant way of storing sparse data sets, which are common in many big data use cases. It is well suited for real-time data processing or random read/write access to large volumes of data.

- Unlike relational database systems, HBase does not support a structured query language like SQL; in fact, HBase isn't a relational data store at all. HBase applications are written in Java much like a typical Apache MapReduce application.

- HBase is a **column-oriented database** and the tables in it are sorted by row. The table schema defines only column families, which are the key value pairs. A table have multiple column families and each column family can have any number of columns. Subsequent column values are stored contiguously on the disk. Each cell value of the table has a timestamp

- in an HBase:
- Table is a collection of rows.
- Row is a collection of column families.
- Column family is a collection of columns.
- Column is a collection of key value pairs.

Base and RDBMS

HBase	RDBMS
HBase is schema-less, it doesn't have the concept of fixed columns schema; defines only column families.	An RDBMS is governed by its schema, which describes the whole structure of tables.
It is built for wide tables. HBase is horizontally scalable.	It is thin and built for small tables. Hard to scale.
No transactions are there in HBase.	RDBMS is transactional.
It has de-normalized data.	It will have normalized data.
It is good for semi-structured as well as structured data.	It is good for structured data.

- *Features of HBase*
- *HBase is linearly scalable.*
- *It has automatic failure support.*
- *It provides consistent read and writes.*
- *It integrates with Hadoop, both as a source and a destination.*
- *It has easy java API for client.*
- *It provides data replication across clusters*

Cassandra

- *Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world. It provides highly available service with no single point of failure.*

- *It is scalable, fault-tolerant, and consistent.*
- *It is a column-oriented database.*
- *Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.*
- *Created at Facebook, it differs sharply from relational database management systems.*
- *Cassandra implements a Dynamo-style replication model with no single point of failure, but adds a more powerful "column family" data model.*
- *Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, ebay, Twitter, Netflix, and more*

Features of Cassandra

- **Elastic scalability** – Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on architecture** – Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast linear-scale performance** – Cassandra is linearly scalable, i. e. , it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.

- **Flexible data storage** – Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy data distribution** – Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction support** – Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).
- **Fast writes** – Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

Components of Cassandra

- **Node** – It is the place where data is stored.
- **Data center** – It is a collection of related nodes.
- **Cluster** – A cluster is a component that contains one or more data centers.
- **Commit log** – The commit log is a crash-recovery mechanism in Cassandra. Every write operation is written to the commit log.

- **Mem-table** – A mem-table is a memory-resident data structure. After commit log, the data will be written to the mem-table. Sometimes, for a single-column family, there will be multiple mem-tables.
- **SSTable** – It is a disk file to which the data is flushed from the mem-table when its contents reach a threshold value.
- **Bloom filter** – These are nothing but quick, nondeterministic, algorithms for testing whether an element is a member of a set. It is a special kind of cache. Bloom filters are accessed after every query

Cassandra Query Language

- users can access Cassandra through its nodes using Cassandra Query Language (CQL). CQL treats the database **(Keyspace)** as a container of tables. Programmers use **cqlsh**: a prompt to work with CQL or separate application language drivers.

- **Write Operations**
- Every write activity of nodes is captured by the **commit logs** written in the nodes. Later the data will be captured and stored in the **mem-table**. Whenever the mem-table is full, data will be written into the **SStable** data file. All writes are automatically partitioned and replicated throughout the cluster. Cassandra periodically consolidates the SSTables, discarding unnecessary data.

- **Read Operations**
- During read operations, Cassandra gets values from the mem-table and checks the bloom filter to find the appropriate SSTable that holds the required data.

RDBMS	Cassandra
RDBMS deals with structured data.	Cassandra deals with unstructured data.
It has a fixed schema.	Cassandra has a flexible schema.
In RDBMS, a table is an array of arrays. (ROW x COLUMN)	In Cassandra, a table is a list of "nested key-value pairs". (ROW x COLUMN key x COLUMN value)
Database is the outermost container that contains data corresponding to an application.	Keyspace is the outermost container that contains data corresponding to an application.
Tables are the entities of a database.	Tables or column families are the entity of a keyspace.
Row is an individual record in RDBMS.	Row is a unit of replication in Cassandra.
Column represents the attributes of a relation.	Column is a unit of storage in Cassandra.
RDBMS supports the concepts of foreign keys, joins.	Relationships are represented using collections.

Module5

XML

- XML stands for eXtensible Markup Language.
- XML was designed to store and transport data.
- XML is a software- and hardware-independent tool for storing and transporting data.
- **The Difference Between XML and HTML**
- XML and HTML were designed with different goals:
- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

- **XML Does Not Use Predefined Tags**
- The XML language has no predefined tags.
- **XML is Extensible**
- Most XML applications will work as expected even if new data is added (or removed).
- **XML Simplifies Things**
- It simplifies data sharing
- It simplifies data transport
- It simplifies platform changes
- It simplifies data availability

XML DTD

- DTD stands for Document Type Definition.
- A DTD defines the structure and the legal elements and attributes of an XML document.
- An XML document with correct syntax is called "Well Formed".
- An XML document validated against a DTD is both "Well Formed" and "Valid".

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

- *The DOCTYPE declaration above contains a reference to a DTD file.*
- *The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:*

Note.dtd:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

CTYPE note - Defines that the root element of the document is note

MENT note - Defines that the note element must contain the elements: "to, from, heading,

MENT to - Defines the to element to be of type "#PCDATA"

MENT from - Defines the from element to be of type "#PCDATA"

MENT heading - Defines the heading element to be of type "#PCDATA"

MENT body - Defines the body element to be of type "#PCDATA"

- *#PCDATA means parseable character data.*

XML Schema

- An XML Schema describes the structure of an XML document, just like a DTD.
- An XML document with correct syntax is called "Well Formed".
- An XML document validated against an XML Schema is both " Well Formed" and "Valid".

```
<xs:element name="note">

<xs:complexType>
  <xs:sequence>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="heading" type="xs:string"/>
    <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:element>
```

- <xs:element name="note"> defines the element called "note"
- <xs:complexType> the "note" element is a complex type
- <xs:sequence> the complex type is a sequence of elements
- <xs:element name="to" type="xs:string"> the element "to" is of type string (text)
- <xs:element name="from" type="xs:string"> the element "from" is of type string
- <xs:element name="heading" type="xs:string"> the element " heading" is of type string
- <xs:element name="body" type="xs:string"> the element "body" is of type string

- XML Schemas are written in XML
- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

XML Applications

- **Storing Data with Complex Structure**
- Many applications need to store data that are structured, but are not easily modeled as relations.
- XML-based representations are now widely used for storing documents, spreadsheet data and other data that are part of office application packages.
- XML is also used to represent data with complex structure that must be exchanged between different parts of an application.

- **Standardized Data Exchange Formats**
- XML-based standards for representation of data have been developed for a variety of specialized applications, ranging from business applications such as banking and shipping to scientific applications such as chemistry and molecular biology

- **Web Services**
- When the information is to be used directly by a human, organizations provide Web-based forms, where users can input values and get back desired information in HTML form. However, there are many applications where such information needs to be accessed by software programs, rather than by end users. Providing the results of a query in XML form is a clear requirement. In addition, it makes sense to specify the input values to the query also in XML format.