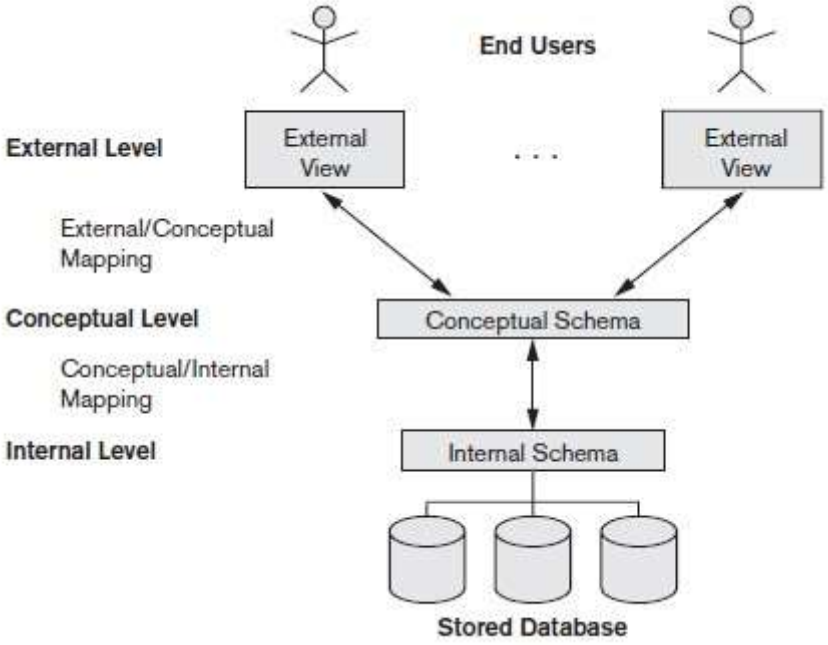
Databases

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.

The Three-Schema Architecture

1. The **internal level** has an internal schema, which describes the physical storage structure of the database. The lowest level of abstraction describes how the data are actually stored.
2. The **conceptual level** has a conceptual schema, describes what data are stored in the database, and what relationships exist among those data. **Database administrators**, who must decide what information to keep in the database, use the logical level.
3. The **external** or **view level** includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group.



Data Independence

The three-schema architecture can be used to further explain the concept of data independence, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well.

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data. A **data-definition language (DDL)** is a language for specifying the database schema and as well as other properties of the data.

ACID Properties:

Transactions access data using two operations:

- **read(X)**, which transfers the data item X from the database to a variable, also called X, in a buffer in main memory belonging to the transaction that executed the read operation.
- **write(X)**, which transfers the value in the variable X in the main-memory buffer of the transaction that executed the write to the data item X in the database.

For now, however, we shall assume that the write operation updates the database immediately. Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as:

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Consistency:-

The consistency requirement here is that **the sum of A and B be unchanged** by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the **application programmer** who codes the transaction.

Atomicity:-

Suppose that, just before the execution of transaction Ti, the values of accounts A and B are $1000 and $2000, respectively. Now suppose that, during the execution of transaction Ti , a failure occurs that prevents Ti from completing its execution successfully. Further, suppose that the failure happened after the write(A) operation but before the write(B) operation. In this case, the values of accounts A and B reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum A + B is no longer preserved.
Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. The system must at some point be in an inconsistent state. Even if transaction Ti is executed to completion, there exists a point at which the value of account Ais $950 and the value of account B is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is $950, and the value of account B is $2050.
If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

Ensuring atomicity is the responsibility of the **database system**; specifically, it is handled by a component of the database called the **recovery system**.

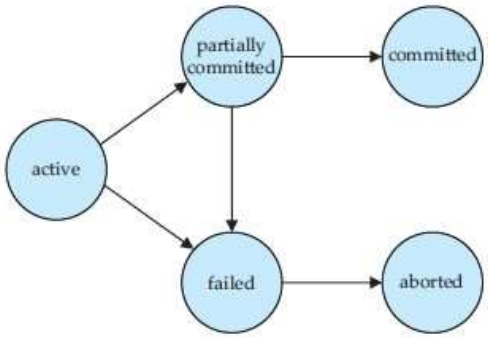
Durability:-

The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.
The **recovery system** of the database, is responsible for ensuring durability, in addition to ensuring atomicity.

Isolation:-
Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. The execution of a transaction should not be interfered with by any other transactions executing concurrently.
Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**.

Note: Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.

- Active, the initial state; the transaction stays in this state while it is executing.
- Partially committed, after the final statement has been executed.
- Failed, after the discovery that normal execution can no longer proceed.
- Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- Committed, after successful completion.



Note: A transaction is said to have **terminated** if it has either **committed** or **aborted**.

Introduction to the Relational Model

A **relational database** consists of a collection of tables, each of which is assigned a unique name. 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.

Relation Instance: specific instance of a relation, i.e. containing a specific set of rows.

Domain: For each attribute of a relation, there is a set of permitted values. Thus, the domain of the salary attribute of the instructor relation is the set of all possible salary values. We require that, 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: is the logical design of the database.
Database Instance: is a snapshot of the data in the database at a given instant in time.

Super key: A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.

Candidate Key: A superkey may contain extraneous attributes, **minimal superkeys** are called candidate keys.

Primary Key: A candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.
Note: The P.K should be chosen such that its attributes are **never or very rarely changed**.

Schema Diagram: A pictorial depiction of the schema of a database that shows the relations in the database, their attributes, and primary keys and foreign keys.

Relational Query Languages: Define a set of operations that operate on tables, and output tables as their results.

Referential Integrity Constraint: A referential integrity constraint requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation. This is the concept of Foreign Key, where the relation which has Primary Key is called referenced relation while the relation contains F.K is called referencing relation.

DATABASE DESIGN &THE E-R MODEL

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.
An entity may be **concrete**, such as a person or a book, or it may be **abstract**, such as a course, a course offering, or a flight reservation.
An **entity set** is a set of entities of the same type that share the same properties, or attributes**. For example:** the entity set student might represent the set of all students in the university.

An entity is represented by a set of **attributes**. Each entity may have its own value for each attribute. Possible attributes of the instructor entity set are ID, name, dept_name, and salary, each entity has a value for each of its attributes.

Note: A database thus includes a collection of entity sets, each of which contains any number of entities of the same type.

Relationship Sets

A **relationship** is an association among several entities. For example, we can define a relationship advisor that associates instructor Katz with student Shankar. This relationship specifies that Katz 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 entity sets. If E1, E2,……,En

are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2, \ldots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

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

The association between entity sets is referred to as participation; that is, the entity sets E1, E2,… En **participate** in relationship set R. A relationship instance in an E-R schema represents an association between the named entities.

When the entity sets of a relationship set are not distinct; that is, the same entity set participates in a relationship set more than once, in different roles. In this type of relationship set, sometimes called a **recursive relationship set**.

Note: **A relationship may also have attributes called "descriptive attributes"**.
A relationship instance in a given relationship set must be uniquely identifiable from its participating entities, without using the descriptive attributes.

Binary Relationship Set: one that involves two entity sets.

Attributes:

For each attribute, there is a set of permitted values, called the **domain**, or **value set**, of that attribute. The domain of attribute course id might be the set of all text strings of a certain length.

a. **Simple** attributes can't be divided into sub-parts.
b. **Composite** attributes can be divided into sub-parts. Name can could be structured as a composite attribute consisting of first_name, middle_name, last_name
c. **Single-valued** single value for a particular.
d. **Multivalued** attribute phone_number may have zero, one or several phone numbers
e. **Derived** value of this type of attributes can be derived from the values of other related attributes or entities.

Note: An attribute takes a **null** value when an entity does not have a value for it. The null value may indicate "not applicable" or "not known".

Mapping Cardinalities

Mapping cardinalities, or **cardinality ratios**, express the number of entities to which another entity can be associated via a relationship set.

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.
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.

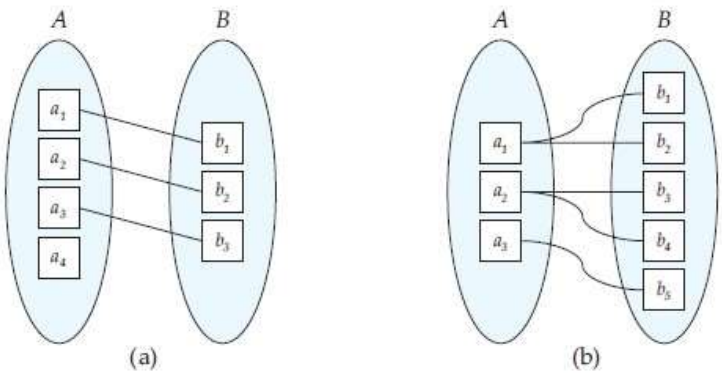


Figure 7.5 Mapping cardinalities. (a) One-to-one. (b) One-to-many.

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.

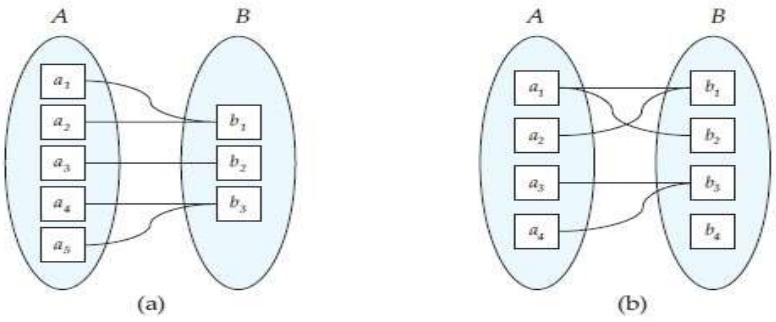


Figure 7.6 Mapping cardinalities. (a) Many-to-one. (b) Many-to-many.

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.

Keys

the values of the attribute values of an entity must be such that they can uniquely identify the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes.

Let R be a relationship set involving entity sets E1, E2, . . . , En. Let primarykey(Ei) denote the set of attributes that forms the primary key for entity set Ei . Assume for now that the attribute names of all primary keys are unique. The composition of the primary key for a relationship set depends on the set of attributes associated with the relationship set R. If the relationship set R has no attributes associated with it, then the set of attributes.

$$\textbf{primary-key(E1)} \cup \textbf{primary-key(E2)} \cup \cdots \cup \textbf{primary-key(En)}$$

If the relationship set R has attributes a1, a2, . . . am, associated with it, then the set of attributes

$$\textbf{primary-key(E1)} \cup \textbf{primary-key(E2)} \cup \quad \cup \cdots \textbf{primary-key(En)} \cup \{\textbf{a1, a2, . . . , am}\}$$

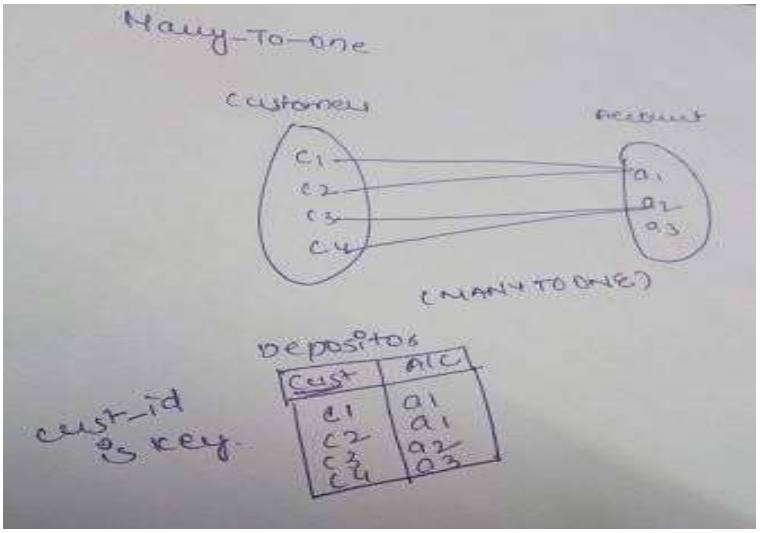
In both of the above cases, the set of attributes **primary-key(E1) ∪ primary-key(E2) ∪ ⋯ ∪ primary-key(En)**
forms a superkey for the relationship set.

Note: If the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them. The name of the entity set combined with the name of the attribute would form a unique name.

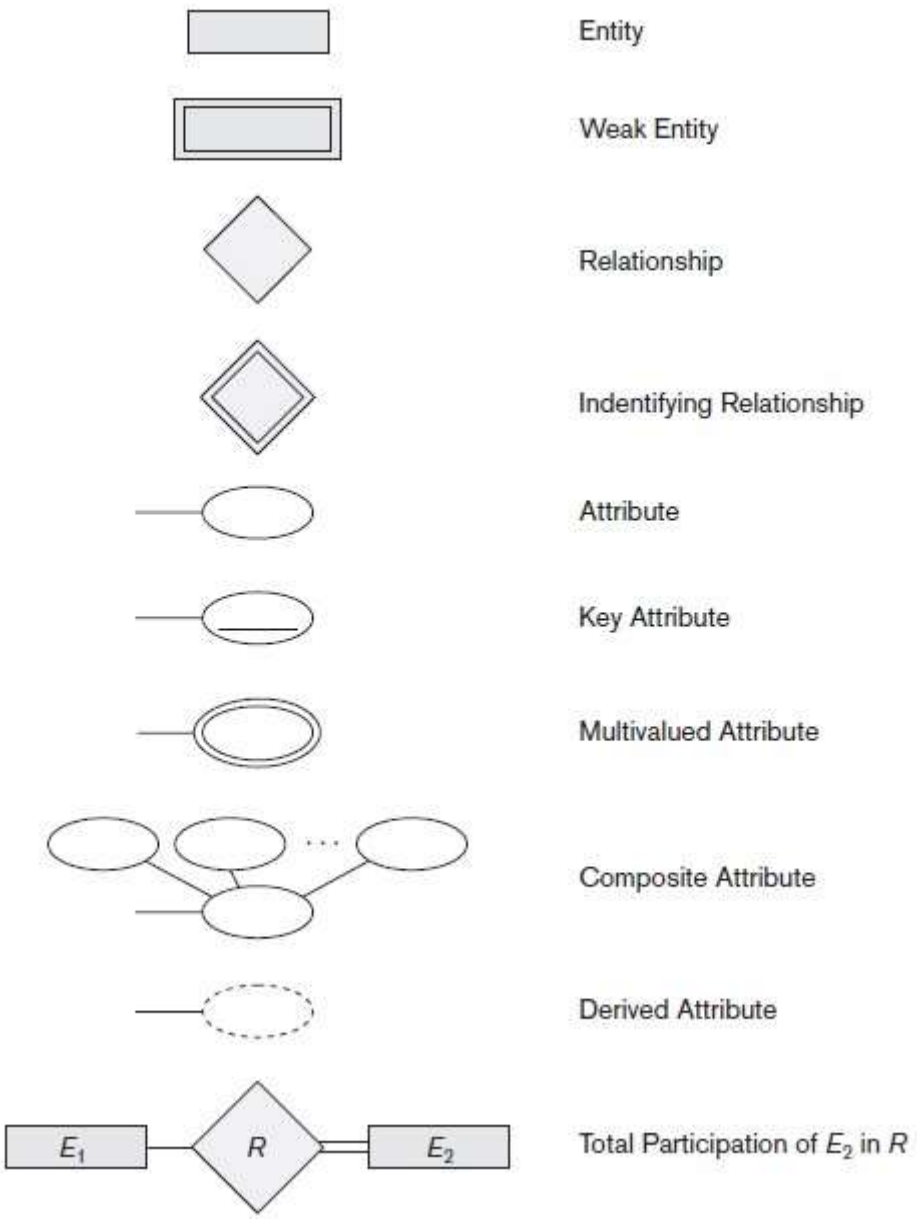
Note: **The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set**.

Consider the entity sets **customer** and **account** and the relationship set **depositor** with attribute access_date.

1. Suppose the relationship set is **many-to-many**. Then the primary key of depositor consist of the union of the primary keys of customer and account.
2. If a customer can have only one account, that is if the depositor relationship is **many-to-one** from customer to account, then primary key of depositor is simply the primary key of customer.



3. If the relationship is **one-to-many** from customer to account, it means an account can be owned by at most one customer, then the primary key of depositor is simply the primary key of account.
4. For **one-to-one** relationship either primary can be used.



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**.

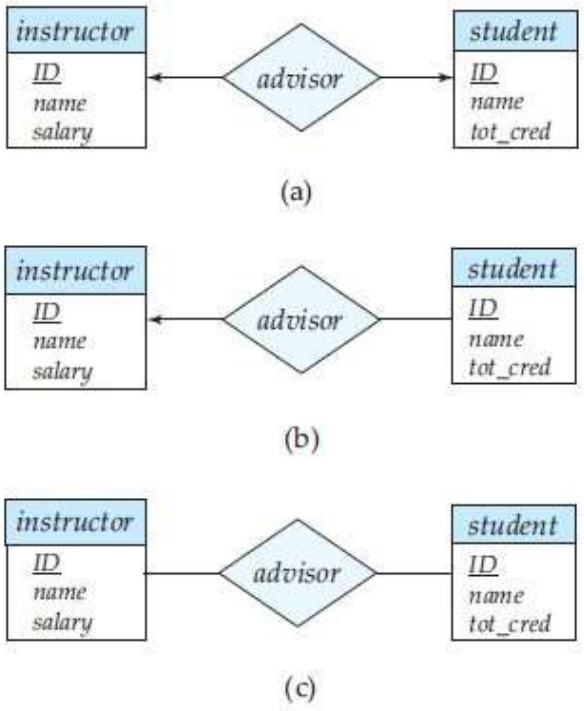
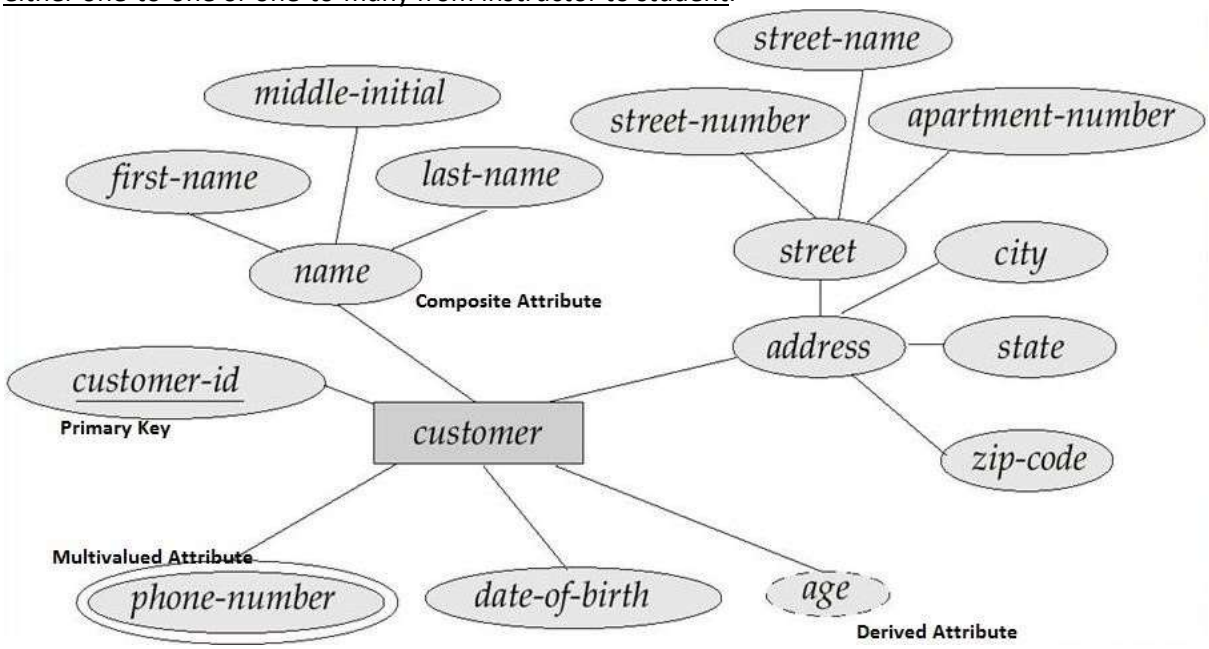


Figure 7.9 Relationships. (a) One-to-one. (b) One-to-many. (c) Many-to-many.

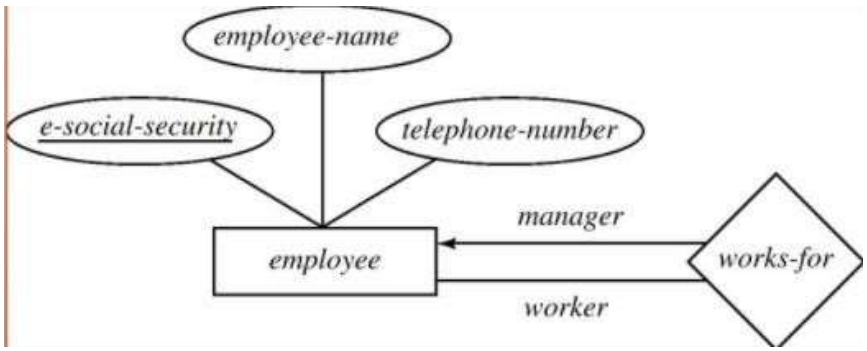
Note: A directed line from relationship set advisor to the entity set instructor specifies that advisor is either one-to-one or one-to-many from instructor to student.



E-R Diagram with composite, multivalued, and derived attributes

Roles:

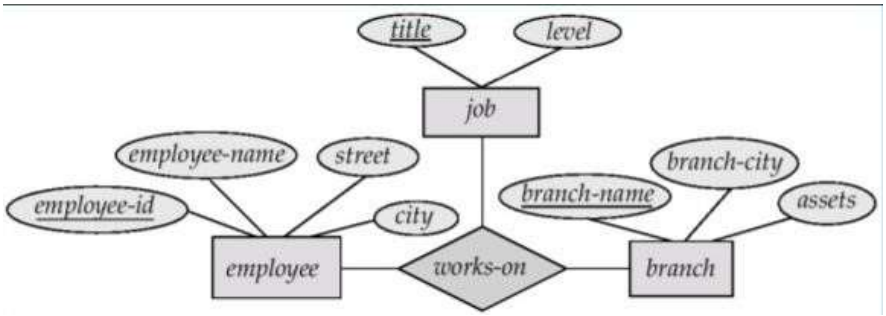
We indicate roles in E-R diagrams by labelling the lines that connect diamonds to rectangles. Below figure shows the role indicators **manager** and **worker** between the **employee** entity set and **works_for** relationship set.



E-R Diagram with roles indicator

Non-Binary Relationship Sets

We can specify some types of **many-to-one** relationships in the case of non-binary relationship sets.



Binary versus n-ary Relationship Sets

Relationships in databases are often binary. Some relationships that appear to be non-binary could actually be better represented by several binary relationships.

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

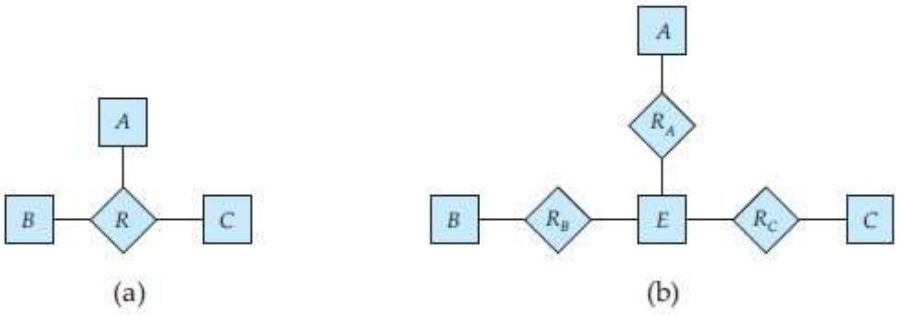


Figure 7.19 Ternary relationship versus three binary relationships.

Consider relationship set R, relating entity sets A, B, and C. We replace the relationship set R by an entity set E, and create three relationship sets as shown above.

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

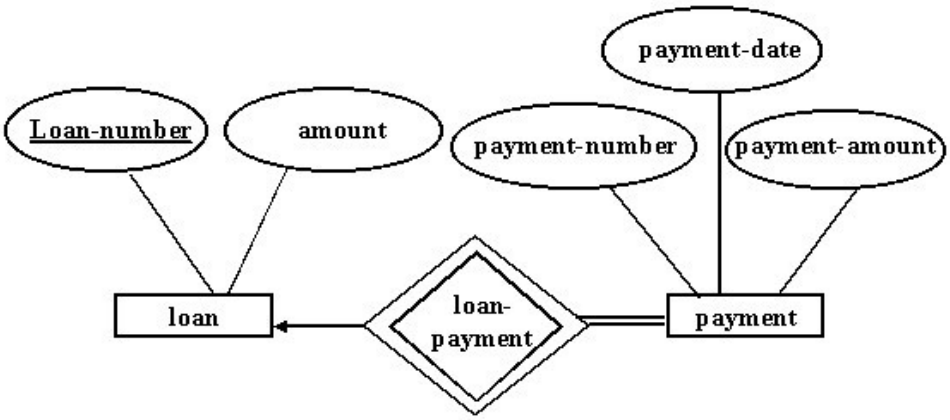
If the relationship set R had any attributes, these are assigned to entity set E; n each of the three new relationship sets, we insert a relationship as follows:

- (e_i, a_i) in R_A.
- (e_i, b_i) in R_B.
- (e_i, c_i) in R_C.

Weak Entity Sets

An entity set may not have sufficient attributes to forma a Primary Key. Such an entity is termed as **weak entity set**. An entity set that has a primary key is termed as **strong entity set**.

Note: A weak entity set can participate in relationships other than the identifying relationship



The weak entity set **payment** depends on the strong entity set **loan** via the relationship set **loan_payment**.

The figure also illustrates the use of double lines to indicate total participation – the participation of the weak set payment in the relationship **loan_payment** is **total**.

Note: Weak relationship set will not have any descriptive attributes. Each payment is for single laon.

Relational Database Design

The goal of relational database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily.

If there were a schema (loan_number, amount), then loan_number is able to serve as the primary key." This rule is specified as a **functional dependency**. **loan_number ->amount**

Atomic Domains and First Normal Form

The E-R model allows entity sets and relationship sets to have substructures such as dependent_name and composite attributes such as address (with component attributes street and city). When we create tables from E-R designs that contain these types of attributes, we **eliminate this substructure**.

1. For composite attributes, we let each component be a **separate attribute** in relation.
2. For multivalued attributes, we create **one tuple for each item** in a multivalued set.

A domain is **atomic** if elements of the domain are considered to be indivisible units.

1. **First Normal Form (1NF)** A relation schema R is in 1NF if the domains of all attributes are atomic. For example, if the schema of a relation employee included an attribute children whose domain elements are sets of names, the schema would not be in first normal form, because there can be more than one children.
   Composite attributes, such as an attribute address with component attributes street, city, state, and zip also have nonatomic domains.

2. **second normal form (2NF)**

A functional dependency $\alpha \rightarrow \beta$ is called a partial dependency if there is a proper subset γ of α such that $\gamma \rightarrow \beta$. We say that β is *partially dependent* on α. A relation schema R is in second normal form (2NF) if each attribute A in R meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Note: where β is not a prime attribute

Decomposition Using Functional Dependencies

We use the notation r (R) to show that the schema is for relation r, with R denoting the set of attributes.

identify the values of certain attributes. Consider a relation schema $r(R)$, and let $\alpha \subseteq R$ and $\beta \subseteq R$.

- Given an instance of $r(R)$, we say that the instance satisfies the functional dependency $\alpha \rightarrow \beta$ if for all pairs of tuples t_1 and t_2 in the instance such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Note: we say that K is a superkey of r (R) if the functional dependency K→R holds on r (R).

Some functional dependencies are said to be **trivial** because they are satisfied by all relations.

1. A → A is satisfied by all relations involving attribute A.
2. AB → Ais satisfied by all relations involving attribute A.

Note: A functional dependency of the form X→Y is trivial if Y⊆X .

Given a schema r (A, B,C), if functional dependencies A→ B and B → C, hold on r , we can infer the functional dependency A→ C must also hold on r .

Boyce–Codd Normal Form

It eliminates all redundancy that can be discovered based on functional dependencies, there may be other types of redundancy remaining. A relation schema R is in BCNF with respect

to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

BCNF Decomposition:

We now state a general rule for decomposing that are not in BCNF. Let R be a schema that is not in BCNF. Then there is at least one nontrivial functional dependency $\alpha \rightarrow \beta$ such that α is not a superkey for R. We replace R in our design with two schemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

In the case of *inst_dept* above, $\alpha = dept\_name$, $\beta = \{building, budget\}$, and *inst_dept* is replaced by

- $(\alpha \cup \beta) = (dept\_name, building, budget)$
- $(R - (\beta - \alpha)) = (ID, name, dept\_name, salary)$

Consider the following example of a relational schema that is not in BCNF:

Inst_dept (ID, name, salary, dept_name, building, budget)

The functional dependency **dept name → budget** holds on inst_dept, but dept_name is not a superkey (because, a department may have a number of different instructors)

In the case of *inst_dept* above, $\alpha = dept\_name$, $\beta = \{building, budget\}$, and *inst_dept* is replaced by

- $(\alpha \cup \beta) = (dept\_name, building, budget)$
- $(R - (\beta - \alpha)) = (ID, name, dept\_name, salary)$

We divide into two parts as:

Instructor (ID,name,Dept_name,salary)

ID is PK, and instructor is in BCNF because {ID→name, dept name, salary}

Department (dept_name,building,budget)

dept_name is PK, and Department is in BCNF because {dept name→building, budget}

Note:

1. **<u>α will be common in both relations to make decomposition lossless</u>**.
2. **<u>BCNF is not always dependency preserving</u>**

<u>Third Normal Form</u>

BCNF requires that all nontrivial dependencies be of the form α→β where α is a superkey.

A relation schema R is in third normal form with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.

- if α is not superkey of R then β must be prime attribute(subset of SK)

<u>Closure of a Set of Functional Dependencies</u>

- **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
- **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Canonical Cover(may not be unique)

A canonical cover F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.
- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

Note: 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.

Minimal Cover (From Navathe)

standard form. To satisfy these properties, we can formally define a set of functional dependencies F to be **minimal** if it satisfies the following conditions:

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency $X \rightarrow A$ in F with a dependency $Y \rightarrow A$, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

Note: Minimal cover is not unique as shown below:

If C is deleted, we get the set $F' = \{A \rightarrow B, B \rightarrow AC, \text{and } C \rightarrow AB\}$. Now, B is not extraneous in the side of $A \rightarrow B$ under F'. Continuing the algorithm, we find A and B are extraneous in the right-hand side of $C \rightarrow AB$, leading to two canonical covers

$$F_c = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$$
$$F_c = \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\}.$$

We are getting two minimal covers.

Lossless Decomposition:

Let r(R) be a relation schema, and let F be a set of functional dependencies on r (R). Let R1 and R2 form a decomposition of R. We say that the decomposition is a lossless decomposition if there is no loss of information by replacing r (R) with two relation schemas r1(R1) and r2(R2).

the same set of tuples as the result of the following SQL query:

```
select *
from (select R1 from r)
     natural join
     (select R2 from r)
```

This is stated more succinctly in the relational algebra as:

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

Note: A decomposition that is not a lossless decomposition is called a **lossy decomposition**. The terms **lossless-join decomposition** and **lossy-join decomposition** are sometimes used in place of lossless decomposition and lossy decomposition.

We can use functional dependencies to show when certain decompositions are lossless. Let R, R_1, R_2, and F be as above. R_1 and R_2 form a lossless decomposition of R if at least one of the following functional dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

Note: if R1 ∩ R2 forms a superkey of either R1 or R2, the decomposition of R is a lossless decomposition.

Dependency Preservation:

Let F be a set of functional dependencies on a schema R, and let R1, R2, . . . , Rn be a decomposition of R. The restriction of F to Ri is the set Fi of all functional dependencies in F+ that include only attributes of Ri.

The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' is a set of functional dependencies on schema R, but, in general, $F' \neq F$. However, even if $F' \neq F$, it may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F', and, if we verify that F' is satisfied, we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a dependency-preserving decomposition.

Note: To test dependency preserving it takes polynomial time unlike exponential time required to compute F+.

Multivalued Dependencies:

Functional dependencies rule out certain tuples from being in a relation. If A → B, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, **they require that other tuples of a certain form be present in the relation**.

For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tuple-generating dependencies**.

Let r (R) be a relation schema and let **α ⊆ R and β ⊆ R**. The multivalued dependency

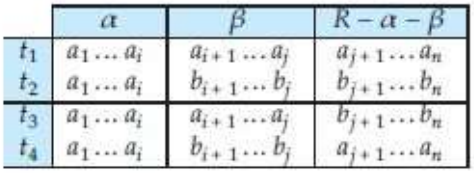
$$\alpha \rightarrow\rightarrow \beta$$

holds on R if, in any legal instance of relation r (R), **for all pairs of tuples t1 and t2 in r such that t1[α] = t2[α], there exist tuples t3 and t4 in r such that**

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

My own definition:

t1[α] = t2[α] = t3[α] = t4[α] t1[β] = t3[β] && t2[β] = t4[β] t1[R-β] = t4[R-β] && t2[R-β] = t3[R-β]



	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figure 8.13 Tabular representation of $\alpha \twoheadrightarrow \beta$.

Note: α→→β is trivial MVD on schema R if α→→β is satisfied by all relations on schema R. thus α→→β is trivial MVD if β ⊆α or α∪β = R.

For example: if R(ABCD) then AB→→CD is trivial MVD because AB U CD = R, and AC→→C is also a trivial MVD as C is subset of AC.

MVD Rules:

1. **Complementation Rule**:
   If α→→β then α→→ R − α − β
   If x→→ y then X→→R - (x U y) Eg: R (ABCD) if A→→B then A→→CD
2. **Trivial MVD:** α→→β is trivial MVD if β ⊆α or αUβ = R.

   Eg: R(AB) then A->B is non-trivial FD but A->->B is trivial MVD as AUB=R

3. **Transitivity**:
   If x→→ y && y→→ z then x→→ (z- y) [all attributes of z except y]

Eg:

a. if AB→→C and C→→DE then AB→→DE
b. if AB→→CD and if CD→→DE then if AB→→E

4. **Augmentation**
   if X→→Y && Z ⊇ W then XZ→→YW

   For eg:
   if A→→B then AC→→B or ACD→→BC or ACD→→BCD

5. **Not Allowed to Split:**
   A->BC then A->B, A->C is allowed, but
   A→→BC then A→→B and A→→C is not Allowed.

6. **Relation b/w FD and MVD:**
   Every FD is also MVD if A->B then A→→B

Fourth Normal Form:

A relation schema r (R) is in fourth normal form (4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow\rightarrow \beta$ is trivial MVD, OR
- α is superkey of R

Relational Algebra

A **query** language is a language in which a user requests information from the database. In a

Procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.

Nonprocedural language the user describes the desired information without giving a specific procedure for obtaining that information.

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, natural join, division and assignment**.

Note:

1. **The relational algebra is a procedural query language.**

2. **The select, project, and rename operations are called unary operations.**

The Select Operation

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection.

$$\sigma_{dept\_name = \text{“Physics”}} (instructor)$$

$$\sigma_{dept\_name = \text{“Physics”} \wedge salary > 90000} (instructor)$$

The selection predicate may include comparisons between two attributes.

$$\sigma_{dept\_name = building}(department)$$

The Project Operation

The **project** operation allows us to produce this relation with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter **pi (π)**.

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

Note: the result of a relational operation is itself a relation.

$$\Pi_{name}\ (\sigma_{dept\_name = \text{“Physics”}}\ (instructor))$$

The Union Operation

Consider a query to find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both.

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year = 2009}\ (section)) \cup$$
$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year = 2010}\ (section))$$

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*

We require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same, for all i.

Note: Note that r and s can be either database relations or temporary relations that are the result of relational-algebra expressions.

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.

We can find all the courses taught in the Fall 2009 semester but not in Spring 2010 semester by writing:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year = 2009}\ (section))\ -$$
$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year = 2010}\ (section))$$

Note: Set differences are taken between compatible relations.

The Cartesian-Product Operation:

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

Note: Relation is by definition a subset of a Cartesian product of a set of domains.

For example, the relation schema for **r = instructor × teaches** is:
(instructor.ID, instructor.name, instructor.dept name, instructor.salary, teaches.ID, teaches.course id, teaches.sec id, teaches.semester, teaches.year)

In general, if we have relations r1(R1) and r2(R2), then r1 × r2 is a relation whose schema is the concatenation of R1 and R2. Relation R contains all tuples t for which there is a tuple t1 in r1 and a tuple t2 in r2 for which t[R1] = t1[R1] and t[R2] = t2[R2].

Since we only want the names of all instructors in the Physics department together with the course id of all courses they taught, we do a projection:

$$\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}(\sigma_{dept\_name = \text{“Physics”}}(instructor\ \times\ teaches)))$$

Note that there is often more than one way to write a query in relational algebra. Consider the following query:

$$\Pi_{name,\ course\_id}\ (\sigma_{instructor.ID = teaches.ID}((\sigma_{dept\_name = \text{“Physics”}}(instructor)) \times\ teaches))$$

The selection that restricts dept name to Physics is applied to instructor, and the Cartesian product is applied subsequently; however, the two queries are **equivalent**.

The Rename Operation:

The rename operator, denoted by the lowercase Greek letter rho (ρ),
Given a relational-algebra expression E, the expression **ρ x (E)**
Returns the result of expression E under the name x.
We can also apply the rename operation to a relation r to get the same relation under a new name.
A second form of the rename operation is as follows: Assume that a relational algebra expression E has arity n. Then, the expression

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

Returns the result of expression E under the name x, and with the attributes renamed to $A_1, A_2 \ldots A_n$.

We consider the query **“Find the highest salary in the university**.” Our strategy is to (1) compute first a temporary relation consisting of those salaries that are not the largest and (2) take the set difference between the relation π salary (instructor) and the temporary relation just computed, to obtain the result.

$$\Pi_{instructor.salary}\ (\sigma_{instructor.salary\ <\ d.salary}\ (instructor\ \times\ \rho_d\ (instructor)))$$

The result contains all salaries except the largest one.

The query to find the largest salary in the university can be written as:

$$\Pi_{salary}(instructor) - \Pi_{instructor.salary}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_d(instructor)))$$

Instructor table has been renamed as d.

The rename operation is not strictly required, since it is possible to use a **positional notation** for attributes. Where $1, $2, . . . refer to the first attribute, the second attribute, and so on.

$$\Pi_{\$4}(\sigma_{\$4 < \$8}(instructor \times instructor))$$

Formal Definition of the Relational Algebra

A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

Note: A constant relation is written by listing its tuples within { }, for example { (22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000) }.

Let E1 and E2 be relational-algebra expressions. Then, the following are all relational-algebra expressions, note that all are basic operations:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, where P is a predicate on attributes in E_1
- $\Pi_S(E_1)$, where S is a list consisting of some of the attributes in E_1
- $\rho_x(E_1)$, where x is the new name for the result of E_1

Additional Relational-Algebra Operations

The fundamental operations of the relational algebra are sufficient to express any relational-algebra query, but we define additional operations that do not add any power to the algebra, but simplify common queries.

1. The Set-Intersection Operation

Suppose that we wish to find the set of all courses taught in both the Fall 2009 and the Spring 2010 semesters.

$$\Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year = 2009}(section)) \cap \Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year = 2010}(section))$$

Which can be written using basic set difference operations: **r ∩ s = r − (r − s)**

2. The Natural-Join Operation

Consider the query "Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount".

Πcustomer_name,loan.loan_number,amount(ρborrower.loan_number=loan.loan_number(borrower X loan))

The **Natural join** is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.
The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

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

Note: Consider two relation schemas R and S we can denote those attribute names that appear in both R and S by R ∩ S, and denote those attribute names that appear in R, in S, or in both by R ∪ S. Similarly, those attribute names that appear in R but not S are denoted by R – S. Note that the union, intersection, and difference operations here are on sets of attributes, rather than on relations.

We are now ready for a formal definition of the natural join. Consider two relations $r(R)$ and $s(S)$. The natural join of r and s, denoted by $r \bowtie s$, is a relation on schema $R \cup S$ formally defined as follows:

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

where $R \cap S = \{A_1, A_2, \ldots, A_n\}$.

Note: Please note that if r (R) and s(S) are relations without any attributes in common, that is, R ∩ S = ∅, then r ⋈ s = r × s (No attributes are same in R and S)

$$\Pi_{name,title}\ (\sigma_{dept\_name = \text{"Comp. Sci."}}\ (instructor \bowtie teaches \bowtie course))$$

The natural join is associative

3. Theta join

The theta join operation **is a variant of the natural-join operation** that allows us to combine a selection and a Cartesian product into a single operation.
If we want to combine tuples from two relations where the combination condition is not simply the equality of shared attributes then it is convenient to have a more general form of join operator, which is the θ-join **θ is a binary relational operator in the set {<, ≤, =, >, ≥}** v is a value constant, and R and S are relations. The θ-join is a binary operator that is written as $R \underset{a\,\theta\,b}{\bowtie} S$ or $R \underset{a\,\theta\,v}{\bowtie} S$

Consider relations r (R) and s(S), and let **θ** be a predicate on attributes in the schema R ∪ S. The theta join operation r ⋈θ s is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

Note: In case the operator θ is the equality operator (=) then this join is also called an "equijoin".

4. <u>The Assignment Operation</u>

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

This relation variable may be used in subsequent expressions. It is, however, a convenient way to express complex queries.

5. <u>Outer join Operations</u>

The outer-join operation is an extension of the join operation to deal with missing information.

> We can use the *outer-join* operation to avoid this loss of information. There are actually three forms of the operation: *left outer join*, denoted ⟕; *right outer join*, denoted ⟖; and *full outer join*, denoted ⟗. All three forms of outer join compute the join, and add extra tuples to the result of the join. For example, the

<u>Note:</u>

1. The left outer join takes all tuples in the l<u>eft relation that did not match with any tuple in the right relation</u>.
2. Note **<u>that the outer-join operations can be expressed by the basic relational-algebra</u>** operations.

6. <u>The Division Operation</u> The division operation denoted by ÷ is suited to queries that include the phrase **<u>"for all"</u>**. Suppose we wish to find all customers who have an account at **all** the branches located at Brooklyn. We can obtain all branches in the Brooklyn by the expression. **r1 =**

Πbrnach_name(σbranch_city="Brooklyn" (branch))

We can find all customer_name, branch_name pairs for which the customer has an account at branch by writing:

R2 = Πcustomer_name,brnach_name(depositor ⋈ Branch)

Now we need to find customers who appear in r2 with every branch name in r1. The operation that provides exactly those customers is the divide operation.

Πcustomer_name,brnach_name(depositor ⋈ Branch)

÷ Πbrnach_name(σbranch_city="Brooklyn" (branch))

<u>Note:</u> Basic Algebra doesn't support aggregate functions, & arithmetic operations.

Formally r(R) and s(S) be relations, and <u>let S is subset of R</u>; that is every attribute of schema S is also in schema R, the relation r ÷ s is a relation on schema R – S(that is, on the schema containing all attributes of schema R that are not in S).

$$r \div s \text{ is defined as } \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

Modifications of Database:

1. **Deletion** we can delete only whole tuples, we cannot delete values on only particular attributes.

r <-- r – E Where

r is a relation and E is a relational-algebra query.

a. Delete all of smith's account records: **depositor ← depositor –**

σcustomer_name='smith'(depositor)

2. **Insertion** tuples inserted must be of the correct arity. **r <-- r U E**

Where r is a relation and E is a relational-algebra expression. E be a constant relation containing one tuple.

Account ← account U {A-973, "Perryridge", 1200}

3. **Update**

Account ← **Π**account_number,branch_name,balance*1.05(**σ**balance>1000 (account))

Increase salary by 5% whose salary is more than 10000.

classroom(*building*, *room_number*, *capacity*)
department(*dept_name*, *building*, *budget*)
course(*course_id*, *title*, *dept_name*, *credits*)
instructor(*ID*, *name*, *dept_name*, *salary*)
section(*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
student(*ID*, *name*, *dept_name*, *tot_cred*)
takes(*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)
advisor(*s_ID*, *i_ID*)
time_slot(*time_slot_id*, *day*, *start_time*, *end_time*)
prereq(*course_id*, *prereq_id*)

University database

Tuple Relational Calculation(Non-Procedural)

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The tuple relational calculus, is a **nonprocedural query** language. Query in the tuple relational calculus is expressed as:

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

It is the set of all tuples t such that predicate P is true for t. **We use t[A] to denote the value of tuple t on attribute A**.

Que 1: Find the ID, name, dept name, salary for instructors (ID, name, dept name, salary) whose salary is greater than $80,000:

$$\{t \mid t \in \text{instructor} \wedge t[\text{salary}] > 80000\}$$

Suppose that we want only the ID attribute, rather than all attributes of the instructor relation. **Que 2: We need those tuples on (ID) such that there is a tuple in instructor with the salary attribute > 80000.**

To express this request, we need the construct "**there exists**" from mathematical logic.

$$\exists\ t \in r\ (Q(t))$$

Means **"there exists a tuple t in relation r such that predicate Q(t) is true."**

We can write the query "Find the instructor ID for each instructor with a salary greater than $80,000" as:

$$\{t \mid \exists\, s \in instructor\ (t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

"The set of all tuples t such that there exists a tuple s in relation instructor for which the values of t and s for the ID attribute are equal, and the value of s for the salary attribute is greater than $80,000."

Que 3: "Find the names of all instructors whose department is in the Watson building." This query involves two relations: instructor and department and it will require two "there exists" connected by and (∧).

$$\{t \mid \exists\, s \in instructor\ (t[name] = s[name] \wedge \exists\, u \in department\ (u[dept\_name] = s[dept\_name] \wedge u[building] = \text{"Watson"}))\}$$

Tuple variable u is restricted to departments that are located in the Watson building, while tuple variable s is restricted to instructors whose dept name matches that of tuple variable u.

Que 4: To find the set of all courses taught in the Fall 2009 semester, the Spring 2010 semester, or both.

We used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by or (∨):

$$\{t \mid \exists\, s \in section\ (t[course\_id] = s[course\_id] \wedge s[semester] = \text{"Fall"} \wedge s[year] = 2009)\} \vee \exists\, u \in section\ (u[course\_id] = t[course\_id]) \wedge u[semester] = \text{"Spring"} \wedge u[year] = 2010)\}$$

This expression gives us the set of all course id tuples for which at least one of the following holds:

- The course id appears in some tuple of the section relation with semester = Fall and year = 2009.
- The course id appears in some tuple of the section relation with semester = Spring and year = 2010.

If the same course is offered in both the Fall 2009 and Spring 2010 semesters, its course id appears only once in the result, because the mathematical definition of a set does not allow duplicate members.

Que 5: If we now want only those course id values for courses that are offered in both the Fall 2009 and Spring 2010 semesters.

We need to do is to change the or (∨) to and (∧) in Que 4 query.

Que 6: Find all the courses taught in the Fall 2009 semester but "not" in Spring 2010 semester."

Query is similar to the expressions of que 4, except for the use of the not (¬) symbol:

$$\{t \mid \exists s \in section\ (t[course\_id] = s[course\_id]) \\ \wedge s[semester] = \text{“Fall”} \wedge s[year] = 2009)\} \\ \wedge \neg \exists u \in section\ (u[course\_id] = t[course\_id])\} \\ \wedge u[semester] = \text{“Spring”} \wedge u[year] = 2010)\}$$

For ALL:

Que 7: Find all students who have taken all courses offered in the Biology department." To write this query in the tuple relational calculus, we introduce the "for all" construct, denoted by ∀. The notation:

$$\forall \mathbf{t} \in \mathbf{r}\ \mathbf{(Q(t))}$$

means **"Q is true for all tuples t in relation r.**

$$\{t \mid \exists r \in student\ (r[ID] = t[ID]) \wedge \\ (\forall u \in course\ (u[dept\_name] = \text{“Biology”} \Rightarrow \\ \exists s \in takes\ (t[ID] = s[ID] \\ \wedge s[course\_id] = u[course\_id]))\}$$

"The set of all students (that is, (ID) tuples t) such that, for all tuples u in the course relation, if the value of u on attribute dept name is 'Biology', then there exists a tuple **in the takes relation** that includes the student ID and the course id."

Note: without the condition

$$\exists\ \mathbf{r} \in \mathbf{student\ (r\ [ID] = t[ID])}$$

if there is no course offered in the Biology department, any value of t (including values that are not student IDs in the student relation) would qualify. **Formal Definition:**

A tuple-relational-calculus expression is of the form:

$$\mathbf{\{t \mid P(t)\}}$$

Where P is a formula. Several tuple variables may appear in a formula, A tuple variable is said to be a free variable unless it is quantified by a ∃ or ∀.

$$t \in instructor \wedge \exists s \in department(t[dept\_name] = s[dept\_name])$$ t

is a free variable. Tuple variable s is said to be a bound variable.

Safety of Expressions

A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression:

{t |¬ (t ∈ instructor)}

There are infinitely many tuples that are not in ***instructor***. Most of these tuples contain values that do not even appear in the database!

To help us define a restriction of the tuple relational calculus, we introduce the concept of the **domain** of a tuple relational formula, P. For example, dom(t ∈ instructor ∧ t[salary] > 80000) is the set containing 80000 as well as the set of all values appearing in any attribute of any tuple in the instructor relation We say that an expression {t | P(t)} is safe if all values that appear in the result are values from dom(P). The expression {t |¬ (t ∈ instructor)} is not safe. Note that dom(¬ (t ∈ instructor)) is the set of all values appearing in instructor.

Expressive Power of Languages

The tuple relational calculus restricted to safe expressions is equivalent in expressive power to the basic relational algebra but without the extended relational operations.

The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses domain variables that take on values from an attributes domain, rather than values for an entire tuple.

Formal Definition:

An expression in the domain relational calculus is of the form:

$$\{< x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

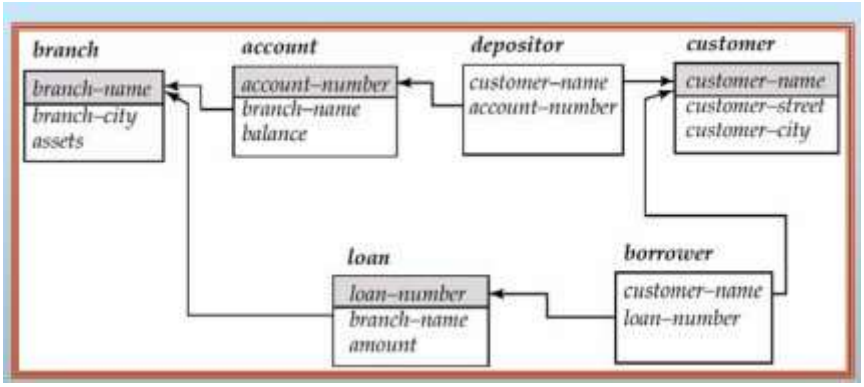
where x1, x2, . . . , xn represent domain variables. P represents a formula composed of atoms. An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- $x \ \Theta \ y$, where x and y are domain variables and Θ is a comparison operator $(<, \leq, =, \neq, >, \geq)$. We require that attributes x and y have domains that can be compared by Θ.

Que1: Find the instructor ID, name, dept name, and salary for instructors whose salary is greater than $80,000.

$$\{< i, n, d, s > \mid < i, n, d, s > \in instructor \wedge s > 80000\}$$

Que2: Find all instructor ID for instructors whose salary is greater than $80,000: {< i > | ∃ n, d, s (< i, n, d, s > ∈ instructor ∧ s > 80000)}



Que3: Find the loan-number, branch-name, and amount for loans of over $1200

$$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$$

Que4: Find all loan numbers for loans with amount greater than $1200.

$$\{< l \mid \exists\, b, a\ (<l, b, a> \in loan \wedge a > 1200)\}$$

Que5: Find the names of all customers who have a loan from the Perryridge branch and find the loan amount:

$$\{ <c,a> \mid \exists\, l\ (<c, l> \in borrower \wedge \exists\, b(<l, b, a> \in loan \wedge b=\text{"Perryridge"})$$

Que6: Find the names of all instructors in the Physics department together with the course id of all courses they teach:

$$\{< n, c > \mid \exists\, i, a\ (< i, c, a, s, y > \in teaches \wedge \exists\, d, s\ (< i, n, d, s > \in instructor \wedge d = \text{"Physics"}))\}$$

NULL Values

1. **ALL >=** Select * from R
   Where a>= ALL (Select A from R where 2=3);
   If in inner sub-query returns **empty**, ALL returns **true** and all records will be selected.
2. **IN**
   Select * from R where a IN (select A from R where 2=3) IN
   returns **FALSE** if inner query returns empty.
3. **NOT IN** returns **TRUE** and select all records.
4. **EXISTS** returns **FALSE** if inner query results EMPTY.
5. **NOT EXISTS** returns **TRUE** if inner query returns EMPTY.

A	B
10	20
30	40
NULL	50
40	60

6. **IN** if inner query contains a NULL value, IN ignores and works as normal **SQL>** select * from temp
   where B IN (select A from temp);
   O/P: 30,40
7. **NOT IN** returns **UNKNOWN** if inner query contains a NULL value; **SQL>** Select * from temp
   Where B NOT IN(Select A from Temp);
   O/P: Nothing
8. **ALL >=**
   If inner query holds a value which is NULL, the query returns EMPTY.
   SQL> select * from temp
   where B ALL>= (select A from temp);
   O/P: Nothing
9. **EXISTS/NOT EXISTS** ignore NULL and behave sane as with empty result in nested sub query or co-related subquery. **SQL>** Select R.B from temp R
   Where exists (Select * from temp where R.B = S.B)

Note: COUNT(*) returns 0 which is not a NULL value if no record exists.

Que:- find avg of all marks.

Name	Marks
A	20
B	40
C	60
D	NULL

Sol: AVG(Marks) = SUM(Marks)/Count(Marks) = 120/3 = 40

Note: SQL doesn't permit 2 attributes to have same name
Null by Aggregate Functions:

Sno	Marks
1	10
2	10
3	NULL
4	20
5	NULL
6	10

In aggregate if two tuples are the same on all grouping attributes. The operation places them in the same group **even if some of their attributes values are NULL**.

SQL> select count (Sno), marks from temp group by marks, the o/p will be:

Count(Sn0	Marks
3	10
1	20
2	NULL

Suppose all marks are NULL then o/p will be

Count(Sn0)	Marks
6	NULL

NULL:

True **AND** unknown = unknown
False **AND** unknown = False
Unknown **AND** unknown = unknown

True **OR** unknown = True
False **OR** unknown = unknown
Unknown **OR** unknown = unknown

Note:

1. <> ANY != NOT IN
2. <> ALL == NOT IN

< > ANY: if don't equal to any single element will come in result.

UNIQUE: The unique construct returns the value **TRUE** if the argument subquery contains no duplicate tuples.

Eg: Find all customers who have at most one account

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id and
                    R.year = 2009);
```

NOT UNIQUE: test the existence of duplicate values.

Note: the unique predicate would evaluate to true on the empty set.

Introduction to SQL

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all.

Note: The **char** data type stores fixed length strings. Consider, for example, an attribute A of type char(10). If we store a string "Avi" in this attribute, 7 spaces are appended to the string to make it 10 characters long. In contrast, if attribute B were of type varchar(10), and we store "Avi" in attribute B, no spaces would be added. When comparing two values of type char, if they are of different lengths extra spaces are automatically added to the shorter one to make them the same size, before comparison.

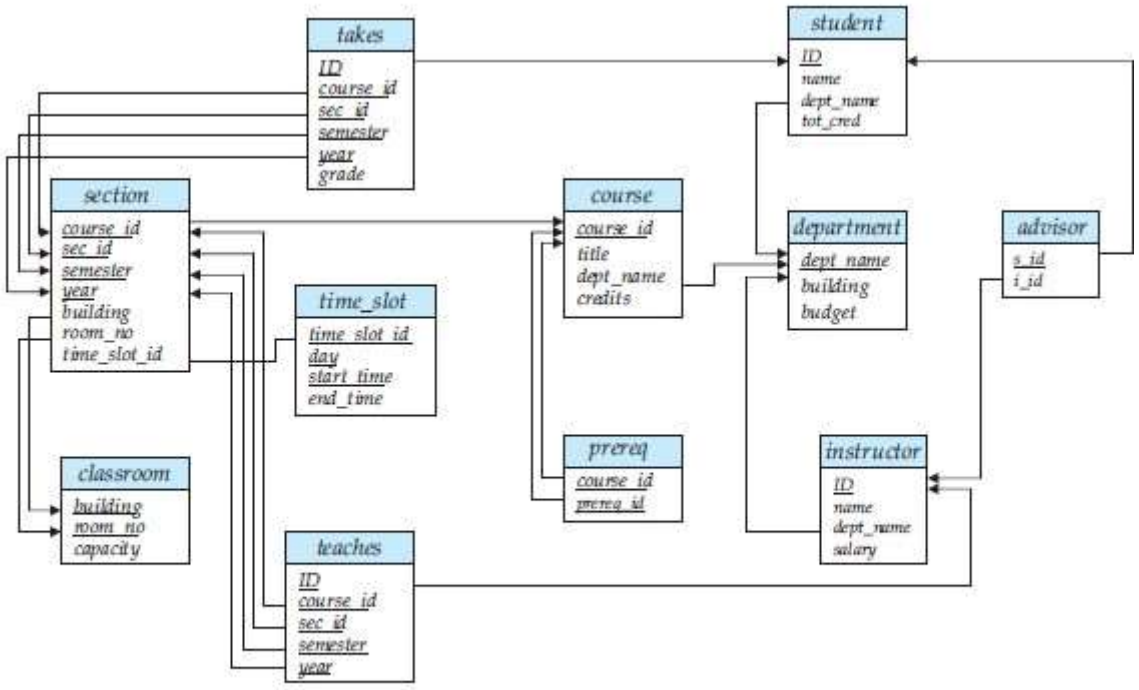


Figure 2.8 Schema diagram for the university database.

Natural Join:

The matching condition in the **from** clause most often requires all attributes with matching names to be equated.

The **natural join** operation operates on two relations and produces a relation as the result.

Note:

1. **We do not repeat those attributes that appear in the schemas of both relations**; rather they appear only once.
2. The order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

Query1: "For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught".

SQL> select name, course id from
instructor, teaches
where instructor.ID= teaches.ID;

This query can be written more concisely using the natural-join operation in SQL as:

SQL> select name, course id
from instructor natural join teaches;

Both of the above queries generate the same result.

from clause in an SQL query can have multiple relations combined using natural join, as shown here:

```
select A1, A2, ..., An
from r1 natural join r2 natural join ... natural join rm
where P;
```

Query2: List the names of instructors along with the titles of courses that they teach.

```
select name, title
from instructor natural join teaches, course
where teaches.course_id= course.course_id;
```

The natural join of instructor and teaches is first computed. And a Cartesian product of this result with course is computed, from which the where clause extracts only those tuples where the course identifier from the join result matches the course identifier from the course relation. Note that **teaches.course_id** in the where clause refers to the course id field of the natural join result.

In contrast the following SQL query does not compute the same result:

```
select name, title
from instructor natural join teaches natural join course;
```

To see why, note that the natural join of instructor and teaches contains the attributes (***ID, name, dept_name, salary, course_id, sec_id***), while the course relation contains the attributes (***course_id, title, dept_name, credits***).
The natural join of these two would require that the **dept_name** attribute values from the two inputs be the same, in addition to requiring that the **course_id** values be the same. The previous query, on the other hand, correctly outputs such pairs.

To provide the benefit of natural join while avoiding the danger of **equating attributes erroneously**, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated
Feature is illustrated by the following query:

SQL> Select name, title
from (instructor natural join teaches) join course using (course id);

The operation **join . . . using** requires a list of attribute names to be specified. Both inputs must have attributes with the specified names. Consider the operation r1 join r2 using (A1, A2). The operation is similar to r1 natural join r2, except that a pair of tuples t1 from r1 and t2 from r2 match if t1.A1 = t2.A1 and t1.A2 = t2.A2; even if r1 and r2 both have an attribute named A3, it is not required that t1.A3 = t2.A3.

Query3: "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

Observe that we could not use the notation **instructor.salary**, since it would not be clear which reference to instructor is intended.

String Operations

Note: The string "It's right" can be specified by "It"s right". Single quote is a part of string hence replaced by double quotes.

Pattern matching can be performed on strings, using the operator **like**. We describe patterns by using two special characters:

- **Percent (%):** The % character matches any substring.
- **Underscore (_):** The character matches any character

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa.

- 'Intro%' matches any string beginning with "Intro".
- '%Comp%' matches any string containing "Comp" as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
- '___' matches any string of exactly three characters.
- '___%' matches any string of at least three characters.

For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an **escape character**. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a **like** comparison using the **escape** keyword. To illustrate, consider the following patterns, which use a **backslash (\)** as the escape character:

- like 'ab\%cd%' escape '\' matches all strings beginning with "ab%cd".
- like 'ab\\cd%' escape '\' matches all strings beginning with "ab\cd".

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator.

Note:

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

can be re-written as:

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

Set Operations

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

The Union Operation

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

Note:-The union operation automatically eliminates duplicates, unlike the select clause.

If we want to retain all duplicates, we must write union all in place of union:

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

The Intersect Operation

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

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

The **intersect** operation automatically eliminates duplicates. If we want to retain all duplicates, we must write **intersect all** in place of intersect. The number of duplicate tuples that appear in the result is equal to **the minimum number of duplicates in both c1 and c2**.

The Except Operation

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

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

The except operation outputs all tuples from its first input that do not occur in the second input; that is, it performs **set difference**.

The operation automatically eliminates duplicates in the inputs before performing set difference. If we want to retain duplicates, we must write **except all in place of except**. The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies in c1 minus the number of duplicate copies in c2, provided that the difference is positive.

Null Values

The result of an arithmetic expression (involving, for example +, −, ∗, or /) is null if any of the input values is null.

Comparisons involving nulls:

Consider the comparison "**1 < null".** It would be wrong to say this is **true** since we do not know what the null value represents. But it would likewise be **wrong to claim this expression is false**; if we did, **"not (1 < null)"** would evaluate to **true**. SQL therefore treats as unknown the result of any comparison involving a null value (other than predicates is **null** and is **not null**)

and: The result of **true and unknown is unknown**, false **and** unknown is false, **while unknown and unknown is unknown**.
or: The result of **true or unknown is true**, **false or unknown is unknown**, while **unknown or unknown is unknown**.

not: The result of **not unknown is unknown**.

Note: If the where clause predicate evaluates to either false or unknown for a tuple, that tuple is not added to the result.

When a query uses the select **distinct** clause duplicate tuples must be eliminated. For this purpose, when comparing values of corresponding attributes from two tuples, the values are treated as **identical if either both are non-null and equal in value**, or **both are null**. Thus two copies of a tuple, such as **{('A',null), ('A',null)},** are treated **as being identical**, even if some of the attributes have a null value. Using the distinct clause then retains only one copy of such identical tuples. Note that the treatment of null above is different from the way nulls are treated in predicates, where a omparison **"null=null" would return unknown**, rather than true.

Aggregate Functions
Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value.

- Avg
- Min
- Max
- Sum
- Count

Note:

1. The input to sum and avg must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.
2. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression.
3. SQL does not allow the **use of distinct with count (*). It is legal to use distinct with max and min,** even though the result does not change.

Any attribute that is not present in the group by clause must appear only inside an aggregate function if it appears in the select clause, otherwise the query is treated as erroneous.

```
select dept_name, count (distinct ID) as instr_count
from instructor natural join teaches
where semester = 'Spring' and year = 2010
group by dept_name;
```

The Having Clause

A condition that applies to groups rather than to tuples. Find those departments where the average salary of the instructors is more than $42,000.

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

Note: any attribute that is present in the having clause without being aggregated must appear in the group by clause, otherwise the query is treated as erroneous.

The meaning of a query containing aggregation, group by, or having clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the from clause is first evaluated to get a relation.
2. If a where clause is present, the predicate in the where clause is applied on the result relation of the from clause.
3. Tuples satisfying the where predicate are then placed into groups by the group by clause if it is present. If the group by clause is absent, the entire set of tuples satisfying the where predicate is treated as being in one group.
4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.
5. The select clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query: "For each course section offered in 2009, find the average total credits (tot cred) of all students enrolled in the section, if the section had at least 2 students."

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Subqueries in the From Clause

Find the average instructors' salaries of those departments where the average salary is greater than $42,000.

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
      from instructor
      group by dept_name)
where avg_salary > 42000;
```

We can give the subquery result relation a name, and rename the attributes, using the as clause, as illustrated below.

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
      from instructor
      group by dept_name)
      as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

The subquery result relation is named **dept_avg**, with the attributes **dept_name** and **avg_salary**.

The with Clause

The **with clause** provides away of defining a temporary relation whose definition is available only to the query in which the with clause occurs.
Query1: Find those departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
    from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

The **with clause** defines the temporary relation max budget, which is used in the immediately following query.
Query2: find all departments where the total salary is greater than the average of the total salary at all departments.

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
    from instructor
    group by dept_name),
dept_total_avg(value) as
    (select avg(value)
    from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Scalar Subqueries

A subquery can be used in the **select** clause provided the subquery returns only one tuple containing a single attribute; such subqueries are called scalar subqueries.

```
select dept_name,
      (select count(*)
       from instructor
       where department.dept_name = instructor.dept_name)
      as num_instructors
from department;
```

The subquery in the above example is guaranteed to return only a single value since it has a **count(*)** aggregate without a **group by**.
Note: Scalar subqueries can occur in **select**, **where**, and **having** clauses.

Case Statement:

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```

Join Conditions

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a where clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression. select * from student join takes **on** student.ID= takes.ID;

Note: ON is working as **where** clause. Results same as the following query:

```
select * from student, takes
where student.ID= takes.ID;
```

Note:

1. The on condition can express any SQL predicate, and thus a join expressions using the **on** condition can express a richer class of join conditions than **natural join**.
2. A join expression with an **on** Condition can be replaced by an equivalent expression with **where** clause always.

Outer Joins

- The **left outer join** preserves tuples only in the relation named before (to the left of) the left outer join operation.
- The **right outer join** preserves tuples only in the relation named after (to the right of) the right outer join operation
- The **full outer** join preserves tuples in both relations
- **inner join** that do not preserve non-matched tuples

Join Types and Conditions

To distinguish normal joins from outer joins, normal joins are called inner joins in SQL. Inner keyword is optional.

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
    from student inner join takes using (ID);
```

Views:

Views can be defined with a with **check option** clause at the end of the view definition. Then, if a tuple inserted into the view does not satisfy the view's **where** clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the **where** clause conditions.

Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.

Examples of integrity constraints are:

- An instructor name cannot be null.
- No two instructors can have the same instructor ID.
- Every department name in the course relation must have a matching department name in the department relation.
- The budget of a department must be greater than $0.00.

Constraints on a Single Relation

The create table command may also include **integrity-constraint** statements. The allowed integrity constraints include:

- not null
- unique
- check(<predicate>)

The check Clause

When applied to a relation declaration, the clause **check(P)** specifies a predicate P that must be satisfied by every tuple in a relation.

```
create table section
   (course_id      varchar (8),
    sec_id         varchar (8),
    semester       varchar (6),
    year           numeric (4,0),
    building       varchar (15),
    room_number    varchar (7),
    time_slot_id   varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

Referential Integrity

We wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

```
create table course
    (...
    foreign key (dept_name) references department
                on delete cascade
                on update cascade,
    ...);
```

If a delete of a tuple in department results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the course relation.

If the constraint is violated: The referencing field (here, dept name) can be set to **null** (**by using set null in place of cascade**)

Default Values

```
create table student
    (ID          varchar (5),
     name        varchar (20) not null,
     dept_name   varchar (20),
     tot_cred    numeric (3,0) default 0,
     primary key (ID));
```

Learnings:

The E-R model allows entity sets and relationship sets to have attributes that have some degree of substructure. Specifically, it allows multivalued attributes such as

---

phone_number in Figure 7.11 and composite attributes (such as an attribute *address* with component attributes *street*, *city*, *state*, and *zip*). When we create tables from E-R designs that contain these types of attributes, we eliminate this substructure. For composite attributes, we let each component be an attribute in its own right. For multivalued attributes, we create one tuple for each item in a multivalued set.

In the relational model, we formalize this idea that attributes do not have any substructure. A domain is atomic if elements of the domain are considered to be

Tree Protocol

In the tree protocol, the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

Nested Loop Join (Korth)

- Number of records of *student*: $n_{student} = 5,000$.
- Number of blocks of *student*: $b_{student} = 100$.
- Number of records of *takes*: $n_{takes} = 10,000$.
- Number of blocks of *takes*: $b_{takes} = 400$.

Nested-loop join algorithm computes the **theta join** $r \bowtie_\theta s$, it basically consists of a pair of nested for loops.
Relation r is called the **outer relation** and relation s the **inner relation** of the join, since the loop for r encloses the loop for s.
The nested-loop join algorithm is **expensive**, since it examines every pair of tuples in the two relations. Consider the cost of the nested-loop join algorithm.

1. The number of pairs of tuples to be considered is **nr ∗ ns**, where nr denotes the number of tuples in r, and ns denotes the number of tuples in s. For each record in r, we have to perform a complete scan on s.
2. **In the worst case, the buffer can hold only one block of each relation**, and a total of **[nr ∗ bs + br]** block transfers would be required, where br and bs denote the number of blocks containing tuples of r and s, respectively.
3. We need only one seek for each scan on the inner relation s since it is read sequentially, and a total of **br seeks to read r**, leading to a total of **nr + br** seeks.
4. In the best case, there is enough space for both relations to fit simultaneously in memory, so each block would have to be read only once; hence, only **br + bs** block transfers would be required, **along with 2 seeks**.

Note:- If one of the relations fits entirely in main memory, it is beneficial to use that relation as the inner relation.

For nested loop example, consider **Student** as outer relation and **takes** as inner relation. We have to examine **5,000*10,000** pair of tuples.

Case 1:- Student is outer and takes is inner relation

In worst case,

1. no. of block transfers is **5000*400 + 100 = 2,000,100**
2. no. of seeks = **5000 + 100 = 51,00** (nr + br)

In best case, we can read both relations only once.

1. No. of block transfer is **100 + 400 = 500**
2. No. of seeks = **2**

Case 2:- Student is inner and takes is outer relation

1. No. of block transfer is = **10,000*100 + 400 =1,000,400**
2. No. of seeks = **10000 + 400 = 10,400** (ns + bs)

Block Nested-Loop Join:

If the buffer is too small to hold either relation entirely in memory, we can still obtain a major saving in block accesses if we process the relations on a per-block basis, rather than on a per-tuple basis. **Within each pair of blocks, every tuple in one block is paired with every tuple in the other block**, to generate all pairs of tuples. As before, all pairs of tuples that satisfy the join condition are added to the result.

If r is outer relation and s is inner relation, total number of block transfer required is:

In worst case,

1. **br*bs + br** where br and bs denote the number of blocks containing records of r and s, respectively.
2. Each scan of the inner relation requires **one seek**, and the scan of the outer relation requires **one seek per block**, leading to a total of **$2 * br$ seeks**.

In best case, where the inner relation fits in memory

1. **br + bs** block transfers
2. **2 seeks**

Example,

In worst case, $student \bowtie takes$

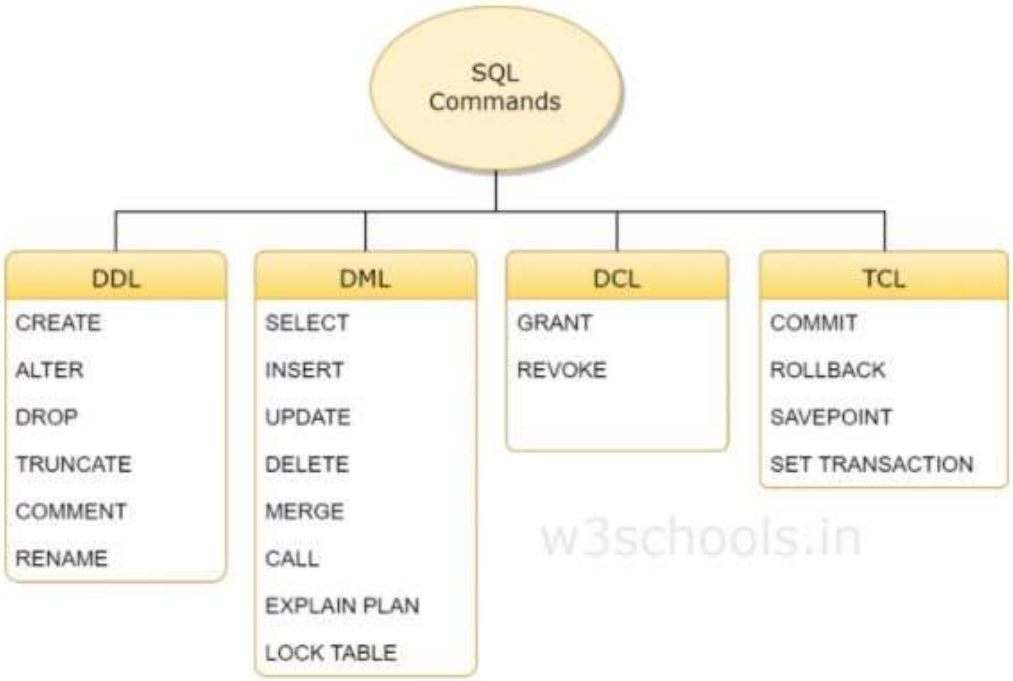
1. No. of block transfers = 100*400 + 100 = **40,100 blocks**
2. No. of seeks = 2*100 (2*br) = **200 seeks**

Best case remain same

1. No. of block transfer is **100 + 400 = 500**
2. No. of seeks = **2**

DDL/DML/DCL/TCL Commands

DDL: deals with database schemas and descriptions.



DML: deals with data manipulation.
DCL: deals with grants and permissions.
TCL: deals with transactions within a database.
_____ BTree/ B+ Tree_____Korth

Indexing and Hashing

Many queries reference only **a small proportion of the records** in a file. For example, a query like "Find all instructors in the Physics department" or "Find the total number of credits earned by the student with ID 22201" references only a fraction of the student records.
It is **inefficient** for the system to read every tuple in the instructor relation to check if the dept name value is "**Physics**".

Basic Concepts

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.

There are two basic kinds of indices:

- **Ordered indices** Based on a sorted ordering of the values. An index search uses the values of the search field itself.
- **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**. A hash directory search uses the binary hash value that is calculated by applying the **hash function to the search field**.

Search key an **attribute** or **set of attributes** used to look up records in a file is called a **search key**.

Ordered Indices

Each index structure is associated with a particular search key. **The records in the indexed file may themselves be stored in some sorted order**.

Types of Single-Level Ordered Indexes (Navathe)

1. Primary Indexes

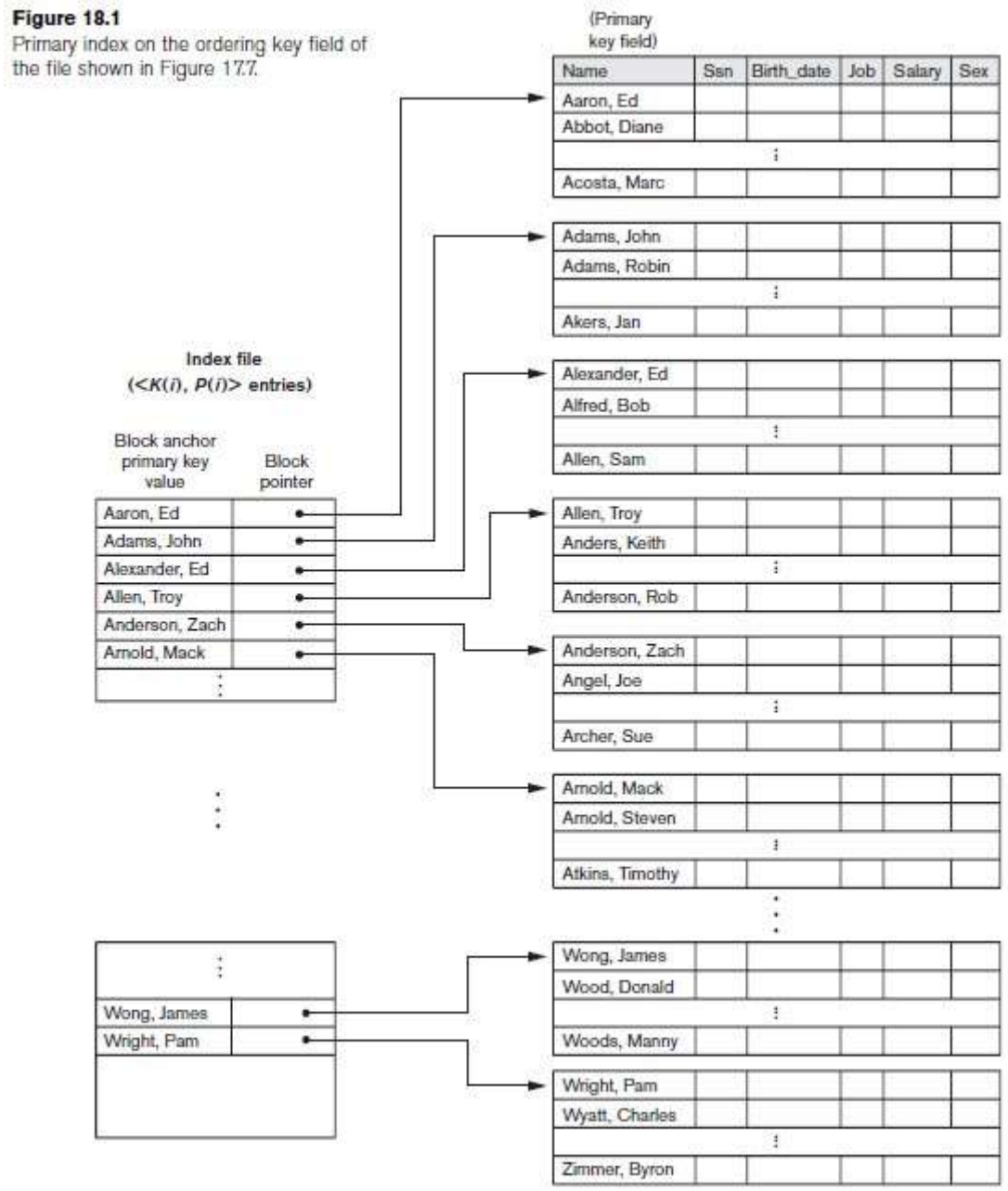
A primary index is specified on the ordering "key field" of an ordered file of records

An ordering key field is used to physically order the file records on disk, and every record has a **unique** value for that field.

A primary index is an **ordered file** whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the **second field is a pointer to a disk block** (a block address).

There is one index entry in the index file for each block in the data file. Each index entry has the value of the primary key field for the **first record in a block** and a pointer to that block as its two field values. We will refer to the two field values of index entry i as **<K(i), P(i)>**.

The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block.



Primary Index on ordering Key field

Note

1. A binary search on the index file requires fewer block accesses than a binary search on the data file.
2. Binary search for an ordered data file required $\log_2 b$ block access, if there are b data blocks.
3. If the primary index file contains only b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

A record whose primary key value is **K lies in the block whose address is P(i)**, where $K(i) \leq K < K(i+1)$ the ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. We do a **binary search on the index file** to find the appropriate index entry i, and then retrieve the data file block whose address is P(i).

Example 1. Suppose that we have an ordered file with $r = 30{,}000$ records stored on a disk with block size $B = 1024$ bytes. File records are of fixed size and are unspanned, with record length $R = 100$ bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (30000/10) \rceil = 3000$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 3000) \rceil = 12$ block accesses.

Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is $P = 6$ bytes long, and we have constructed a primary index for the file. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 45$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ block accesses. To search for a record using the index, we need one additional block access To **the data file for a total of 6 + 1 = 7 block accesses**.

Disadvantages

A major problem with a primary index—as with any ordered file—is **insertion** and **deletion** of records.

With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since **moving records will change the anchor records of some blocks**.

2. Clustering Indexes

If file records are **physically ordered on a nonkey field**—which does not have a distinct value for each record—that **field is called the clustering field** and the data file is called a **clustered file**.

A clustering index is another example of a **nondense** index because it has an entry for every distinct value of the indexing field.

We can create a different type of index, called a **clustering index**, to **speed up retrieval of all the records that have the same value for the clustering field**. This differs from a primary index, which requires that the ordering field of the data file have a distinct value for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the **clustering field of the data file**, and the second field is a **disk block pointer**.

1. There is **one entry in the clustering index for each distinct value** of the clustering field.
2. It contains **the value and a pointer to the first block in the data file** that has a record with that value for its clustering field.
3. Notice that record insertion and deletion still cause problems because the data records are physically ordered.

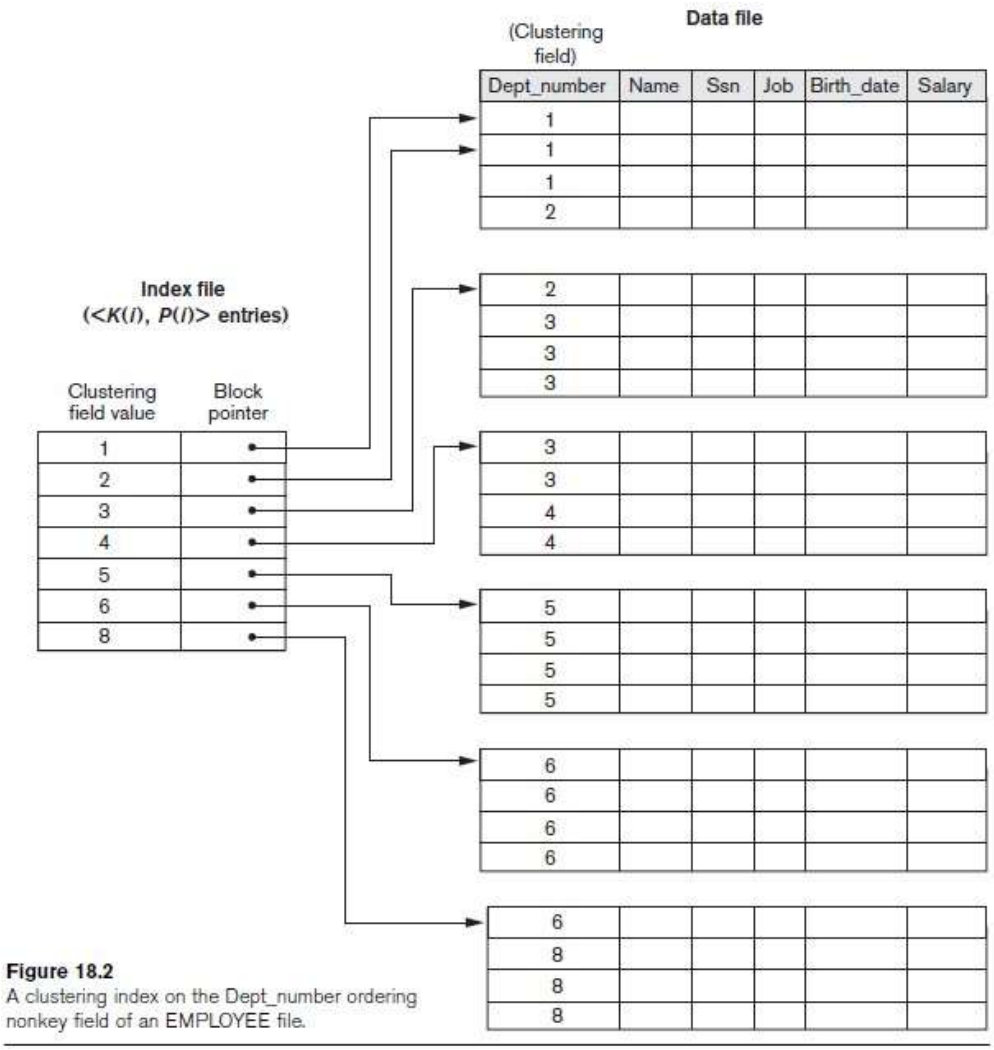


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Clustering Index non-key ordered field

3. Secondary Indexes

3.1 Secondary Index on Key field

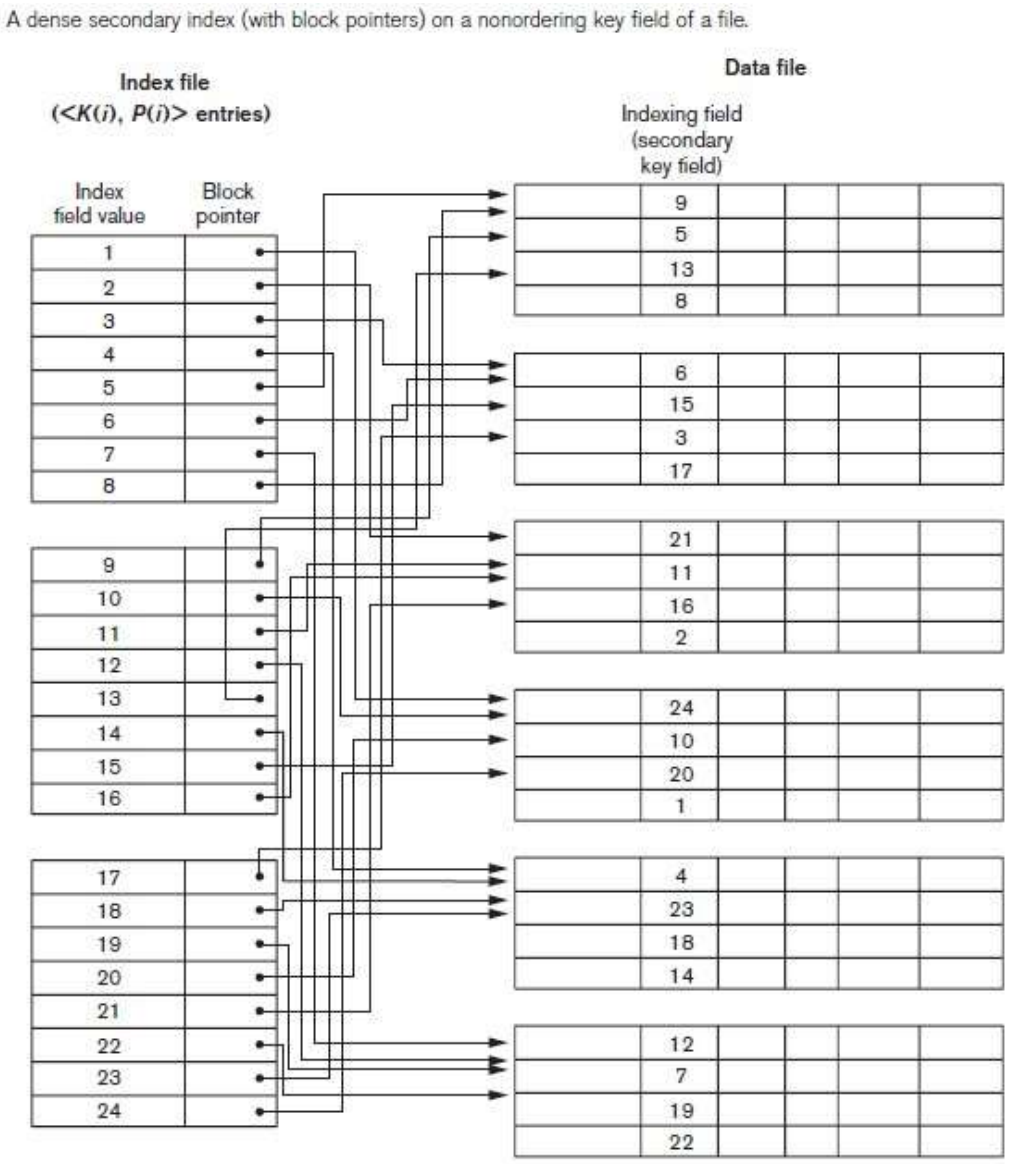
A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. **The data file records could be ordered, unordered, or hashed**.

The secondary index may be created on a field that is a **candidate key and has a unique value in every record, or on a nonkey field with duplicate values**.

The index is again an ordered file with two fields. The **first field** is of the same data type as some non-ordering field of the data file that is an indexing field. The **second field is either a block pointer or a record pointer**.
Many secondary indexes (and hence, indexing fields) can be created for the same file.

1. If a secondary index access structure is on a **key** (unique) field that has a distinct value for every record. Such a field is sometimes called a **secondary key**. **Such an index is dense**.

2. **P(i)** in the index **entries are block pointers**, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.



Dense Secondary Index on unordered key field

3. A **secondary index** usually needs more storage space and **longer search** time than does a **primary index**, because of its larger number of entries.
4. We would have to do a **linear search** on the data file if the secondary index did not exist.
5. For a primary index, we could still use a binary search on the main file, even if the index did not exist.

Example
Consider the file of Example 1 with r = 30,000 fixed-length records of size R = 100 bytes stored on a disk with block size B = 1024 bytes. The file has b = 3000 blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is V = 9 bytes long. Without the secondary index, to do a **linear search** on the file would require **b/2 = 3000/2 = 1500 block accesses on the average**. Suppose that we construct a secondary index on that nonordering key field of the file.
A block pointer is P = 6 bytes long, so each index entry is Ri = (9 + 6) = 15 bytes, and
the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 30,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 442$ blocks.
A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ block accesses.
To search for a record using the index, we need an additional block access to the data file for a total of **9 + 1 = 10 block accesses**.

3.2 Secondary index on non-key field

We can also create a **secondary index** on a **nonkey**, **nonordering** field of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

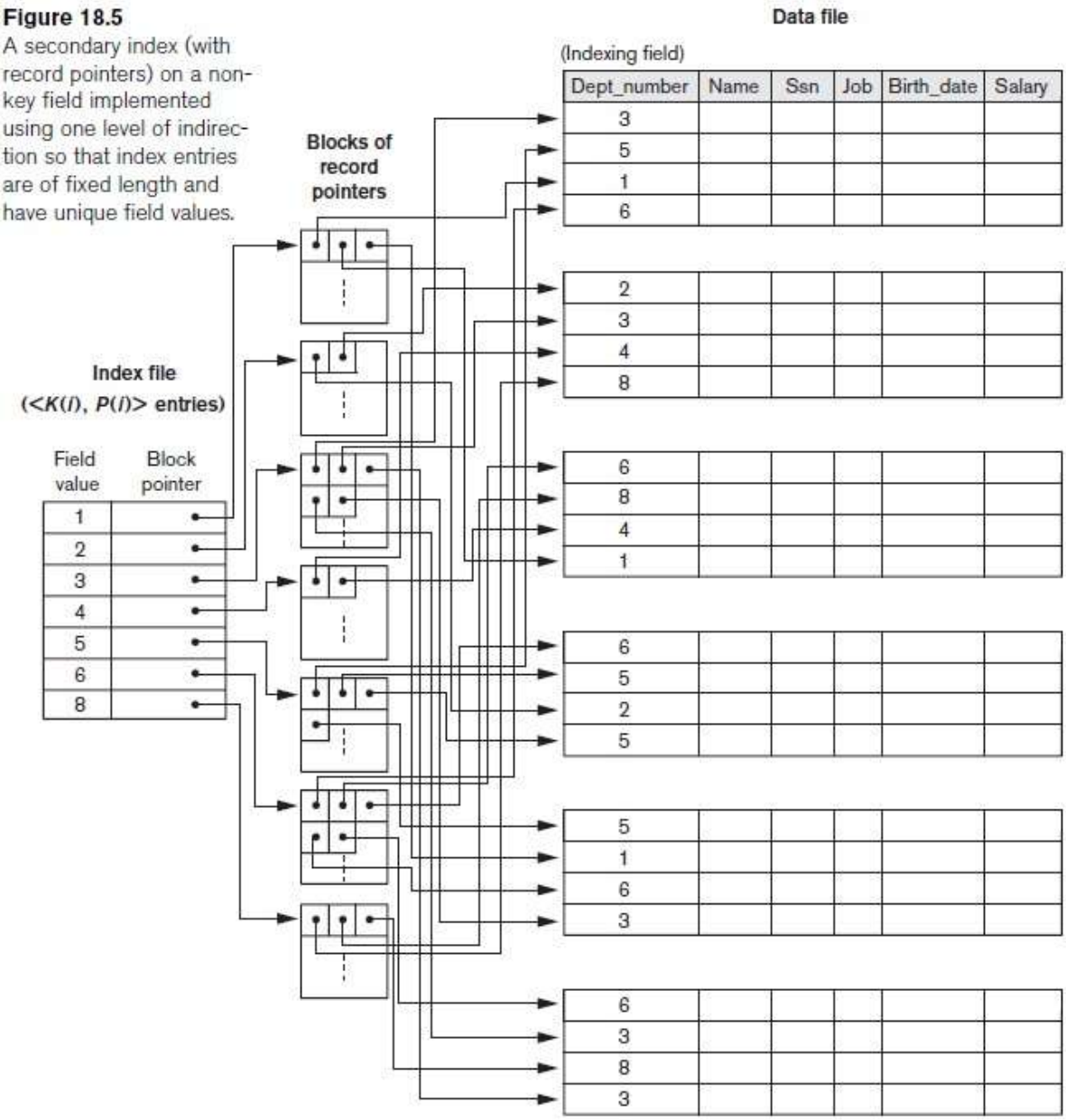
1. Option 1 is to **include duplicate index entries with the same K(i) value**—**one for each record**. This would be a **dense index**.
2. Option 2, which is more commonly used, is to keep the index entries themselves at a fixed length and **have a single entry for each index field value (nondense)**, but to create an extra level of indirection to handle the multiple pointers.

   In this **nondense scheme**, the pointer P(i) in index entry <K(i), P(i)> points to a disk block, **which contains a set of record pointers**; each record pointer in that disk block points to one of the data file records with value K(i) for the indexing field.
   If some value K(i) occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used.

Table 18.1 Types of Indexes Based on the Properties of the Indexing Field

	Index Field Used for Physical Ordering of the File	Index Field Not Used for Physical Ordering of the File
Indexing field is key	Primary index	Secondary index (Key)
Indexing field is nonkey	Clustering index	Secondary index (NonKey)

Figure 18.5
A secondary index (with record pointers) on a non-key field implemented using one level of indirection so that index entries are of fixed length and have unique field values.



NonDense Secondary Index on non-key unordered field

Table 18.2 Properties of Index Types

Type of Index	Number of (First-level) Index Entries	Dense or Nondense (Sparse)	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

Indices on Multiple Keys

A search key containing more than one attribute is referred to as a **composite search key**. The structure of the index is the same as that of any other index, the only difference being that the search key is not a single attribute, but rather is a list of attributes. The search key can be represented as a tuple of values, of the form (a1, . . . , an), where the indexed attributes are A1, . . . , An.
The ordering of search-key values is the **lexicographic ordering**. For example, for the case of **two attribute search keys**, (a1, a2) < (b1, b2) if either a1 < b1 or a1 = b1 and a2 < b2. Lexicographic ordering is basically the same as alphabetic ordering of words.

1. Using Multiple Single-Key Indices

Assume that the **instructor** file has **two indices**: one for **dept_name** and one for **salary**. Consider the following query: "Find all instructors in the Finance department with salary equal to $80,000." We write:

```
select ID
from instructor
where dept_name ='Finance' and salary= 80000;
```

There are three strategies possible for processing this query:

1. Use the index on dept_name to find all records pertaining to the Finance department. Examine each such record to see whether salary= 80000.
2. Use the index on salary to find all records pertaining to instructors with salary of $80,000. Examine each such record to see whether the department name is "Finance".
3. Use the index on dept_name to find pointers to all records pertaining to the Finance department. Also, use the index on salary to find pointers to all records pertaining to instructors with a salary of $80,000. **Take the intersection of these two sets of pointers**. Those pointers that are in the intersection point to records pertaining to instructors of the Finance department and with salary of $80,000.

The third strategy is the only one of the three that takes advantage of the existence of multiple indices. However, even this strategy may be a poor choice if all of the following hold:

- There are many records pertaining to the Finance department.
- There are many records pertaining to instructors with a salary of $80,000.
- There are only a few records pertaining to both the Finance department and instructors with a salary of $80,000.

If these conditions hold, we must scan a large number of pointers to produce a small result.

Note: - **An index structure called a "bitmap index" can in some cases greatly speed up the intersection operation used in the third strategy**.

2. Indices on Multiple Keys

An alternative strategy for this case is to create and use an index on a composite search key **(dept name, salary)**—that is, the search key consisting of the department name concatenated with the instructor salary.

1. We can use an ordered (B+-tree) index on the above composite search key to answer efficiently queries of the form (both conditions are with equal operator)

```
select ID
from instructor
where dept_name = 'Finance' and salary= 80000;
```

2. Queries such as the following query, which specifies **an equality condition on the first attribute** of the search key (dept_name) **and a range on the second attribute** of the search key (salary), can also **be handled efficiently** since they correspond to a range query on the search attribute.

```
select ID
from instructor
where dept_name = 'Finance' and salary< 80000;
```

3. We can even use an ordered index on the search key (**dept_name, salary**) to answer the following query on **only one attribute efficiently**

```
select ID
from instructor
where dept_name = 'Finance';
```

4. The use of an ordered-index structure on a composite search key, however, has a few shortcomings. As an illustration, consider the query

```
select ID
from instructor
where dept_name < 'Finance' and salary< 80000;
```

   We can answer this query by using an ordered index on the search key (dept_name, salary): For each value of dept_name that is less than "Finance" in alphabetic order, the system locates records with a salary value of 80000. However, e**ach record is likely to be in a different disk block, because of the ordering of records in the file, leading to many I/O operations**.

The difference between this query and the previous two queries is that the condition on the first attribute (dept_name) **is a comparison condition, rather than an equality condition**. The condition does not correspond to a range query on the search key.

Multilevel Indexes

A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) level of a multilevel index, as an ordered file with a distinct value for each K(i).
Therefore, by considering the first-level index file as a sorted data file, we can create a **primary index** for the first level; this index to the first level is called the second level of the multilevel index.
Because the second level is a primary index, we can use **block anchors** so that the second level has one entry for each block of the first level.

The blocking factor **bf** for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has **one field value and one block address**.

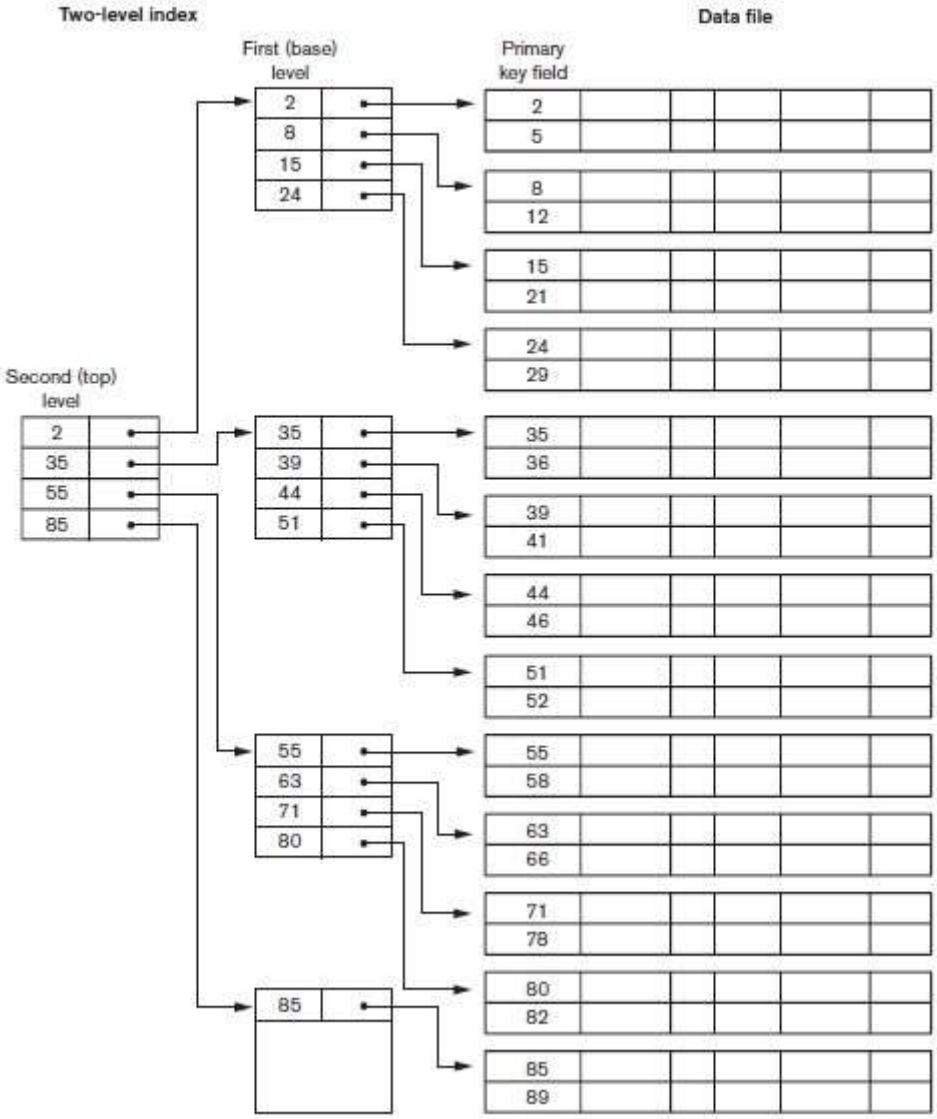
Note: -

1. When searching the index, **a single disk block is retrieved at each level**. Hence, t disk blocks are accessed for an index search, where **t is the number of index levels**.
2. The multilevel scheme described here can be used on any type of index—whether it is **primary**, **clustering**, or **secondary**—as long as the first-level index has **distinct values** for K(i) and **fixed-length entries**.

Example 3. Suppose that the dense secondary index of Example 2 is converted into a multilevel index. We calculated the index blocking factor $bfr_i = 68$ index entries per block, which is also the fan-out fo for the multilevel index; the number of first-level blocks $b_1 = 442$ blocks was also calculated. The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.

To access a record by searching the multilevel index, **we must access one block at each level plus one block from the data file**, so we need **t + 1 = 3 + 1 = 4 block accesses**.

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



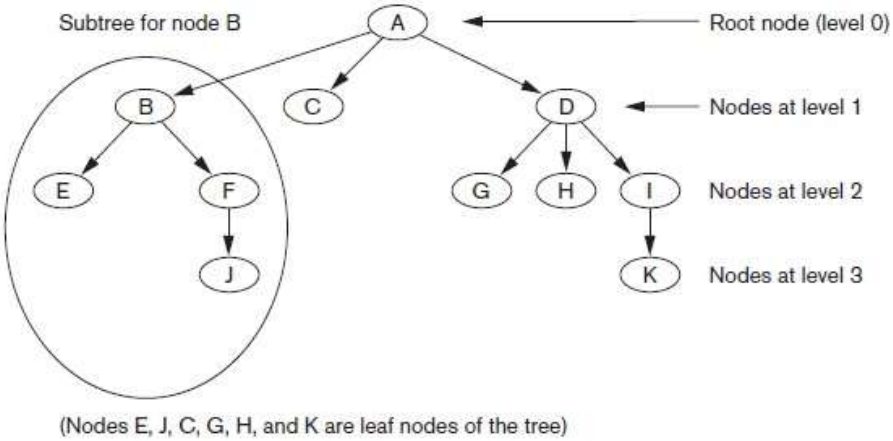
Multilevel Index: 2-level Primary Index

Dynamic Multilevel Indexes Using B-Trees and B+-Trees

Subtree

A precise recursive definition of a subtree is that it consists of a node n **(consider that node itself)** and the subtrees of all the child nodes of n.

In the following tree, since the **leaf nodes are at different levels of the tree**, this tree is called **unbalanced**.



A tree data structure that shows an unbalanced tree

Search Tree

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields.

A search tree is slightly different from a **multilevel index**. A **search tree** of order p is a tree such that each node contains **at most p − 1 search values and p pointers** in the order $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer).

B-Trees

The B-tree has additional constraints that ensure that **the tree is always balanced** and that the space wasted by deletion, if any, **never becomes excessive**. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints.

A B-tree of **order p**, when used as an access structure on a key field to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree is of the form

   $$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$$

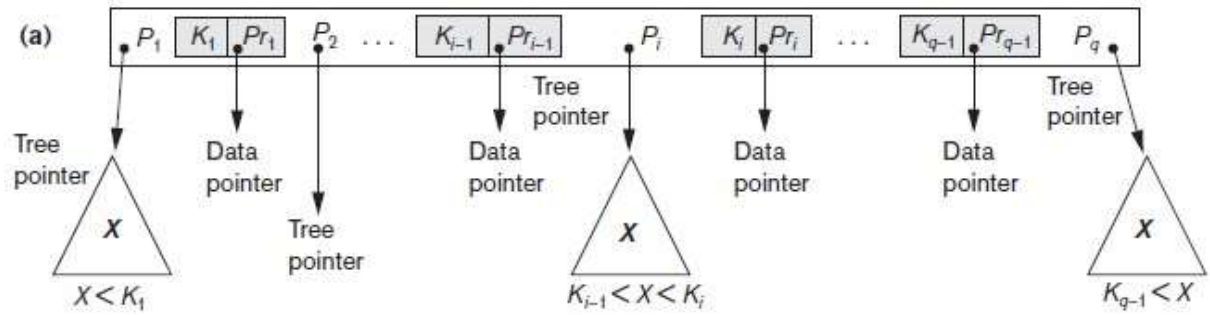
   where $q \leq p$. Each P_i is a **tree pointer**—a pointer to another node in the B-tree. Each Pr_i is a **data pointer**[8]—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).

2. Within each node, $K_1 < K_2 < ... < K_{q-1}$.

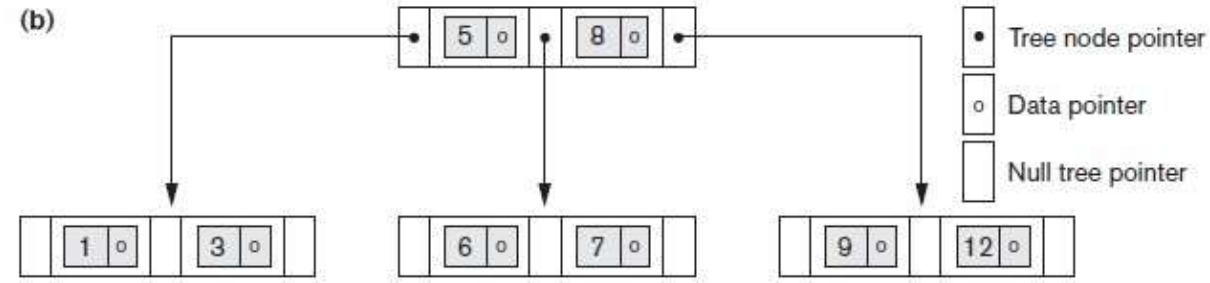
3. For all search key field values X in the subtree pointed at by Pi (the ith subtree, see Figure 18.10(a)), we have:

$$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q.$$

4. Each node has at most p tree pointers.



A node in a B-tree with q – 1 search values



A B-tree of order p = 3.The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

5. Each node, **except the root and leaf nodes, has at least** $\lceil (p/2) \rceil$ **tree pointers**. The root node has at least two tree pointers unless it is the only node in the tree.
6. A node with q tree pointers, q ≤ p, has q – 1 search key field values (and hence has **q – 1 data pointers**).
7. **All leaf nodes are at the same level**. **Leaf nodes have the same structure as internal nodes** except that all of their tree pointers Pi are NULL.

Note:

1. Notice that all **search values K in the B-tree are unique** because we assumed that the tree is used as an access structure on a **key field**.
2. If we use a **B-tree on a nonkey field**, we must change the definition of the file pointers Pri.
3. A **B-tree allows search-key values to appear only once** (if they are unique), unlike a B+-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node.

Insert into B Tree

1. A **B-tree starts with a single root node** (which is also a **leaf node**) at **level 0** (zero).
2. Once the root node is **full with p – 1 search key values** and we attempt to insert another entry in the tree, **the root node splits into two nodes at level 1**.
3. Only the **middle value is kept in the root node**, and **the rest of the values are split evenly between the other two nodes**.

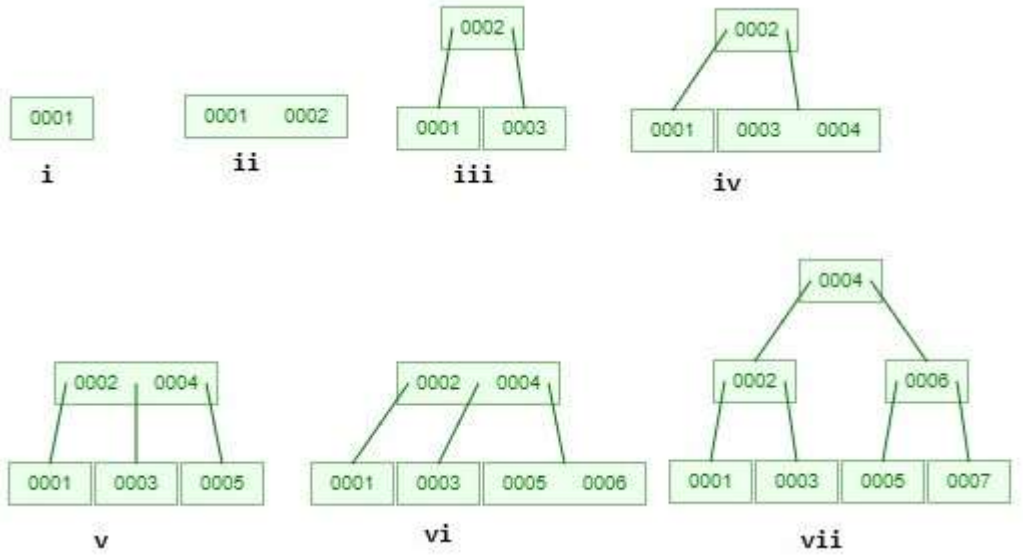
4. When a **nonroot node** is full and a new entry is inserted into it, that **node is split into two nodes at the same level**, and the **middle entry is moved to the parent node along with two pointers to the new split nodes**.
5. **If the parent node is full**, it is also split. **Splitting can propagate all the way to the root node**, **creating a new level if the root is split**.
6. Splitting of nodes may occur, so some insertions will take more time.

Example insert the following keys in a B Tree which has order = 3. 1,
2, 3, 4, 5, 6, 7

Solution

Order is 3, it means max keys per node allowed is 2. When keys are inserted 3 in a node, then split the node and send the middle key above.

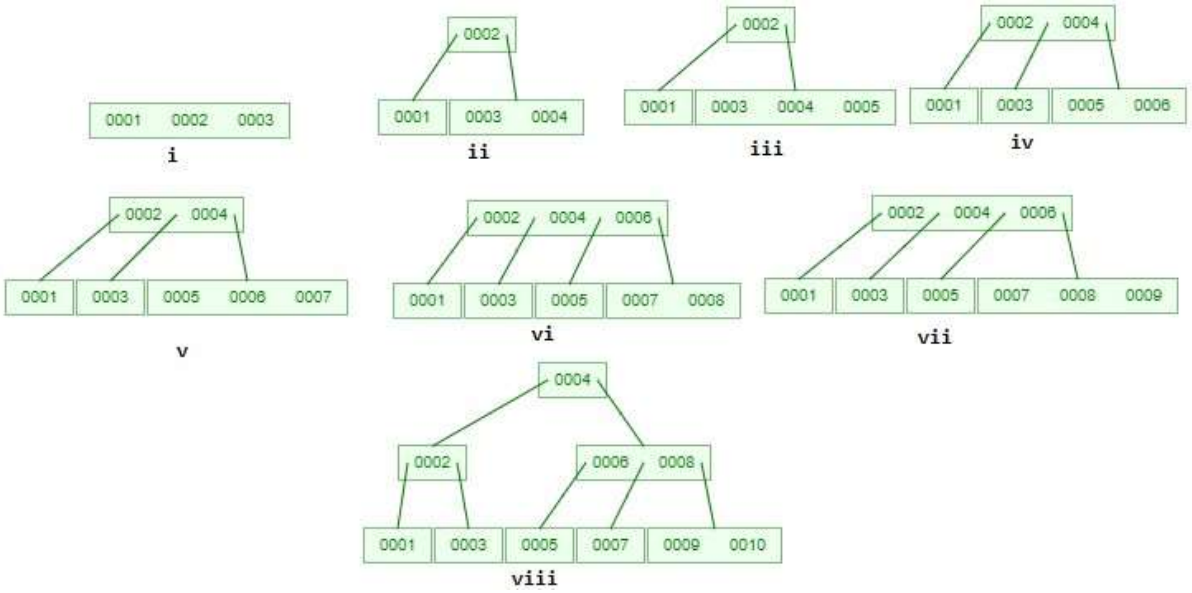
(1) Order = 3



(ii) Order = 4, max keys allowed is 3

Suppose our node already has |1|2|3| if we insert 4 then there will be node overflow. First split the node, send 2 to above and then insert the key 4.

Keys: 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10



Deletion from B Tree

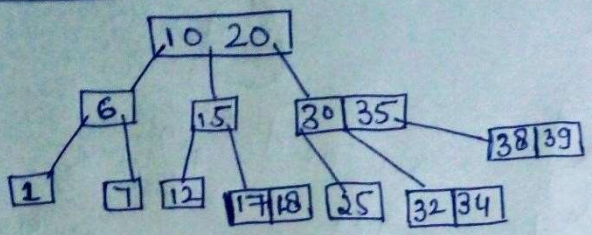
1. If deletion of a value causes a node to be **less than half full, it is combined with its neighboring nodes**, and this can also propagate all the way to the root.
2. Hence, deletion can reduce the number of tree levels.

Case 1: Deletion in leaf node (GateBook)

1. If leaf node has sufficient keys, delete the key and stop.
2. If on deletion a key from leaf node, if leaf node underflows then borrow a key from its sibling(s) via parent node. Parent key comes down and a key from sibling takes the place of parent key.
3. If siblings are not able to give the key, then bring the parent down and merge, parent key, the node from which deletion is done and the other sibling.
4. If we merge the nodes and it causes an internal node to be underflow, then internal node will ask its sibling for borrowing a key, if sibling can't give the key, bring the parent key down and merge the nodes.

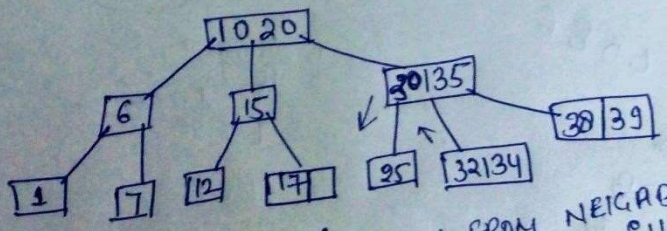
Example Delete the keys from the following B tree.
Order = 3
Max keys = 2, Min keys = 1

Deletion in BTREE

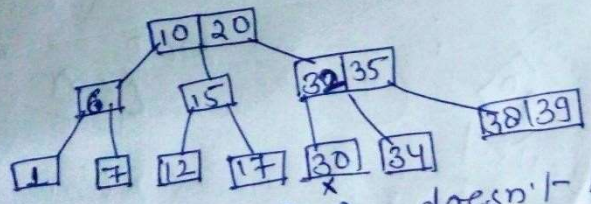


Case 1: Deletion in Leaf Node

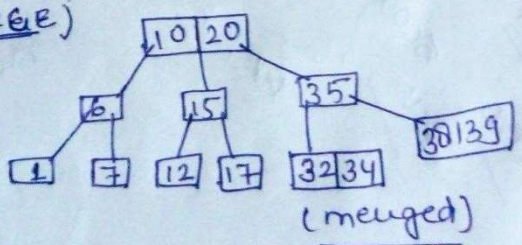
(i) - delete(18). If we delete 18, leaf node is not underflow. delete the key 18 and stop.



(ii) delete(25) (BORROW FROM NEIGABOR)
If we delete 25, Node will underflow. It will as its sibling to share key via parent. it has only one sibling.
- 32 will go up.
- 30 will come down.



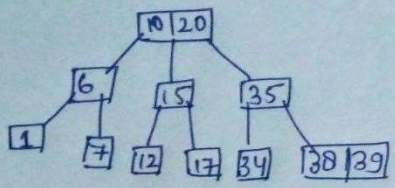
(iii) Delete(30) sibling doesn't have enough key.
(MERGE)



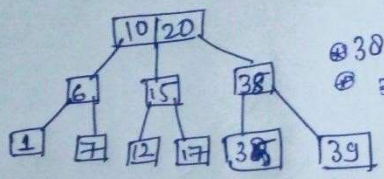
(merged)

Bring Parent key down and merge.

(iv) Delete 32



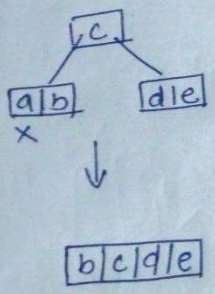
(v) delete 34



* 38 went up
* 35 came down.

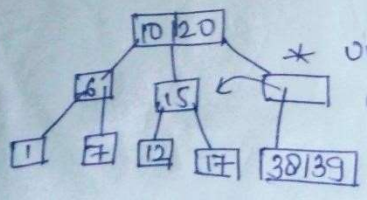
(vi) delete 35 (underflow) -

Suppose following B+ tree with min key = 2

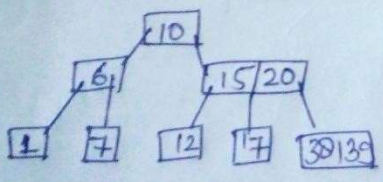


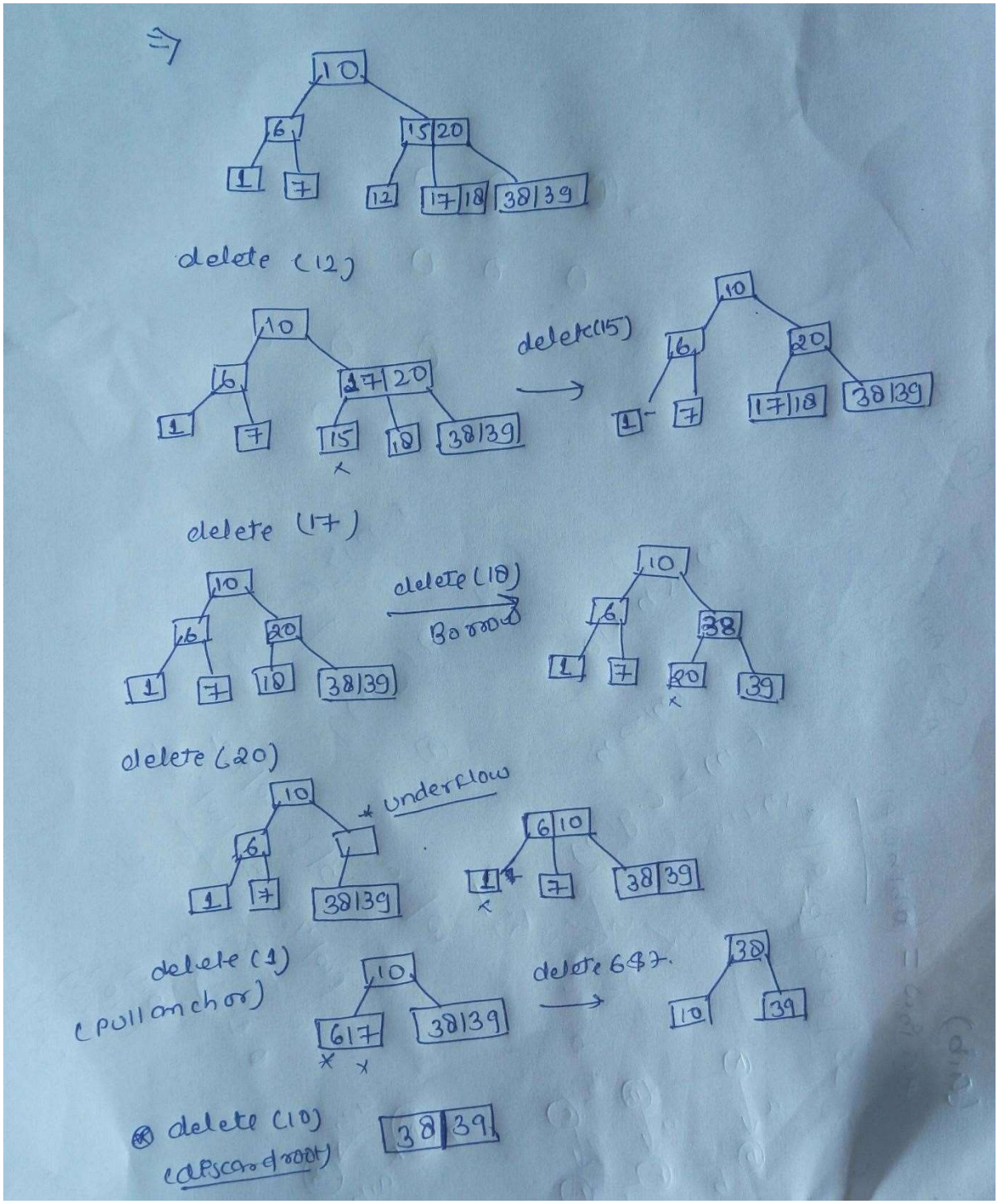
delete 'a'.
underflow.
merge sibling1 and sibling2 and bring parent down.

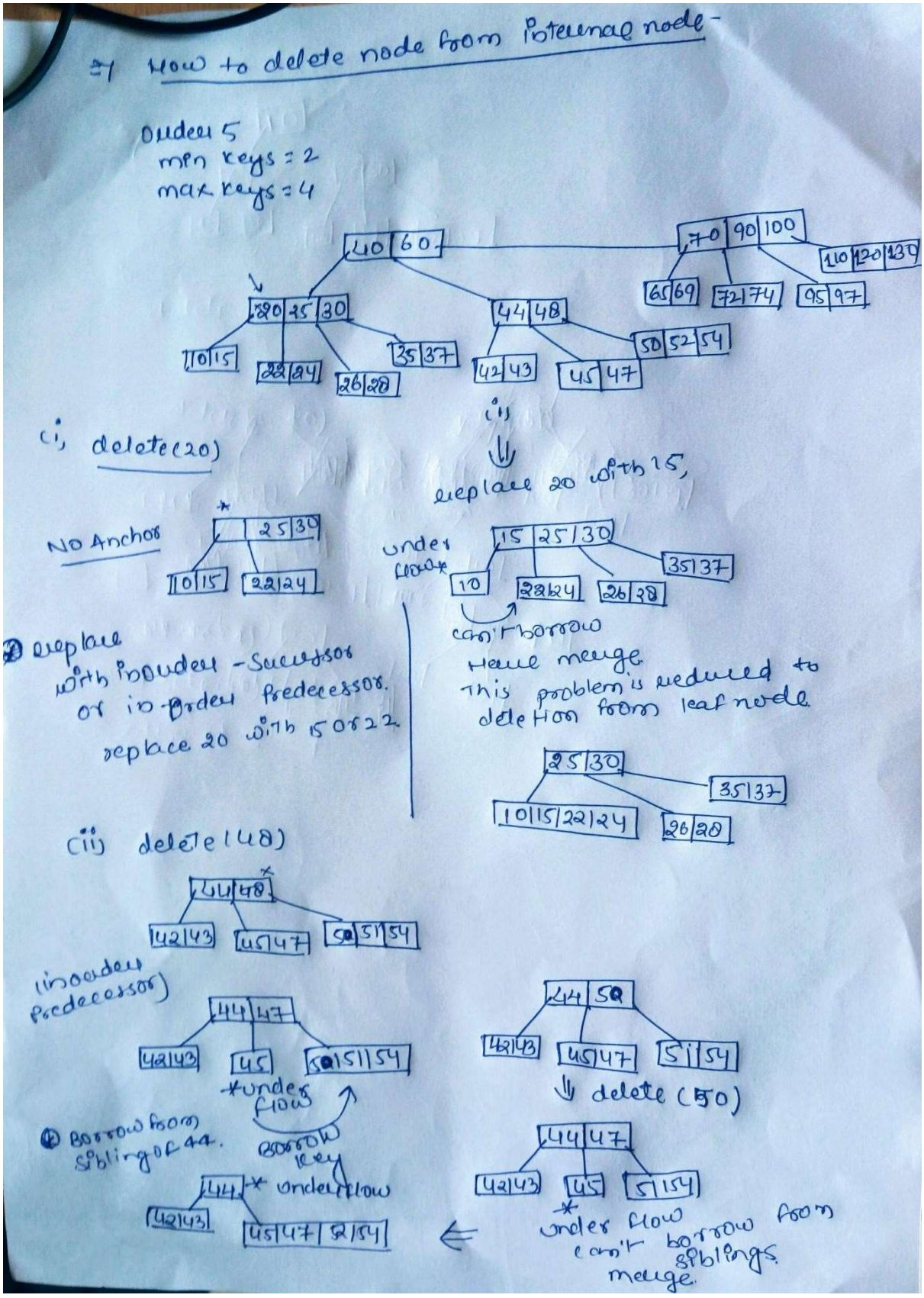
If we delete 35, we bring 38 down and merge with 39.



* underflow
* will borrow from sibling but borrow not possible as it is single key, Hence merge.
* pull parent key (20)
* there will be 3 pointers

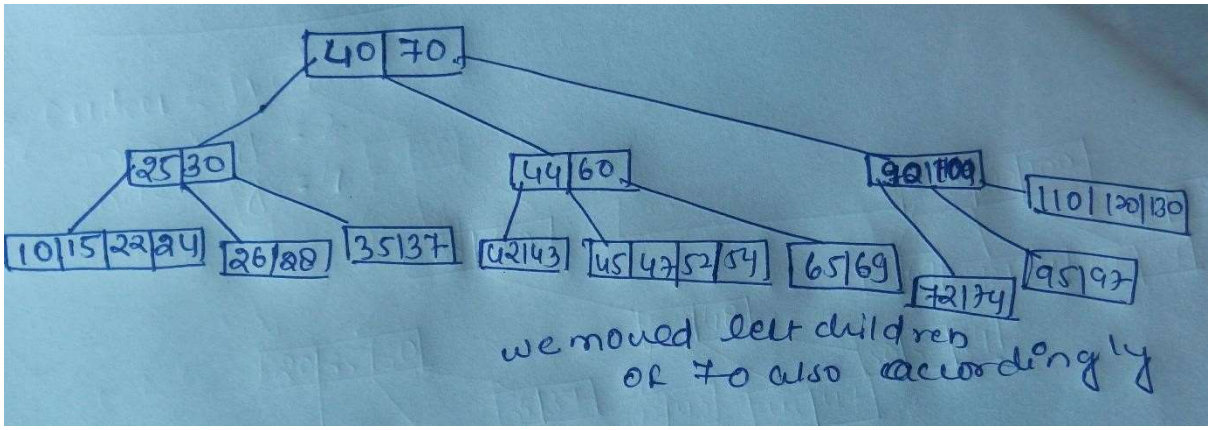



⇒
10
6
15 20
1
7
12
17 18
38 39
delete (12)
10
6
17 20
1
7
15
18
38 39
delete(15)
10
6
20
1
7
17 18
38 39
delete (17)
10
6
20
1
7
18
38 39
delete (18)
Borrow
10
6
38
1
7
20
39
delete (20)
10
6
1
1
7
38 39
* underflow
6 10
1
7
38 39
delete (1)
(pull anchor)
10
6 17
38 39
delete 6 & 7.
38
10
39
delete (10)
(collapse root)
38 39



(i) replace with Inorder - Successor or in-order predecessor.
replace 20 with 15 or 22.

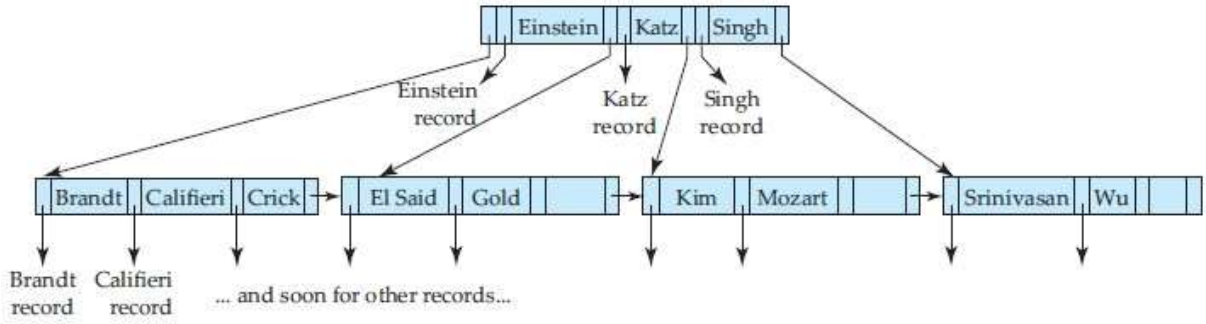
(ii) Borrow from sibling of 44.



Deletion from Non-leaf node

We replace nonleaf key with inorder predecessor or inorder successor. And then problem reduces to a deletion from leaf node.

Note -
In general, a B-tree node may contain **additional information** needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node.



B-Tree example

Example Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with **p = 23**. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have **p * 0.69 = 23 * 0.69 =15.87** or approximately **16 pointers** and, hence, 15 search key field values. The average **fan-out fo = 16**.We can start at the root and see how many values and pointers can exist, **on the average**, at each subsequent level:

Root:	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3840 key entries	4096 pointers
Level 3:	4096 nodes	61,440 key entries	

15 + 240 + 3840 + 61440 = 65,535

B+Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B+tree**.

In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer.

In a B+ tree, **data pointers are stored only at the leaf nodes of the tree**; hence, the structure of leaf nodes differs from the structure of internal nodes.

The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field.

For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

Note

1. A B+-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file.
2. Leaf nodes are similar to the first (base) level of an index. Internal nodes of the B+-tree correspond to the other levels of a multilevel index.
3. Some search field values from the **leaf nodes are repeated in the internal nodes** of the B+-tree to guide the search.

The structure of the **internal nodes** of a B+ tree of order p is as follows:

1. Each internal node is of the form
   $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$
   where $q \leq p$ and each P_i is a **tree pointer.**
2. Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

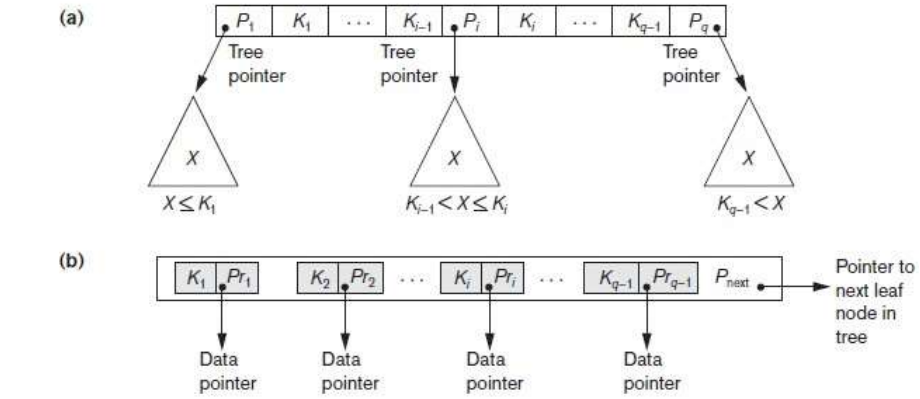
The structure of the **leaf nodes of a B+ tree** of order p

1. Each leaf node is of the form

   $<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}>$

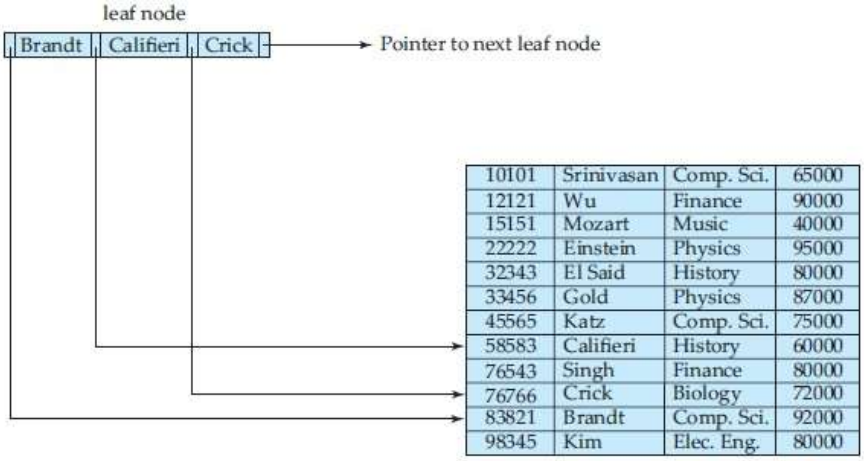
   where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B+-tree.

The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values. (b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.



2. Within each leaf node, $K_1 \leq K_2 \ldots, K_{q-1}, q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

The pointers in **internal nodes are tree pointers** to blocks that are tree nodes, whereas the pointers in leaf nodes are data pointers to the data file records or blocks—except for the **Pnext** pointer, which is a tree pointer to the next leaf node.



Leaf node of a relation

The nonleaf nodes of the B+-tree form a multilevel (sparse) index on the leaf nodes.

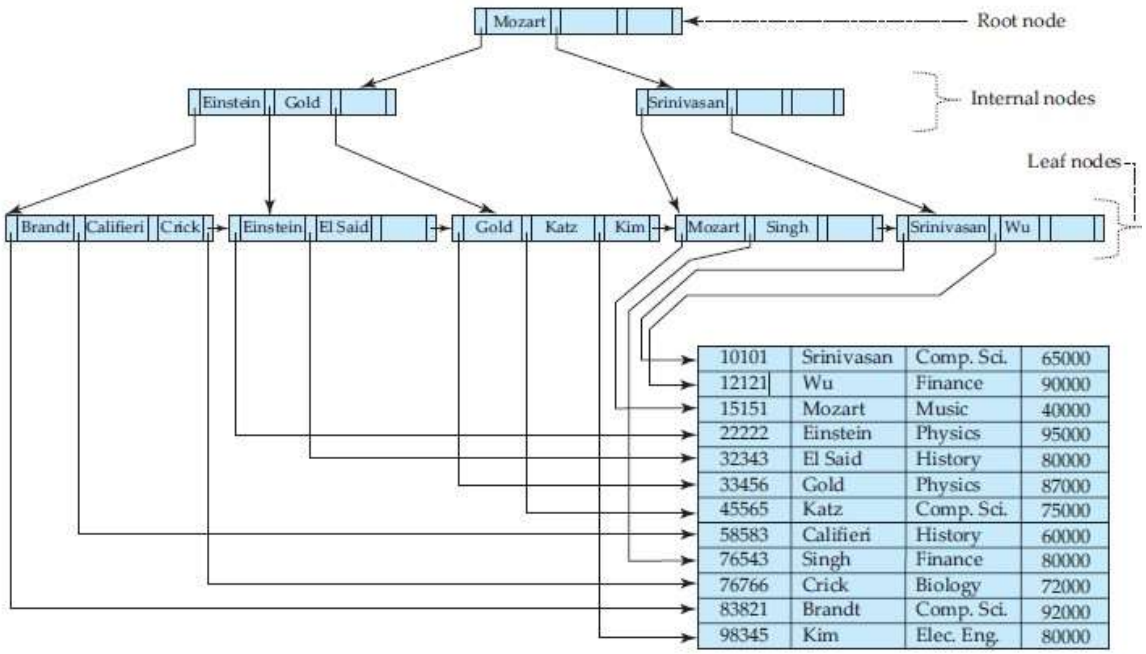


Figure 11.9 B^+-tree for *instructor* file ($n = 4$).

Queries on B+Trees

The function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree.

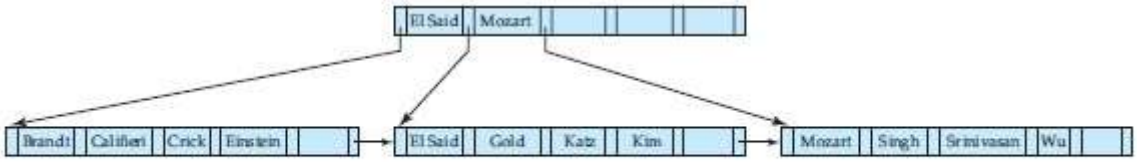


Figure 11.10 B^+-tree for *instructor* file with $n = 6$.

1. For the **same block** (node) size, the order p will be larger for the B+-tree than for the B-tree**.**
2. B+-trees can also be used to find all records with search key values in a **specified range**.

Updates on B+Trees (Korth)

When a record is inserted into, or deleted from a relation, **indices on the relation must be updated** correspondingly.

When a node is split or a pair of nodes is combined, **we must ensure that balance is preserved**.

- **Insertion** we first find the leaf node in which the new key will be inserted. We then insert an entry (that is, a **search-key value and record pointer pair**) in the leaf node, positioning it such that the search keys are still in order.
- **Deletion** We find the leaf node containing the entry to be deleted, if there are multiple entries with the same search key value, we search across all entries with the same search-key value until

we find the entry that points to the record being deleted. We then remove the entry from the leaf node. All entries in the leaf node that are to the right of the deleted entry are shifted left by one position, so that there are no gaps in the entries after the entry is deleted.

Insertion (GateBook)

⇒ GATE BOOK (B+ TREE) –

Internal Node

Order p

- p = tree pointers (max)
- $\lceil \frac{p}{2} \rceil$ = Tree pointers (min)
- p − 1 = Max keys.
- $\lceil \frac{p}{2} \rceil - 1$ = min keys
- Not applied on root Node. for min condition.

Leaf Node

Order p

- max keys = p
- min keys = $\lceil \frac{p}{2} \rceil$

NOTE – In GATE questions, if only one order is given suppose order = 4

- take order 4 for internal as well as leaf nodes

Internal Node

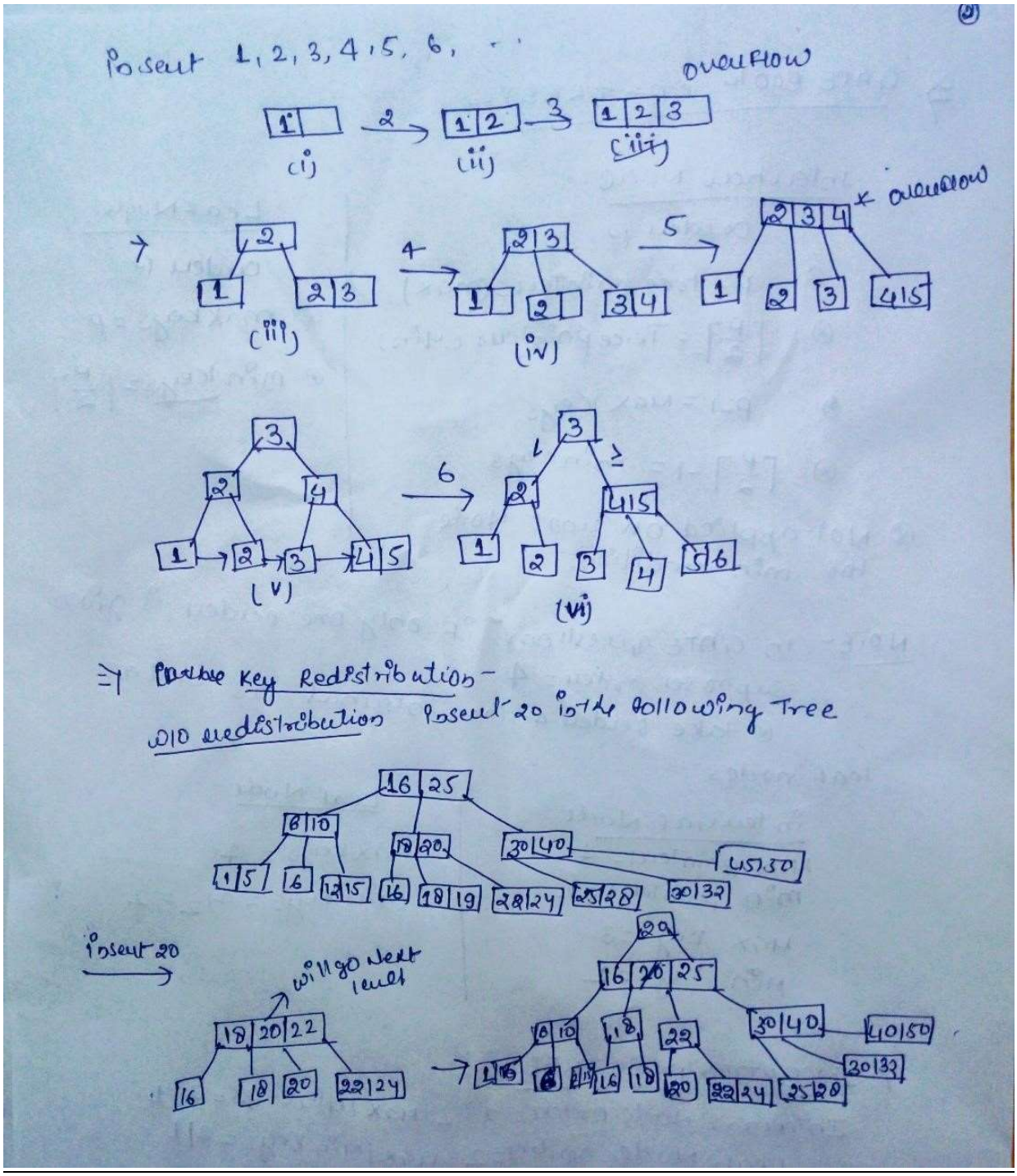
Max pointers = 4
min pointers = 2
Max keys = 3
Min keys = 1

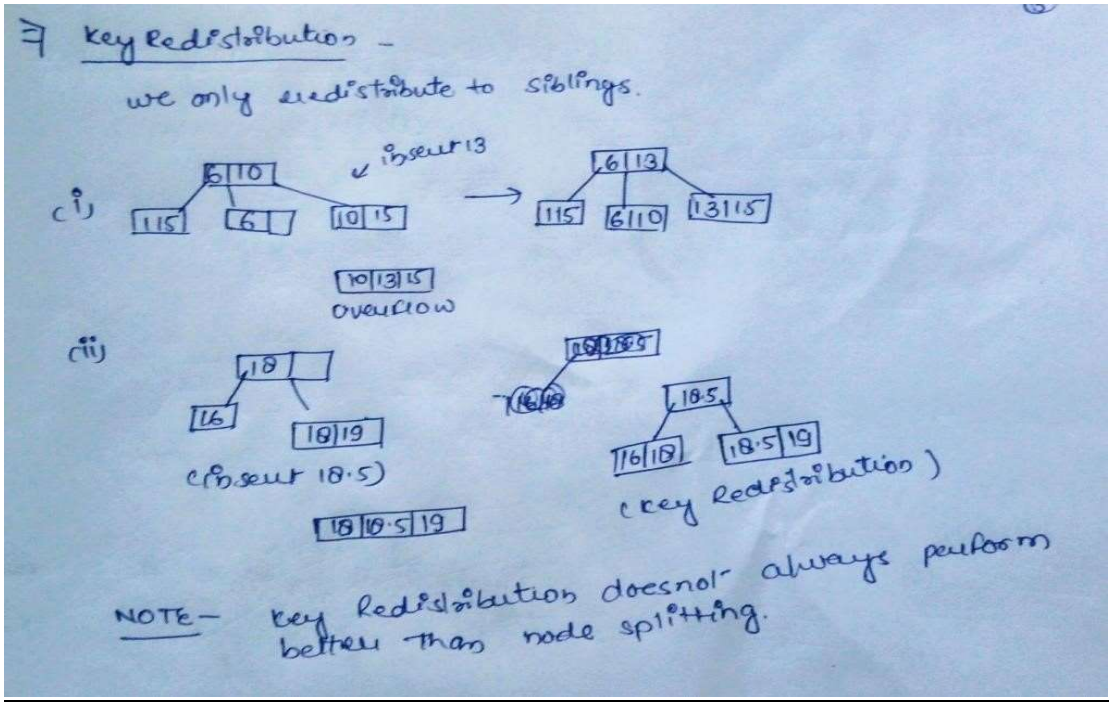
Leaf Node

Max keys = 4
min keys = 4/2 = 2.

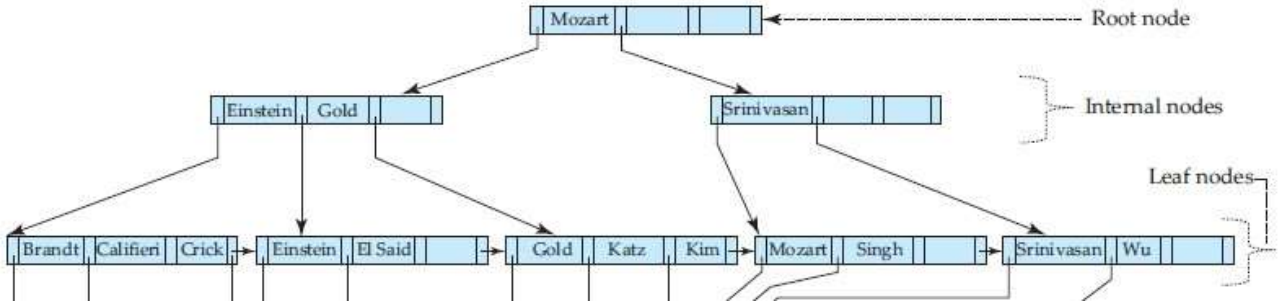
⇒ INSERTION IN B+ TREE

Internal Node order = 3 – Max/Min keys = 2/1
Leaf Node order = 2 Max/Min keys = 2/1


Insert 1, 2, 3, 4, 5, 6, ...
overflow
(i)
(ii)
(iii)
(iv)
(v)
(vi)
Key Redistribution -
No redistribution
Insert 20 in the following Tree
Insert 20
will go next level



Insertion in B+ Tree (Korth)



Insert **Adams** into this tree

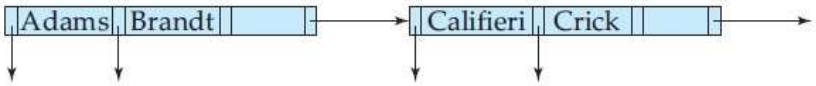


Figure 11.12 Split of leaf node on insertion of "Adams"

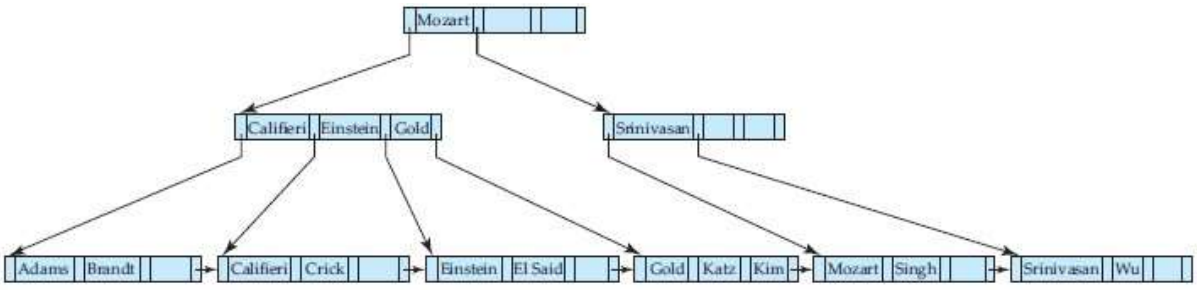
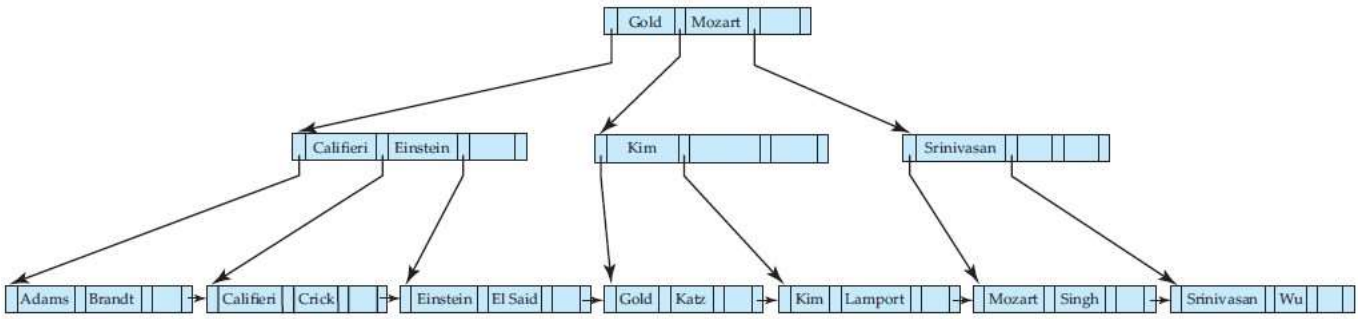


Figure 11.13 Insertion of "Adams" into the B+-tree of Figure 11.9.



Insertion of "Lamport"

Deletion (Korth) P=4

In Leaf node, max p keys = 4 Min keys=p/2 = 2

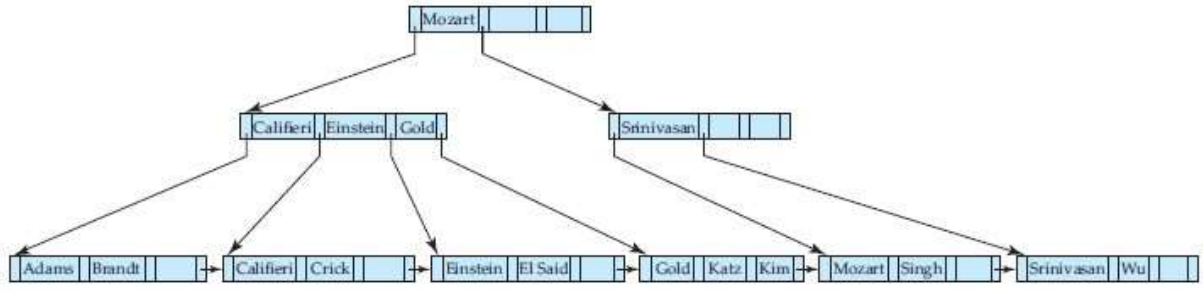


Figure 11.13 Insertion of "Adams" into the B+-tree of Figure 11.9.

Delete "**Srinivasan**"

If we delete Srinivasan from the tree, right most node will be left Wu and we will update anchor also. Min allowed key in leaf node is 2 (p/2 = 4/2 = 2). Hence this node is under flow.

1. We must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full.
2. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node.
3. In our example, the underfull node with the entry for "Wu" can be merged with its left sibling node, and deleting the now empty right sibling.
4. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

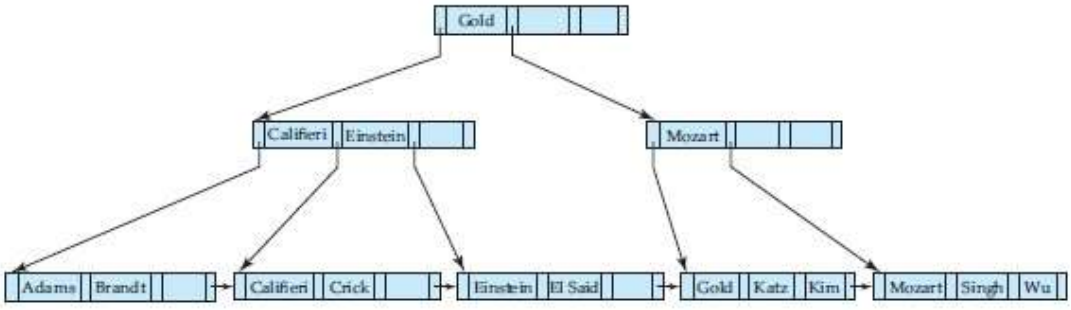
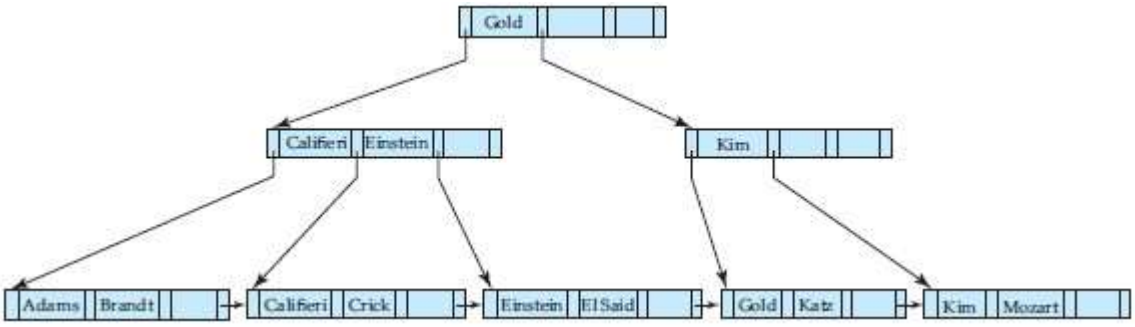


Figure 11.16 Deletion of "Srinivasan" from the B+-tree of Figure 11.13.

Delete Singh and wu.

Kim Is borrowed from sibling.



Deletion of "Singh" and "Wu"

Delete Gold

If we delete Gold from leaf node, node will be underflow and it can't borrow any key from its sibling.

1. Kim is brought down and merge Kim, Katz, and Mozart.
2. Internal node, parent of Gold is underflow now and it can't borrow a key from its sibling.
3. Califieri, Einstein and Gold are merged together.

Note It is worth noting that, as a result of deletion, a key value (Gold) that is present in a nonleaf node of the B+-tree may not be present at any leaf of the tree.

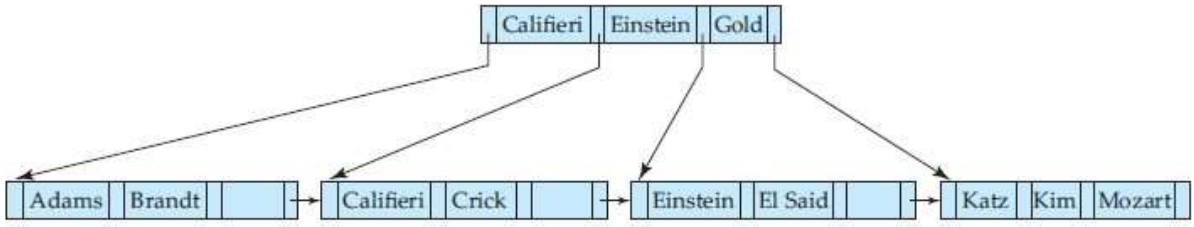


Figure 11.18 Deletion of "Gold" from the B+-tree of Figure 11.17.

Note

1. **If possible redistribute from the sibling.**
2. **If redistribution is not possible then bring down the key from parent node and merge.**
3. **If underflow is propagated to parent node, check with parent's sibling for redistribution then merge with parent's parent key.**

Transaction Management:

View Serializability:

View equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be **view equivalent** if three conditions are met:

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.
2. For each data item Q, if transaction T_i executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T_j, then the read(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction T_j.
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

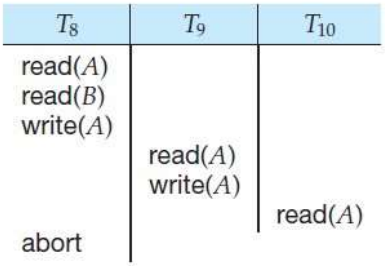
Note:-

Blind writes appear in any view-serializable schedule that is not conflict serializable.

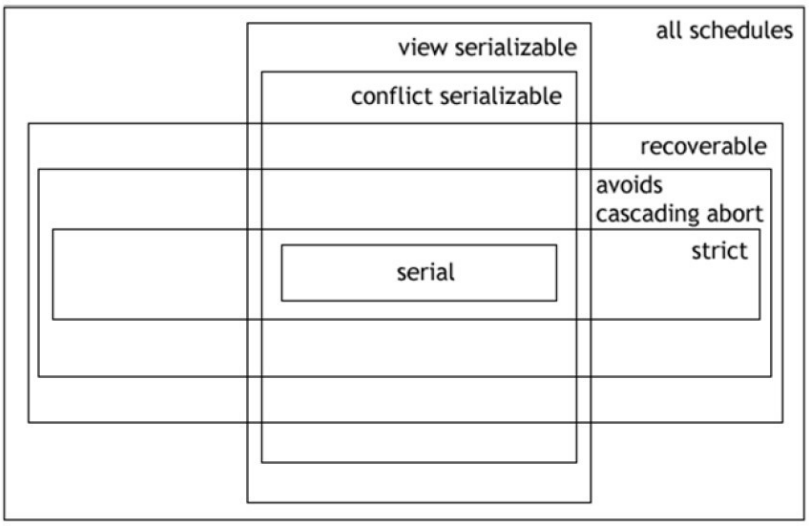
1. **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. The validation test (described below) is applied to transaction T_i. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write phase**. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database. Read-only transactions omit this phase.

Transaction Concurrency Control (Korth)

1. A **transaction** is a unit of program execution that accesses and possibly updates various data items.
2. The transaction consists of all operations executed between the begin transaction and end transaction.
3. A transaction is indivisible, it either executes in its entirety or not at all.
4. **Atomicity**: Either all operations of the transaction are reflected properly in the database, or none are. – Recovery Management
5. **Consistency**: Execution of a transaction in isolation preserves the consistency of the database. – Application Programmer
6. **Isolation**: Each transaction is unaware of other transactions executing concurrently in the system. – Con-concurrency control.
7. **Durability**: A transaction's actions must persist across crashes. Recovery Management
8. There exist n factorial (n!) different valid **serial** schedules.
9. If a schedule S can be transformed into a schedule
   S
   by a series of swaps of nonconflicting instructions, we say that S and S' are **conflict equivalent**.
10. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.
11. **View serializability** is not used in practice due to its high degree of computational complexity.
12. A **recoverable 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 commit operation of Tj .
13. In which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.



14. Where cascading rollbacks cannot occur. Such schedules are called **cascadeless schedules**.
15. 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** . It means no dirty read.
16. Each cascadeless schedule is also recoverable.
17. **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction **may not be serializable** with respect to other transactions.
18. **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
19. **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level.


all schedules
view serializable
conflict serializable
recoverable
avoids
cascading abort
strict
serial