KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module 3: SQL DML (Data Manipulation Language), Physical Data Organization

SQL DML (Data Manipulation Language) - SQL queries on single and multiple tables, Nested queries (correlated and non-correlated), Aggregation and grouping, Views, assertions, Triggers, SQL data types.

Physical Data Organization - Review of terms: physical and logical records, blocking factor, pinned and unpinned organization. Heap files, Indexing, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files.

MODULE 3

Basic structure of SQL Queries

The basic structure of an SQL expression consists of three clauses: **select**, **from** and **where**.

Syntax:

SELECT <attribute list> FROM <table list> WHERE <condition>;

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

SQL queries on single tables

Select Clause:

- It corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.

Eg: Find the names of all branches in the loan relation.

select *branch-name* from *loan;*

Output:

branch-name
Round Hill
Perryridge
Down Town
Perryridge

- To eliminate the duplicates, the keyword **distinct** is inserted after select.

select distinct branch-name **from** loan;

Output:

branch-name
Round Hill
Perryridge
Down Town

- ***** is used to denote "all attributes".

Example: select * from loan;

- means all attributes of loan are to be selected.

Output:

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Perryridge	1500
L-15	Down Town	1500
L-16	Perryridge	1000

- Select clause also contains arithmetic expressions involving operators +,-,*,/ on attributes of tuples or constants

select loan-number, branch-name, amount*100 **from** loan;

Output:

loan-number	branch-name	amount
L-11	Round Hill	90000
L-14	Perryridge	150000
L-15	Down Town	150000

L-16	Perryridge	100000

From clause:

It corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.

SQL queries on multiple tables

- Queries often need to access information from multiple relations.

Eg 1: find customer names, loan numbers and loan amount of all customers who have a loan from the bank

select customer-name, borrower.loan-number, amount **from** borrower, loan **where** borrower.loan-number=loan.loan-number;

Output:

customer-name	borrower.loan_number	amount
Adams	L-16	1300
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-11	900

Eg 2: find customer names, loan numbers and loan amount for all loans at the Perryridge branch

select customer-name,borrower.loan-number,amount **from** borrower, loan **where** borrower.loan-number=loan.loan-number **and** branch-name='Perryridge';

Where clause:

It corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

SQL uses logical connectives **and**, **or**, **not** rather than the mathematical symbols ^,v,¬.

Eg 1: find all loan numbers for loans made at the Perryridge branch with loan amounts greater than $1200.

select loan-number **from** loan **where** branch-name='Perryridge' and amount >1200;

output:

loan_number	branch_name	amount
L-15	Perryridge	1500
L-16	Perryridge	1300

Eg 2: find the loan number of those loans with loan amounts between $90000 and $100000.

select loan-number **from** loan **where** amount **between** 90000 **and** 100000;

The rename operation

- SQL provides a mechanism for renaming both relations and attributes.
- It uses the **as** clause.
  Syntax: *old-name* **as** *new-name*

Eg: Consider the query:

select customer-name, borrower.loan-number, amount **from** borrower, loan **where** borrower.loan-number=loan.loan-number;

If we want to replace the attribute name loan_number with the name loan_id, the query can be rewritten as

select customer-name, borrower.loan-number **as** loan_id, amount **from** borrower, loan **where** borrower.loan-number=loan.loan-number;

OUTPUT:

customer-name	loan_id	amount
Adams	L-16	1300
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-11	900

String operations

- The most commonly used operation on strings is *pattern matching* using the operator **like.**
- Two special characters are used to describe patterns: **percent (%)** and **underscore(_).**
- **percent (%):** The % character matches any substring.
- **underscore(_):**The _ character matches any character.

- Patterns are case sensitive. ie, uppercase characters do not match lower case characters or vice versa.
- Egs: 'Perry%' matches any string beginning with "Perry".
- '%idge%' matches any string containing "idge" as a substring. Egs: 'Perryridge','Rock Ridge', 'Mianus Bridge' and 'Ridgeway'.

1. Find the names of all customers whose street address includes the substring 'Main'.

   select customer_name **from** customer **where** customer_street **like** '%Main%';

2. Consider *customer(name,address ,place, phone).*Find the names of all customers whose place is either 'Munnar' or 'Mannar'.

   select name **from** customer **where** place **like** 'M_nnar';

- If an underscore or % (special pattern characters) is needed as a literal character in the string, the character should be preceded by an escape character.

Eg: 'AB_CD\%EF' escape '\' represents the string 'AB_CD%EF' because \ is specified as the escape character.

Ordering the display of tuples:

- The **order by** clause causes the tuples in the result of a query to appear in sorted order.
- Eg: list in alphabetic order all customers who have a loan at the Perryridge branch.

  select distinct customer_name **from** borrower, loan **where** borrower.loan_number = loan.loan_number **and** branch_name='Perryridge' **order by** customer_name;

- By default, the **order by** clause lists items in ascending order.
- To specify the sorted order, we specify **desc** for descending order and **asc** for ascending order.

  Eg: list the entire loan relation in descending order of amount.

  If several loans have the same amount, order them in ascending order by loan_number.

  select * **from** loan **order by** amount **desc**, loan_number **asc**;

output:

loan_number	branch_name	amount
L-23	Redwood	2000
L-14	Downtown	1500
L-15	Perryridge	1500

L-16	Perryridge	1300
L-17	Downtown	1000
L-11	Round Hill	900
L-93	Mianus	500

Set operations:

SQL operations **Union, intersect** and **except** correspond to the relational-algebra operations U, n, -

Note: *The relations participating in these operations must be compatible ie, they must have the same set of attributes.*

Union:

- The UNION command is used to select related information from two tables. When using the UNION command all selected columns need to be of the same data type.
- With UNION, only distinct values are selected.

Eg: Find all customers having a loan, an account, or both at the bank.

(select customer-name from depositer) **union** (select customer-name from borrower);

DEPOSITER

customer-name	account-number
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Smith	A-215

BORROWER

customer-name	loan-number
Adams	L-16
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11

Output:

customer-name
Hayes
Johnson
Jones
Smith
Adams
Jackson

- If we want to retain all duplicates, we must write **union all**

(select customer-name from depositer) **union all** (select customer-name from borrower);

- The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both the relations.

O/P:

customer_name
Hayes
Hayes
Johnson
Johnson
Jones
Jones
Smith
Smith
Adams
Jackson

Intersect:

- The INTERSECT command is used to find the intersection of two tables.

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

(select customer-name from depositer) **intersect** (select customer-name from borrower);

Output:

customer-name
Hayes
Jones
Smith

- The intersect operation automatically eliminates duplicates.
- If we want to retain all duplicates, we must write i**ntersect all**

  (select customer-name from depositer) **intersect all** (select customer-name from borrower);
- The number of duplicate tuples in the result is equal to the minimum number of duplicates that appear in both the relations.

Except:

- It returns rows from first select statement.

Eg: Find all customers who have an account but no loan at the bank.

(select customer-name from depositer) **except** (select customer-name from borrower);

Output:

customer-name
Johnson

- The except operation automatically eliminates duplicates.
- If we want to retain all duplicates, we must write **except all**

  (select customer-name from depositer) **except all** (select customer-name from borrower)

- The number of duplicate tuples in the result is equal to the number of duplicate tuples in depositer minus the number of duplicate tuples in borrower.

Aggregate functions

- Functions that take a collection of values as input and return a single value. SQL offers five built-in aggregate functions
    1. Average: **avg**
    2. Minimum: **min**
    3. Maximum: **max**
    4. Total: **sum**
    5. Count: **count**

Eg 1: Find the average account balance at the Perryridge branch.

ACCOUNT

account-number	branch-name	balance
A-101	Downtown	500
A-102	Perryridge	400
A-222	Redwood	700
A-305	Perryridge	350

select avg (balance) **from** account **where** branch-name='Perryridge';

Output: avg(balance)

375

- We can give a name to the attribute of the result relation by using the **as** clause.
  select avg (balance) **as** avg_balance **from** account **where** branch-name='Perryridge';
  Output:

avg_balance
375

- Aggregate functions can be applied to a group of sets of tuples by using the **group by** clause. Tuples with the same value on all attributes are placed in one group.

  Eg 2: Find the average account balance at each branch.

  select branch-name, **avg**(balance) **from** account **group** by branch-name;

  output:

branch-name	balance
Downtown	250
Perryridge	375
Redwood	350

- If we want to eliminate duplicates before computing an aggregate function, we use the keyword **distinct** in the aggregate expression.

  Eg 3: Find the number of depositors for each branch

  select branch-name, **count** (**distinct** customer-name) **as** customer_count **from** depositer, account **where** depositor.account-number=account.account-number **group by** branch-name;

  Output:

branch-name	customer-count
Downtown	1
Perryridge	1
Redwood	null
Perryridge	null

- In some cases, it is useful to state a condition that applies to groups rather than to tuples. It can be expressed with the **having** clause of SQL.

  Eg 4: List the branches where the average account balance is more than $1200.

  select branch-name, **avg**(balance) **from** account **group by** branch-name **having** avg (balance) >1200;

Nested Subqueries

SQL provides a mechanism for nesting subqueries.

*A subquery is a **select-from where** expression that is nested within another SQL query.* That other query is called the **outer query.**

A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality, by nesting subqueries in the **where** clause.

Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

Eg 1: Find all customers who have both an account and a loan at the bank.

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

Eg 2: Find all customers who have a loan at the bank but do not have an account at the bank

select distinct customer_name **from** borrower **where** customer_name **not in** (**select** customer_name **from** depositor);

Eg 3: Find all customers who have both an account and a loan at the Perryridge branch

select distinct customer_name **from** borrower, loan **where** borrower.loan_number = loan.loan_number **and** branch_name = 'Perryridge' **and** (branch_name, customer_name) **in** (**select** branch_name, customer_name **from** depositor, account **where** depositor.account_number = account.account_number)

- The **in** and **not in** operators can also be used on enumerated sets.

  Eg 4: Find the name of customers who have a loan at the bank and whose names are neither Smith nor Jones.

  select distinct customer_name **from** borrower **where** customer_name **not in** ('Smith','Jones');

Set Comparison

- To compare sets.

- The **= ANY** (or **= SOME**) operator returns TRUE if the value v is equal to some value in the set V.
- The comparison condition (**v > ALL V**) returns TRUE if the value v is greater than all the values in the set (or multiset) V.

Eg 1: Find the names of employees whose salary is greater than the salary of all the employees in department 5.

SELECT Lname, Fname **FROM** EMPLOYEE **WHERE** Salary **> ALL** (**SELECT** Salary **FROM** EMPLOYEE **WHERE** Dno = 5);

SQL offers an alternative style for writing the preceding query. The phrase *"greater than at least one"* is represented in SQL by > **some**.

- The above query can be written using > some clause as:

SELECT Lname, Fname **FROM** EMPLOYEE **WHERE** Salary **> SOME** (**SELECT** Salary **FROM** EMPLOYEE **WHERE** Dno = 5);

Eg 2: Find the names of all branches that have greater assets than all branches located in Brooklyn.

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

>all corresponds to the phrase *"greater than all".*

- SQL also allows **<all**,**<=all**,**>=all**,**=all** and **<>all** comparisons.
- Aggregate functions cannot be composed in SQL. Thus we cannot use **max(avg(..)).**

Eg 3: Find the branch that has the highest average balance.

First find all average balances and then nest it as a subquery of a larger query that finds hose branches for which the average balance is greater than or equal to all average balances.

select branch_name **from** account **group by** branch_name **having avg**(balance) >=**all** (**select avg** (balance) **from** account **group by** branch_name)

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

CORRELATED NESTED QUERIES

Two queries are said to be correlated whenever a condition in the *WHERE* clause of a nested query references some attribute of a relation declared in the outer query.

Eg: consider *employee(name, ssn, address, salary, Dno)*

dependent(Essn, dep_name, Bdate)

1. Retrieve the name of each employee who has a dependent with the same name as the employee.

SELECT name **FROM** employee, dependent **WHERE** employee.ssn= dependent.Essn **and** employee.name= dependent.dep_name;

Joined Tables in SQL

- It permit users to specify a table resulting from a join operation in the FROM clause of a query.
- This construct may be easier to comprehend than mixing together all the select and join conditions in the WHERE clause.
- Eg: Retrieves the name and address of every employee who works for the 'Research' department.

EMPLOYEE (Fname, Minit, Lname, Ssn, Bdate ,Address ,Sex ,Salary ,Super_ssn ,Dno)

DEPARTMENT (Dname, Dnumber, Mgr_ssn, Mgr_start_date)

SELECT Fname, Lname, Address **FROM** (EMPLOYEE **JOIN** DEPARTMENT **ON** Dno = Dnumber) **WHERE** Dname = 'Research';

In a **NATURAL JOIN** on two relations R and S, ***no join condition is specified.***

- If the ***names of the join attributes are not the same*** in the base relations, it is possible to ***rename the attributes*** so that they match, and then to apply NATURAL JOIN.
- In this case, the **AS** construct can be used to rename a relation and all its attributes in the FROM clause.

SELECT Fname, Lname, Address **FROM** (EMPLOYEE **NATURAL JOIN** (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate))) **WHERE** Dname = 'Research';

- The default type of join in a joined table is called an **inner join**, where a tuple is included in the result only if a matching tuple exists in the other relation.
- If the user requires that all employees be included, a different type of join called OUTER JOIN must be used explicitly.

- There are several variations of **OUTER JOIN.**
- **LEFT OUTER JOIN** (every tuple in the left table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the right table).
- **RIGHT OUTER JOIN** (every tuple in the right table must appear in the result; if it does not have a matching tuple, it is padded with NULL values for the attributes of the left table), and
- **FULL OUTER JOIN.**

ASSERTIONS AND TRIGGERS

Two additional features of SQL are:

1. the **CREATE ASSERTION** statement and
2. the **CREATE TRIGGER** statement.

CREATE ASSERTION can be used to ***specify additional types of constraints*** that are outside the scope of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity)

CREATE TRIGGER can be used to ***specify automatic actions that the database system will perform when certain events and conditions occur.***

Specifying General Constraints as Assertions in SQL

- In SQL, users can specify general constraints that do not fall into any of the *built-in relational model constraints* (primary and unique keys, entity integrity, and referential integrity) via **declarative assertions**, using the **CREATE ASSERTION** statement of the DDL.
- Each assertion is given a constraint name and is specified via a condition similar to the WHERE clause of an SQL query.
- For example, to specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for* in SQL, we can write the following assertion:

CREATE ASSERTION SALARY_CONSTRAINT **CHECK** (**NOT EXISTS** (**SELECT** * **FROM** EMPLOYEE E, EMPLOYEE M, DEPARTMENT D **WHERE** E.Salary>M.Salary **AND** E.Dno=D.Dnumber **AND** D.Mgr_ssn=M.Ssn));

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition** in parentheses that must hold true on every database state for the assertion to be satisfied.
- The constraint name can be used later to refer to the constraint or to modify or drop it.
- The DBMS is responsible for ensuring that the condition is not violated.

- Any WHERE clause condition can be used, but many constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**. The constraint is **satisfied** by a database state if *no combination of tuples* in that database state violates the constraint.
- The basic technique for writing such assertions is to specify a query that selects any tuples *that violate the desired condition*.
- By including this query inside a NOT EXISTS clause, the assertion will specify that the result of this query must be empty so that the condition will always be TRUE.
- Thus, the assertion is violated if the result of the query is not empty.
- In the preceding example, the query selects all employees whose salaries are greater than the salary of the manager of their department. If the result of the query is not empty, the assertion is violated.
- The CHECK clause and constraint condition can also be used to specify constraints on *individual* attributes and domains and on *individual* tuples.
- A major difference between CREATE ASSERTION and the individual domain constraints and tuple constraints is that the CHECK clauses on individual attributes, domains, and tuples are checked in SQL *only when tuples are inserted or updated*. Hence, constraint checking can be implemented more efficiently by the DBMS in these cases. The schema designer should use CHECK on attributes, domains, and tuples only when he or she is sure that the constraint can *only be violated by insertion or updating of tuples*. On the other hand, the schema designer should use CREATE ASSERTION only in cases where it is not possible to use CHECK on attributes, domains, or tuples, so that simple checks are implemented more efficiently by the DBMS.

Triggers in SQL

It is mainly used to ***specify the type of action that is to be taken when certain events occur and when certain conditions are satisfied.***

For example, it may be useful to specify a condition that, if violated, causes some user to be informed of the violation. A manager may want to be informed if an employee's travel expenses exceed a certain limit by receiving a message whenever this occurs. The action that the DBMS must take in this case is to send an appropriate message to that user.

Triggers are used for mainly 2 purposes:

1. To maintain business data uniformly.
2. To maintain audit information of table data.

A trigger has *three components*:

1. The **events**: These are usually database update or delete or insert operations that are explicitly applied to the database.

These events are specified after the keyword **BEFORE**, which means that the trigger should be executed before the triggering operation is executed or after the keyword **AFTER**, which specifies that the trigger should be executed after the operation specified in the event is completed.

2. The **condition:** that determines whether the rule action should be executed

The condition is used to **monitor** the database. If *no condition* is specified, the action will be executed once the event occurs. If a condition is specified, it is first evaluated, and only *if it evaluates to true* will the rule action be executed. The condition is specified in the WHEN clause of the trigger.

3. The **action** to be taken: The action is usually a sequence of SQL statements, but it could also be a database transaction or an external program that will be automatically executed.

- Two keywords are used in trigger:

:new – get new value from column and **:old** – get old value from column

:new is associated with insert and after update.

:old is associated with delete and before update.

- Two types of triggers: row level/record level and statement level

Row level: in row/record level a trigger is executing for each row/ record.

It can be indicated by writing in the syntax **FOR EACH ROW**.

Eg: suppose we are inserting 10 rows to a table in row level trigger, then trigger is executing 10 times one for each row.

Statement level: for a DML operation, trigger is executing only once.

It can be indicated by not writing in the syntax **FOR EACH ROW**.

Syntax:

CREATE [OR REPLACE] TRIGGER <trigger name>

[BEFORE/AFTER] trigger_event[**INSERT/UPDATE/DELETE**] **OF** [COLUMN] **ON** [TABLE NAME]

[FOR EACH ROW] WHEN <condition>

Eg 1: consider *employee(name,id,ssn,address,super_ssn,Dno)*

write a trigger to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database.

Events that can trigger this rule are inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to be taken would be to call an external stored procedure SALARY_VIOLATION, which will notify the supervisor. The trigger could then be written as below. Here we are using the syntax of the Oracle database system.

CREATE TRIGGER SALARY_VIOLATION **BEFORE INSERT OR UPDATE OF** SALARY, SUPER_SSN **ON** EMPLOYEE

FOR EACH ROW WHEN (**NEW**.SALARY > (**SELECT** SALARY **FROM** EMPLOYEE **WHERE** SSN = **NEW**.SUPERVISOR_SSN))

INFORM_SUPERVISOR(**NEW**.Supervisor_ssn, **NEW**.Ssn);

Eg 2: consider an *employee(empname,city)* relation:

a) First create table employee with syntax:

create table employee (empname varchar(20), city varchar(20));

b) Then insert values into the table.

insert into employee values ('Anu', 'Delhi');

insert into employee values ('Appu', 'Bombay');

insert into employee values ('ram', 'Hyderabad');

insert into employee values ('smith', 'texas');

c) if we write the query as slect * from employee, we get output as

empname	city
Anu	Delhi
Appu	Bombay
ram	Hyderabad
smith	texas

d) After this, Write a trigger to insert or update data of employee in uppercase.

CREATE TRIGGER TRIGGER_UPPER **BEFORE INSERT OR UPDATE ON** EMPLOYEE **FOR EACH ROW**

begin

:new.empname=upper(:new.empname),:new.city=upper(:new.city));

e) After this write an SQL query to update or insert into employee with:

update employee set city='Chicago' where empname='smith';

insert into employee values ('Ben','delhi');

insert into employee values ('chithra','texas');

f) If we write select * from employee the output becomes

empname	city
Anu	Delhi
Appu	Bombay
ram	Hyderabad
SMITH	CHICAGO
BEN	DELHI
CHITHRA	TEXAS

- Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.

Difference between trigger and constraint

- Constraint is applicable in both old and new data in the database.
- Trigger is applicable from which time they are defined.

Eg 3: consider account(accnumber,bname,bcity,balance)

Write a trigger to update branch name of an account when name of the branch in city'Bombay' is changed.

create trigger update_branch **after** update **of** bname **on** branch **for each row when** bcity='Bombay'.

begin

update account set bname=new.bname where bname=old.bname;

Views in SQL

- A **view** in SQL terminology is *a single table that is derived from other tables.*
- These other tables can be *base tables* or previously defined views. A view does not necessarily exist in physical form; it is considered to be a **virtual table**, in contrast to **base tables**, whose tuples are always physically stored in the database. This limits the possible

update operations that can be applied to views, but it does not provide any limitations on querying a view.

- A view can be think of as a way of specifying a table that we need to reference frequently.
- In SQL, the command to specify a view is **CREATE VIEW.**
- The view is given a table name(or view name), a list of attribute names, and a query to specify the contents of the view.

Eg: consider company database

EMPLOYEE(name,ssn,address,Dno,salary)

WORKS_ON(Essn,Pno,Hours)

DEPARTMENT(Dname,Dnumber)

PROJECT(Pname,Pnumber,Plocation,Dnum)

1. Retrieve the employee name and the project name that the employee works on.

Rather than specifying join of the three tables EMPLOYEE, WORKS_ON and PROJECT every time, define a view that is specified as the result of these joins.

a) **CREATE VIEW** WORKS_ON1 **AS SELECT** name, Pname **FROM** EMPLOYEE, PROJECT, WORKS_ON **WHERE** ssn=Essn **and** Pno=Pnumber;

- View attributes can also result from applying aggregate functions or arithmetic operations.

b) **CREATE VIEW** DEPT_INFO (Dept_name, No_of_emps, Total_sal) **AS SELECT** name, COUNT(*), SUM (salary) **FROM** DEPARTMENT, EMPLOYEE **WHERE** Dnumber=Dno **GROUP BY** Dname;

Eg: write an SQL query to retrieve name of all employees who work on the 'x' project.

SELECT name **FROM** WORKS_ON1 **WHERE** Pname='x';

Advantages of views

1. Simplify the specification of certain queries.
2. Views are also used as a security and authorization mechanism.

- A view is supposed to be always up-to-date.ie, if the tuples in the base tables on which the view is defined is modified, the view must automatically reflect these changes. DBMS is responsible for keeping the view up-to-date.
- To dispose the view, command **DROP VIEW** is used.

  Eg: **DROP VIEW** WORKS_ON1;

View implementation

- The problem of efficiently implementing a view for querying is complex.

- Two approaches: **query modification** and **view materialization**.
- **query modification:** modifying or transforming the view query into a query on the underlying base tables.

Eg: the query:

SELECT name **FROM** WORKS_ON1 **WHERE** Pname='x';

can be modified as

SELECT name **FROM** EMPLOYEE, PROJECT, WORKS_ON **WHERE** ssn=Essn and Pno=Pnumber and Pname='x'.

Disadvantage: time-consuming to execute.

Physical Data Organization

Databases are stored physically as files of records, which are typically **stored on magnetic disks.**

Databases typically store large amounts of data that must persist over long periods of time, and hence the data is often referred to as persistent data. Parts of this data are accessed and processed repeatedly during the storage period.

Most databases are stored permanently (or persistently) on magnetic disk secondary storage, for the following reasons:

■ Generally, databases are too large to fit entirely in main memory.

■ The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as nonvolatile storage, whereas main memory is often called volatile storage.

■ The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main

memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as **files** of **records.**

Each record is a collection of data values that can be interpreted as facts about **entities, their attributes, and their relationships.**

Records should be stored on disk in a manner that makes it possible to locate them efficiently when they are needed.

There are several **primary file organizations,** which determine how the file records are physically placed on the disk, and hence how the records can be accessed.

A **heap file** (or unordered file) places the records on disk in no particular order by appending new records at the end of the file.

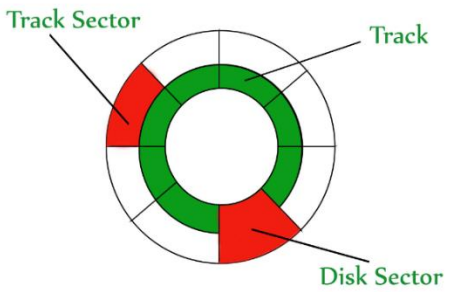
A **sorted file** (or sequential file) keeps the records ordered by the value of a particular field (called the sort key).

A **hashed file** uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk.

Other primary file organizations, such as **B-trees**, use tree structures.

Data Storage on Hard Disks

Each Hard Drive is usually composed of a set of disks. Each Disk has a layer of magnetic material deposited on its surface. The entire disk can contain a large amount of data, which is organized into smaller packages called BLOCKS (or pages). Typical disk block sizes range from 512 to 8192 bytes.



A **block** is the smallest unit of data transfer between the hard disk and the processor of the computer. Each block therefore has a fixed, assigned, address. Typically, the computer processor will submit a read/write request, which includes the address of the block, and the address of RAM in the computer memory area called a buffer (or cache) where the data must be stored/taken from.

How are tables stored on Disk ?

Each record of a table can contain different amount of data. This is because in some records, some attribute values may be 'null'. Or, some attributes may be of type varchar (), and therefore each record may have a different length string as the value of this attribute.

RECORDS

- Data is usually stored in the form of **records.**
  - Each record consists of a collection of related **data values** or **items.**
    - where each value is formed of one or more bytes and corresponds to a particular field of the record.

Eg: An EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor.

Collection of field names and their corresponding data type constitutes a **record type** or **record format** definition. A data type, associated with each field, specifies the types of values a field can take.

The number of bytes required for each data type is fixed for a given computer system.

An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of k characters k bytes. Variable-length strings may require as many bytes as there are characters in each field value.

For example, an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

```
struct employee{
    char name[30];
    char ssn[9];
```

```
    int salary;
    int job_code;
    char department[20];
} ;
```

FILES

- A FILE is a sequence of records. In many cases, all records in a file are of same record type.
- If every record in the file has exactly the same size (in bytes), the file is called **fixed-length record.**
- If different records in the file have different sizes, the file is called **variable-length records**.

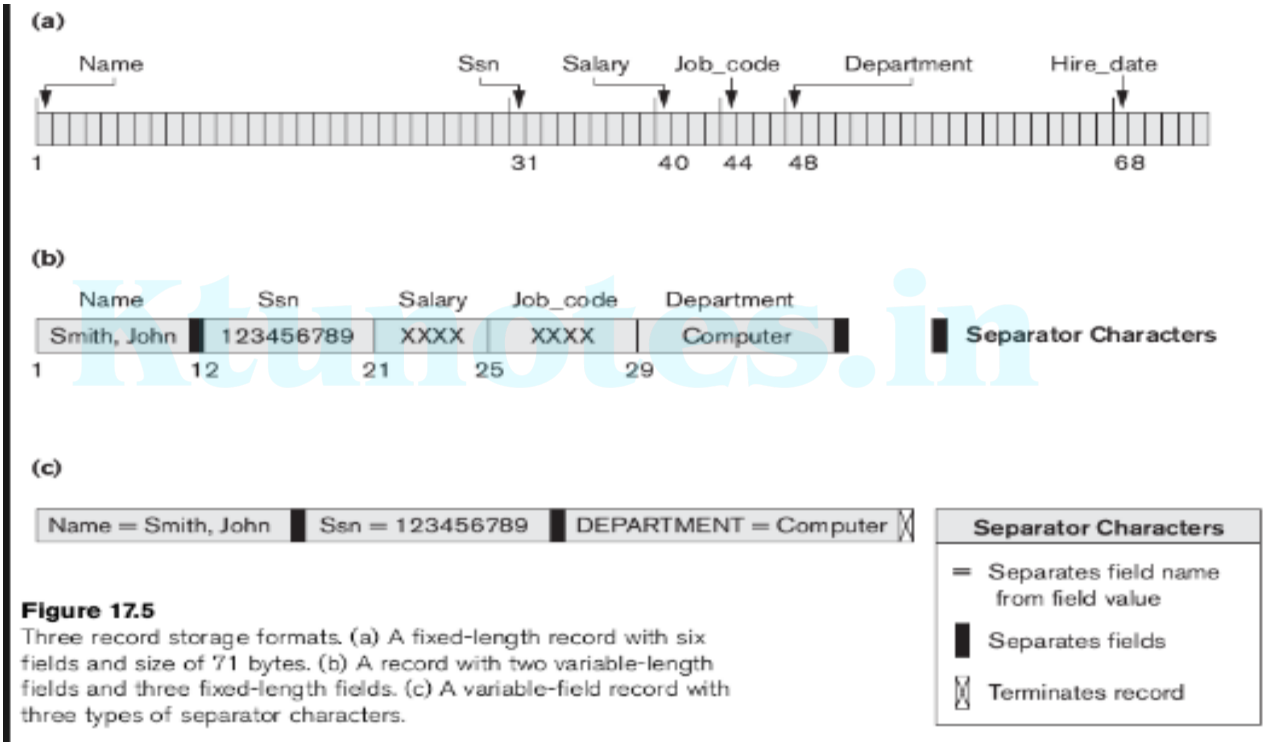


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

The fixed-length EMPLOYEE records in Figure 17.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record.

For variable-length fields, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special separator characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields as in Figure 17.5(b)

SPANNED Vs UNSPANNED RECORDS

The records of a file must be allocated to disk blocks because a block is a unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records.

Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with B>=R, we can fit $bfr = \lfloor B/R \rfloor$ records per block, where the $\lfloor(x)\rfloor$ (floor function) rounds down the number x to an integer. The value **bfr** is called the **blocking factor** for the file.

In general, R may not divide B exactly, so the unused space in each block is equal to

B - (bfr * R) bytes.

To utilize this unused space, we can store part of a record on one block and the rest on another.

A **pointer** at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on the disk. This organization is called **spanned** because records can span more than one block.

Whenever a record is larger than a block, we use a spanned organization.

If the records are not allowed to cross block boundaries, the organization is called **unspanned.** This is used with fixed length records with B > R.

For variable length records using spanned organization, each block may store different number of records. Here bfr represents the average number of records per block for the file. bfr can be used to calculate ***the number of blocks b needed for a file of r records:***

B= $\lceil(r/bfr)\rceil$ blocks.

where the $\lceil(x)\rceil$ (ceiling function) rounds the value x up to the next integer.

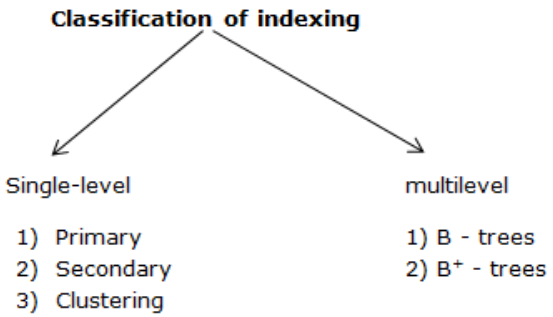
Files of Unordered Records (Heap Files)

- Simplest and most basic type of organization.
- Records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap** or **pile** file.
- This organization is often used with additional access paths, such as the secondary indexes.
- **Inserting a new record** is very efficient. The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk.

- **Searching for a record** using any search condition involves a linear search through the file block by block—an expensive procedure.
  - If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record.
  - For a file of b blocks, this requires searching (b/2) blocks, on average.
  - If no records or several records satisfy the search condition, the program must read and search all b blocks in the file.
- To **delete a record**, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a deletion marker, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value for the marker indicates a valid (not deleted) record.
- We can use either **spanned** or **unspanned organization** for an unordered file, and it may be used with either fixed-length or variable-length records.
- For a file of unordered fixed-length records using unspanned blocks and contiguous allocation, it is straightforward to access any record by its position in the file. If the file records are numbered 0, 1, 2, ... , r − 1 and the records in each block are numbered 0, 1, ..., bfr − 1, where bfr is the blocking factor, then the ith record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the (i mod bfr)th record in that block. Such a file is often called a **relative** or **direct file** because records can easily be accessed directly by their relative positions.
- Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates the construction of access paths on the file, such as the **indexes.**

INDEXES

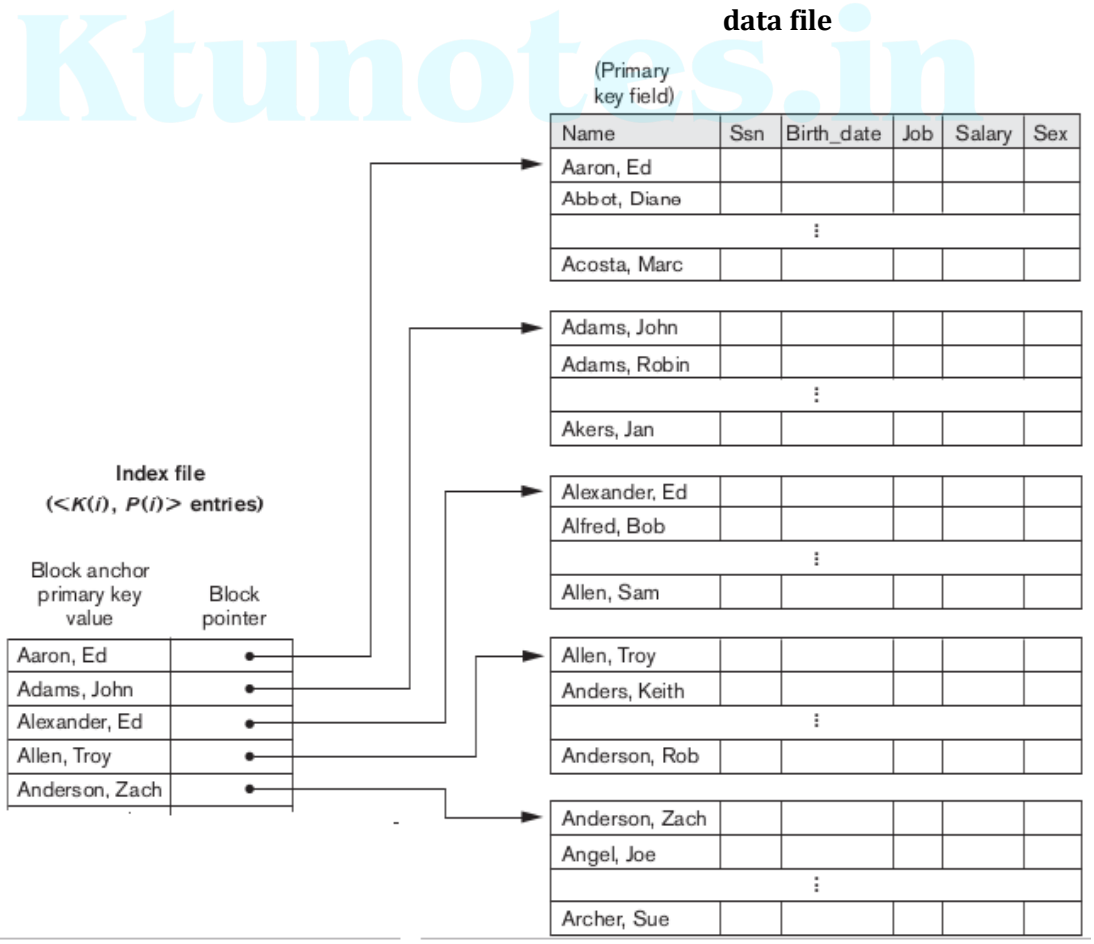
- ✓ Assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations
- ✓ Indexes are additional access structures used to speed up the retrieval of records in response to certain search conditions.
- ✓ The index structure provides secondary access paths.
  - Provides alternative ways to access the records without affecting the physical placement of records on disk.
  - Enables efficient access to records based on the indexing fields that are used to construct index.
  - Any field of the file can be used to create an index.



Single- level indexing:

Primary indexes

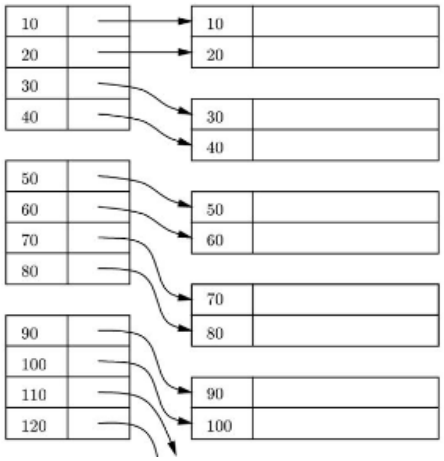
- ✓ It is an ***ordered indexing file*** whose records are of fixed length with **two fields.** *The first field is the primary key of the data file and the second is the pointer to a disk block (disk block address).*
- ✓ There is one index entry in the index file for each block in the data file.
- ✓ The primary key field of each index entry has the value of the first record in a block. And a pointer to that block is written in the second field of the index.



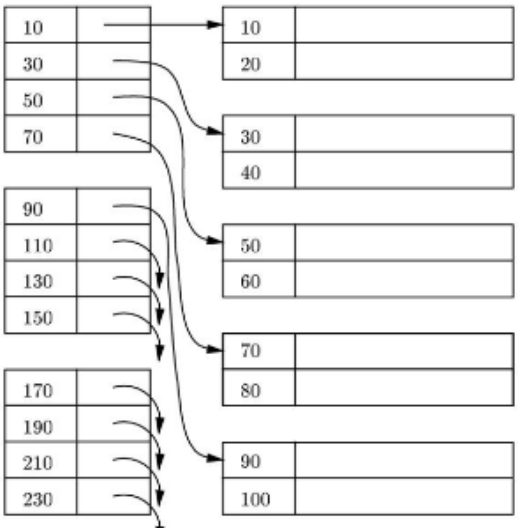
- ✓ The total number of entries in the index is the same as the number of disk blocks in the ordered data file.
- ✓ The first record in each block of the data file is called the **anchor record** or the **block anchor.**

Dense Vs sparse index

Dense index: has an index entry for every search key value in the data file.



Sparse index: has index entries for only some of the search values.



- ✓ A primary index is a **sparse index**.
- ✓ The index file for a primary index needs fewer blocks than the the data file.
- ✓ The binary search for an ordered data file requires $\log_2 b$ block access.
- ✓ *If the primary index file contains b_i blocks, then to locate a record with a search key value requires a total of* ***$\log_2 b_i + 1$*** *accesses.*

Disadvantages:

- ✓ Insertion and deletion – to insert a record in its correct position in the data file, records have to be moved to make space for the new record which in turn changes some index entries.

Example 1. Consider an ordered file with these parameters:

Block size (B)	1024 B
Record count (r)	30,000
Record length (R)	100 B *fixed size, unspanned*

The blocking factor is bfr = ⌊ (B/R)⌋ = 10 $^{record}/_{block}$. The number of blocks needed by the file is b = ⌈ (r/bfr)⌉ = 3000 blocks. A binary search would, on average, need to access ⌈ lg b⌉ = 12 blocks.

Now consider a primary index on that file with these parameters:

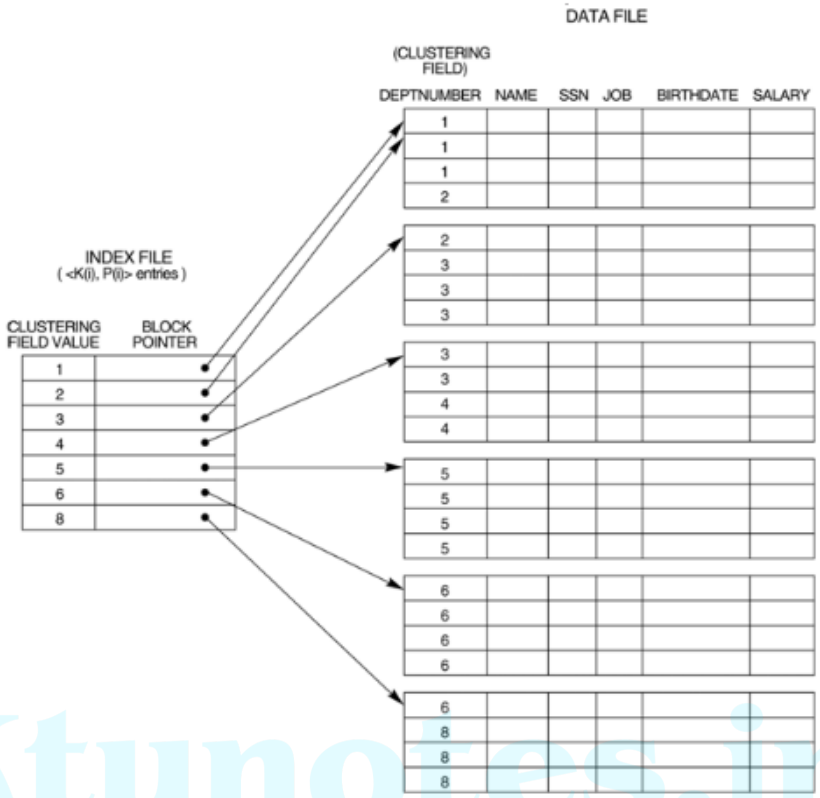
Ordering key length (V)	9 B
Block pointer length (P)	6 B
Index entry length (R_i)	15 B *(9+6)*

The blocking factor of the index file is bfr_i = ⌊ (B/R_i)⌋ = 68 $^{record}/_{block}$. The total number of index entries will be the same as the number of blocks in the main file, or r_i = 3000. Thus the number of blocks needed by the index file is b_i = ⌈ (r_i/bfr_i)⌉ = 45 blocks. A binary search on this need access on average only ⌈ lg b_i⌉ = 7 blocks.

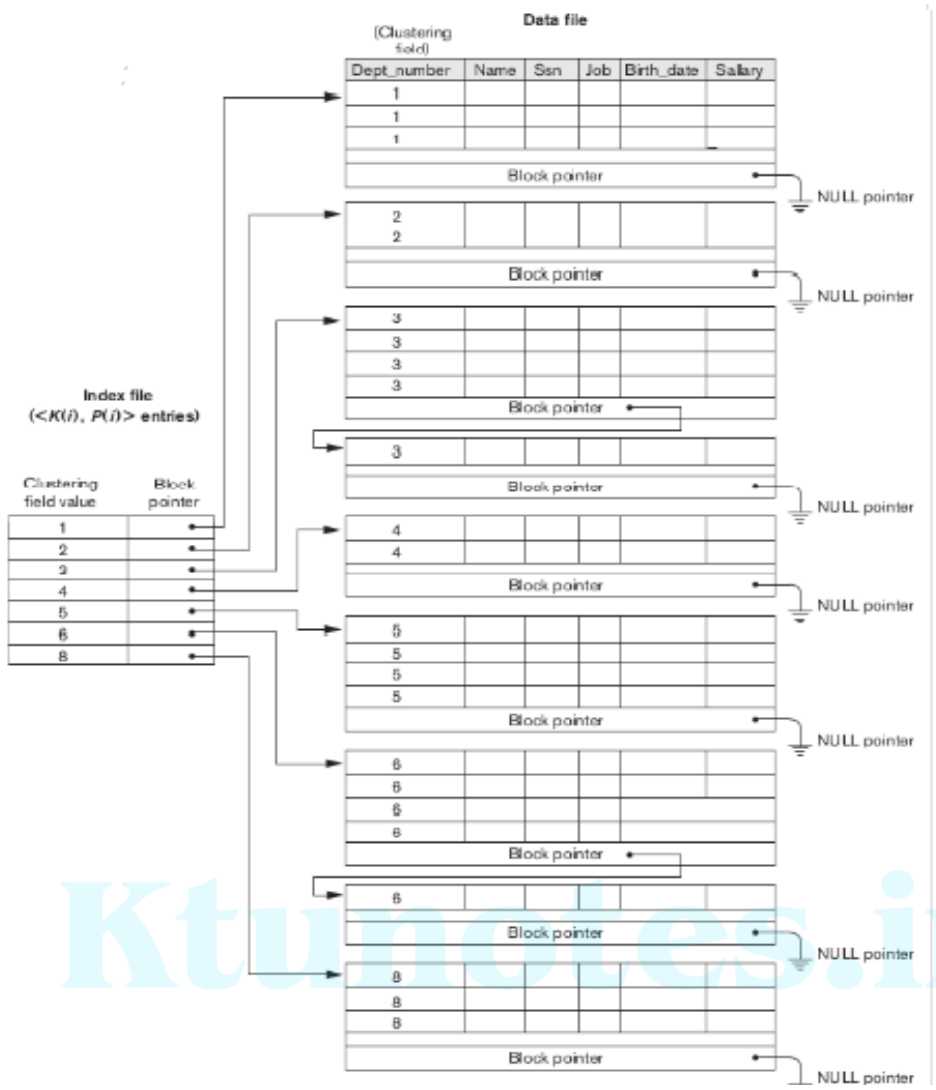
Clustering indexes

- ✓ The file records are **ordered** *based on a non-key field.*
- ✓ Advantage: speed up retrieval of records that have the same value for the clustering field.
- ✓ A clustering index is an ordered file with two fields: *first field contains the non-key value and the second field contains a block pointer.*

- ✓ One entry in the clustering index for each distinct value of the clustering field, containing the value and a pointer to the first block in the data file that has a record with that value for its clustering field.



- ✓ Disadvantages: insertion and deletion.
- ✓ To alleviate the problem of insertion, it is common to reserve a whole block for each value of the clustering field. All records with that value are placed in the block.
- ✓ A clustering index is a ***sparse index***.



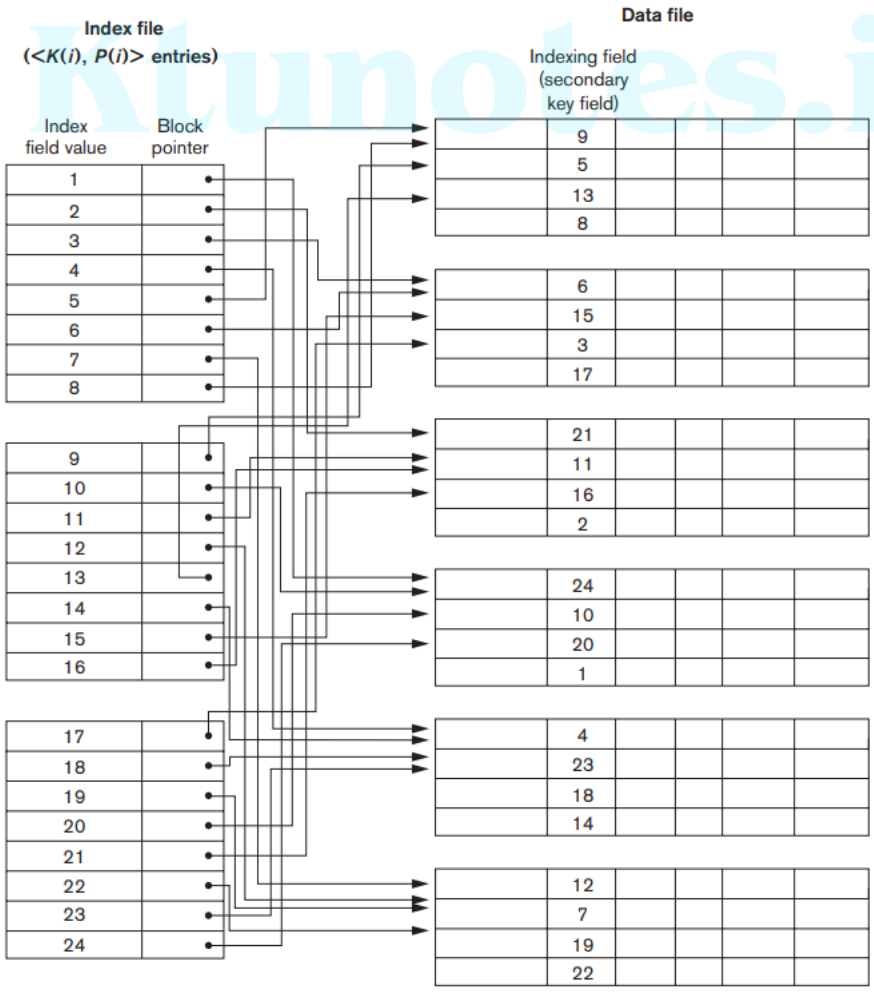
Secondary indexes

- ✓ A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- ✓ The secondary index may be on a field which
  - is a ***candidate key*** and has a ***unique value*** in every record or
  - a ***non-key with duplicate values.***
- ✓ The *index is an ordered file* with two fields. The first field is of the same data type as some nonordering field of the data file. The second field is either a block pointer or record pointer.

- ✓ For example: suppose we want to create an index for Employee by (fname, lname), assumed it to be a secondary **key**. The record file itself is ordered by SSN, which is the primary key. An index on a **secondary key** will necessarily be dense, as the file won't be ordered by the secondary key; we cannot use block anchors.

Indexing on a key field that has a distinct value:

- ✓ First consider a secondary index on a key field that has a distinct value for every record. Such fields are called secondary key.
- ✓ There is one *index entry for each record in the data file* – containing value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself. Such an index is **dense**.
- ✓ Because the ***data records are not physically ordered*** by values of the secondary key field, we *cannot use block anchors* as in primary indexes. That is why an index entry is created for each record in the data file, rather than for each block.
- ✓ ***Disadvantages:*** A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries.
- ✓ ***Advantages:*** Greater improvement in search time for an arbitrary record can be obtained by using the secondary index, because we would have to do a linear search on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main data file, even if the index did not exist.



- ✓ ***Example :*** Consider the file with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes.
  - bfr = $\lfloor(B/R)\rfloor$ = $(1024/100)$ = 10 records per block.
  - Number of blocks in this file, b = $\lceil(r/bfr)\rceil$ = $\lceil(40{,}000/10)\rceil$ =4000 blocks.
  - To do a linear search on the file, we would require b/2 = 4000/2 = 2000 block accesses on average to locate a record.
- ✓ Suppose that we construct a secondary index on a non-ordering key field of the file that is V = 11 bytes long and block pointer is P = 8 bytes long. Thus,
  - The size of an index entry is Ri = (11 + 8) = 19 bytes, and
  - The blocking factor for the index file is bfri = $\lfloor(B/Ri)\rfloor$ = $\lfloor(1024/19)\rfloor$= 53 entries per block.
  - In a dense secondary index, the total number of index entries ri is equal to the number of records in the data file, which is 40000.
  - The number of blocks needed for the index is hence bi = $\lceil(ri/bfri)\rceil$ = $\lceil(40{,}000/53)\rceil$ = 755 blocks.
  - To perform a binary search on the index file would need $\lceil(\log_2 bi)\rceil$= $\lceil(\log_2 755)\rceil$ = 10 block accesses.
  - To search for the actual data record using the index, one additional block access is needed. In total, we need 10 + 1 = 11 block accesses, which is a huge improvement over the 2000 block accesses needed on average for a linear search on the data file.
  -

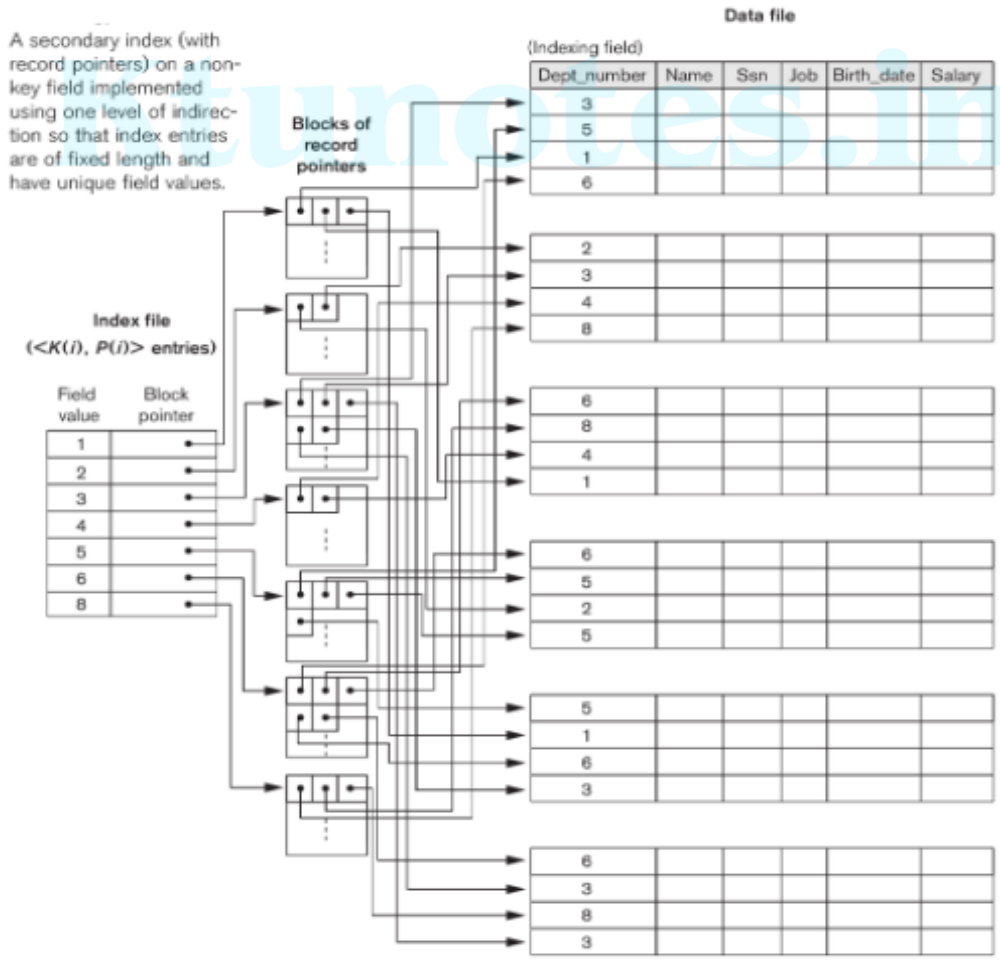
Indexing on a non-key field:

- ❖ We can also build a *secondary index on a non-key field* of a file. In this case, many data records can have the same value for the indexing field. There are *several options* for implementing such an index.
  - ➢ ***Option 1:*** We can create several entries in the index file with the same K(i) value - one for each record sharing the same K(i) value. The other field P(i) may have different block addresses, depending on where those records are stored. Such an index would be a ***dense index.***
  - ➢ ***Option 2:*** Alternatively, we can use variable-length records for the index entries, with a repeating field for the pointer. We maintain a list of pointers in the index entry for K(i) - one pointer to each block that contains a record whose indexing field value equals K(i). In other words, an index entry will look like this: <K(i), [P(i, 1), P(i, 2), P(i, 3), ...]>.
    In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately.

- ***Option 3:*** This is the most commonly adopted approach. In this option, we keep the index entries themselves at a fixed-length and have a single entry for each indexing field value. Additionally, an extra level of indirection is created to handle the multiple pointers. Such an index is a ***sparse*** scheme, and the pointer P(i) in index entry <K(i), P(i)> points to a block of record pointers (this is the extra level); each record pointer in that block points to one of the data file blocks containing the record 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 block, a linked list of blocks is used.

Retrieval via the index requires one or more additional block accesses because of the extra level.

- A secondary index provides a logical ordering on the data records by the indexing field. If we access the records in order of the entries in the secondary index, the records can be retrieved in order of the indexing field values.



Multilevel indexing

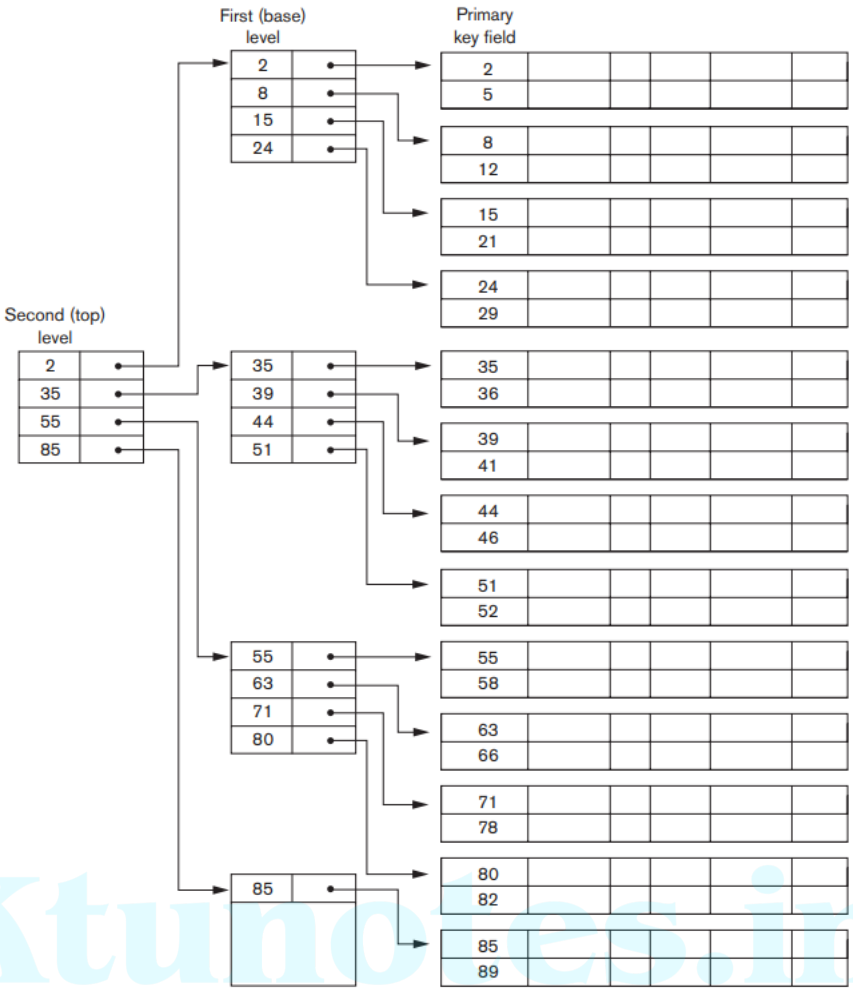
- A multilevel index considers the index file, referred to as the ***first (or base) level*** of the multilevel structure, as a sorted file with a distinct value for each K(i).
- An index file is a special type of data file with two fields. Thus, we can build a primary index for an index file itself (i.e. on top of the index at the first level). This new 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 index entries.
- The blocking factor bfri for the second level and for all subsequent levels is the same as that for the first-level index, because all index entries are of the same size; each has one field value and one block address.
  - If the first level has r1 entries, and the blocking factor which is also the fan-out for the index is bfri = fo, then the first level needs $\lceil (r1/fo) \rceil$ blocks, which is therefore the number of entries r2 needed at the second level of the index.
- The above process can be repeated and a third-level index can be created on top of the second-level one. The third level, which is a primary index for the second level, has an entry for each second-level block. Thus, the number of third-level entries is r3 =$\lceil (r2/fo) \rceil$

Important note: We require a second level only if the first level needs more than one block of disk storage, and similarly, we require a third level only if the second level needs more than one block.

- We can continue the index-building process until all the entries of index level **t** fit in a single block.

This block at the **t** th level is called the ***top index level***.

A multilevel index with r1 first-level entries will need t levels, where **t=** $\lceil (\log_{fo}(r1)) \rceil$**.**



- ✓ ***Example***: Consider a multilevel index, with r = 40,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed-length and are unspanned, with a record size R = 100 bytes. Key field of the file, V = 11 bytes long and block pointer is P = 8 bytes long.
  - The index blocking factor bfri =⌊(B/Ri)⌋ =⌊(1024/19)⌋ = 53 entries per block, which is also the fan-out for the multilevel index.
  - The number of first-level blocks b1 =⌈(ri/bfri)⌉ = ⌈(40,000/53)⌉ = 755.
  - Hence, the number of second-level blocks will be b2 = $\lceil(b1/fo)\rceil$= ⌈(755/53)⌉ = 15 blocks, and the number of third-level blocks will be b3 = $\lceil(b2/fo)\rceil$ = ⌈(15/53)⌉ = 1 block.
  - Now at the third level, because all index entries can be stored in a single block, it is also the top level and t= 3.

    t= $\lceil(\log_{fo}(r1))\rceil$. = $\lceil(\log_{53} 40{,}000)\rceil = 3$ where r1 = r = 40000.
- ✓ To search for a record based on a non-ordering key value using the multilevel index, we must access one block at each level plus one block from the data file. Thus, we need t + 1 = 3 + 1 = 4 block accesses.

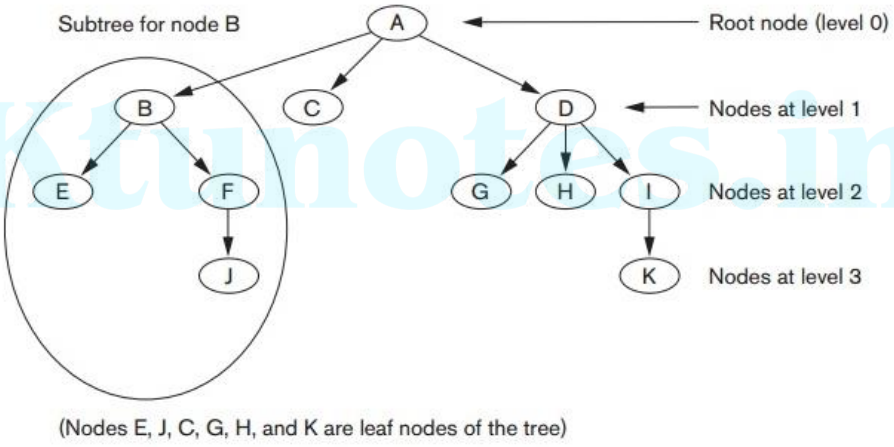
Dynamic Multilevel Indexes Using B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known search data structure known as a **tree**.

A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*.

A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node *n* and the subtrees of all the child nodes of *n*.

Figure given below illustrates a tree data structure.



Difference between B-trees and B+-trees

B-trees can be used as dynamic multilevel indexes to guide the search for records in a data file. ***Pointers to the data blocks are stored in both internal nodes and leaf nodes*** of the B-tree structure.

B+-trees, a variation of B-trees in which ***pointers to the data blocks of a file are stored only in leaf nodes,*** which can lead to fewer levels and higher-capacity indexes.

Search Trees

A search tree is slightly different from a multilevel index.

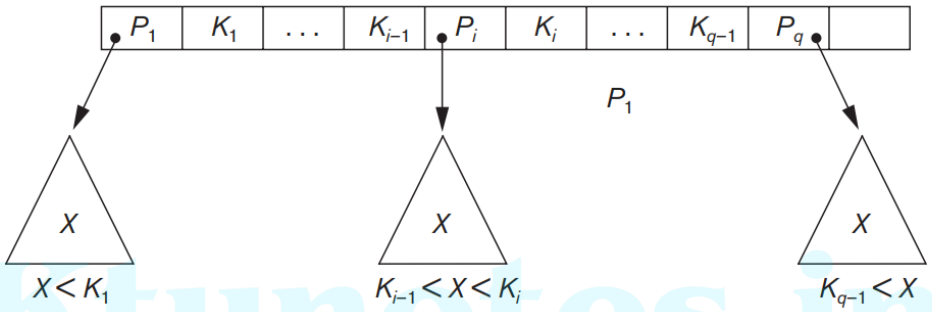
A search tree of **order p** is a tree such that each node contains **at most p – 1 search values** and p pointers in the order

$$\langle P_1, K_1, P_2, K_2, \ldots\ldots P_{q-1}, K_{q-1}, P_q \rangle, \text{ where } q \leq p.$$

Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values.

Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < \ldots < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$



- Whenever we search for a value X, we follow the appropriate pointer Pi according to the formulas in condition 2 above.
- We can **use a search tree** as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called **the search field** (which is the same as the index field if a multilevel index guides the search).
- Each **key value** in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record.

B-tree

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

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

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**—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 P_i, 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.

5. Each node, except the root and leaf nodes, has at least $(p/2)$ 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, $q \leq p$, 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* P_i are NULL.

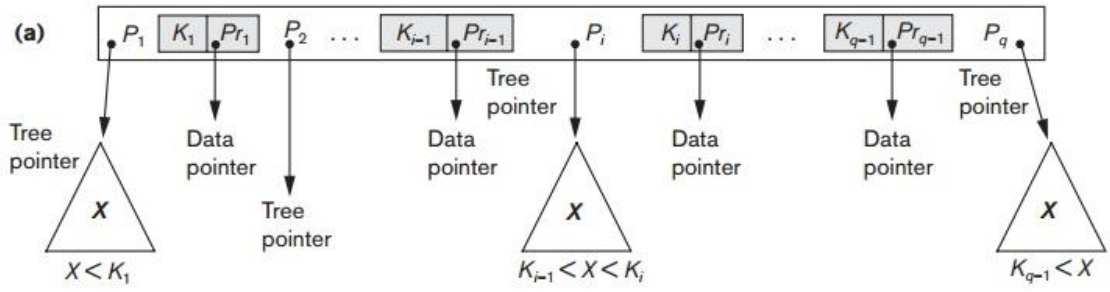
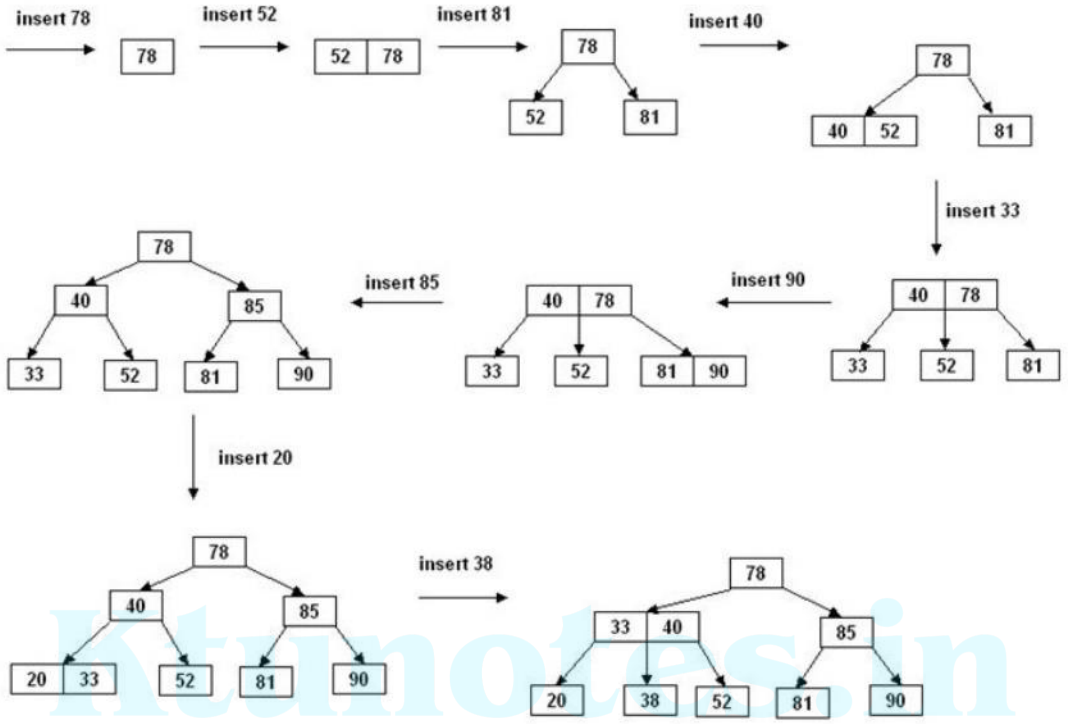


Figure: B-tree structures. (a) A node in a B-tree with q – 1 search values

- A **B-tree** starts with a single root node (which is also a leaf node) at level 0 (zero).
- 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.
- Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes.
- When a non-root 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.

- 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.

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



- 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.

B+-Trees

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.

The leaf nodes of the B+-tree are usually linked to provide ordered access on the search field to the records. These 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.

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, $K_1 < K_2 < ... < K_{q-1}$.

3. For all search field values X in the subtree pointed at by P_i, we have

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

4.Each internal node has at most p tree pointers.

5.Each internal node, except the root, has at least $(p/2)$ 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 is as follows:

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.

2. Within each leaf node, $K_1 \leq K_2 ... , K_{q-1}, q \leq p$

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 $(p/2)$ values.

5. All leaf nodes are at the same level.

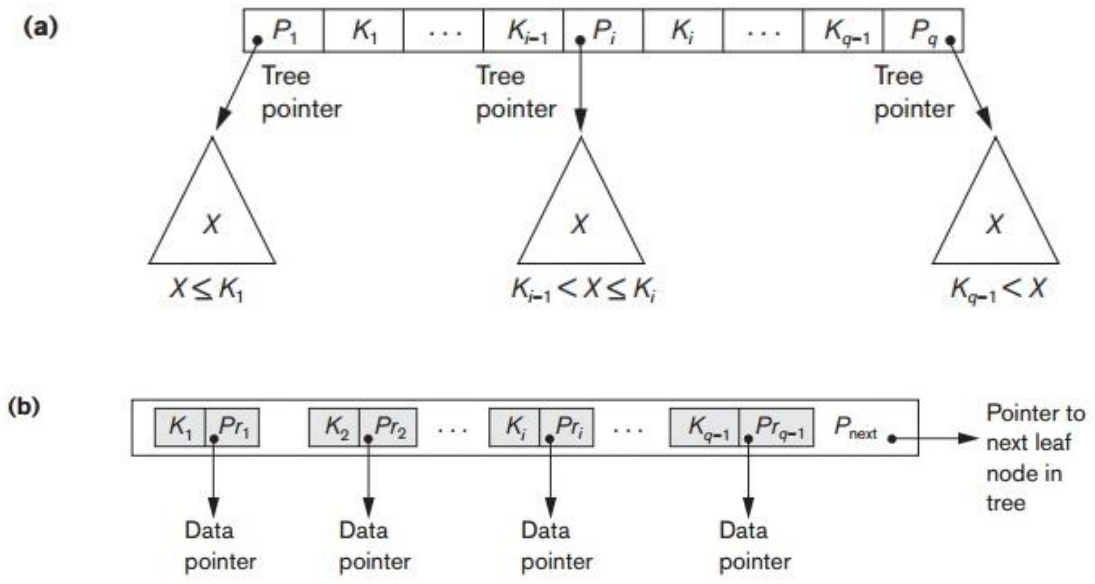
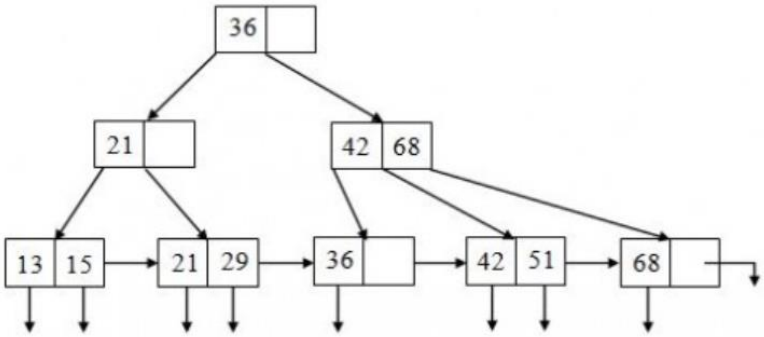


Figure
The nodes of a B^+-tree. (a) Internal node of a B^+-tree with $q - 1$ search values. (b) Leaf node of a B^+-tree with $q - 1$ search values and $q - 1$ data pointers.

Example:



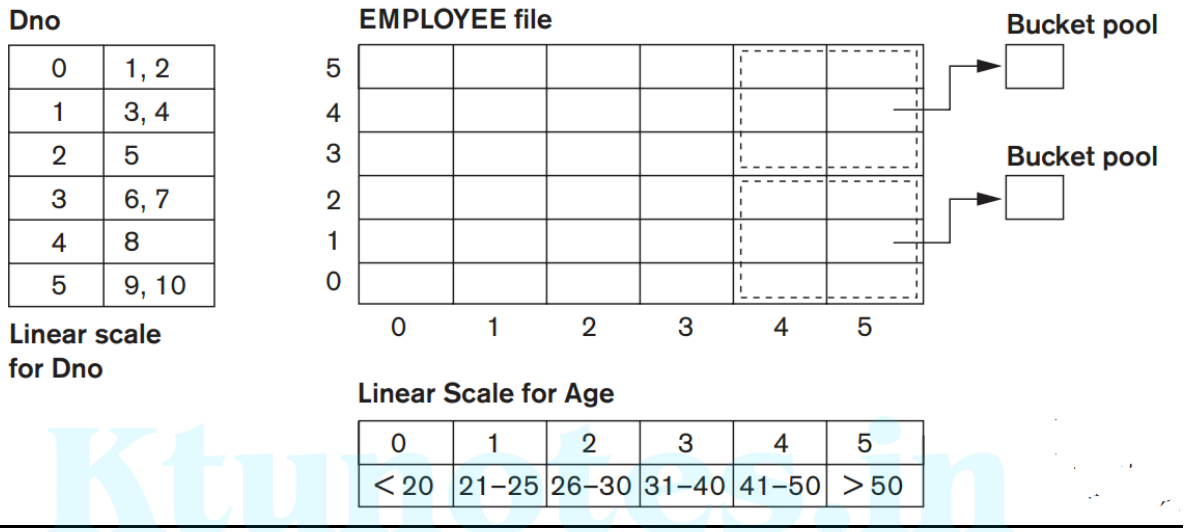
Indexes on Multiple Keys

- Till now, we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields).
- In many retrieval and update requests, multiple attributes are involved.
- If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.
- keys containing multiple attributes are referred to as **composite keys.**
- For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip_code, Salary and Skill_code, with the key of Ssn (Social Security number).
- Consider the query: List the employees in department number 4 whose age is 59.
- Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records.

Grid Files

- The EMPLOYEE file is organized as a grid file. If we want to access a file on two keys, say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes.
- Below figure shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute.

- The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, whereas Dno = 5 corresponds to the value 2 on that scale.
- Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age.
- The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored.



Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket.