

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

PREPARED BY SHARIKAT  
R, SNGCE

## SYLLABUS

- 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, Single 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

PREPARED BY SHARIKA T.R.  
SNGCE

## Data-manipulation language(DML)

- The SQL DML provides the ability to query information from the database and to **insert tuples into, delete tuples from, and modify tuples in the database.**
  - **Integrity**
    - The SQL DDL includes commands for specifying **integrity constraints** that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
  - **View definition**
    - The SQL DDL includes commands for **defining views.**
  - **Transaction control**
    - SQL includes commands for specifying the beginning and ending of **transactions.**

PREPARED BY SHARIKA T.R.  
SNGCE

## Basic Retrieval Queries in SQL

- SQL has one basic statement for retrieving information from a database; the **SELECT** statement
- **This is not the same as the SELECT operation of the relational algebra**
- Important distinction between SQL and the formal relational model;
  - SQL allows a table (relation) to have **two or more tuples that are identical** in all their attribute values
  - Hence, an SQL relation (table) is a **multi-set** (sometimes called a bag) of tuples;
  - it is not a set of tuples SQL relations can be **constrained to be sets** by specifying **PRIMARY KEY** or **UNIQUE** attributes, or by using the **DISTINCT** option in a query

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

- <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

**EMPLOYEE**

FNAME	MINIT	LNAME	<u>SSN</u>	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
-------	-------	-------	------------	-------	---------	-----	--------	----------	-----

**DEPARTMENT**

DNAME	<u>DNUMBER</u>	MGRSSN	MGRSTARTDATE
-------	----------------	--------	--------------

**DEPT\_LOCATIONS**

<u>DNUMBER</u>	<u>DLOCATION</u>
----------------	------------------

**PROJECT**

PNAME	<u>PNUMBER</u>	PLOCATION	DNUM
-------	----------------	-----------	------

**WORKS\_ON**

<u>ESSN</u>	<u>PNO</u>	HOURS
-------------	------------	-------

**DEPENDENT**

<u>ESSN</u>	<u>DEPENDENT_NAME</u>	SEX	BDATE	RELATIONSHIP
-------------	-----------------------	-----	-------	--------------

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith		123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong		333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya		999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace		987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan		666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English		453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar		987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg		888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPT_LOCATIONS	DNUMBER	DLOCATION
	1	Houston
	4	Stafford
	5	Bellaire
	5	Sugarland
	5	Houston

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research		5	333445555	1988-05-22
Administration		4	987654321	1995-01-01
Headquarters		1	888665555	1981-06-19

WORKS_ON	ESSN	PNO	HOURS
	123456789	1	32.5
	123456789	2	7.5
	666884444	3	40.0
	453453453	1	20.0
	453453453	2	20.0
	333445555	2	10.0
	333445555	3	10.0
	333445555	10	10.0
	333445555	20	10.0
	999887777	30	30.0
	999887777	10	10.0
	987987987	10	35.0
	987987987	30	5.0
	987654321	30	20.0
	987654321	20	15.0
	888665555	20	null

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
ProductX		1	Bellaire	5
ProductY		2	Sugarland	5
ProductZ		3	Houston	5
Computerization		10	Stafford	4
Reorganization		20	Houston	1
Newbenefits		30	Stafford	4

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

PREPARED BY SHARIKAT R.  
NIGGE

Q0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

**SELECT** Bdate, Address  
**FROM** EMPLOYEE  
**WHERE** Fname='John' AND Minit='B' AND Lname='Smith';

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith		123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong		333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya		999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace		987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan		666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English		453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar		987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg		888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

Bdate	Address
1965-01-09	731Fondren, Houston, TX

Q1. Retrieve the name and address of all employees who work for the 'Research' department.

**SELECT** Fname, Lname, Address  
**FROM** EMPLOYEE, DEPARTMENT  
**WHERE** Dname='Research' AND Dnumber=Dno;

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5	
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5	
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4	
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4	
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5	
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5	
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4	
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1	

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
Research	Research	5	333445555	1988-05-22
Administration	Administration	4	987654321	1995-01-01
Headquarters	Headquarters	1	888665555	1981-06-19

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Q 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

EMPLOYEE									
FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT			
DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS	
DNUMBER	DLOCATION

PROJECT			
PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON		
ESSN	PNO	HOURS

DEPENDENT				
ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

PREPARED BY SHARIKA T.R.  
SNGCE

**SELECT** Pnumber, Dnum, Lname, Address, Bdate  
**FROM** PROJECT, DEPARTMENT, EMPLOYEE  
**WHERE** Dnum=Dnumber **AND** Mgr\_ssn=Ssn **AND**  
Plocation='Stafford';

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPARTMENT	DNAME	DNUMBER	MGRSSN	MGRSTARTDATE
	Research	5	333445555	1988-05-22
	Administration	4	987654321	1995-01-01
	Headquarters	1	888665555	1981-06-19

PROJECT	PNAME	PNUMBER	PLOCATION	DNUM
	ProductX	1	Bellaire	5
	ProductY	2	Sugarland	5
	ProductZ	3	Houston	5
	Computerization	10	Stafford	4
	Reorganization	20	Houston	1
	Newbenefits	30	Stafford	4

**SELECT** Pnumber, Dnum, Lname, Address, Bdate  
**FROM** PROJECT, DEPARTMENT, EMPLOYEE  
**WHERE** Dnum=Dnumber **AND** Mgr\_ssn=Ssn **AND**  
Plocation='Stafford';

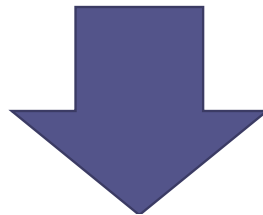
Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

## Ambiguous Attribute Names, Aliasing, Renaming, and Tuple Variables

- In SQL, we can use the same name for two (or more) attributes as long as the attributes are in different relations
- a **multitable query** refers to two or more attributes with the same name, we must **qualify the attribute name** with the relation name to **prevent ambiguity**
- This is done by **prefixing the relation name** to the attribute name and separating the two by a **period(.)**.

PREPARED BY SHARIKA T.R.  
SNGCE

**Q1:**     **SELECT**     Fname, Lname, Address  
             **FROM**       EMPLOYEE, DEPARTMENT  
             **WHERE**      Dname='Research' **AND** Dnumber=Dno;



**Q1A:**    **SELECT**       Fname, EMPLOYEE.Name, Address  
             **FROM**       EMPLOYEE, DEPARTMENT  
             **WHERE**      DEPARTMENT.Name='Research' **AND**  
                         DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;

PREPARED BY SHARIKA T.R.  
SNGCE

**Q8.** For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

**SELECT**     E.Fname, E.Lname, S.Fname, S.Lname  
**FROM**       EMPLOYEE **AS** E, EMPLOYEE **AS** S  
**WHERE**      E.Super\_ssn=S.Ssn;

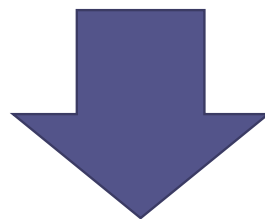
<u>E.Fname</u>	<u>E.Lname</u>	<u>S.Fname</u>	<u>S.Lname</u>
John	Smith	Franklin	Wong
Franklin	Wong	James	Borg
Alicia	Zelaya	Jennifer	Wallace
Jennifer	Wallace	James	Borg
Ramesh	Narayan	Franklin	Wong
Joyce	English	Franklin	Wong
Ahmad	Jabbar	Jennifer	Wallace



- Alternative relation names E and S are called aliases or tuple variables, for the EMPLOYEE relation.
- An alias follow the keyword AS
- It is also possible to rename the relation attributes within the query in SQL by giving them aliases.

EMPLOYEE **AS** E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno)

**Q1A:**    **SELECT**    Fname, EMPLOYEE.Name, Address  
          **FROM**        EMPLOYEE, DEPARTMENT  
          **WHERE**      DEPARTMENT.Name='Research' **AND**  
                         DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;



**SELECT**    E.Fname, E.LName, E.Address  
**FROM**        EMPLOYEE E, DEPARTMENT D  
**WHERE**      D.DName='Research' **AND** D.Dnumber=E.Dno;

## Unspecified WHERE Clause and Use of the Asterisk

- missing WHERE clause indicates no condition on tuple selection;
  - hence, all tuples of the relation specified in the FROM clause qualify and are selected for the query result
- If more than one relation is specified in the FROM clause and there is no WHERE clause, then the CROSS PRODUCT all possible tuple combinations of these relations is selected

Q 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

**Q9:**      **SELECT**      Ssn  
              **FROM**        EMPLOYEE;

**Q10:**     **SELECT**      Ssn, Dname  
              **FROM**        EMPLOYEE, DEPARTMENT;

(e)

E.Fname
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

Ssn	Dname
123456789	Research
333445555	Research
999887777	Research
987654321	Research
666884444	Research
453453453	Research
987987987	Research
888665555	Research
123456789	Administration
333445555	Administration
999887777	Administration
987654321	Administration
666884444	Administration
453453453	Administration
987987987	Administration
888665555	Administration
123456789	Headquarters
333445555	Headquarters
999887777	Headquarters
987654321	Headquarters
666884444	Headquarters
453453453	Headquarters
987987987	Headquarters
888665555	Headquarters

- To retrieve all the attribute values of the selected tuples, we do not have to list the attribute names explicitly in SQL;
- we just specify an asterisk (\*), which stands for all the attributes

Q1C retrieves all the attribute values of any  
EMPLOYEE who works in DEPARTMENT number  
5

**Q1C:      SELECT      \***  
             **FROM        EMPLOYEE**  
             **WHERE       Dno=5;**

(g)

<u>Fname</u>	<u>Minit</u>	<u>Lname</u>	<u>Ssn</u>	<u>Bdate</u>	<u>Address</u>	<u>Sex</u>	<u>Salary</u>	<u>Super_ssn</u>	<u>Dno</u>
John	B	Smith	123456789	1965-09-01	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

PREPARED BY SHARIKA T.R.  
SNGCE

Q1D Retrieves all the attributes of an EMPLOYEE and the attributes of the DEPARTMENT in which he or she works for every employee of the 'Research' department

```
Q1D:  SELECT *
      FROM EMPLOYEE, DEPARTMENT
      WHERE Dname='Research' AND Dno=Dnumber;
```

PREPARED BY SHARIKA T.R.  
SNGCE

Q10A specifies the CROSS PRODUCT of the EMPLOYEE and DEPARTMENT relations

```
Q10A:  SELECT *
      FROM EMPLOYEE, DEPARTMENT;
```

## Tables as Sets in SQL

- SQL usually treats a table not as a set but rather as a multiset;
  - duplicate tuples can appear more than once in a table, and in the result of a query.
- SQL does not automatically eliminate duplicate tuples in the results of queries, for the following reasons
  - **Duplicate elimination is an expensive operation.**
    - One way to implement it is to **sort the tuples first and then eliminate duplicates.**
  - The user may want to see duplicate tuples in the result of a query.
  - When an **aggregate function** is applied to tuples, in most cases we do not want to eliminate duplicates

## DISTINCT Keyword

- to eliminate duplicate tuples from the result of an SQL querys we use the keyword DISTINCT in the SELECT clause
- only distinct tuples should remain in the result
- a query with SELECT DISTINCT eliminates duplicates, whereas a query with SELECT ALL does not.
- SELECT with neither ALL nor DISTINCT is equivalent to SELECT ALL

PREPARED BY SHARIKA T.R.  
SNGCE

Q11 retrieves the salary of every employee without distinct

**Q11: SELECT ALL Salary  
FROM EMPLOYEE;**

(a)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

Salary
30000
40000
25000
43000
38000
25000
25000
55000

PREPARED BY SHARIKA T.R.  
SNGCE

Q11A retrieves the salary of every employee using keyword DISTINCT

**Q11A: SELECT DISTINCT Salary  
FROM EMPLOYEE;**

Duplicate  
eliminated

(b)

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

Salary
30000
40000
25000
43000
38000
55000

## UNION, EXCEPT and INTERSECT

- set union (UNION), set difference (EXCEPT), and set intersection (INTERSECT) operations.
- The relations resulting from these set operations are sets of tuples; that is, duplicate tuples are eliminated from the result.
- These set operations apply only to union-compatible relations, so we must make sure that the two relations on which we apply the operation have the same attributes and that the attributes appear in the same order in both relations

**Q4.** Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project

```
Q4A: ( SELECT   DISTINCT Pnumber
      FROM      PROJECT, DEPARTMENT, EMPLOYEE
      WHERE     Dnum=Dnumber AND Mgr_ssn=Ssn
              AND Lname='Smith' )

      UNION
      ( SELECT   DISTINCT Pnumber
      FROM      PROJECT, WORKS_ON, EMPLOYEE
      WHERE     Pnumber=Pno AND Essn=Ssn
              AND Lname='Smith' );
```

The first SELECT query retrieves the projects that involve a 'Smith' as manager of the department that controls the project, and the second retrieves the projects that involve a 'Smith' as a worker on the project. Notice that if several employees have the last name 'Smith', the project names involving any of them will be retrieved. Applying the UNION operation to the two SELECT queries gives the desired result.

## UNION ALL

- The UNION ALL command combines the result set of two or more SELECT statements (allows duplicate values).
- The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```
SELECT supplier_id  
FROM suppliers  
UNION ALL  
SELECT supplier_id  
FROM orders  
ORDER BY supplier_id;
```

This SQL UNION ALL example would return the **supplier\_id multiple times** in the result set if that same value appeared in both the suppliers and orders table. The **SQL UNION ALL operator does not remove duplicates**. If you wish to remove duplicates, try using the UNION operator.

If you had the *suppliers* table populated with the following records: And the *orders* table populated with the following records

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

order_id	order_date	supplier_id
1	2015-08-01	2000
2	2015-08-01	6000
3	2015-08-02	7000
4	2015-08-03	8000



PREPARED BY SHARIKA T R,

If you had the *suppliers* table populated with the following records: And the *orders* table populated with the following records

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

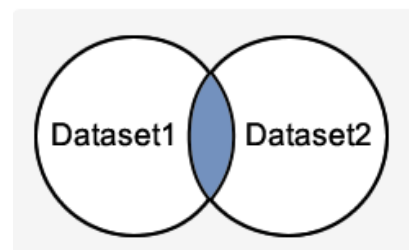
order_id	order_date	supplier_id
1	2015-08-01	2000
2	2015-08-01	6000
3	2015-08-02	7000
4	2015-08-03	8000

supplier_id
1000
2000
2000
3000
4000
6000
7000
8000

PREPARED BY SHARIKA T R,  
SNGCE

## INTERSECT Operator

- INTERSECT operator is used to return the records that are in common between two SELECT statements or data sets.
- If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.
- It is the intersection of the two SELECT statements.



```
SELECT supplier_id
FROM suppliers
INTERSECT
SELECT supplier_id
FROM orders;
```

- In this SQL INTERSECT example, if a `supplier_id` appeared in both the `suppliers` and `orders` table, it would appear in your result set.

```
SELECT supplier_id
FROM suppliers
WHERE supplier_id > 78
INTERSECT
SELECT supplier_id
FROM orders
WHERE quantity <> 0;
```

In this example, the WHERE clauses have been added to each of the datasets. The first dataset has been filtered so that only records from the *suppliers* table where the *supplier\_id* is greater than 78 are returned. The second dataset has been filtered so that only records from the *orders* table are returned where the *quantity* is not equal to 0.

# EXCEPT

- The SQL EXCEPT clause/operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.
  - This means EXCEPT returns only rows, which are not available in the second SELECT statement.
- Just as with the UNION operator, the same rules apply when using the EXCEPT operator.

## Syntax

The basic syntax of **EXCEPT** is as follows.

```
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]  
  
EXCEPT  
  
SELECT column1 [, column2 ]  
FROM table1 [, table2 ]  
[WHERE condition]
```

Here, the given condition could be any given expression based on your requirement.

## Example

Consider the following two tables.

**Table 1** – CUSTOMERS Table is as follows.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

**Table 2** – ORDERS table is as follows.

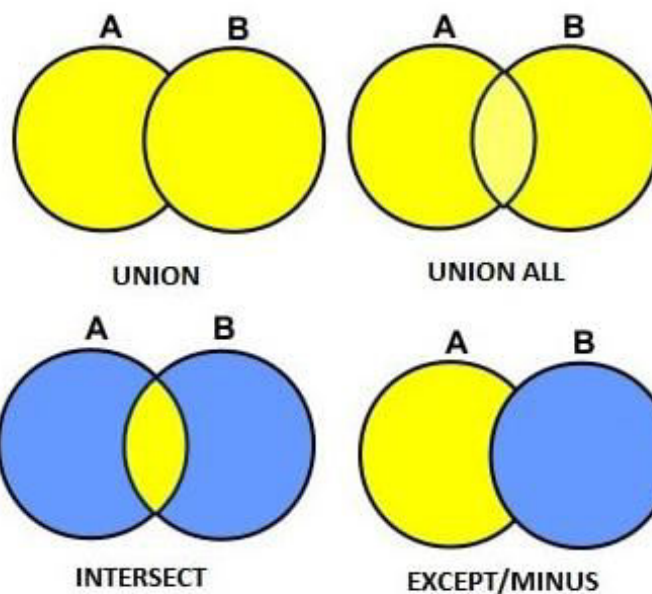
OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

PREPARED BY SHARIKA T.R.  
SNGCE

```
SQL> SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
LEFT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
SELECT ID, NAME, AMOUNT, DATE
FROM CUSTOMERS
RIGHT JOIN ORDERS
ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL



PREPARED BY SHARIKA T.R.  
SNGCE

## Substring Pattern Matching and Arithmetic Operators

- **LIKE comparison operator**
  - This can be used for string pattern matching
  - Partial strings are specified using two reserved characters
    - % replaces an arbitrary number of zero or more characters, and
    - the underscore ( ) replaces a single character

**Q 12.** Retrieve all employees whose address is in Houston, Texas.

```
Q12:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Address LIKE '%Houston,TX%';
```

## Q 12A. Find all employees who were born during the 1950s.

```
Q12:  SELECT  Fname, Lname
       FROM    EMPLOYEE
       WHERE   Bdate LIKE '__ 5 _____';
```

- To retrieve all employees who were born during the 1950s,
- Here, '5' must be the third character of the string (according to our format for date),
- so we use the value '\_\_ 5 \_\_\_\_\_', with each underscore serving as a placeholder for an arbitrary character.

- If an underscore or % is needed as a literal character in the string, the character should be preceded by an escape character, which is specified after the string using the keyword ESCAPE.
- For example, 'AB\\_CD\%EF' ESCAPE '\ ' represents the literal string 'AB\_CD%EF' because \ is specified as the escape character.
- Any character not used in the string can be chosen as the escape character.
- Also, we need a rule to specify apostrophes or single quotation marks ( ' ) if they are to be included in a string because they are used to begin and end strings.
- If an apostrophe ( ' ) is needed, it is represented as two consecutive apostrophes ( '' ) so that it will not be interpreted as ending the string.

- The standard arithmetic operators for addition (+), subtraction (−), multiplication (\*), and division (/) can be applied to numeric values or attributes with numeric domains

**Q13.** Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
Q13:  SELECT  E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal  
FROM    EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P  
WHERE   E.Ssn=W.Essn AND W.Pno=P.Pnumber AND  
        P.Pname='ProductX';
```

suppose that we want to see the effect of giving all employees who work on the 'ProductX' project a 10 percent raise; we can issue Query 13 to see what their salaries would become. This example also shows how we can rename an attribute in the query result using AS in the SELECT clause.

- For string data types, the concatenate operator || can be used in a query to append two string values.
- For date, time, timestamp, and interval data types, operators include incrementing (+) or decrementing (–) a date, time, or timestamp by an interval.
- In addition, an interval value is the result of the difference between two date, time, or timestamp values.
- Another comparison operator, which can be used for convenience, is BETWEEN

**Q 14.** Retrieve all employees in department 5 whose salary is between \$30,000 and \$40,000.

```
Q14:  SELECT *  
      FROM   EMPLOYEE  
      WHERE  (Salary BETWEEN 30000 AND 40000) AND Dno = 5;
```

The condition (Salary BETWEEN 30000 AND 40000) in Q14 is equivalent to the condition ((Salary >= 30000) AND (Salary <= 40000))



## Ordering of Query Results

- The **ORDER BY** statement in sql is used to **sort the fetched data in either ascending or descending** according to one or more columns.
  - By default ORDER BY sorts the data in ascending order.
  - We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.

Q. Fetch all data from the table Student and sort the result in descending order according to the column ROLL\_NO

```
SELECT * FROM Student ORDER BY ROLL_NO DESC;
```

Student Table	ROLL_NO	NAME	ADDRESS	PHONE	Age
	1	HARSH	DELHI	XXXXXXXXXX	18
	2	PRATIK	BIHAR	XXXXXXXXXX	19
	3	RIYANKA	SILIGURI	XXXXXXXXXX	20
	4	DEEP	RAMNAGAR	XXXXXXXXXX	18
	5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
	6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
	7	ROHIT	BALURGHAT	XXXXXXXXXX	18
	8	NIRAJ	ALIPUR	XXXXXXXXXX	19

PREPARED BY SHARIKA T R,

```
SELECT * FROM Student ORDER BY ROLL_NO DESC;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
8	NIRAJ	ALIPUR	XXXXXXXXXX	19
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
3	RIYANKA	SILIGURI	XXXXXXXXXX	20
2	PRATIK	BIHAR	XXXXXXXXXX	19
1	HARSH	DELHI	XXXXXXXXXX	18

## Sort according to multiple columns

PREPARED BY SHARIKA T R,  
SNGCE

Q. Fetch all data from the table Student and then sort the result in ascending order first according to the column Age. and then in descending order according to the column ROLL\_NO.

```
SELECT * FROM Student ORDER BY Age ASC , ROLL_NO DESC;
```

Output:

ROLL_NO	NAME	ADDRESS	PHONE	Age
7	ROHIT	BALURGHAT	XXXXXXXXXX	18
4	DEEP	RAMNAGAR	XXXXXXXXXX	18
1	HARSH	DELHI	XXXXXXXXXX	18
8	NIRAJ	ALIPUR	XXXXXXXXXX	19
5	SAPTARHI	KOLKATA	XXXXXXXXXX	19
2	PRATIK	BIHAR	XXXXXXXXXX	19
6	DHANRAJ	BARABAJAR	XXXXXXXXXX	20
3	RIYANKA	SILIGURI	XXXXXXXXXX	20

Q15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

```
Q15:  SELECT  D.Dname, E.Lname, E.Fname, P.Pname
      FROM    DEPARTMENT D, EMPLOYEE E, WORKS_ON W,
              PROJECT P
      WHERE   D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
              W.Pno= P.Pnumber
      ORDER BY D.Dname, E.Lname, E.Fname;
```

## NESTING OF QUERIES

- A complete SELECT query, called a nested query , can be specified within the WHERE-clause of another query, called the outer query
- Many of the previous queries can be specified in an alternative form using nesting
- ☐

**Query 1:** Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1: SELECT FNAME,LNAME, ADDRESS  
FROM EMPLOYEE  
WHERE DNO IN (SELECT DNUMBER  
FROM DEPARTMENT  
WHERE DNAME='Research')
```

- The nested query selects the number of the 'Research' department
- The outer query select an EMPLOYEE tuple if its DNO value is in the result of either nested query
- The comparison operator IN compares a value v with a set (or multi-set) of values V, and evaluates to TRUE if v is one of the elements in V
- In general, we can have several levels of nested queries
- A reference to an unqualified attribute refers to the relation declared in the innermost nested query
- In this example, the nested query is not correlated with the outer query

PREPARED BY SHARIKA T R,  
SNGCE

**Q.** Returns 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 );
```

## CORRELATED NESTED QUERIES

- If a condition in the WHERE-clause of a nested query **references an attribute of a relation declared in the outer query**, the two queries are said to be correlated
- The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) the outer query

Query: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
SELECT  E.FNAME, E.LNAME
FROM    EMPLOYEE AS E
WHERE   E.SSN IN (SELECT  ESSN
                  FROM    DEPENDENT
                  WHERE   ESSN=E.SSN AND
                          E.FNAME=DEPENDENT_NAME)
```

Here the nested query has a different result for each tuple in the outer query.

A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can always be expressed as a single block query.

## A Simple Retrieval Query in SQL

- A simple retrieval query in SQL can consist of up to four clauses,
  - but only the first two **SELECT** and **FROM** are mandatory
  - the clauses between square brackets [ ... ] being optional:

```
SELECT    <attribute list>
FROM      <table list>
[ WHERE    <condition> ]
[ ORDER BY <attribute list> ];
```

- The **SELECT**-clause lists the attributes or functions to be retrieved
- The **FROM**-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The **WHERE**-clause specifies the conditions for selection and join of tuples from the relations specified in the **FROM**-clause
- **GROUP BY** specifies grouping attributes
- **HAVING** specifies a condition for selection of groups
- **ORDER BY** specifies an order for displaying the result of a query
- A query is evaluated by first applying the **WHERE**-clause, then **GROUP BY** and **HAVING**, and finally the **SELECT**-clause
- There are three SQL commands to modify the database; **INSERT**, **DELETE**, and **UPDATE**

## The EXISTS and UNIQUE Functions in SQL

- **EXISTS** function in SQL is used to **check whether the result of a correlated nested query is empty** (contains no tuples) or not
- The result of EXISTS is a Boolean value
  - **TRUE** if the nested query result contains **at least one tuple**, or
  - **FALSE** if the nested query result contains **no tuples**.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE EXISTS
( SELECT *
  FROM DEPENDENT AS D
  WHERE E.Ssn=D.Essn AND E.Sex=D.Sex
);
```

Fname	Lname
John	Smith
Franklin	Wong
Ahamad	Jabbar

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

OUTPUT

employee (2×3)	
Fname	Lname
John	Smith
Franklin	Wong
Ahamad	Jabbar

PREPARED BY SHARIKA T R,  
SNGCE

- **EXISTS(Q)** returns **TRUE** if there is at least one tuple in the result of the nested query Q, and it returns **FALSE** otherwise.
- On the other hand, **NOT EXISTS(Q)** returns **TRUE** if there are **no tuples in the result of nested query Q**, and it returns **FALSE** otherwise



Q. Retrieve the names of employees who have no dependents.

Q6:      **SELECT**      Fname, Lname  
         **FROM**        EMPLOYEE  
         **WHERE**       **NOT EXISTS** ( **SELECT**      \*  
   **FROM**        DEPENDENT  
   **WHERE**        Ssn=Essn );

employee (2×5)	
Fname	Lname
Joyce	English
Ramesh	Narayan
James	Borg
Jennifer	Wallace
Alicia	Zelaya

EMPLOYEE	FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO
	John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
	Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
	Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
	Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
	Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
	Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
	Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
	James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPENDENT	ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP
	333445555	Alice	F	1986-04-05	DAUGHTER
	333445555	Theodore	M	1983-10-25	SON
	333445555	Joy	F	1958-05-03	SPOUSE
	987654321	Abner	M	1942-02-28	SPOUSE
	123456789	Michael	M	1988-01-04	SON
	123456789	Alice	F	1988-12-30	DAUGHTER
	123456789	Elizabeth	F	1967-05-05	SPOUSE

OUTPUT

employee (2×5)	
Fname	Lname
Joyce	English
Ramesh	Narayan
James	Borg
Jennifer	Wallace
Alicia	Zelaya

## Aggregate Functions in SQL

- Aggregate functions are used to summarize information from multiple tuples into a single-tuple summary.
- Grouping is used to create sub-groups of tuples before summarization.
- A number of built-in aggregate functions exist:
  1. COUNT,
  2. SUM,
  3. MAX,
  4. MIN, and
  5. AVG

- COUNT function returns the number of tuples or values as specified in a query.
- The functions SUM, MAX, MIN, and AVG can be applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average (mean) of those values
- These functions can be used in the SELECT clause or in a HAVING clause

PREPARED BY SHARIKA T.R.  
SNGCE

**Q19.** Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

**Q19:**    **SELECT**    **SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)**  
          **FROM**        **EMPLOYEE;**

Result #1 (4×1)			
SUM(Salary)	MAX(Salary)	MIN(Salary)	AVG(Salary)
281,000	55,000	25,000	35,125.0000

PREPARED BY SHARIKA T.R.  
SNGCE

**Q21.** Retrieve the total number of employees in the company

**Q21:**    **SELECT**    **COUNT (\*)**  
          **FROM**        **EMPLOYEE;**

SELECT COUNT(\*)  
FROM EMPLOYEE;

Result #1 (1×1)	
COUNT(*)	8

## Q22. The number of employees in the 'Research' department

**Q22:**    **SELECT**    **COUNT (\*)**  
          **FROM**        **EMPLOYEE, DEPARTMENT**  
          **WHERE**      **DNO=DNUMBER AND DNAME='Research';**

Result #1 (1×1)	
COUNT(*)	4

Here the asterisk (\*) refers to the rows (tuples), so COUNT (\*) returns the number of rows in the result of the query. We may also use the COUNT function to count values in a column rather than tuples

## Query 23. Count the number of distinct salary values in the database.

**Q23:**    **SELECT**    **COUNT (DISTINCT Salary)**  
          **FROM**        **EMPLOYEE;**

Result #1 (1×1)	
COUNT(DISTINCT Salary)	6

If we write COUNT(SALARY) instead of COUNT(DISTINCT SALARY) then **duplicate values will not be eliminated**. However, any tuples with NULL for SALARY will not be counted. In general, **NULL values are discarded** when aggregate functions are applied to a particular column

## Q. Retrieve the names of all employees who have two or more dependents

Q5:     **SELECT**     Lname, Fname  
          **FROM**     EMPLOYEE  
          **WHERE**    ( **SELECT**   **COUNT** (\*)  
                      **FROM** DEPENDENT  
                      **WHERE**   Ssn=Essn ) >= 2;

employee (2×2)	
Lname	Fname
Smith	John
Wong	Franklin

The correlated nested query counts the number of dependents that each employee has; if this is greater than or equal to two, the employee tuple is selected.

## Grouping: The GROUP BY and HAVING Clauses

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.
- For example,
  - we may want to find the average salary of employees in each department or
  - the number of employees who work on each project.
- In these cases we need to **partition the relation into nonoverlapping subsets** (or groups) of tuples.
- Each group (partition) will consist of the tuples that have the same value of some attributes called the **grouping attributes**.

## GROUP BY

- The GROUP BY clause **specifies the grouping attributes**, which should also appear in the SELECT clause,
- so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attributes

**Q24.** For each department, retrieve the department number, the number of employees in the department, and their average salary

```
Q24:  SELECT  Dno, COUNT (*), AVG (Salary)  
        FROM  EMPLOYEE  
        GROUP BY Dno;
```

the EMPLOYEE tuples are partitioned into groups—each group having the same value for the grouping attribute Dno. Hence, each group contains the employees who work in the same department.

PREPARED BY SHARIKA T.R.  
SNGCE

(a)

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

If NULLs exist in the grouping attribute, then a separate group is created for all tuples with a NULL value in the grouping attribute.  
For example, if the EMPLOYEE table had some tuples that had NULL for the grouping attribute Dno, there would be a separate group for those tuples in the result of Q24

PREPARED BY SHARIKA T.R.  
SNGCE

**Query 25.** For each project, retrieve the project number, the project name, and the number of employees who work on that project.

**Q25:**    **SELECT**    Pnumber, Pname, **COUNT (\*)**  
              **FROM**    PROJECT, WORKS\_ON  
              **WHERE**    Pnumber=Pno  
              **GROUP BY** Pnumber, Pname;

PREPARED BY SHARIKA T.R.  
SNGCE

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

project (3x6)		
Pnumber	Pname	COUNT(*)
1	ProductX	2
2	ProductY	3
3	ProductZ	2
10	Computerization	3
20	Reorganization	3
30	Newbenefits	3

PREPARED BY SHARIKA T.R.  
SNGCE

## HAVING clause

- In GROUP BY the grouping and functions are applied after the joining of the two relations.
- Sometimes we want to retrieve the values of these functions only for groups that satisfy certain conditions
- HAVING clause, which can appear in conjunction with a GROUP BY clause
- HAVING provides a condition on the summary information regarding the group of tuples associated with each value of the grouping attributes.
- Only the groups that satisfy the condition are retrieved in the result of the query.



Q 26. For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

**Q26:**    **SELECT**    Pnumber, Pname, **COUNT (\*)**  
              **FROM**    PROJECT, WORKS\_ON  
              **WHERE**    Pnumber=Pno  
              **GROUP BY** Pnumber, Pname  
              **HAVING**   **COUNT (\*) > 2;**

After applying the WHERE clause but before applying HAVING

(b)

Pname	Pnumber	...	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	...	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

## After applying the HAVING clause condition

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26  
(Pnumber not shown)

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

**Q27:**    **SELECT**    Pnumber, Pname, COUNT (\*)  
              **FROM**     PROJECT, WORKS\_ON, EMPLOYEE  
              **WHERE**    Pnumber=Pno AND Ssn=Essn AND Dno=5  
              **GROUP BY** Pnumber, Pname;

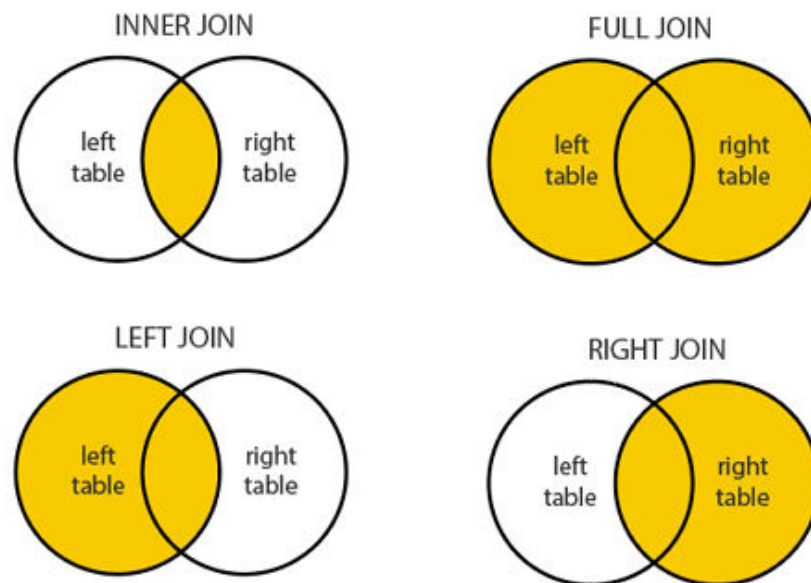
project (3x5)		
Pnumber	Pname	COUNT(*)
1	ProductX	2
2	ProductY	3
3	ProductZ	2
10	Computerization	1
20	Reorganization	1

# SQL JOIN

- How do I get data from multiple tables?
  - A SQL JOIN combines records from two tables.
  - A JOIN locates related column values in the two tables.
  - A query can contain zero, one, or multiple JOIN operations.
  - INNER JOIN is the same as JOIN; the keyword INNER is optional

- Four different types of JOINS
  1. (INNER) JOIN:
    - Select records that have matching values in both tables.
  2. FULL (OUTER) JOIN:
    - Selects all records that match either left or right table records.
  3. LEFT (OUTER) JOIN:
    - Select records from the first (left-most) table with matching right table records.
  4. RIGHT (OUTER) JOIN:
    - Select records from the second (right-most) table with matching left table records

PREPARED BY SHARIKA T R,  
SNGCE



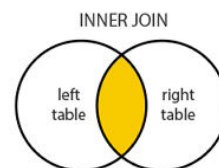
PREPARED BY SHARIKA T R,  
SNGCE

## JOIN Syntax

- The general syntax is

```
SELECT column-names  
FROM table-name1 INNER JOIN table-name2  
ON column-name1 = column-name2  
WHERE condition
```

- The INNER keyword is optional: it is the default as well as the most commonly used JOIN operation.



## Example

- **Problem: List all orders with customer information**

```
SELECT OrderNumber, TotalAmount, FirstName, LastName, City,
Country
```

```
FROM Order JOIN Customer
```

```
ON Order.CustomerId = Customer.Id
```

OrderNumber	TotalAmount	FirstName	LastName	City	Country
542378	440.00	Paul	Henriot	Reims	France
542379	1863.40	Karin	Josephs	Münster	Germany
542380	1813.00	Mario	Pontes	Rio de Janeiro	Brazil
542381	670.80	Mary	Saveley	Lyon	France
542382	3730.00	Pascale	Cartrain	Charleroi	Belgium
542383	1444.80	Mario	Pontes	Rio de Janeiro	Brazil
542384	625.20	Yang	Wang	Bern	Switzerland
...					

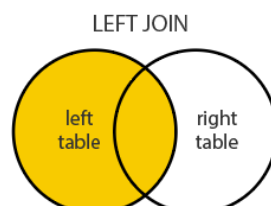
ORDER	
Id	PK
OrderDate	
OrderNumber	
CustomerId	
TotalAmount	

CUSTOMER	
Id	PK
FirstName	
LastName	
City	
Country	
Phone	

PREPARED BY SHARIKA T R,  
SNGCE

## SQL LEFT JOIN

- A LEFT JOIN performs a join starting with the first (left-most) table.
- Then, any matched records from the second table (right-most) will be included.
- LEFT JOIN and LEFT OUTER JOIN are the same.



## The SQL LEFT JOIN syntax

- The general LEFT JOIN syntax is

```
SELECT column-names
  FROM table-name1 LEFT JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

- The general LEFT OUTER JOIN syntax is

```
SELECT column-names
  FROM table-name1 LEFT OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

## SQL LEFT JOIN Example

CUSTOMER	ORDER
Id	Id
FirstName	OrderDate
LastName	OrderNumber
City	CustomerId
Country	TotalAmount
Phone	

- Problem: List all customers and the total amount they spent irrespective whether they placed any orders or not.

```
SELECT OrderNumber, TotalAmount, FirstName, LastName, City,
Country
  FROM Customer C LEFT JOIN [Order] O
    ON O.CustomerId = C.Id
 ORDER BY TotalAmount
```

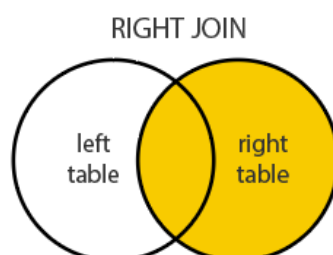
- The ORDER BY TotalAmount shows the customers without orders first (i.e. TotalMount is NULL).

			CUSTOMER		
			Id	PK	
			FirstName		
			LastName		
			City		
			Country		
			Phone		
			ORDER		
			Id	PK	
			OrderDate		
			OrderNumber		
			CustomerId		
			TotalAmount		
OrderNumber	TotalAmount	FirstName	LastName	City	Country
NULL	NULL	Diego	Roel	Madrid	Spain
NULL	NULL	Marie	Bertrand	Paris	France
542912	12.50	Patricio	Simpson	Buenos Aires	Argentina
542937	18.40	Paolo	Accorti	Torino	Italy
542897	28.00	Pascale	Cartrain	Charleroi	Belgium
542716	28.00	Maurizio	Moroni	Reggio Emilia	Italy
543028	30.00	Yvonne	Moncada	Buenos Aires	Argentina
543013	36.00	Fran	Wilson	Portland	USA

PREPARED BY SHARIKAT R.  
SNGCE

## SQL RIGHT JOIN

- A RIGHT JOIN performs a join starting with the second (right-most) table and then any matching first (left-most) table records. RIGHT JOIN and RIGHT OUTER JOIN are the same.



## The SQL RIGHT JOIN syntax

- The general syntax is

```
SELECT column-names
  FROM table-name1 RIGHT JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

The general RIGHT OUTER JOIN syntax is:

```
SELECT column-names
  FROM table-name1 RIGHT OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

## SQL RIGHT JOIN Examples

- Problem: List customers that have not placed orders

```
SELECT TotalAmount, FirstName, LastName, City,
Country
  FROM [Order] O RIGHT JOIN Customer C
    ON O.CustomerId = C.Id
 WHERE TotalAmount IS NULL
```

ORDER	
Id	PK
OrderDate	
OrderNumber	
CustomerId	
TotalAmount	

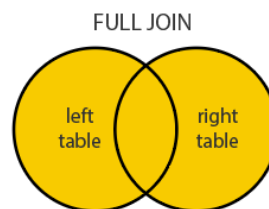
CUSTOMER	
Id	PK
FirstName	
LastName	
City	
Country	
Phone	

TotalAmount	FirstName	LastName	City	Country
NULL	Diego	Roel	Madrid	Spain
NULL	Marie	Bertrand	Paris	France



## SQL FULL JOIN Statement

- FULL JOIN returns all matching records from both tables whether the other table matches or not.
- Be aware that a FULL JOIN can potentially return very large datasets.
- These two: FULL JOIN and FULL OUTER JOIN are the same.



## The SQL FULL JOIN syntax

- The general syntax is:

```
SELECT column-names
  FROM table-name1 FULL JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

- The general FULL OUTER JOIN syntax is:

```
SELECT column-names
  FROM table-name1 FULL OUTER JOIN table-name2
    ON column-name1 = column-name2
 WHERE condition
```

## SQL FULL JOIN Examples

PREP.

CUSTOMER	
Id	≠0
FirstName	
LastName	
City	
Country	
Phone	

SUPPLIER	
Id	≠0
CompanyName	
ContactName	
City	
Country	
Phone	
Fax	

- **Problem: Match all customers and suppliers by country**

```
SELECT C.FirstName, C.LastName, C.Country AS
CustomerCountry,
      S.Country AS SupplierCountry, S.CompanyName
FROM Customer C FULL JOIN Supplier S
ON C.Country = S.Country
ORDER BY C.Country, S.Country
```

- This returns suppliers that have no customers in their country and customers that have no suppliers in their country, and customers and suppliers that are from the same country.

**Q 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.**

**Q20:    SELECT    SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)**  
**FROM        (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)**  
**WHERE        Dname='Research';**

Result #1 (4×1)			
SUM(Salary)	MAX(Salary)	MIN(Salary)	AVG(Salary)
133,000	40,000	25,000	33,250.0000

**Q26:**    **SELECT**    Pnumber, Pname, **COUNT (\*)**  
          **FROM**     PROJECT, WORKS\_ON  
          **WHERE**    Pnumber=Pno  
          **GROUP BY** Pnumber, Pname  
          **HAVING**   **COUNT (\*) > 2;**

## SQL Query Order of Execution

ORDER	CLAUSE	FUNCTION
1	from	Choose and join tables to get base data.
2	where	Filters the base data.
3	group by	Aggregates the base data.
4	having	Filters the aggregated data.
5	select	Returns the final data.
6	order by	Sorts the final data.
7	limit	Limits the returned data to a row count.

PREPARED BY SHARIKA T R,  
SNGCE

## Views (Virtual Tables) 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.

- We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.
- Views, which are a type of virtual tables allow users to do the following
  - Structure data in a way that users or classes of users find natural or intuitive.
  - Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
  - Summarize data from various tables which can be used to generate reports.

## Example

- In COMPANY database we may frequently issue queries that retrieve the employee name and the project names that the employee works on.
- Rather than having to specify the join of the three tables EMPLOYEE, WORKS\_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins.
- Then we can issue queries on the view, which are specified as single\_xxxx2\_table retrievals rather than as retrievals involving two joins on three tables.
- We call the EMPLOYEE, WORKS\_ON, and PROJECT tables the defining tables of the view.

# Specification of Views in SQL

- In SQL, the command to specify a view is **CREATE VIEW**.
- The view is given a (virtual) table name (or view name), a list of attribute names, and a query to specify the contents of the view.
- If none of the view attributes results from applying functions or arithmetic operations,
  - we do not have to specify new attribute names for the view,
  - since they would be the same as the names of the attributes of the defining tables in the default case

```

V1:  CREATE VIEW  WORKS_ON1
      AS SELECT   Fname, Lname, Pname, Hours
      FROM        EMPLOYEE, PROJECT, WORKS_ON
      WHERE       Ssn=Essn AND Pno=Pnumber;

V2:  CREATE VIEW  DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT   Dname, COUNT (*), SUM (Salary)
      FROM        DEPARTMENT, EMPLOYEE
      WHERE       Dnumber=Dno
      GROUP BY    Dname;
    
```

company.works\_on1

V1

Fname	Lname	Pname	Hours
Franklin	Wong	Computerization	10.0
Ahamad	Jabbar	Computerization	35.0
Alicia	Zelaya	Computerization	10.0
Jennifer	Wallace	Newbenefits	20.0
Ahamad	Jabbar	Newbenefits	5.0
Alicia	Zelaya	Newbenefits	30.0
John	Smith	ProductX	32.5
Joyce	English	ProductX	20.0
John	Smith	ProductY	7.5
Franklin	Wong	ProductY	10.0
Joyce	English	ProductY	20.0
Franklin	Wong	ProductZ	10.0
Ramesh	Narayan	ProductZ	40.0
Franklin	Wong	Reorganization	10.0
James	Borg	Reorganization	0.0
Jennifer	Wallace	Reorganization	15.0

company.dept\_info V2

Dept_name	No_of_emps	Total_sal
Administration	3	93,000
Headquarters	1	55,000
Research	4	133,000

- We can specify SQL queries on a view in the same way we specify queries involving base tables.
- For example,
  - to retrieve the last name and first name of all employees who work on the 'ProductX' project, we can utilize the WORKS\_ON1 view and specify the query as in QV1:

**QV1:**     **SELECT**     Fname, Lname  
              **FROM**        WORKS\_ON1  
              **WHERE**      Pname='ProductX';

**QV1:**     **SELECT**     Fname, Lname  
              **FROM**        WORKS\_ON1  
              **WHERE**      Pname='ProductX';

company.works\_on1

Fname	Lname	Pname	Hours
Franklin	Wong	Computerization	10.0
Ahamad	Jabbar	Computerization	35.0
Alicia	Zelaya	Computerization	10.0
Jennifer	Wallace	Newbenifits	20.0
Ahamad	Jabbar	Newbenifits	5.0
Alicia	Zelaya	Newbenifits	30.0
John	Smith	ProductX	32.5
Joyce	English	ProductX	20.0
John	Smith	ProductY	7.5
Franklin	Wong	ProductY	10.0
Joyce	English	ProductY	20.0
Franklin	Wong	ProductZ	10.0
Ramesh	Narayan	ProductZ	40.0
Franklin	Wong	Reorganization	10.0
James	Borg	Reorganization	0.0
Jennifer	Wallace	Reorganization	15.0

employee (2x2)	
Fname	Lname
John	Smith
Joyce	English

- A view is supposed to be always up-to-date;
  - if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes.
- Hence, the view is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view.
- It is the responsibility of the DBMS and not the user to make sure that the view is kept up-to-date

## DROP VIEW


- If we do not need a view any more, we can use the DROP VIEW command to dispose of it.
- For example, to get rid of the view V1, we can use the SQL statement in V1A:

**V1A:    DROP VIEW    WORKS\_ON1;**

## View Implementation, View Update, and Inline Views

- query modification
  - involves **modifying or transforming** the view query (submitted by the user) into a query on the underlying base tables.
  - For example, the query QV1 would be automatically modified to the following query by the DBMS

QV1:    **SELECT**    Fname, Lname  
         **FROM**    WORKS\_ON1  
         **WHERE**    Pname='ProductX';



**SELECT**    Fname, Lname  
**FROM**    EMPLOYEE, PROJECT, WORKS\_ON  
**WHERE**    Ssn=Essn **AND** Pno=Pnumber  
         **AND** Pname='ProductX';

- The **disadvantage** of this approach is that it is inefficient for views defined via **complex queries** that are **time-consuming** to execute,
- especially if multiple queries are going to be applied to the same view within a short period of time.

## View Materialization

- involves **physically creating a temporary view table** when the view is first queried and keeping that table on the assumption that other queries on the view will follow
- an efficient strategy for **automatically updating the view table when the base tables are updated** must be developed in order to keep the **view up-to-date**.
- Techniques using the concept of **incremental update** have been developed for this purpose,
  - where the DBMS can determine what **new tuples must be inserted, deleted, or modified in a materialized view table** when a database update is applied to one of the defining base tables



- The view is generally kept as a materialized (physically stored) table as long as it is being queried.
- If the **view is not queried** for a certain period of time, the system may then **automatically remove** the physical table and **recompute it from scratch** when future queries reference the view

## UPDATING VIEWS

- There are certain conditions needed to be satisfied to update a view
  1. The view is defined based on one and only one table.
  2. The view must include the PRIMARY KEY of the table based upon which the view has been created.
  3. The view should not have any field made out of aggregate functions.
  4. The view must not have any DISTINCT clause in its definition.
  5. The view must not have any GROUP BY or HAVING clause in its definition.
  6. The view must not have any SUBQUERIES in its definitions.
  7. If the view you want to update is based upon another view, the later should be updatable.
  8. Any of the selected output fields (of the view) must not use constants, strings or value expressions.

## Uses of a View

- **Restricting data access**
  - Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
- **Hiding data complexity**
  - A view can hide the complexity that exists in a multiple table join.
- **Simplify commands for the user**
  - Views allows the user to select information from multiple tables without requiring the users to actually know how to perform a join.

- **Store complex queries**
  - Views can be used to store complex queries.
- **Rename Columns**
  - Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to to hide the names of the columns of the base tables.
- **Multiple view facility**
  - Different views can be created on the same table for different users.

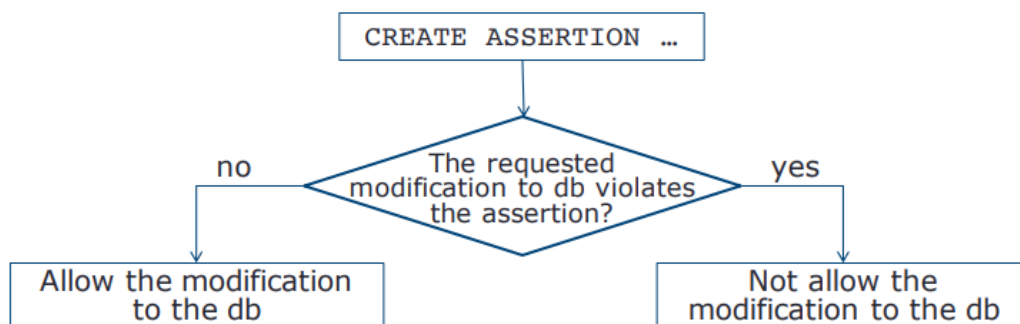
## Specifying Constraints as Assertions and Actions as Triggers

## Specifying General Constraints as Assertions in SQL

- Assertions = conditions that the database must always satisfy
- Domain constraints and referential-integrity constraints are specific forms of assertions
- CHECK – verify the assertion on one-table, one-attribute
- ASSERTION – verify one or more tables, one or more attributes
- Some constraints cannot be expressed by using only domain constraints or referential-integrity constraints; for example,
  - “Every department must have at least five courses offered every semester” – must be expressed as an assertion

```
CREATE ASSERTION <assertion-name>  
CHECK <predicate>;
```

```
DROP ASSERTION <assertion-name>
```



## Example 1

- For each tuple in the student relation, the value of the attribute tot\_cred must equal the sum of credits of courses that the student has completed successfully.

```
CREATE ASSERTION credits_earned_constraint
CHECK (NOT EXISTS
      (SELECT ID
       FROM student
       WHERE tot_cred <> (SELECT SUM(credits)
                          FROM takes
                          NATURAL JOIN course
                          WHERE student.ID=takes.ID
                          AND grade IS NOT NULL
                          AND grade <> 'F' ) ) )
```

- The total length of all movies by a given studio shall not exceed 10,000 minutes

```
CREATE ASSERTION sumLength
CHECK (10000 >= ALL
      (SELECT SUM(length)
       FROM Movies
       GROUP BY studioName ) )
```

- Since this constraint involves only the relation Movies, it can be expressed as a tuple-based CHECK constraint

```
CHECK (10000 >= ALL
      (SELECT SUM(length)
       FROM Movies
       GROUP BY studioName ) )
```

Type of constraint	Where declared	When activated	Guaranteed to hold?
Attributed-based <b>CHECK</b>	With attribute	On insertion to relation or attribute update	No if subqueries
Tuple-based <b>CHECK</b>	Element of relation schema	On insertion to relation or tuple update	No if subqueries
<b>ASSERTION</b>	Element of database scheme	On any change to any mentioned relation	Yes

## TRIGGER

- A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.
- action to be taken when certain events occur and when certain conditions are satisfied.
- For example,
  - a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.
  - it may be useful to specify a condition that, if violated, causes some user to be informed of the violation

## Benefits of Triggers

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

## Creating Triggers

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
    Declaration-statements
BEGIN
    Executable-statements
EXCEPTION
    Exception-handling-statements
END;
```

PREPARED BY SHARIKA T R,  
SNGCE

- **CREATE [OR REPLACE] TRIGGER trigger\_name**
  - Creates or replaces an existing trigger with the trigger\_name.
- **{BEFORE | AFTER | INSTEAD OF}**
  - This specifies when the trigger will be executed.
  - The INSTEAD OF clause is used for creating trigger on a view.
- **{INSERT [OR] | UPDATE [OR] | DELETE}**
  - This specifies the DML operation.
- **[OF col\_name]**
  - This specifies the column name that will be updated.
- **[ON table\_name]**
  - This specifies the name of the table associated with the trigger.

PREPARED BY SHARIKA T R,  
SNGCE

- **[REFERENCING OLD AS o NEW AS n]**
  - This allows you to refer new and old values for various DML statements, such as INSERT, UPDATE, and DELETE.
- **[FOR EACH ROW]**
  - This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
  - Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- **WHEN (condition)**
  - This provides a condition for rows for which the trigger would fire.
  - This clause is valid only for row-level triggers.



## Example

- creates a row-level trigger for the customers table that would fire for **INSERT** or **UPDATE** or **DELETE** operations performed on the CUSTOMERS table.
- This trigger will **display the salary difference between the old values and new values**

```
Select * from customers;
```

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
    sal_diff number;
BEGIN
    sal_diff := :NEW.salary - :OLD.salary;
    dbms_output.put_line('Old salary: ' || :OLD.salary);
    dbms_output.put_line('New salary: ' || :NEW.salary);
    dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

- The following points need to be considered here –
  - OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
  - If you want to query the table in the same trigger, then you should use the AFTER keyword,
    - because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
  - The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table,
    - but you can write your trigger on a single or multiple operations, for example BEFORE DELETE,
    - which will fire whenever a record will be deleted using the DELETE operation on the table.

## Triggering a Trigger

- Here is one INSERT statement, which will create a new record in the table

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

- When a record is created in the CUSTOMERS table, the above create trigger, display\_salary\_changes will be fired and it will display the following result

Because this is a new record, old salary is not available and the above result comes as null.

```
Old salary:
New salary: 7500
Salary difference:
```

- The UPDATE statement will update an existing record in the table

```
UPDATE customers  
SET salary = salary + 500  
WHERE id = 2;
```

- When a record is updated in the CUSTOMERS table, the above create trigger, display\_salary\_changes will be fired and it will display the following result

```
Old salary: 1500  
New salary: 2000  
Salary difference: 500
```

## Physical Data Organization

# SYLLABUS

- 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, Single 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

## Physical Files

- Physical files contain the **actual data** that is stored on the system, and a description of how data is to be presented to or received from a program.
- They contain only one record format, and one or more members.
- Records in database files can be externally or program-described.
- A physical file can have a keyed sequence access path.
  - This means that data is presented to a program in a sequence based on one or more key fields in the file.

## Logical files

- Logical files **do not contain data**.
- They contain a **description of records** found in one or more physical files.
- A logical file is a **view** or representation of one or more physical files.
- Logical files that contain more than one format are referred to as multi-format logical files.
- If your program processes a logical file which contains more than one record format, you can use a read by record format to set the format you wish to use.

Physical File	Logical File
It occupies the portion of memory. It contains the original data.	It does not occupy memory space. It does not contain data.
A physical file contains one record format.	It can contain upto 32 record formats.
It can exist without logical file.	It cannot exist without physical file.
If there is a logical file for physical file, the physical file cannot be deleted until and unless we delete the logical file.	If there is a logical file for a physical file, the logical file can be deleted without deleting the physical file.

- 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

## File organizations

- **File organization** refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked.
- An **access method**, on the other hand, provides a group of operations that can be applied to a file.
- In general, it is possible to apply several access methods to a file organization

## Primary file organizations

- Primary file organizations determines how the file records are physically placed on the disk, and hence how the records can be accessed.
- 1. **Heap file** (or unordered file) places the records on disk in no particular order by appending new records at the end of the file
- 2. **Sorted file** (or sequential file) keeps the records ordered by the value of a particular field (called the sort key).
- 3. **Hashed file** uses a hash function applied to a particular field (called the hash key) to determine a record's placement on disk.
- 4. Other primary file organizations, such as B-trees, use tree structures

## Secondary organization

- A secondary organization or auxiliary access structure allows efficient access to file records based on **alternate fields** than those that have been used for the primary file organization

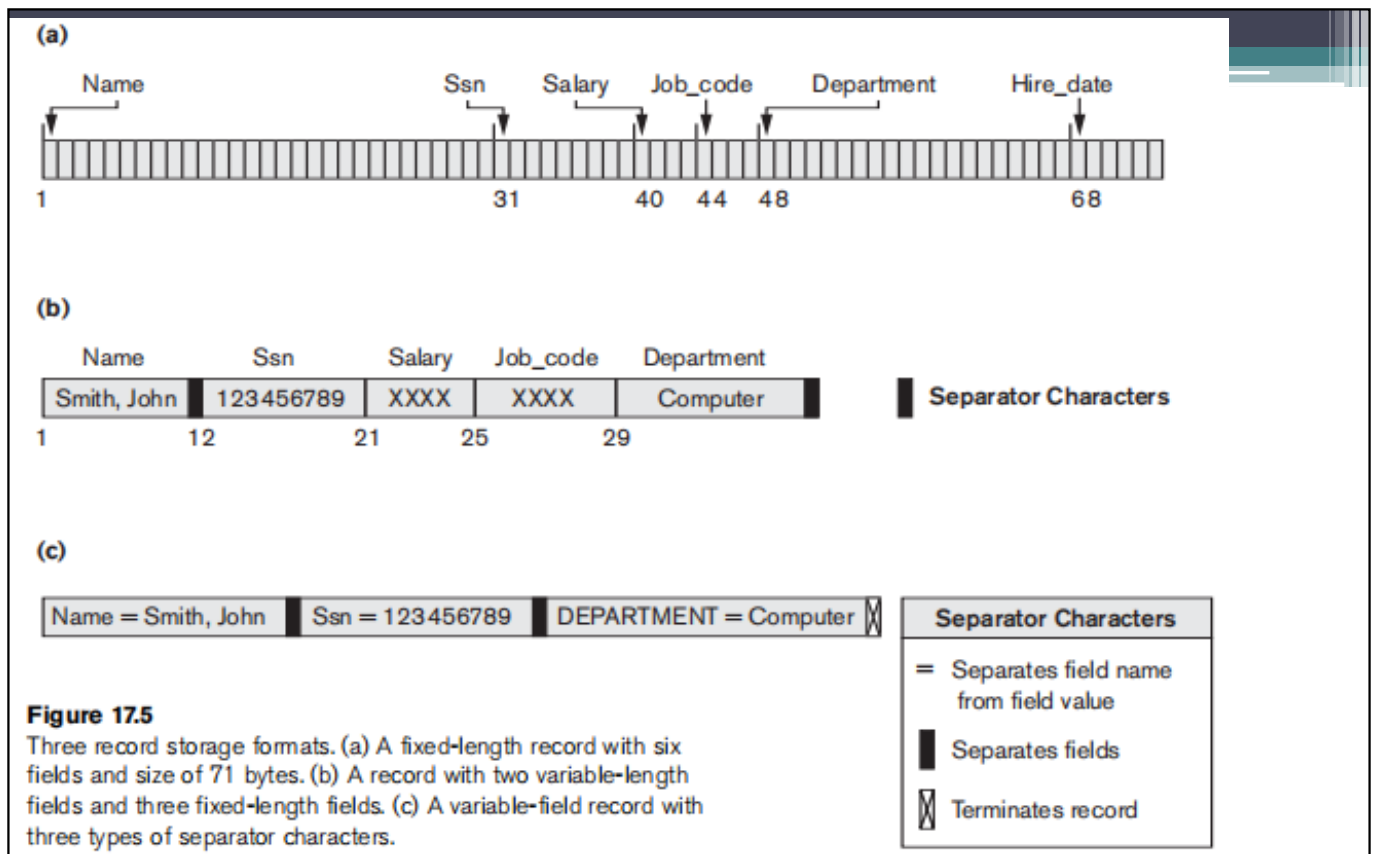
## Records and Record Types

- 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.
- Records usually describe entities and their attributes.
- A collection of field names and their corresponding data types constitutes a record type or record format definition.
- A data type, associated with each field, specifies the types of values a field can take

## Files, Fixed-Length Records, and Variable-Length Records

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





PREPARED BY SHARIKAT R.  
SNGCE

## Spanned and Unspanned Organization

- A block is the unit of data transfer between disk and memory
- When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.
- Part of the record can be stored 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. This organization is called **spanned** because records can span more than one block.
- Whenever a record is larger than a block, we must use a spanned organization.
- If records are **not allowed to cross block boundaries**, the organization is called **unspanned**.

## Blocking factor for the file

- The records of a file must be allocated to disk blocks because a **block is the unit of data transfer** between disk and memory.
- When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block.
- Suppose that the block size is B bytes.
- For a file of fixed-length records of size R bytes, with  $B \geq R$ , we can fit  $\text{bfr} = \lfloor B / R \rfloor$  **records per block**.
- The value bfr is called the blocking factor for the file.

- In general, R may not divide B exactly, so we have some **unused space in each block equal** to

$$B - (bfr * R) \text{ bytes}$$

- 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 disk.
- This organization is called spanned because records can span more than one block.
- Whenever a record is larger than a block, we must use a spanned organization.
- If records are not allowed to cross block boundaries, the organization is called unspanned
- If the average record is large, it is advantageous to use spanning to reduce the lost space in each block

- For variable-length records using spanned organization, each block may store a different number of records.
- In this case, the blocking factor bfr represents the **average number of records per block** for the file.
- We can use bfr to calculate the number of blocks b needed for a file of r records:

$$b = \lceil (r/bfr) \rceil \text{ blocks}$$

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

## Question

- Number of records=100000
- record size=100 bytes
- block size=512 byte
- for spanned organization number of  
blocks needed=  $100000 \times 100 \div 512$   
= **19523 blocks**
- In case of unspanned organization
- No of record per block=  $\lfloor 512/100 \rfloor = 5$  record/block
- Total no of blocks=  $100000/5 = 20000$  blocks
- so we can see in unspanned need 468 disk blocks in case of unspanned organisation

- How many disk block required to store 2000 records each of size 100bytes. Given block size is 512 bytes
- For spanned organisation  
no. of disk block=  $2000 \times 100 \div 512 = \lceil 390.6 \rceil = 391$  block
- For unspanned organisation  
No. of record per block=  $\lfloor 512/100 \rfloor = 5$  records per block  
No. disk block=  $2000/5 = 400$

## Allocating File Blocks on Disk

- **Contiguous allocation,**
  - the file blocks are allocated to consecutive disk blocks.
  - This makes reading the whole file very fast using double buffering, but it makes `expand_xxxxxx`ing the file difficult.
- **Linked allocation**
  - each file block contains a pointer to the next file block.
  - This makes it easy to expand the file but makes it slow to read the whole file.
- **Clusters allocation**
  - A combination of the two allocates clusters of consecutive disk blocks, and the clusters are linked.
  - Clusters are sometimes called file segments or extents.
- **Indexed allocation**
  - where one or more index blocks contain pointers to the actual file blocks

## File Headers

- A file header or file descriptor contains information about a file that is needed by the system programs that access the file records.
- The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions,
  - which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

- To search for a record on disk, one or more blocks are copied into main memory buffers.
- Programs then search for the desired record or records within the buffers, using the information in the file header.
- If the address of the block that contains the desired record is not known, the search programs must do a linear search through the file blocks.
- Each file block is copied into a buffer and searched until the record is located or all the file blocks have been searched unsuccessfully.
- This can be very time-consuming for a large file. The goal of a good file organization is to locate the block that contains a desired record with a minimal number of block transfers.

## Operation on Files

- Typical file operations include:
  - OPEN: Readies the file for access, and associates a pointer that will refer to a current file record at each point in time.
  - FIND: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
  - FINDNEXT: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
  - READ: Reads the current file record into a program variable.
  - INSERT: Inserts a new record into the file & makes it the current file record.

PREPARED BY SHARIKA T.R.  
SNGCE

- **DELETE:** Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY:** Changes the values of some fields of the current file record.
- **CLOSE:** Terminates access to the file.
- **REORGANIZE:** Reorganizes the file records.
- For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **READ\_ORDERED:** Read the file blocks in order of a specific field of the file.

PREPARED BY SHARIKA T.R.  
SNGCE

## Files of Unordered Records (Heap Files)

- 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.
- This organization is often used with additional access paths, such as the secondary indexes
- **Insertion**
  - 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.
  - The address of the last file block is kept in the file header.

- **Searching**

- searching 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

- **Deletion**

- 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. Search programs consider only valid records in a block when conducting their search.
- Both of these deletion techniques require periodic reorganization of the file to reclaim the unused space of deleted records



- 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.
- Modifying a variable-length record may require deleting the old record and inserting a modified record because the modified record may not fit in its old space on disk

- 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  $0, 1, 2, \dots, r - 1$  and the records in each block are numbered  $0, 1, \dots, bfr - 1$ , where  $bfr$  is the blocking factor, then the  $i$ th record of the file is located in block  $\lfloor i/bfr \rfloor$  and is the  $(i \bmod bfr)$ 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

## Files of Ordered Records (Sorted Files)

- We can physically order the records of a file on disk based on the values of one of their fields called the ordering field.
- This leads to an ordered or sequential file.
- If the ordering field is also a key field of the file a field guaranteed to have a unique value in each record then the field is called the ordering key for the file.

**Figure 17.7**

Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

	Name	Ssn	Birth_date	Job	Salary	Sex
Block 1	Aaron, Ed					
	Abbott, Diane					
	Acosta, Marc					
Block 2	Adams, John					
	Adams, Robin					
	Akers, Jan					
Block 3	Alexander, Ed					
	Alfred, Bob					
	Allen, Sam					
Block 4	Allen, Troy					
	Anders, Keith					
	Anderson, Rob					
Block 5	Anderson, Zach					
	Angeli, Joe					
	Archer, Sue					

## Ordered records advantages over unordered files

- First, reading the records in order of the ordering key values becomes extremely efficient because no sorting is required.
- Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one
- Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files.
- Ordered files are blocked and stored on contiguous cylinders to minimize the seek time

- A binary search for disk files can be done on the blocks rather than on the records.
- Suppose that the file has  $b$  blocks numbered 1, 2, ...,  $b$ ; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is  $K$ .
- A binary search usually accesses  $\log_2(b)$  blocks, whether the record is found or not an improvement over linear searches, where, on the average,  $(b/2)$  blocks are accessed when the record is found and  $b$  blocks are accessed when the record is not found.
- Ordering does not provide any advantages for random or ordered access of the records based on values of the other nonordering fields of the file. In these cases, we do a linear search for random access.
- To access the records in order based on a nonordering field, it is necessary to create another sorted copy in a different order of the file

- Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered.
- To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position.
- For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record.
- This means that half the file blocks must be read and rewritten after records are moved among them.
- For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

## Average Access Times

- The following table shows the average access time to access a specific record for a given type of file

**TABLE 13.2 AVERAGE ACCESS TIMES FOR BASIC FILE ORGANIZATIONS**

TYPE OF ORGANIZATION	ACCESS/SEARCH METHOD	AVERAGE TIME TO ACCESS A SPECIFIC RECORD
Heap (Unordered)	Sequential scan (Linear Search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary Search	$\log_2 b$

PREPARED BY SHARIKA T.R.  
SNGCE

- Consider an unordered file of 100000 records with record size of 100 bytes stored on block of 4096 byte with an unspanned record organization. Assume that no system relation information stored in block. On average how many block would be access to find particular records

Ans. No. of records per block =  $\lfloor 4096/100 \rfloor = 40$

No. of blocks =  $100000/40 = 2500$  blocks

Average number of block access to search a record =  $b/2$   
 $= 2500/2 = 1250$  block access

PREPARED BY SHARIKA T.R.  
SNGCE

- Consider an sorted file of 100000 records with record size of 100 bytes stored on block of 4096 byte with an unspanned record organization. Assume that no system relation information stored in block. On average how many block would be access to find particular records if search is based on ordering field .

Ans. No. of records per block =  $\lfloor 4096/100 \rfloor = 40$

No. of blocks =  $100000/40 = 2500$  blocks

Average number of block access to search a record =  $\log_2 b$   
 $= \log_2 2500 = 12$  blocks

## Hashed Files

- Hashing for disk files is called External Hashing
- The file blocks are divided into **M equal-sized buckets**, numbered bucket<sub>0</sub>, bucket<sub>1</sub>, ..., bucket<sub>M-1</sub>
  - Typically, a bucket corresponds to one (or a fixed number of) disk block.
- One of the file fields is designated to be the **hash key of the file**.
- The **record with hash key value K is stored in bucket i**, where  $i=h(K)$ , and h is the hashing function.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.
  - Overflow records that hash to each bucket can be linked together.

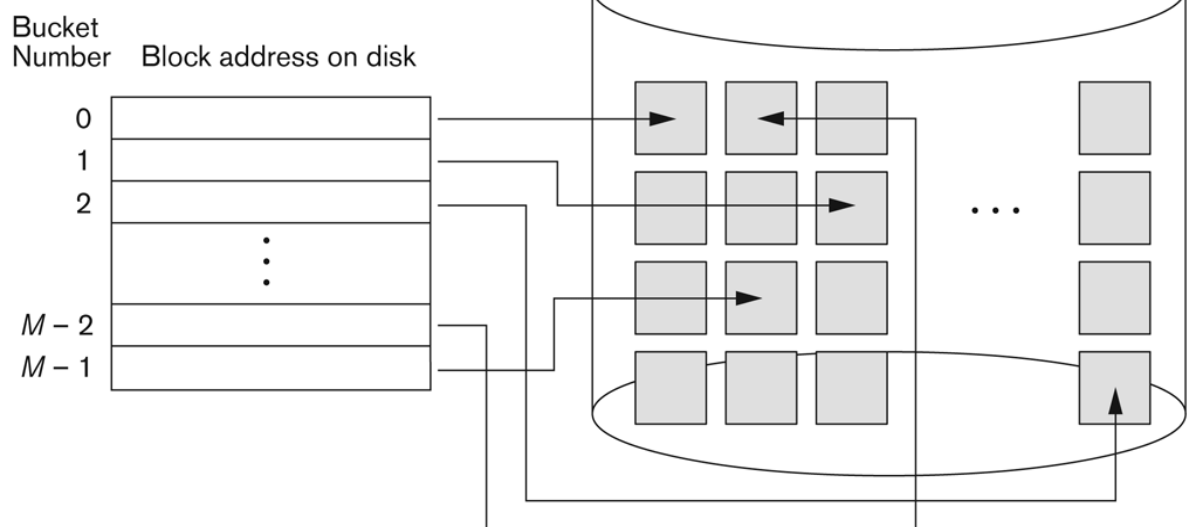
- There are numerous methods for collision resolution, including the following:
  - **Open addressing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
  - **Chaining**: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

## Static External Hashing for Disk Files

- Hashing for disk files is called external hashing.
- To suit the characteristics of disk storage, the target address space is made of buckets, each of which holds multiple records.
- A bucket is either one disk block or a cluster of contiguous disk blocks.
- The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket.
- A table maintained in the file header converts the bucket number into the corresponding disk block address
- Here we have fixed number of buckets allocated so it is also called static external hashing

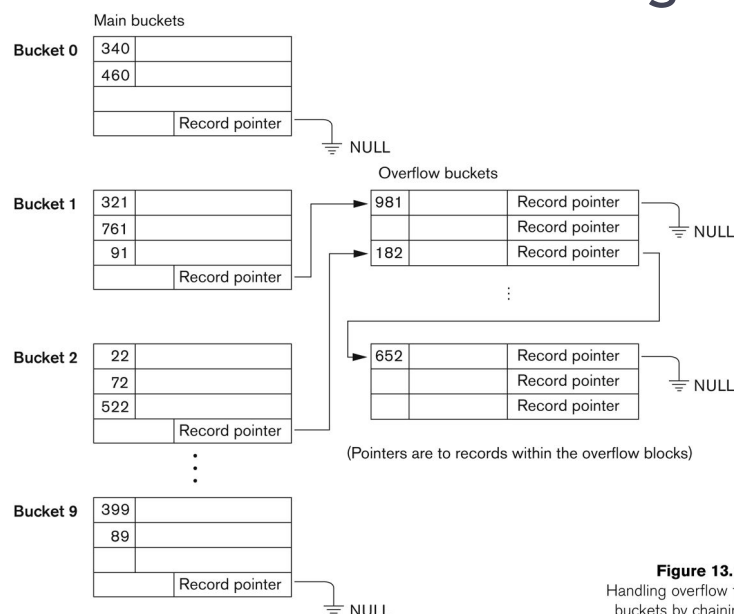
**Figure 13.9**

Matching bucket numbers to disk block addresses.



- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function  $h$  should distribute the records uniformly among the buckets
  - Otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of static external hashing:
  - **Fixed number of buckets**  $M$  is a problem if the number of records in the file grows or shrinks.
  - **Ordered access** on the hash key is quite inefficient (requires sorting the records).
  - searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file

## Hashed Files - Overflow handling



**Figure 13.10**  
Handling overflow for buckets by chaining.

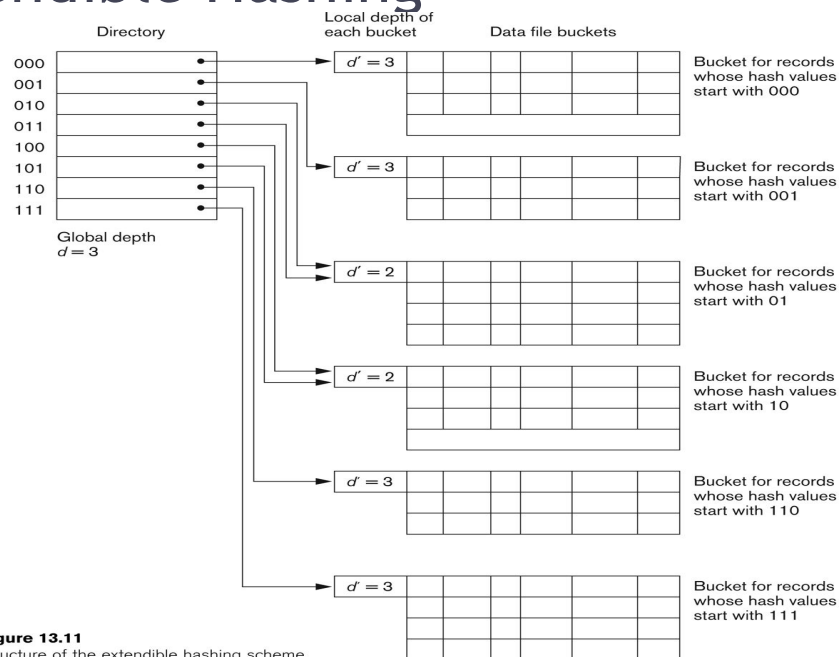


## Dynamic And Extendible Hashed Files

- Dynamic and Extendible Hashing Techniques
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the number of file records.
  - These techniques include the following: dynamic hashing, extendible hashing, and linear hashing.
- Both **dynamic and extendible hashing use the binary representation** of the hash value  $h(K)$  in order to access a directory.
  - In **dynamic hashing the directory is a binary tree**.
  - In **extendible hashing the directory is an array of size  $2^d$  where  $d$  is called the global depth**.

- The directories can be stored on disk, and they expand or shrink dynamically.
  - Directory entries point to the disk blocks that contain the stored records.
- An **insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks**.
  - The directory is updated appropriately.
- **Dynamic and extendible hashing do not require an overflow area.**
- Linear hashing does require an overflow area but does not use a directory.
  - **Blocks are split in linear order as the file expands.**

# Extendible Hashing



PREPARED BY SHARIKA T R,  
SNGCE

## INDEXING

### Types of Single-level Ordered Indexes

- Primary Indexes

- Clustering Indexes

- Secondary Indexes

### Multilevel Indexes

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

- Indexes on Multiple Keys

# Index Structures

- Indexing is a data structure technique to efficiently retrieve records from the database files based on some attributes on which the indexing has been done.
- An index on a database table provides a convenient mechanism for locating a row (data record) without scanning the entire table and thus greatly reduces the time it takes to process a query.
- The index is usually specified on one field of the file.
- One form of an index is a file of entries <field value, pointer to record>, which is ordered by field value.
- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller.

- Advantages:
  - Stores and organizes data into computer files.
  - Makes it easier to find and access data at any given time.
  - It is a data structure that is added to a file to provide faster access to the data.
  - It reduces the number of blocks that the DBMS has to check.
- Disadvantages
  - Index needs to be updated periodically for insertion or deletion of records in the main table.

## Types of index

- Indexes can be characterized as
  1. Dense index
  2. Sparse index
- A dense index has an index entry for every search key value (and hence every record) in the data file.
- A sparse (or nondense) index, on the other hand, has index entries for only some of the search values.

## Structure of index

- An index is a small table having only two columns.
- The first column contains a copy of the primary or candidate key of a table
- The second column contains a set of pointers holding the address of the disk block where that particular key value can be found.
- If the indexes are sorted, then it is called as ordered indices.

## Primary Index

- Primary index is defined on an **ordered data file**. The data file is ordered on a key field.
- The key field is generally the primary key of the relation.
- A primary index is a **nondense (sparse) index**, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

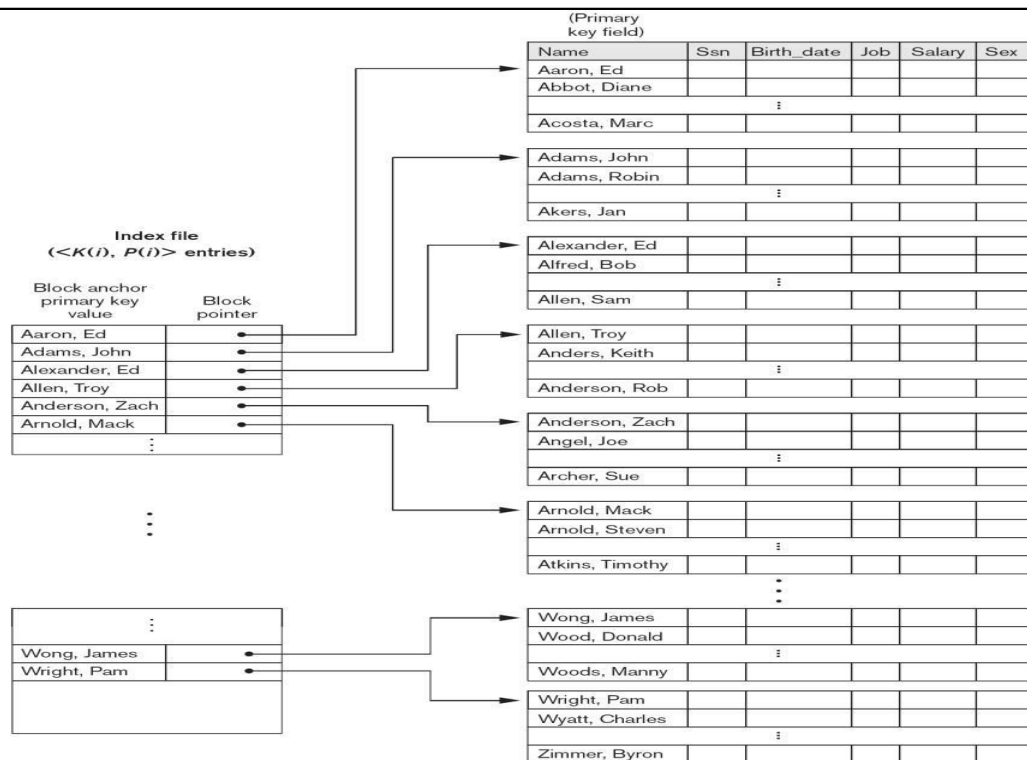


Figure 5.1: Primary Index on the Ordering Key Field of the File

## Example

- Suppose we have an ordered file with 30,000 records and stored on a disk of block size 1024 bytes and records are of fixed size, unspanned organisation.
- Record length = 100 bytes. How many block access if using a primary index file, with an ordering key field of the file 9 bytes and block pointer size 6 bytes.

Ans. Ordering key field of file  $V=9$  byte

Block pointer  $P=6$  byte

we need to construct a primary index for the file

size of each index entry =  $9+6=15$  byte

so blocking factor of index entry =

$$\lfloor 1024/15 \rfloor = 68 \text{ entries per block}$$

Total no of entries per block =  $\lceil 3000/68 \rceil = 45$

To do binary search =  $\lceil \log_2 bi \rceil = \lceil \log_2 45 \rceil = 6$  blocks

To search for a record using index we need additional one block access =  $6+1=7$  block access

# Clustering Index

- 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.
- clustering index speeds 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.

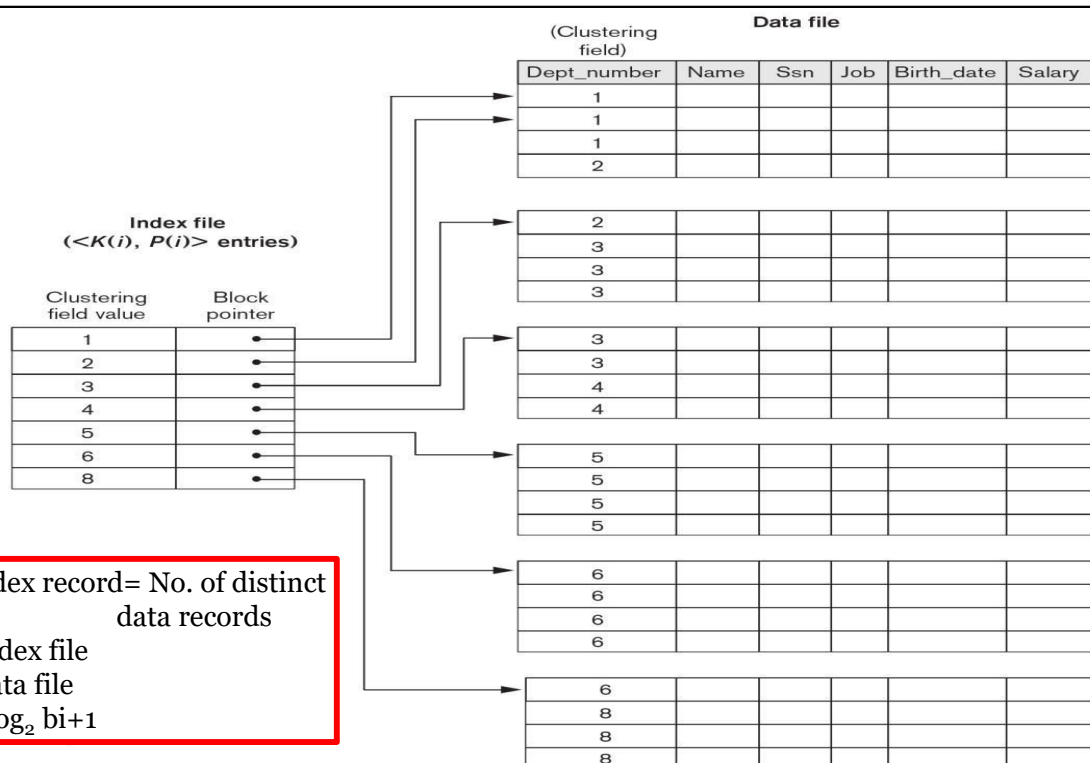


Figure 5.2: A Clustering Index on Dept\_number Ordering Nonkey Field of a File

## Question

- Suppose we have an ordered file with 30,0000 records and stored on a disk of block size 4096 bytes and records are of size 100 byte, unspanned organisation.
- The **ordered file is based on non key field (department\_name)**. How many block access if using a primary index file, with an ordering key field of the file 5 bytes and block pointer size 6 bytes. There are 1000 department and 300 employees per department.

No of record=300000

Block size=4096 b

Record size=100 b

Block ptr=6 b

Clustered Index

Without index:

No of blocks=  $\lceil 300000/40 \rceil = 7500$

No. of block access=  $\log_2 7500 = 13$

With clustered index

No of index record=  $\lceil 4096/(6+5) \rceil = 4096/11 = 372$

Total no of index record=1000 department

No of clustered index blocks=  $\lceil 1000/372 \rceil = 3$  blocks

No of clock access=  $\lceil \log_2 3 \rceil + 1 = 2 + 1 = 3$  block access



## Secondary Index

- 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 non-ordering field of the data file that is an indexing field.
- The second field is either a block pointer or a record pointer.
- There can be many secondary indexes (and hence, indexing fields) for the same file.
- Includes one entry for each record in the data file; hence, it is a dense index.

## Example

- Consider an unordered file with  $10^8$  records with record size of 400 bytes with an unspanned organization. Suppose that we construct a single level secondary index for the file where search key field is 16 bytes and block pointer is 4 byte. Assume disk block size is 4096
- How many blocks are required to store the data file?
- How many blocks are required to store the index file?

- Disk file
  - No of data records/ block =  $4096/400 = 10$
  - No of data block required =  $\text{ceil}(10^8/10) = 10^7$
- Index file
  - Size of one index record =  $16 + 4 = 20$  byte
  - No of index record = No of data record =  $10^8$ 
    - secondary index on key field
  - No of index record per block =  $4096/20 = 204$
  - No of index block required =  $\text{ceil}(10^8/204) = 490197$

## Example 2

- Consider an unordered file with  $10^8$  records with record size of 400 bytes with an spanned organization. Suppose that we construct a single level secondary index for the file where search key field is 16 bytes and block pointer is 4 byte. Assume disk block size is 4096
- How many blocks are required to store the data file?
- How many blocks are required to store the index file?

- Disk file
  - No of data block required =  $\text{ceil}(10^8 * 400 / 4096) = 9765625$
- Index file
  - Size of one index record =  $16 + 4 = 20$  byte
  - No of index record = No of data record =  $10^8$ 
    - secondary index on key field
  - No of index block required =  $\text{ceil}(10^8 * 20 / 4096) = 488282$

## Example

- Consider an **unordered file** with 30000 records with record size of 100 bytes with an unspanned organization. Suppose that we construct a secondary index for the key field with key field size is 9 bytes and block pointer is 6 byte. Assume disk block size is 1024
- Average number of disk block access to search for record without index
- Average number of disk block access to search for record with index

- Disk file
  - No of data records/ block =  $1024/100 = 10$
  - No of data block required =  $\text{ceil}(30000/10) = 3000$
  - Average no of disk access =  $\text{ceil}(3000/2) = 1500$
- Index file
  - Size of one index record =  $9 + 6 = 15$  byte
  - No of index record = No of data record = 3000
    - secondary index on **key field**
  - No of index record per block =  $1024/15 = 68$
  - No of index block required =  $\text{ceil}(30000/68) = 442$
  - Avg no of disk block access required =  $\text{ceil}(\log_2 442) + 1 = 10$

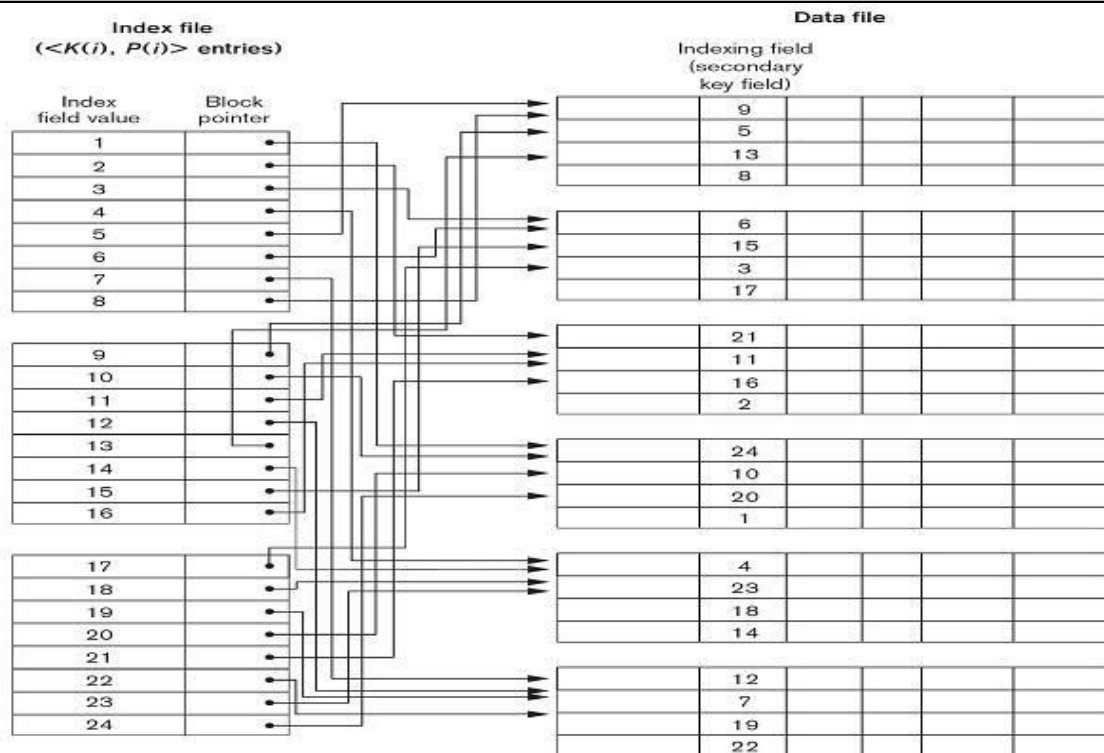


Figure 5.3: Dense Secondary Index (with Block Pointer) on a Non Ordering Key Field of the File

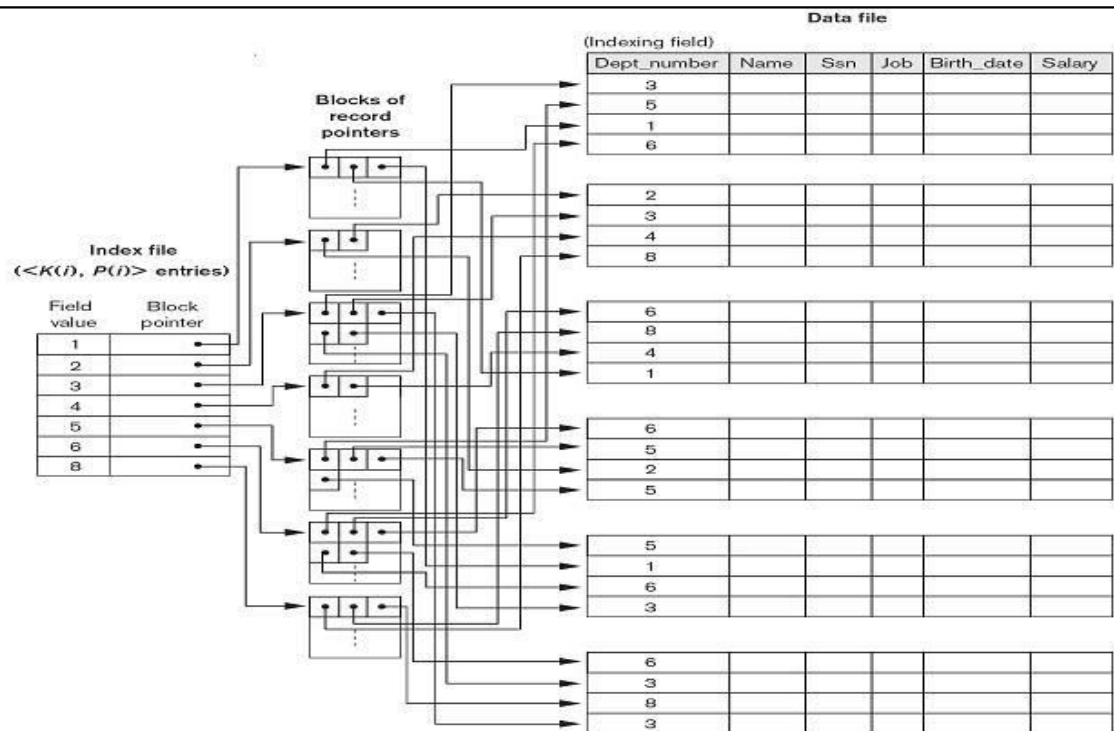


Figure 5.4: Secondary Index (with Record Pointer) on a Non Key Field implemented using one level of indirection so that Index entries are of Fixed Length and have unique field values

PREPARED BY SHARIKA T R,  
SNGCE

**Table 5.1: 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 <sup>a</sup>
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup>	Dense or Nondense	No

## Single level and Multi-level indexing

- Because a single-level index is an ordered file, we can create a primary index to the index itself;
- In this case, the original index file is called the first-level index and the index to the index is called the second-level index.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the top level fit in one disk block.
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of more than one disk block

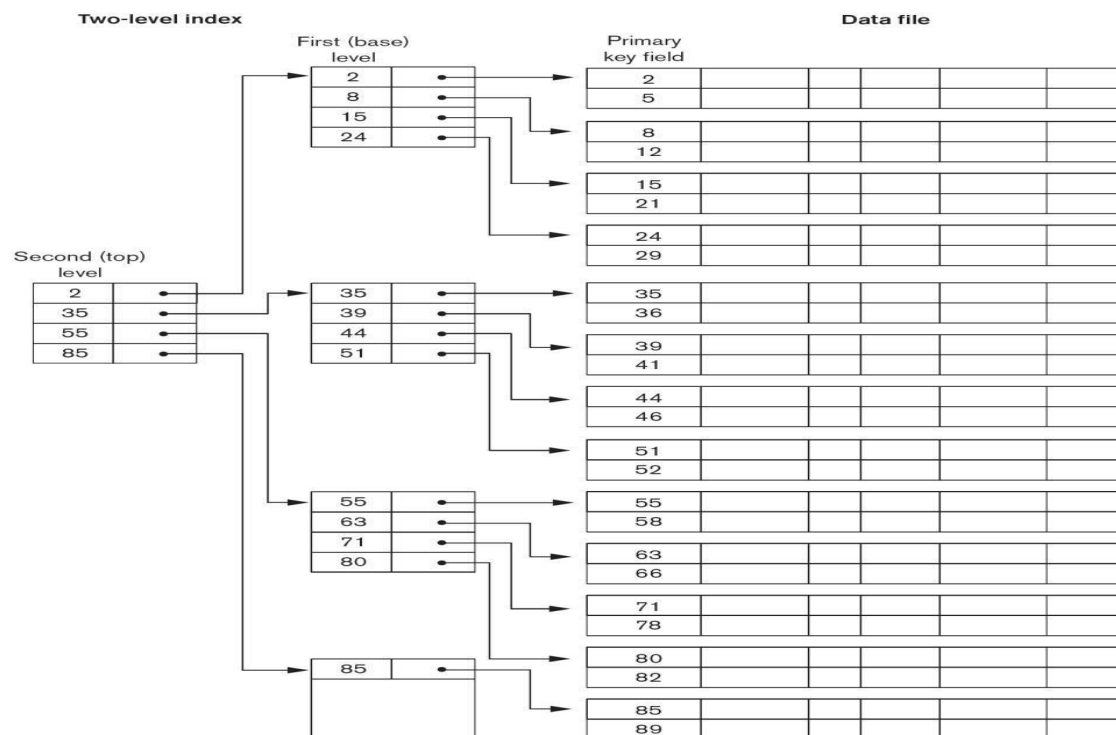


Figure 5.5: Two-Level Primary Index

## Example

- Consider a file of **16384 records**. Each **record** is **32 bytes** long and its **key field** is of size **6 bytes**. The file is **ordered on a non-key field**, and the file organization is **unspanned**. The **file is stored in a file system** with **block size 1024 bytes**, and the size of a **block pointer is 10 bytes**. If the **secondary index is built on the key field of the file**, and a **multi-level index scheme is used** to store the secondary index, the number of first-level and second-level blocks in the multi-level index are?

- Number of records in file = 16384
- Record size = 32 bytes
- Key Size = 6 bytes
- Block Size on file system = 1024 bytes
- Size of Block Pointer = 10 bytes
- Size of a record or index Entry =  $10 + 6 = 16$   
**ordered on a non-key field**  
**secondary index is built on the key field of the file,**  
**multi-level index scheme is used**

- Without Indexing
  - No of data record per block =  $1024/32=32$
  - No of data block =  $16384/32=512$
- Indexing First Level:
  - No of index record = 16384
    - **secondary index is built on the key field of the file**
    - ordered
  - No of index record per block =  $1024/(6+10)=64$
  - No of index blocks =  $16384/64=256$

- Indexing Second Level
  - No of index records = 256
  - **ordered on a non-key field**
  - No of index record per block =  $1024/16=64$
  - No of index block =  $256/64=4$  blocks



## Example 2

Consider an EMPLOYEE file with 10000 records where each record is of size 80 bytes. The file is sorted on employee number (15 bytes long), which is the primary key. Assuming un-spanned organization and block size of 512 bytes compute the number of block accesses needed for selecting records based on employee number if, (9)

- i. No index is used
- ii. Single level primary index is used
- iii. Multi-level primary index is used

Assume a block pointer size of 6 bytes.

## Given

- No of records=10000
- record size=80byte
- key field=15byte
- block size=512byte
- block pointer size=6 byte

**SORTED ON PRIMARY KEY**  
**UNSPANNED ORGANISATION**

- No index used( data file)
  - No of record per block= $512/80=6$
  - No of data block needed= $10000/6=1667$
  - No of block access needed =  $\log_2 1667=11$
- Single level primary index
  - No of index record= $1667$
  - No of index record per block =  $512/(15+6)=24$
  - No of index block needed= $1667/24=70$
  - No of block access needed= $\log_2 70 +1=7+1=8$
- Multilevel Primary Index
  - No of multilevel index record= $70$
  - No of multilevel index record/block= $512/(15+6)=24$
  - No of multilevel index record =  $70/24=3$
  - No of block access= $\log_2 3+2=5$

## 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  $\langle P_1K_1, P_2K_2, \dots, P_{q-1}K_{q-1}, P_q \rangle$ , 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,  $K_1 < K_2 < \dots < K_{q-1}$ .
  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  $i = q$

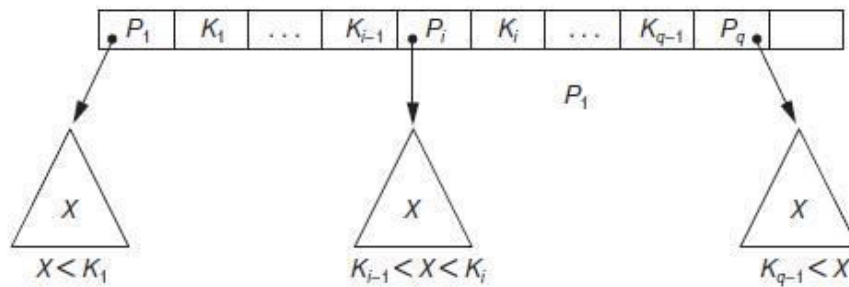
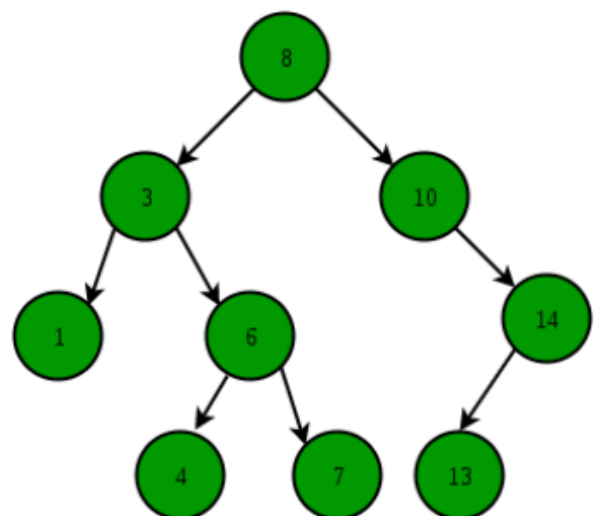


Figure 5.6: A node in a search tree with pointers to subtrees below it

## Binary Search Tree

- Binary Search Tree is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - The left and right subtree each must also be a binary search tree.



## B-Trees

PREPARED BY SHARIKA T.R.  
SNGCE

- The B-tree has additional constraints that ensure that the tree is always **balanced**.
- 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
    - $\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$  where  $q \leq p$ .
    - Each  $P_i$  is a tree pointer a pointer to another node in the Btree.
    - Each  $Pr_i$  is a data pointer a pointer to the record whose search key field value is equal to  $K_i$ .

PREPARED BY SHARIKA T.R.  
SNGCE

2. Within each node,  $K_1 < K_2 < \dots < K_{q-1}$ .
3. For all search key field values  $X$  in the subtree pointed at by  $P_i$ , we have:  $K_{i-1} < X < K_i$  for  $1 < i < q$ ;  $X < K_i$  for  $i = 1$ ; and  $K_{i-1} < X$  for  $i = q$ .
4. Each node has **at most p tree pointers**.
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**,  $q \leq p$ , 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**.

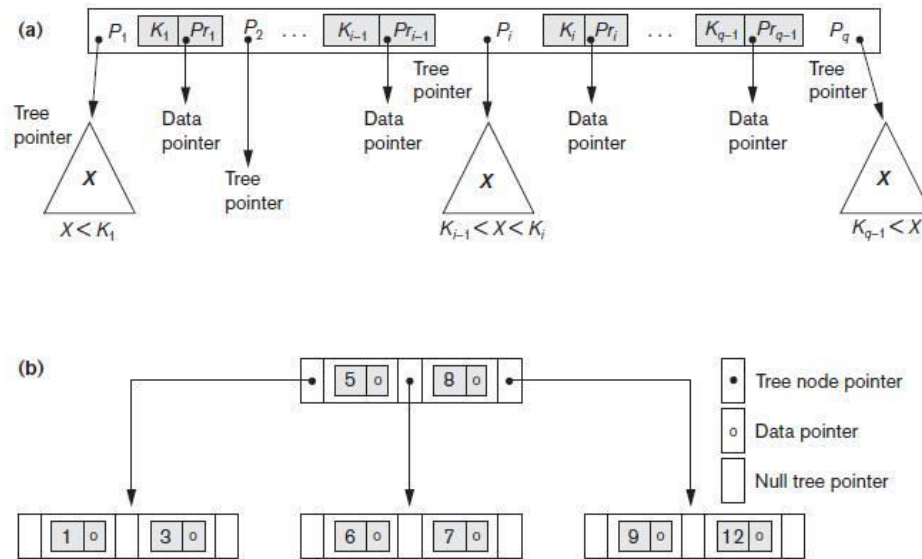


Figure 5.7: B-tree structures. (a) A node in a B-tree with  $q - 1$  search values.

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

PREPARED BY SHARIKAT R.  
SNGCE

- 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 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.
- 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.

- 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.
  - Hence, deletion can reduce the number of tree levels.

## Properties of a B-tree

- For a tree to be classified as a B-tree, it must fulfill the following conditions:
  - the nodes in a B-tree of **order m** can have a **maximum of m children**
  - each **internal node** (non-leaf and non-root) can have **at least  $(m/2)$  children** (rounded up)
  - the **root** should have **at least two children** – unless it's a leaf
  - a **non-leaf** node with **k children** should have **k-1 keys**
  - all leaves must appear on the **same level**

## Building a B-tree

- Since we're starting with an empty tree, the first item we insert will become the root node of our tree.
- At this point, the root node has the key/value pair.
- The key is 1, but the value is depicted as a star to make it easier to represent, and to indicate it is a reference to a record.
- The root node also has pointers to its left and right children shown as small rectangles to the left and right of the key.
- Since the node has no children, those pointers will be empty for now:

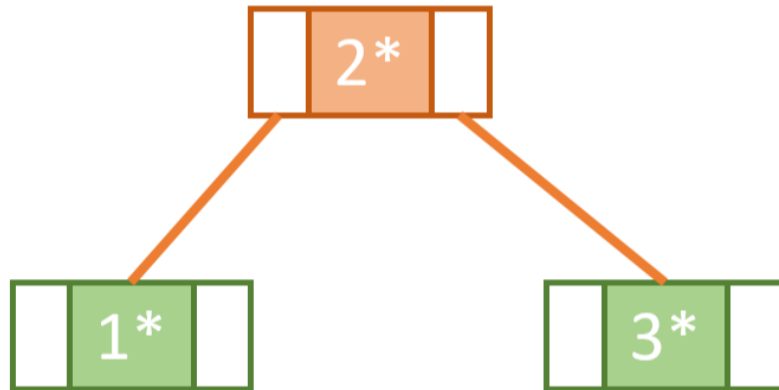


We know that this tree has **order of 3**, so it can have only up to 2 keys in it. So we can add the payload with key 2 to the root node in ascending order:

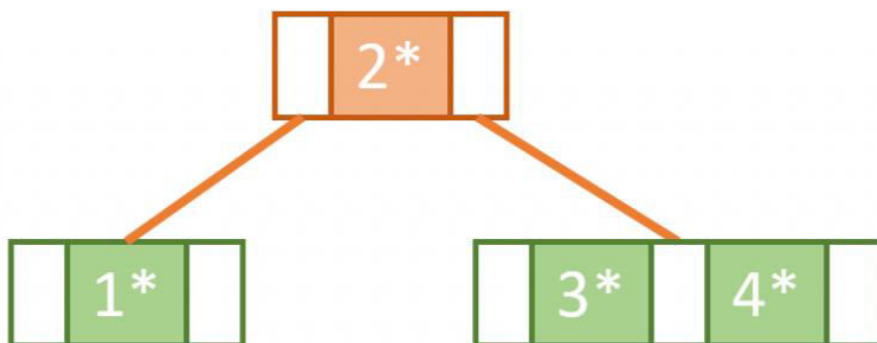


Next, if we wanted to insert 3, for us to keep the tree balanced and fulfilling the conditions of a B-tree we need to perform what is called a split operation. We can determine how to split the node by picking the middle key.

PREPARED BY SHARIKA T R,  
SNGCE



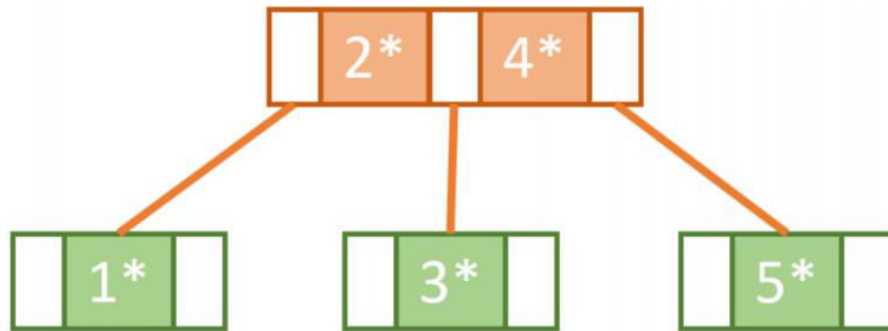
PREPARED BY SHARIKA T R,  
SNGCE



Now, let's insert 4. To determine where this needs to be placed we must remember that B-trees are organized such that sub-trees on the right have greater keys than sub-trees on the left. Consequently, Key 4 belongs in the right sub-tree. And since the right sub-tree still has the capacity, we can simply add 4 to it alongside 3 in ascending order:

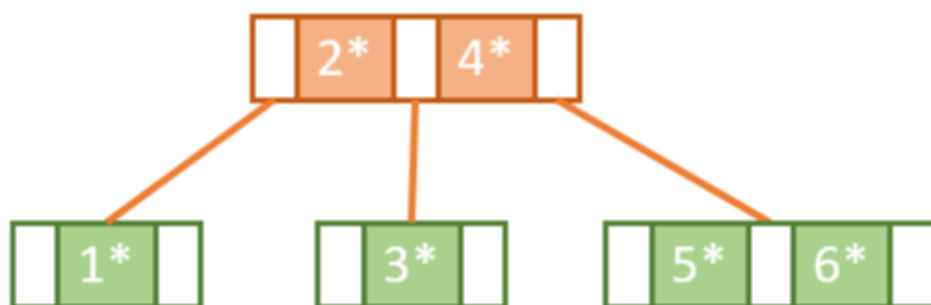


PREPARED BY SHARIKA T R,  
SNGCE

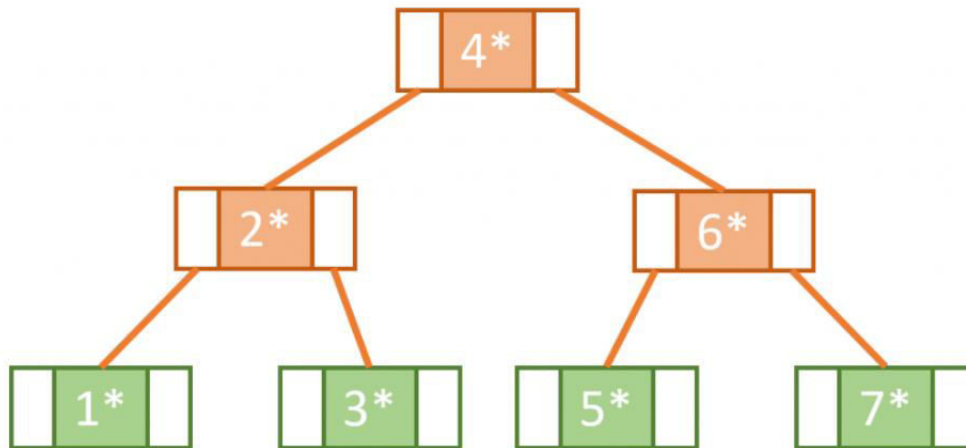


Our right sub-tree is now at full capacity, so to insert 5 we need to use the same splitting logic explained above. We split the node into two so that Key 3 goes to a left sub-tree and 5 goes to a right sub-tree leaving 4 to be promoted to the root node alongside 2.

PREPARED BY SHARIKA T R,  
SNGCE



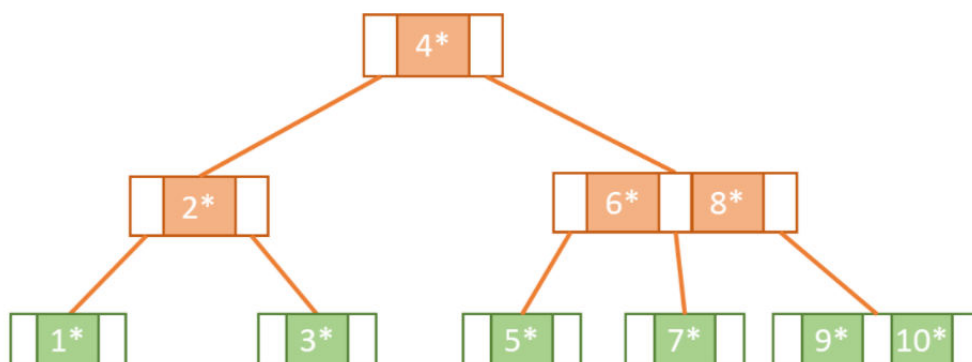
This rebalancing gives us space in the rightmost sub-tree to insert 6:



Next, we try to insert 7. However, since the rightmost tree is now at full capacity we know that we need to do another split operation and promote one of the keys. But wait! The root node is also at full capacity, which means that it also needs to be split.

So, we end up doing this in two steps. First, we need to split the right nodes 5 and 6 so that 7 will be on the right, 5 will be on the left, and 6 will be promoted. Then, to promote 6, we need to split the root node such that 4 will become a part of new root and 6 and 2 become the parents of the right and left subtree.

PREPARED BY SHARIKA T R,  
SNGCE



Continuing in this way, we fill the tree by adding Keys 8,9 and 10 until we get the final tree:

## 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.

- The structure of the internal nodes of a B+ tree of order  $p$  is as follows:
  1. Each internal node is of the form  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$  where  $q \leq p$  and each  $P_i$  is a tree pointer.
  2. Within each internal node,  $K_1 < K_2 < \dots < K_{q-1}$ .
  3. For all search field values  $X$  in the subtree pointed at by  $P_i$ , we have  $K_{i-1} < X \leq K_i$  for  $1 < i < q$ ;  $X \leq K_1$  for  $i = 1$ ; and  $K_{q-1} < X$  for  $i = q$ .
  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  $p$  is as follows:
  1. Each leaf node is of the form  $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{next} \rangle$  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,  $K_1 \leq K_2 \dots, K_{q-1}, q \leq p$ .
  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
  4. Each leaf node has at least  $\lceil (p/2) \rceil$  values.
  5. All leaf nodes are at the same level.

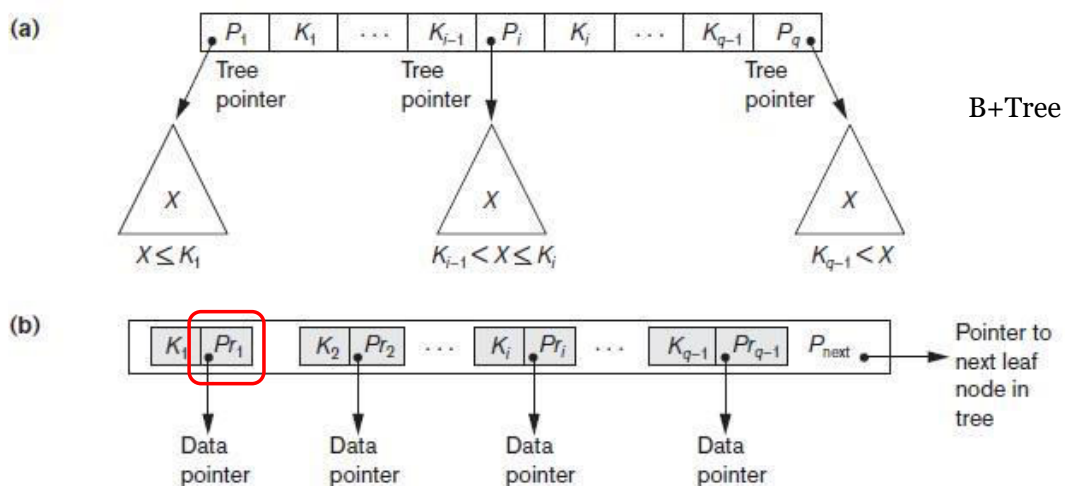
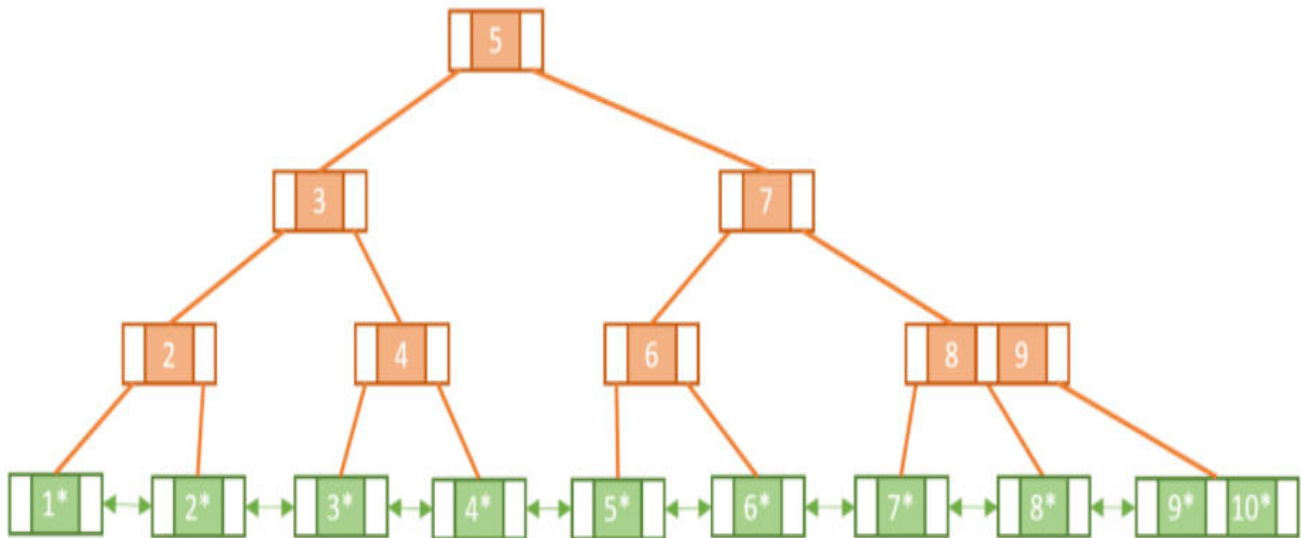


Figure 5.8: The nodes of a B+-tree. (a) Internal node of a B+-tree with  $q - 1$  search values.

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

PREPARED BY SHARIKA T R,  
SNGCE

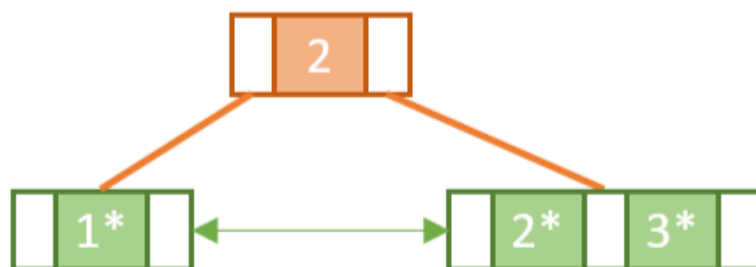


PREPARED BY SHARIKA T R,  
SNGCE

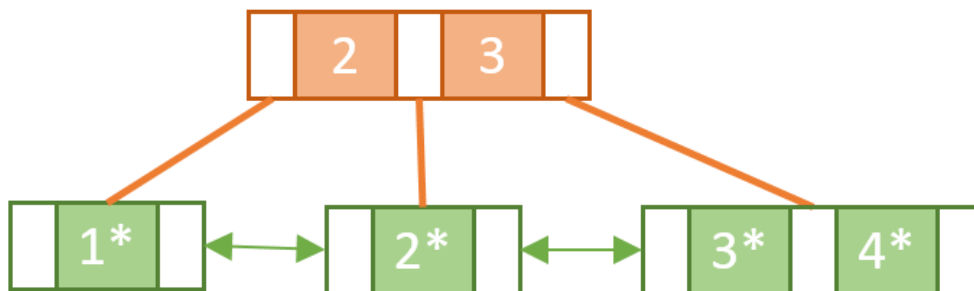
- When a leaf node is full and a new entry is inserted there, the node overflows and must be split.
- The first  $j = \lceil ((p_{leaf} + 1)/2) \rceil$  entries in the original node are kept there, and the remaining entries are moved to a new leaf node.
- The  $j$ th search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent.
- These must be inserted in the parent node in their correct sequence.
- If the parent internal node is full, the new value will cause it to overflow also, so it must be split.
- The entries in the internal node up to  $P_j$ —the  $j$ th tree pointer after inserting the new value and pointer, where  $j = \lfloor ((p + 1)/2) \rfloor$ —are kept, while the  $j$ th search value is moved to the parent, not replicated.
- A new internal node will hold the entries from  $P_{j+1}$  to the end of the entries in the node
- This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree.

# Building a B+tree

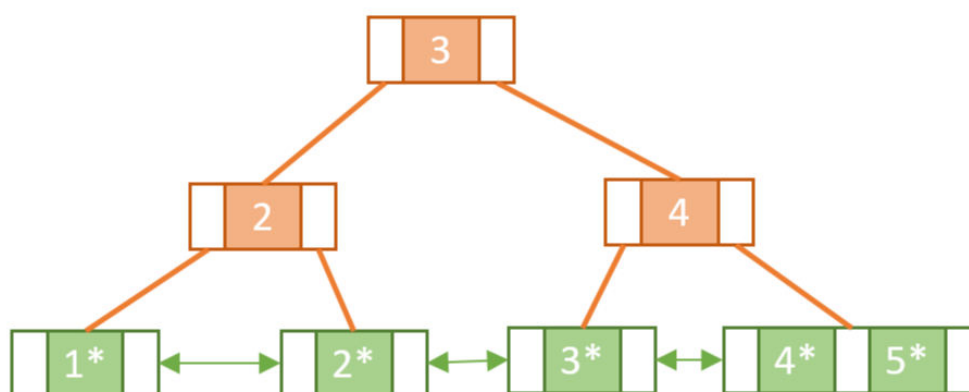
To start with, we'll insert Keys 1 and 2 into the root node in ascending order:



When we come to insert Key 3, we find that in doing so we will exceed the capacity of the root node. Similar to a normal B-tree this means we need to perform a split operation. However, unlike with the B-tree, we must copy-up the first key in the new rightmost leaf node. As mentioned, this is so we can make sure we have a key/value pair for Key 2 in the leaf nodes:



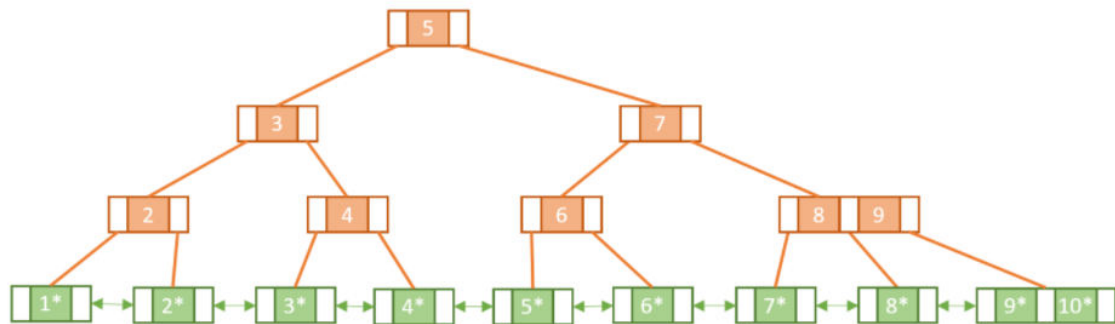
Next, we add Key 4 to the rightmost leaf node. Since it's full, we need to perform another a split operation and copy-up Key 3 to the root node:



Now, let's add 5 to the rightmost leaf node. Once again to keep the order, we'll split the leaf node and copy-up 4. Since that will overflow the root node, we'll have to perform another split operation splitting the root node into two nodes and promoting 3 into a new root node

Notice the difference between splitting a leaf node and splitting an internal node. When we split the internal node in the second split operation we didn't copy-up Key 3.

PREPARED BY SHARIKA T R,  
SNGCE



In the same way, we keep adding the keys from 6 to 10, each time splitting and copying-up when necessary until we have reached our final tree:

PREPARED BY SHARIKA T R,  
SNGCE

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem.
- This leaves space in each tree node (disk block) to allow for new index entries.
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block.
- Each node is kept between half-full and completely full.
- An insertion into a node that is not full is quite efficient.
- If a node is full the insertion causes a split into two nodes.
- Splitting may propagate to other tree levels.
- A deletion is quite efficient if a node does not become less than half full.
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes



## B-trees in the Context of Databases

- DBMSs leverage the logarithmic efficiency of B-tree indexing to reduce the number of reads required to find a particular record.
- B-trees are typically constructed so that each node takes up a single page in memory and they are designed to reduce the number of accesses by requiring that each node be at least half full.

## Difference between B-tree and B+-tree

- In B+trees, search keys can be repeated but this is not the case for B-trees
- B+trees allow satellite data to be stored in leaf nodes only, whereas B-trees store data in both leaf and internal nodes
- In **B+trees**, **data stored on the leaf node makes the search more efficient** since we can **store more keys in internal nodes**
  - this means we need to access fewer nodes
- **Deleting data from a B+tree is easier** and less time consuming because we only need to remove data from leaf nodes
- Leaf nodes in a B+tree are linked together making range search operations efficient and quick

- Finally, although B-trees are useful, B+trees are more popular. In fact, 99% of database management systems use B+trees for indexing.
- This is because the B+tree holds no data in the internal nodes.
- This maximizes the number of keys stored in a node thereby minimizing the number of levels needed in a tree.
- Smaller tree depth invariably means faster search.

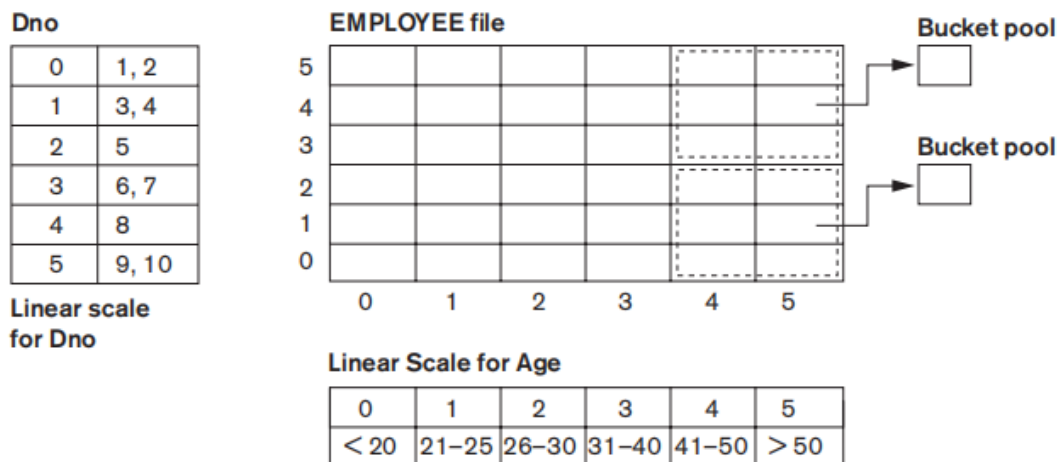
## Indexes on Multiple Keys

- 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 as composite keys

## Grid File

**Figure 18.14**

Example of a grid array on Dno and Age attributes.



- for n search keys, the grid array would have n dimensions.
- The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions.
- Grid files perform well in terms of reduction in time for multiple key access.
- However, they represent a space overhead in terms of the grid array structure.
- Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.

## References

- Elmasri R. and S. Navathe, Database Systems: Models, Languages, Design and Application Programming, Pearson Education, 2013.
- <https://www.baeldung.com/cs/b-trees-vs-btrees>
- <https://www.cs.usfca.edu/~galles/visualization/BTree.htm>
- <https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

**MODULE 3 ENDS**