KTU NOTES

The learning companion.

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

Website: www.ktunotes.in

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

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, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files

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.

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	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

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

QO. Retrieve the birth date andaddress 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	731 Fondren, 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	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
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.

```
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(.).

```
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;
```

Suppose *Dno* and *Lname* attributes of EMPLOYEE relation were called *Dnumber* and *Name* and the *Dname* attribute was called *Name*. To prevent ambiguity Q1 should be rephrased as Q1A

Q8. For each employee, retrieve the eemployee'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;
```

E.Fname	E.Lname	S.Fname	S.Lname
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)

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


```
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 sclause 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(Q 9) 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)

Ssn
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)

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
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

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;
```

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

Q11 retrieves the salary of every employee without distinct

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

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

(a)

Salary
30000
40000
25000
43000
38000
25000
25000
55000

Q11A :retrieves the salary of every employee using keyword DISTINCT

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


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

(b)

Salary
30000
40000
25000
43000
38000
55000

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:

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

And the *orders* table populated with the following records

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

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

supplier_id	supplier_name
1000	Microsoft
2000	Oracle
3000	Apple
4000	Samsung

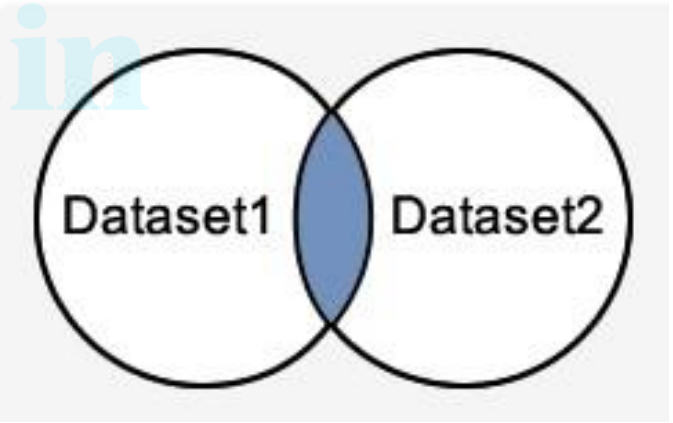
And the *orders* table populated with the following records

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

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.



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

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

INTERSECT

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

```
SELECT  ID, NAME, Amount, Date
     FROM Customers
     LEFT JOIN Orders
     ON Customers.ID = Orders.Customer_id
INTERSECT
     SELECT  ID, NAME, Amount, Date
     FROM Customers
     RIGHT JOIN Orders
     ON Customers.ID = Orders.Customer_id;
```

Customers Table:

ID	Name	Address	Age	Salary
1	Harsh	Delhi	20	3000
2	Pratik	Mumbai	21	4000
3	Akash	Kolkata	35	5000
4	Varun	Madras	30	2500
5	Souvik	Banaras	25	6000
6	Dhanraj	Siliguri	22	4500
7	Riya	Chennai	19	1500

Orders Table:

Oid	Date	Customer_id	Amount
102	2017-10-08	3	3000
100	2017-10-08	3	1500
101	2017-11-20	2	1560
103	2016-5-20	4	2060

```
SELECT  ID, NAME, Amount, Date
     FROM Customers
     LEFT JOIN Orders
     ON Customers.ID = Orders.Customer_id
INTERSECT
     SELECT  ID, NAME, Amount, Date
     FROM Customers
     RIGHT JOIN Orders
     ON Customers.ID = Orders.Customer_id;
```


Output:

ID	Name	Amount	Date
3	Akash	3000	2017-10-08
3	Akash	1500	2017-10-08
2	Pratik	1560	2017-11-20
4	Varun	2060	2016-05-20

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

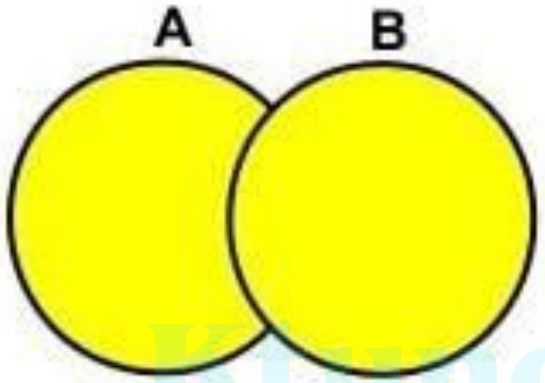
Table 2 – ORDERS table is as follows.

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

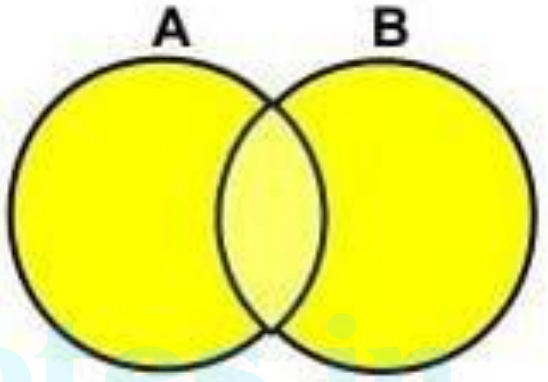
```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
EXCEPT
   SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

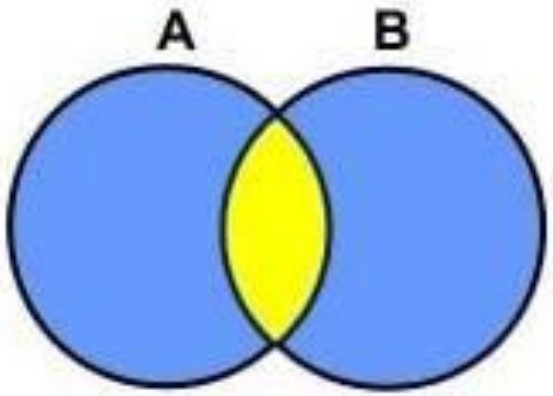
ID	NAME	AMOUNT	DATE
1	Ramesh	NULL	NULL
5	Hardik	NULL	NULL
6	Komal	NULL	NULL
7	Muffy	NULL	NULL



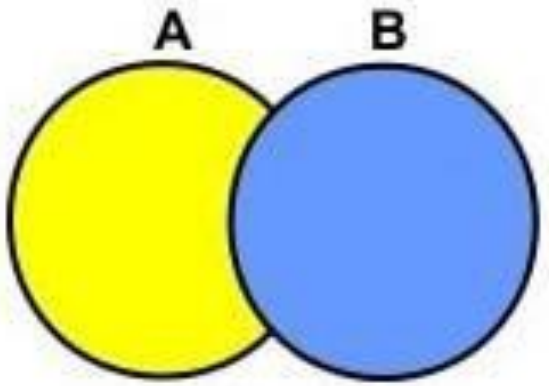
UNION



UNION ALL



INTERSECT



EXCEPT/MINUS

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))

```
SELECT id, first_name, last_name, first_name || last_name,
                salary, first_name || salary FROM myTable
```

Output (Third and Fifth Columns show values concatenated by operator ||)

id	first_name	last_name	first_name||last_name	salary	first_name||sa
1	Rajat	Rawat	RajatRawat	10000	Rajat10000
2	Geeks	ForGeeks	GeeksForGeeks	20000	Geeks20000
3	Shane	Watson	ShaneWatson	50000	Shane50000
4	Kedar	Jadhav	KedarJadhav	90000	Kedar90000

Example 1: Using character literal

Syntax:

```
SELECT id, first_name, last_name, salary,
       first_name||' has salary '||salary as "new"  FROM myTable
```

```
Output : (Concatenating three values and giving a name 'new')
id   first_name   last_name   salary              new
1    Rajat         Rawat      10000       Rajat has salary 10000
2    Geeks         ForGeeks   20000       Geeks has salary 20000
3    Shane         Watson     50000       Shane has salary 50000
4    Kedar         Jadhav     90000       Kedar has salary 90000
```


Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'

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 ascending order according to the column ROLL_NO

SELECT * FROM Student ORDER BY ROLL_NO ASC;

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

```
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

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 employee 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;
```

if we want descending alphabetical order on Dname and ascending order on Lname, Fname, the ORDER BY clause of Q15 can be written as

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

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

Q. Returns the names of employees whose salary is greater than the salary of all the employ

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

The keyword ALL can be combined with each of >, >=, <, <=, and <>, these operators. For example, the comparison condition (*v > ALL V*) *returns TRUE if the value v is greater than all the values in the set (or multiset) V.*

Q . Select the Essns of all employees who work the same (project, hours)combination on some project that employee ‘John Smith’ (whose Ssn =‘123456789’) works on.

```
SELECT   DISTINCT Essn
FROM     WORKS_ON
WHERE    (Pno, Hours) IN ( SELECT   Pno, Hours
                           FROM     WORKS_ON
                           WHERE    Essn='123456789' );
```

In this example, the IN operator compares the subtuple of values in parentheses (Pno, Hours) within each tuple in WORKS_ON with the set of type-compatible tuples produced by the nested query.

Q 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:  SELECT  E.Fname, E.Lname
      FROM    EMPLOYEE AS E
      WHERE   E.Ssn IN ( SELECT  Essn
                         FROM    DEPENDENT AS D
                         WHERE   E.Fname=D.Dependent_name
                         AND E.Sex=D.Sex );
```

It is generally advisable to create tuple variables (aliases) for all the tables referenced in an SQL query to avoid potential errors and ambiguities

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)
```

```
Q16A:  SELECT    E.Fname, E.Lname
       FROM      EMPLOYEE AS E, DEPENDENT AS D
       WHERE     E.Ssn=D.Essn AND E.Sex=D.Sex
                 AND E.Fname=D.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 alwaysbe 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
        );
```

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

Fname	Lname
John	Smith
Franklin	Wong

- 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	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
Ahamad	Jabbar
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

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

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

Q21. Retrieve the total number of employees in the company

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

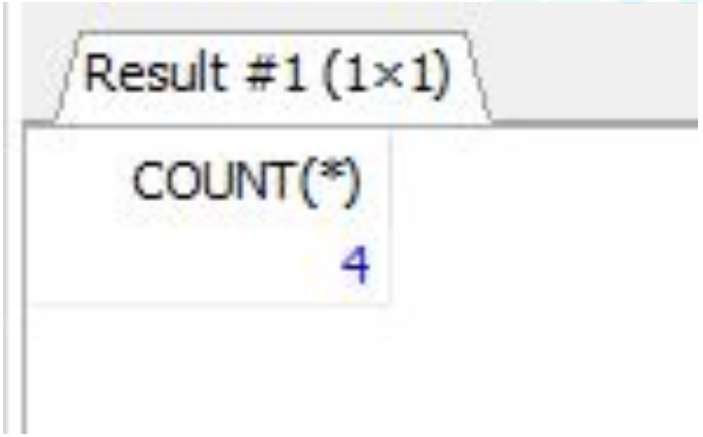
```
SELECT COUNT(*)
FROM EMPLOYEE;
```

Result #1 (1×1)

COUNT(*)
8

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

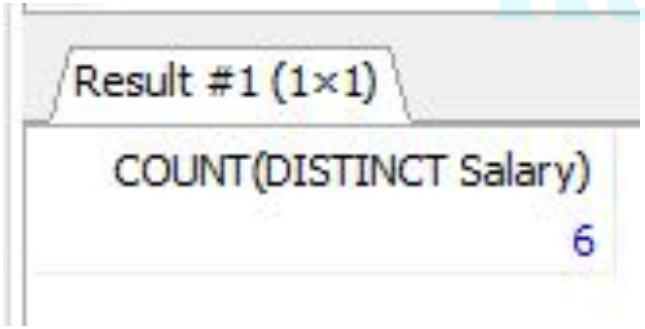
```
Q22:    SELECT    COUNT (*)
        FROM      EMPLOYEE, DEPARTMENT
        WHERE     DNO=DNUMBER AND DNAME='Research';
```



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.

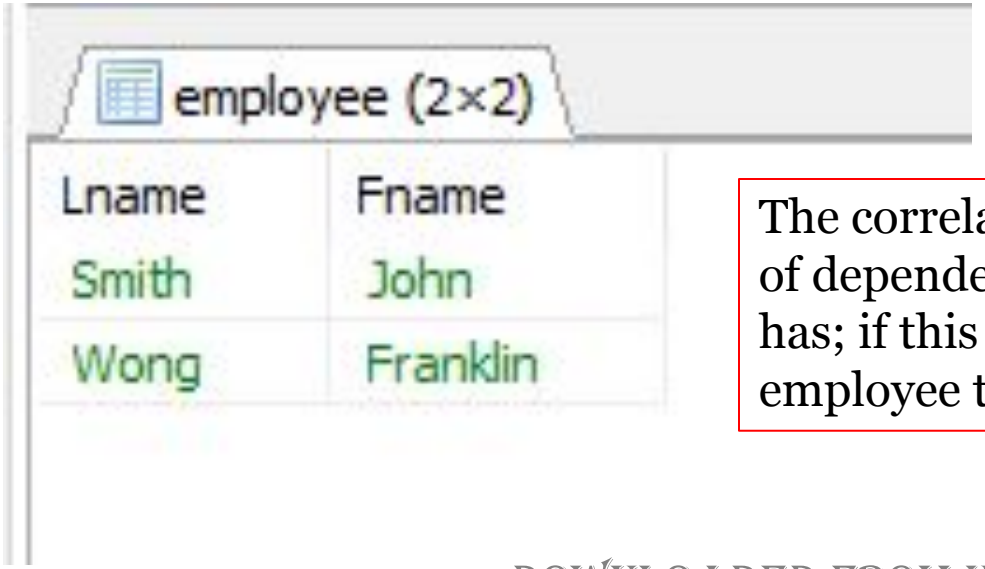
```
Q23:  SELECT  COUNT (DISTINCT Salary)
      FROM    EMPLOYEE;
```

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.

(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

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;
```

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 (3×6)

Pnumber	Pname	COUNT(*)
1	ProductX	2
2	ProductY	3
3	ProductZ	2
10	Computarization	3
20	Reorganization	3
30	Newbenifits	3

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 (3×5)

Pnumber	Pname	COUNT(*)
1	ProductX	2
2	ProductY	3
3	ProductZ	2
10	Computarization	1
20	Reorganization	1

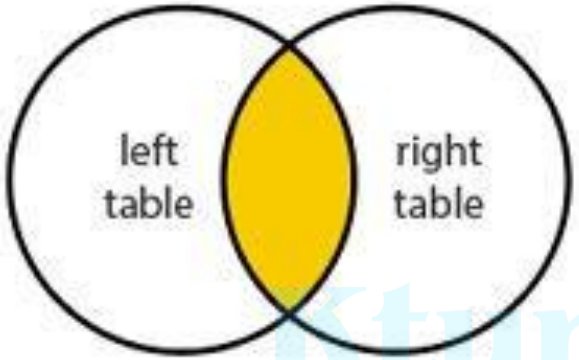
SQL JOIN

- 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

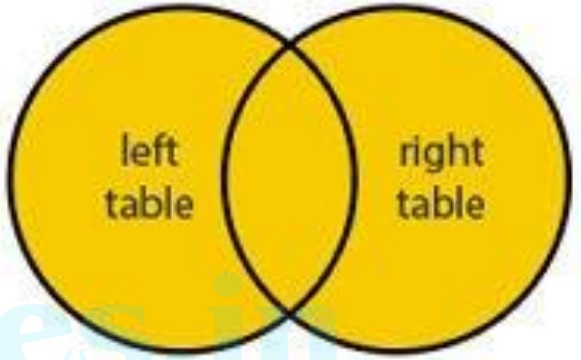
- Four different types of JOINs

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.

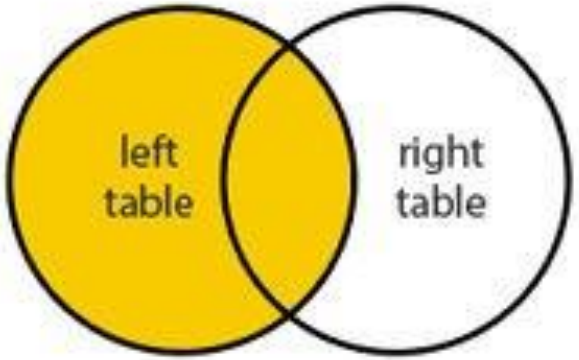
INNER JOIN



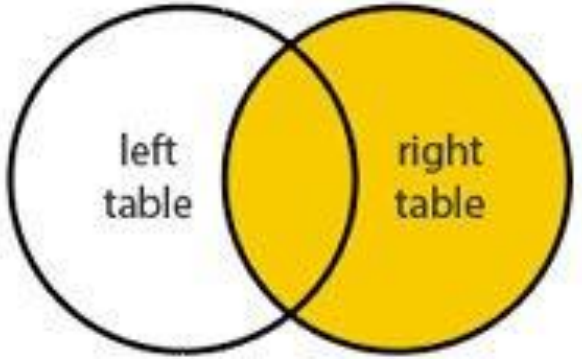
FULL JOIN



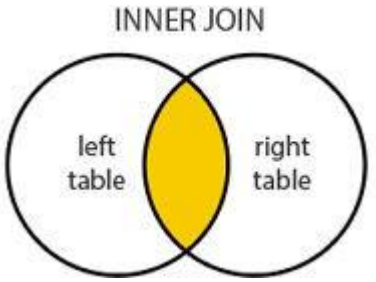
LEFT JOIN



RIGHT JOIN



JOIN Syntax

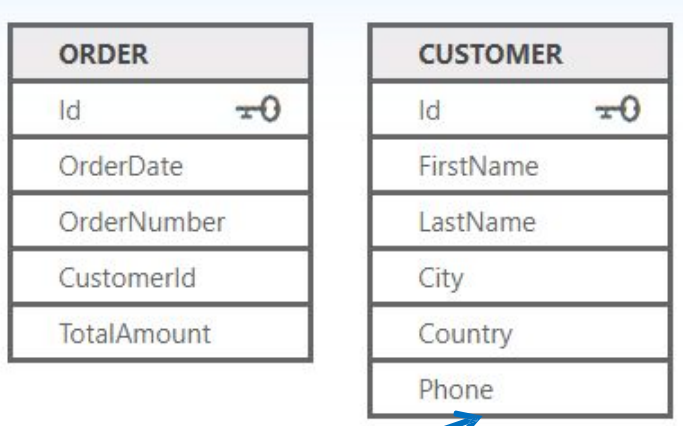


- The general syntax is

```
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

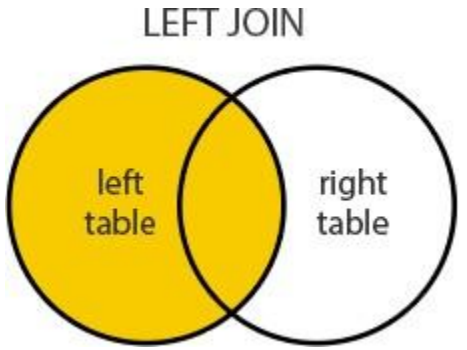
```
SELECT   OrderNumber,TotalAmount,FirstName,  LastName,City,
Country
FROM  Order JOIN  Customer
ON Order.CustomerId  =  Customer.Id
```


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

...

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

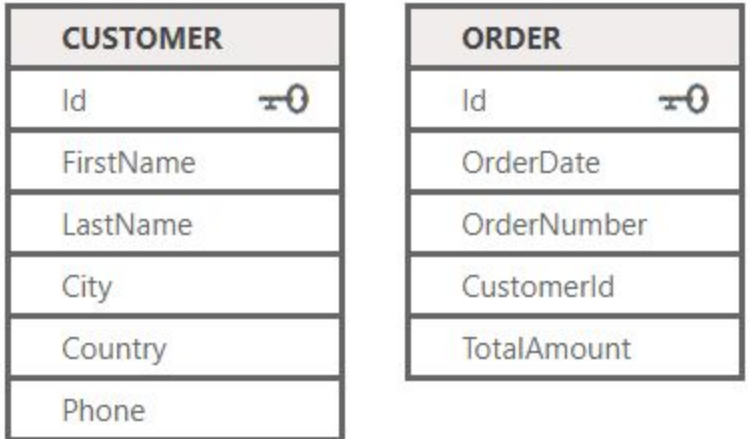
- The general LEFT JOIN syntax is

```
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  ONcolumn-name1  =
    column-name2
```

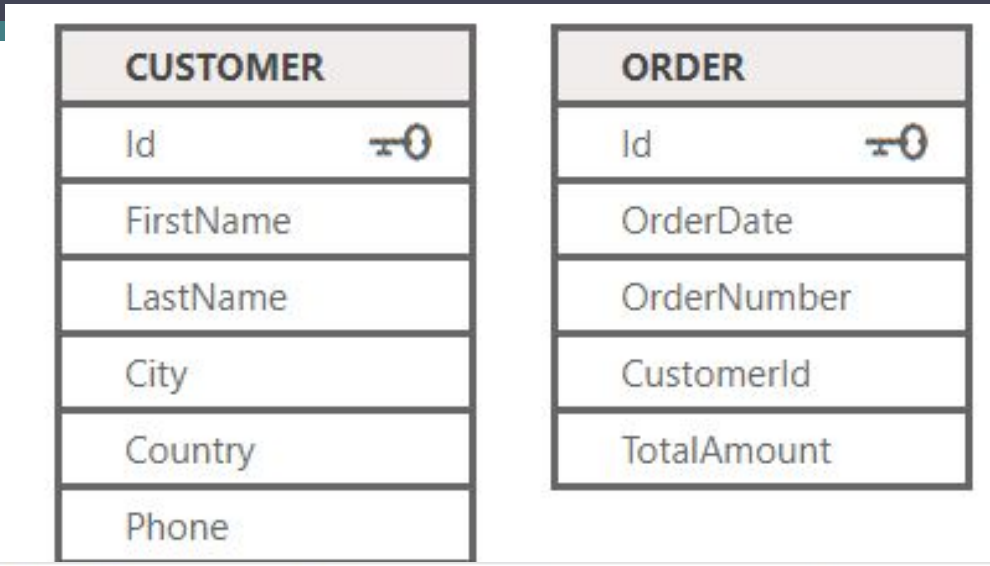
SQL LEFT JOIN
Example



- 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
```

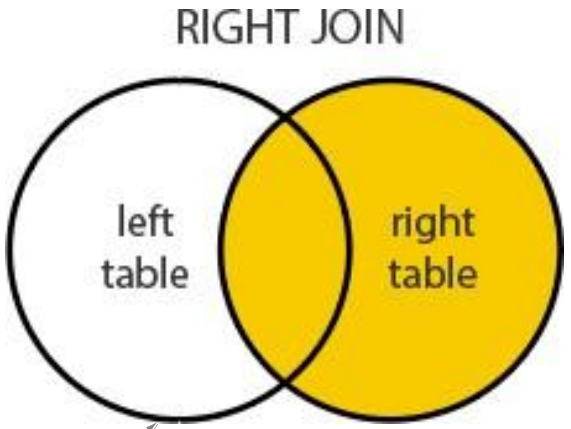
- The ORDER BY TotalAmount shows the customers without orders first (i.e. TotalMount is NULL).



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

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

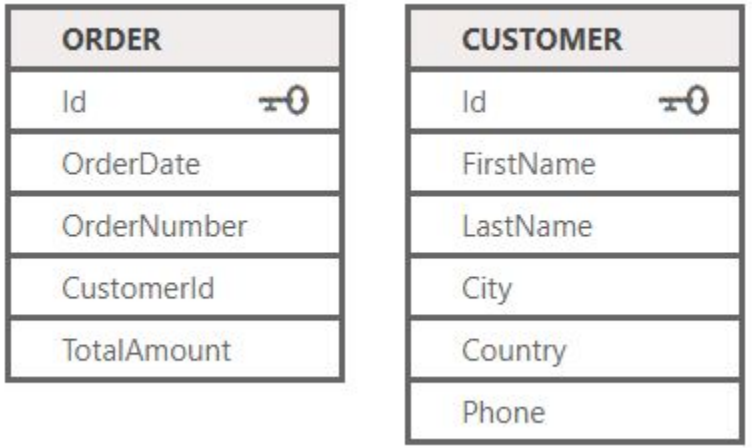
- The general syntax is

```
SELECT column-names
  FROM table-name1 RIGHT JOIN
    table-name2 ONcolumn-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 =
```

SQL RIGHT JOIN
Examples



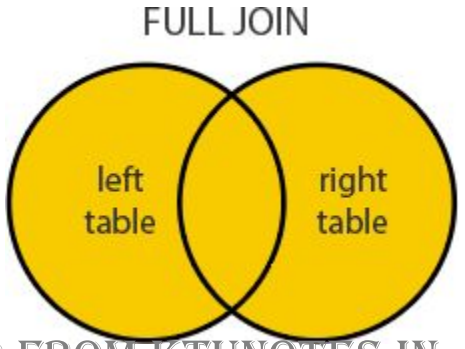
- **Problem: List customers that have not placed orders**

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


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

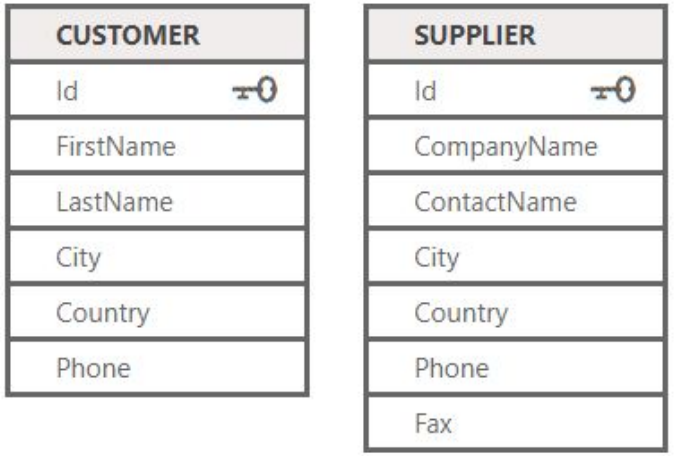
- The general syntax is:

```
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  ONcolumn-name1   =
    column-name2
```

SQL FULL JOIN
Examples



- Problem: Match all customers and suppliers by country

```
SELECT  C.FirstName,  C.LastName,
C.Country  AS  CustomerCountry,
       S.CountryAS SupplierCountry, S.CompanyName
  FROM  CustomerC     FULL  JOIN  SuppliersS
    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.

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 databasewe 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 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	Computarization	10.0
Ahamad	Jabbar	Computarization	35.0
Alicia	Zelaya	Computarization	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

company.dept_info V2

Dept_name	No_of_emps	Total_sal
Administration	3	93,000
Headquaters	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	Computarization	10.0
Ahamad	Jabbar	Computarization	35.0
Alicia	Zelaya	Computarization	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 (2×2)

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

```
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 of views is complicated and can be ambiguous

- an update on a view defined on a single table without any aggregate functions can be mapped to an update on the underlying base table under certain conditions.
- For a view involving joins,
  - an update operation may be mapped to update operations on the underlying base relations in multiple ways.
- Hence, it is often not possible for the DBMS to determine which of the updates is intended.

- consider the WORKS_ON1 view, and suppose that we issue the command to update the PNAME attribute of 'John Smith' from 'ProductX' to 'ProductY'.
- This view update is shown in UV1

```
UV1:    UPDATE WORKS_ON1
        SET    Pname = 'ProductY'
        WHERE  Lname='Smith' AND Fname='John'
               AND Pname='ProductX';
```

- a view update is feasible when only one possible update on the base relations can accomplish the desired update effect on the view.
- Whenever an update on the view can be mapped to more than one update on the underlying base relations,
  - we must have a certain procedure for choosing one of the possible updates as the most likely one.
  - Some researchers have developed methods for choosing the most likely update, while other researchers prefer to have the user choose the desired update mapping during view definition.

- A view with a single defining table is updatable if the view attributes contain the primary key of the base relation, as well as all attributes with the NOT NULL constraint that do not have default values specified.
- Views defined on multiple tables using joins are generally not updatable.
- Views defined using grouping and aggregate functions are not updatable.

Specifying Constraints as Assertions and Actions as Triggers

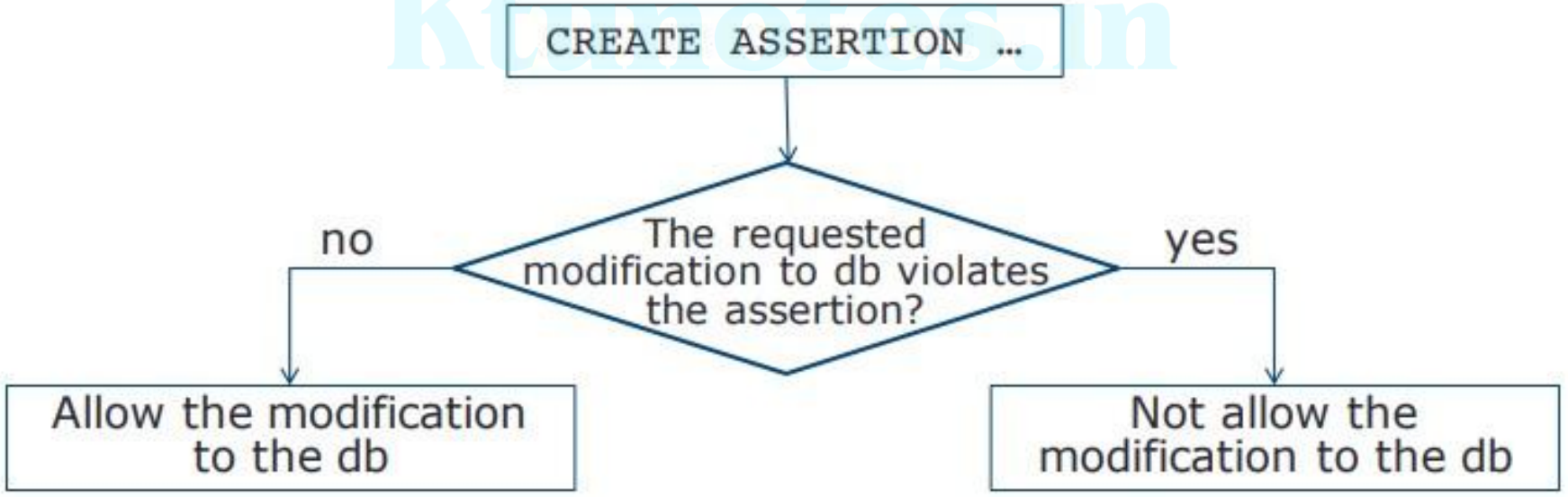
Specifying General Constraints as Assertions in SQL

- 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

To specify the constraint that *the salary of an employee must not be greater than the salary of the manager of the department that the employee works for*

in SQL, we can write the following assertion:.

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

- The constraint name SALARY_CONSTRAINT is followed by the keyword CHECK, which is followed by a **condition in parentheses that must hold true on every database** state for the assertion to be satisfied.
- The constraint name can be used later to refer to the constraint or to modify or drop it.
- constraints can be specified using the EXISTS and NOT EXISTS style of SQL conditions.
- Whenever some tuples in the database cause the condition of an ASSERTION statement to evaluate to FALSE, the constraint is **violated**

Type of constraint	Where declared	When activated	Guaranteed to hold?
Attributed-based CHECK	With attribute	On insertion to relation or attribute update	No if subqueries
Tuple-based CHECK	Element of relation schema	On insertion to relation or tuple update	No if subqueries
ASSERTION	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 Tri

```
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;
```

- 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.

- [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, Singe level indices, numerical examples, Multi-level-indices, numerical examples, B-Trees & B+-Trees (structure only, algorithms not required), Extendible Hashing, Indexing on multiple keys – grid files

Fixed 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.

Spanned and Unspanned Organization

- 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 $bfr = \lfloor B / R \rfloor$ records per block.
- The value bfr is called the blocking factor for the file.

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.

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.

- 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.

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.

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 needed to search a record?

No. of records,	r	= 30,000	
Block size,	B	= 1024 bytes	
Record size,	R	= 100 bytes	
Blocking factor,	bfr	= $\lfloor B / R \rfloor$	= $\lfloor 1024 / 100 \rfloor$ = 10 records/block
No. blocks,	b	= $\lceil r/bfr \rceil$	= $\lceil 30,000/ 10 \rceil$ = 3000 blocks
If linear search		= 3000 / 2	= 1500 block access
If binary search		= $\log_2(3000)$	= 12 block access

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.

Blocking factor,	bfr	= ⌊B / R⌋	= ⌊1024 / (9 + 6)⌋	= 68 entries/block
Total no. of index entries		= No. of blocks in data file		= 3000
No. of blocks needed		= ⌈r/bfr⌉	= ⌈3000/68⌉	= 45 blocks
If binary search		= $\log_2(45)$	= 6 block access	
Total no. of search		= 6 block access + 1 block access (data file)		
		= 7 block access		

Clustering Index

- Defined on an ordered data file.
- The data file is ordered on a non-key field unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
- Includes one index entry for each distinct value of the field;
  - the index entry points to the first data block that contains records with that field value.
- It is another example of nondense index where Insertion and Deletion is relatively straightforward with a clustering index.

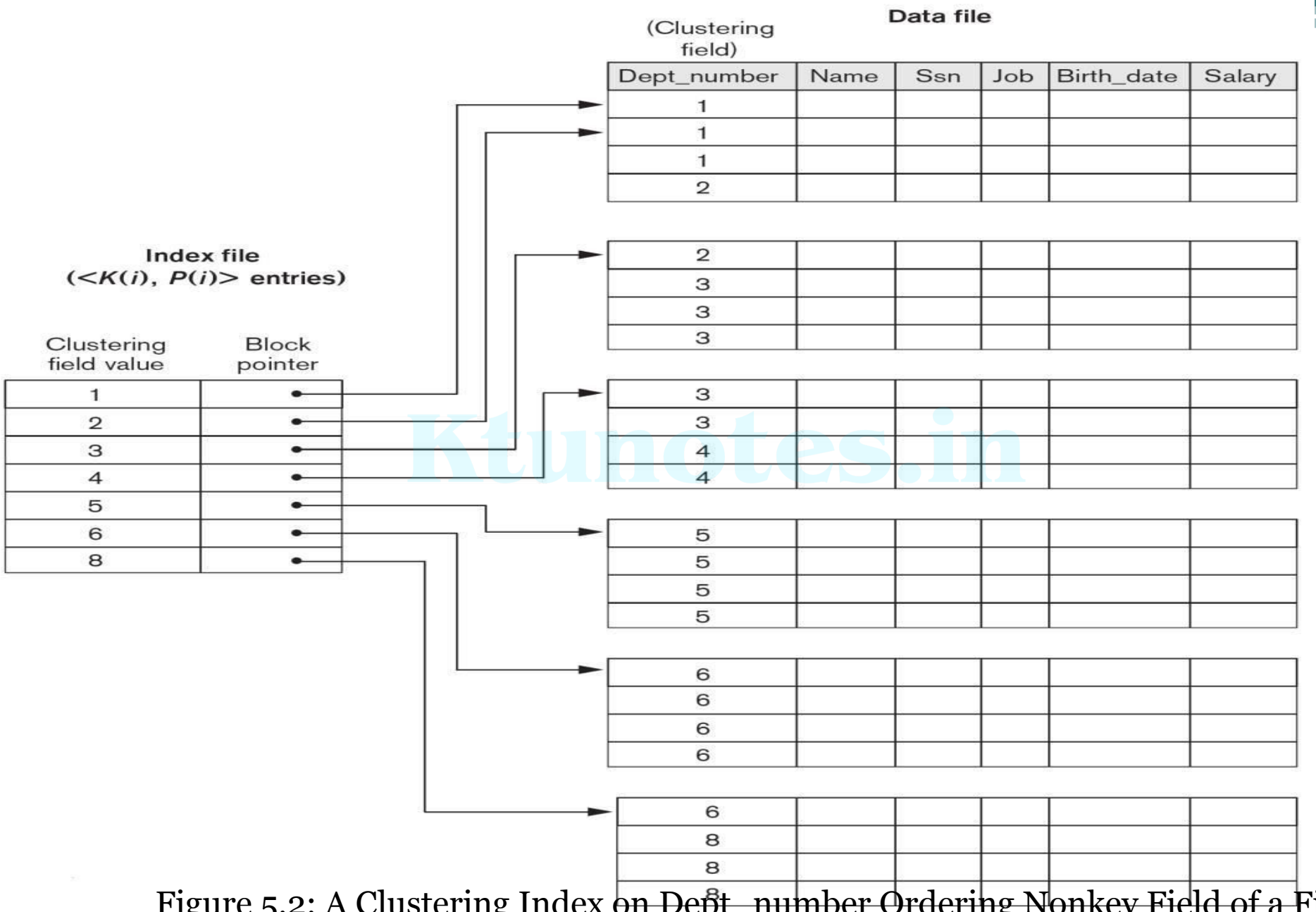


Figure 5.2: A Clustering Index on Dept_number Ordering Nonkey Field of a File

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

- 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 secondary index file.

Total no. of index entries	= No. of records in data file		= 30000
No. of blocks needed	= ⌐ r/bfr ¬	= ⌐30000/68 ¬	= 442 blocks
If binary search	= $\log_2(442)$	= 9 block access	
Total no. of search	= 9 block access + 1 block access (data file)		
	= 10 block access		

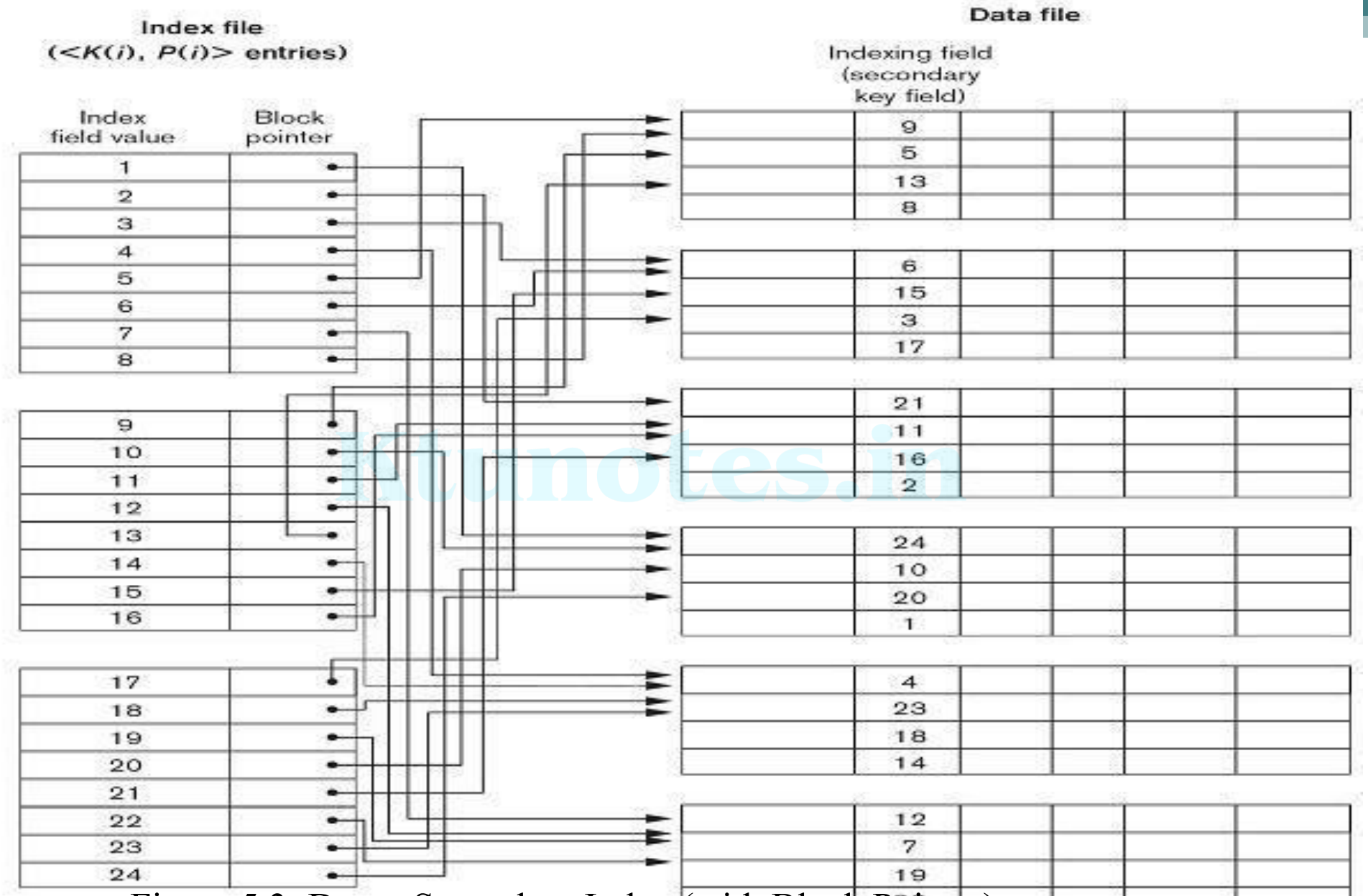


Figure 5.3: Dense Secondary Index (with Block Pointer) on a Non-Ordering Key Field of the File

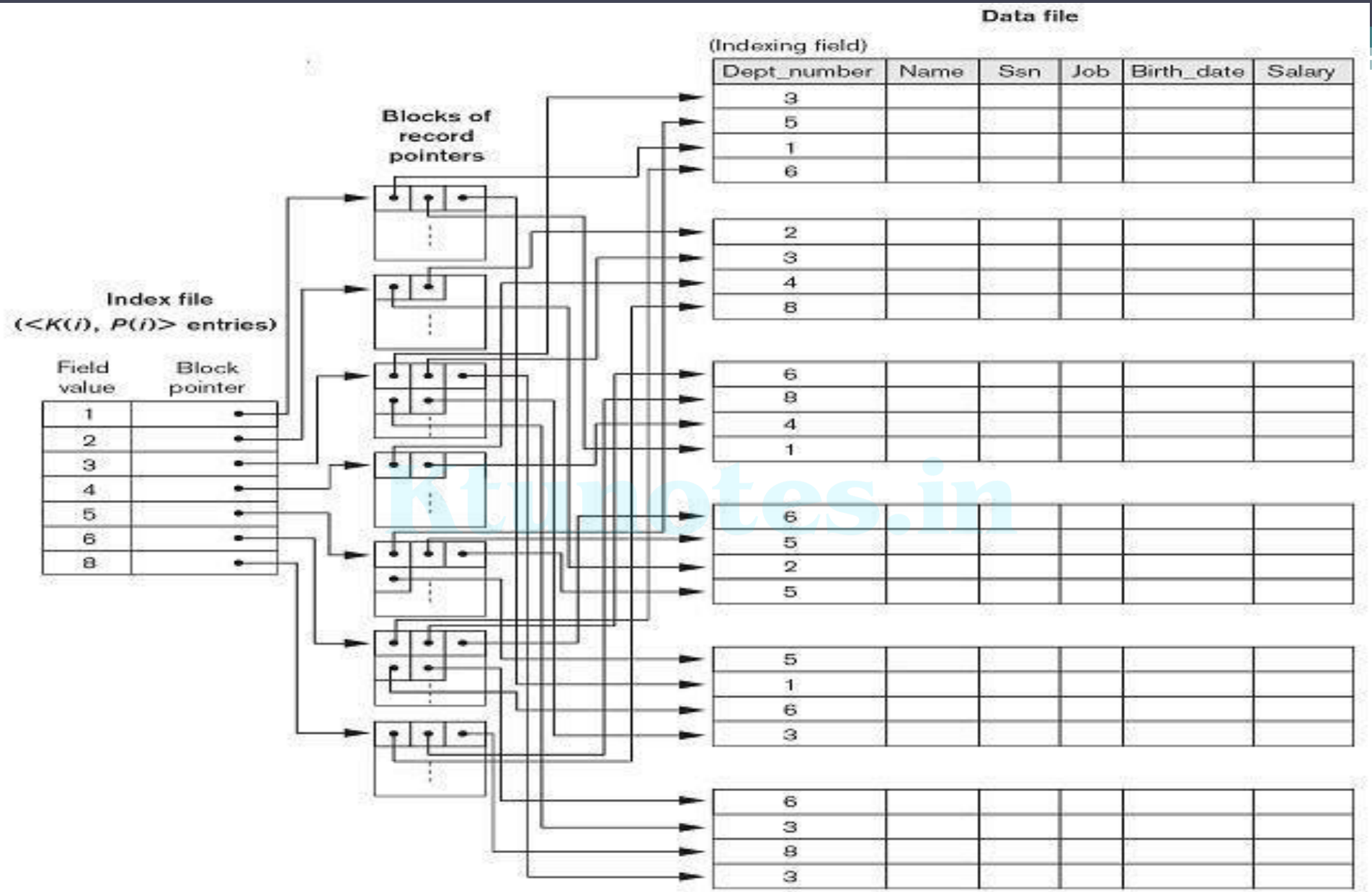


Figure 5.4: Secondary Index (with Record Pointer) on a Non Key Field implemented using one level of indirection so that Index entries are of Fixed Length and have unique field values

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[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or number of distinct index field values[c]	Dense or Nondense	No

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

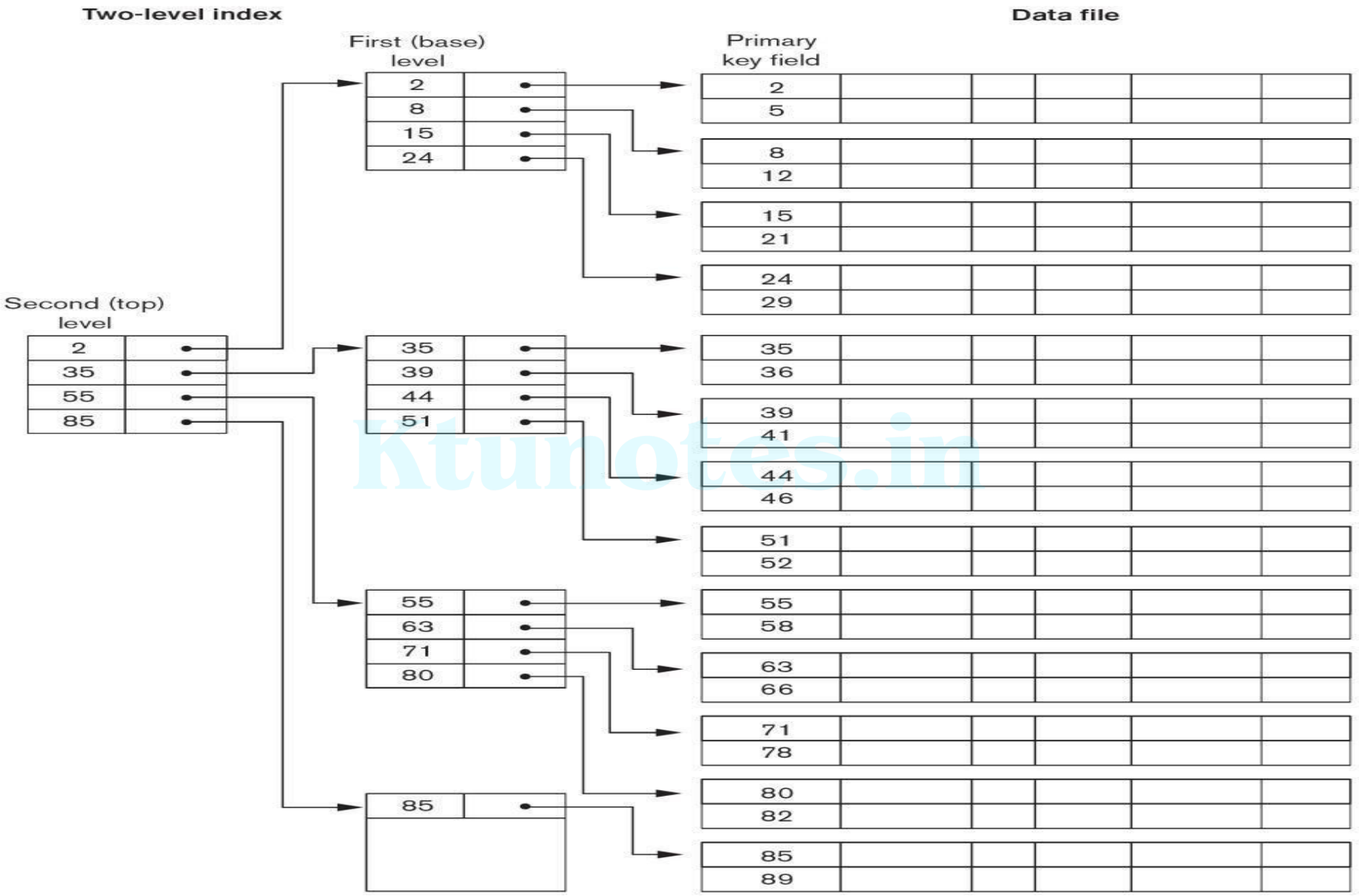


Figure 5.5: Two-Level Primary Index

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 <P1, K1, P2, K2, ..., Pq–1, Kq–1, Pq>, where q ≤ p.
- Each Pi is a pointer to a child node (or a NULL pointer), and each Ki 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, K1 < K2 < ... < Kq–1.
  2. For all values X in the subtree pointed at by Pi, we have Ki–1<X < Ki for 1<i < q; X < Ki for i=1; and Ki–1 <X for i = q

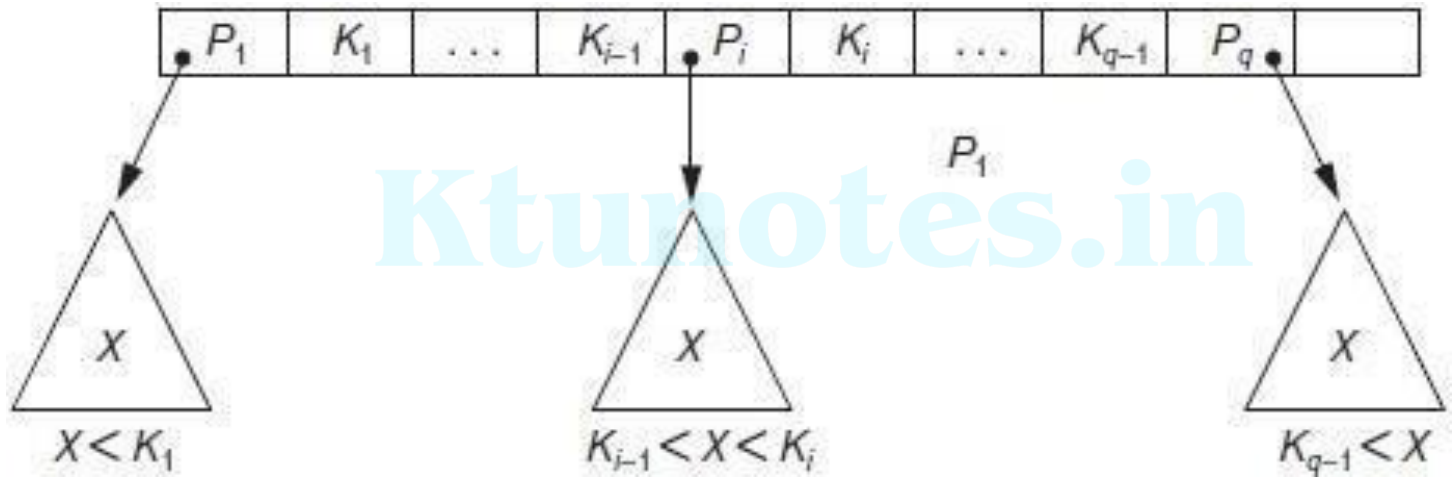


Figure 5.6: A node in a search tree with pointers to subtrees below it

B-Trees

- 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
   - <P1, <K1, Pr1>, P2, <K2, Pr2>, ..., <Kq–1, Prq–1>, Pq> where q ≤ p. Each Pi is a tree pointer—a pointer to another node in the Btree. Each Pri is a data pointer—a pointer to the record whose search key field value is equal to Ki.

2. Within each node, $K1 < K2 < \ldots < Kq-1$.
3. For all search key field values X in the subtree pointed at by Pi, we have: $Ki-1 < X < Ki$ for $1 < i < q$; $X < Ki$ for $i = 1$; and $Ki-1 < X$ 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 Pi are NULL

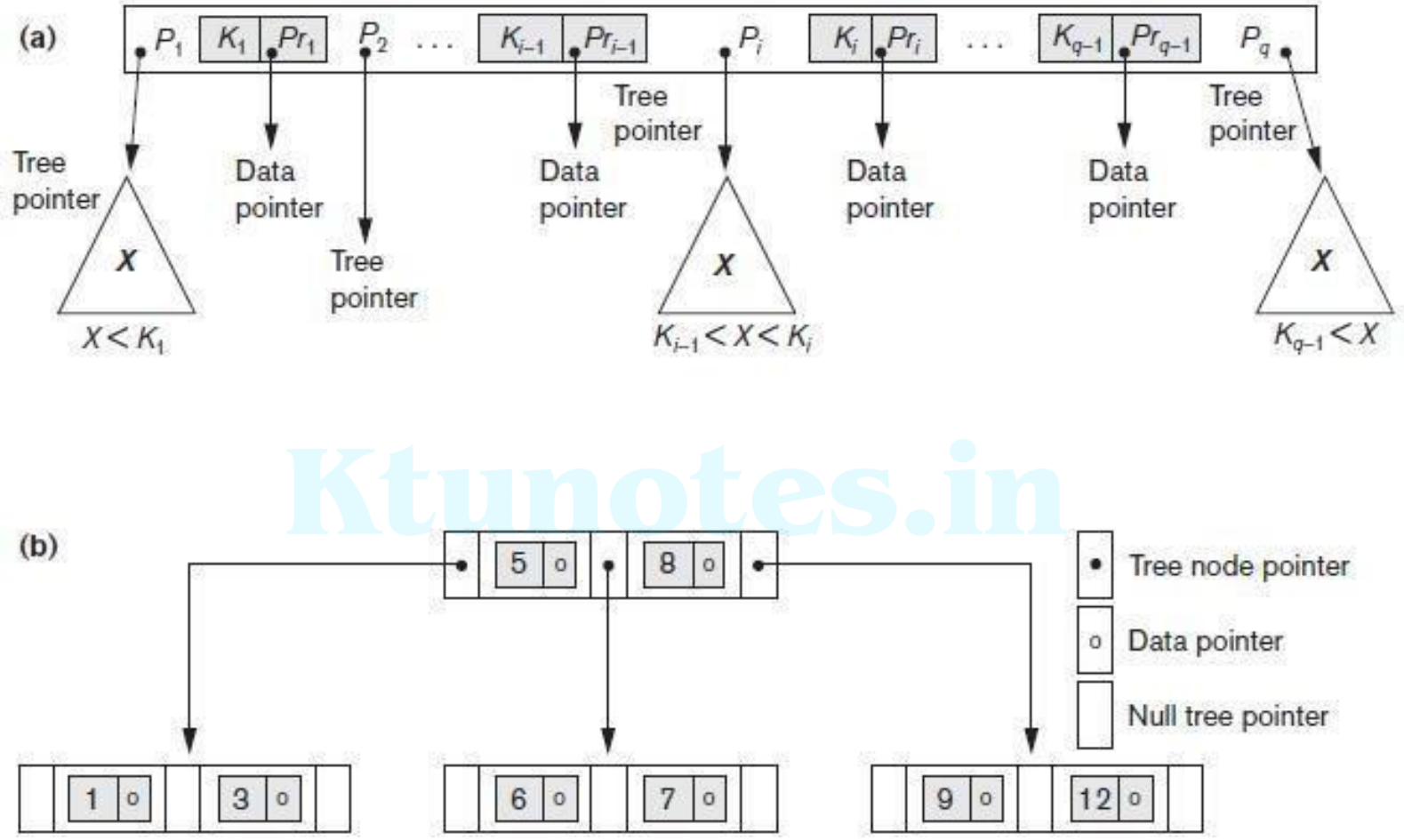


Figure 5.7: B-tree structures. (a) A node in a B-tree with $q - 1$ search values.

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

- 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

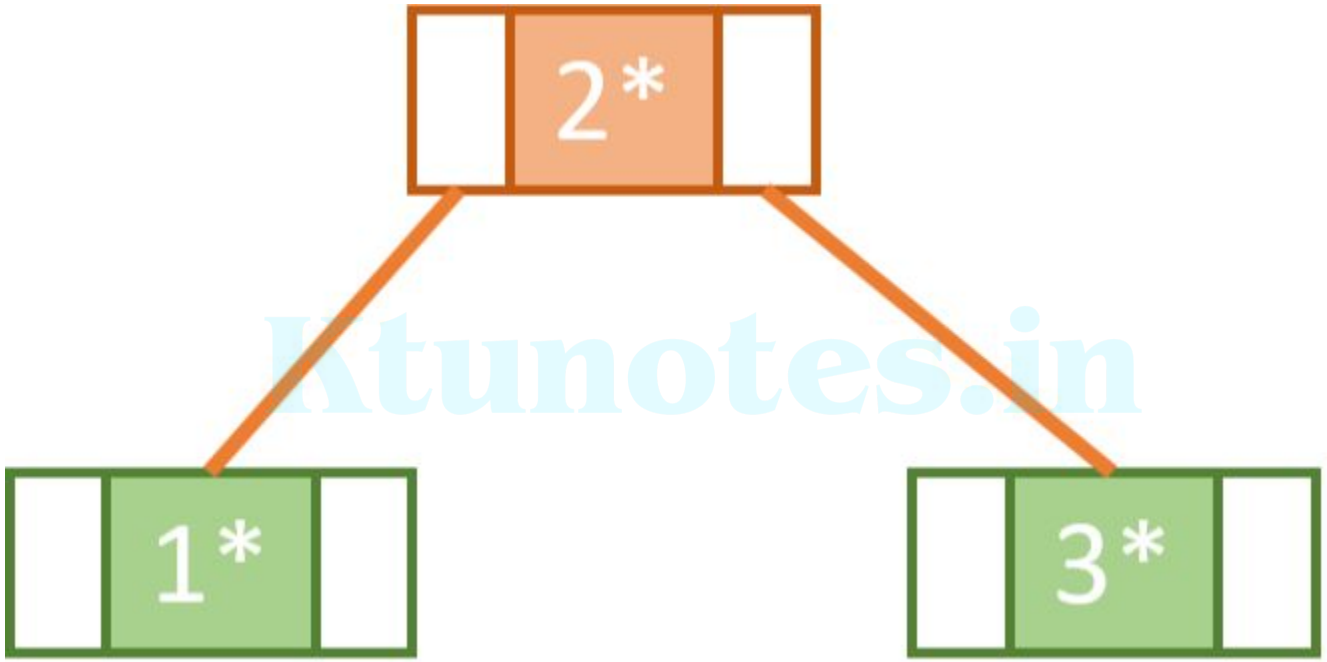
Building a B-tree

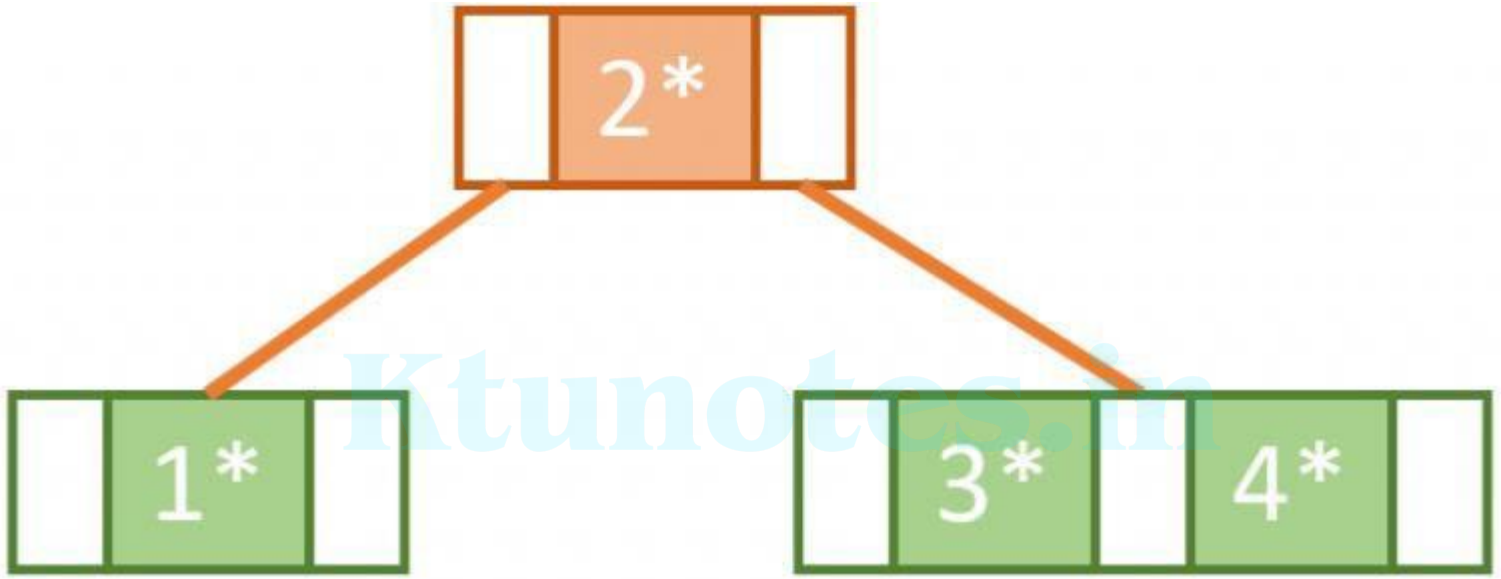
- Since we're starting with an empty tree, the first item we insert will become the root node of our tree.
- At this point, the root node has the key/value pair.
- The key is 1, but the value is depicted as a star to make it easier to represent, and to indicate it is a reference to a record.
- The root 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:

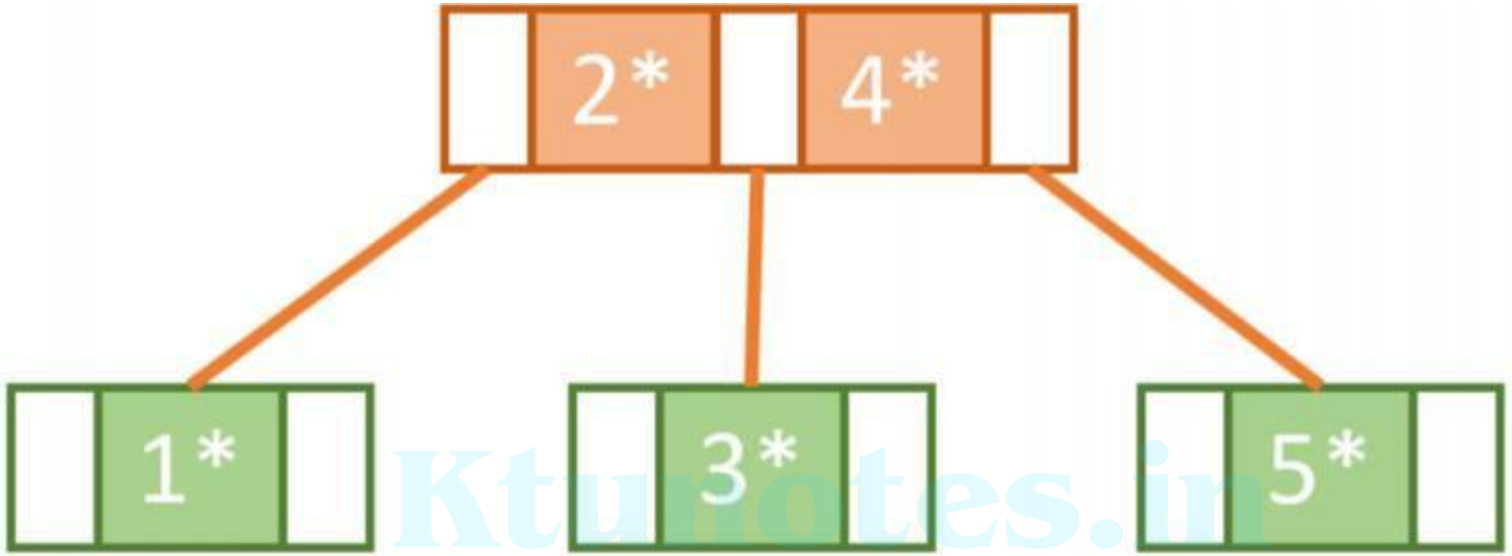
Next, if we wanted to insert 3, for us to keep the tree balanced and fulfilling the conditions of a B-tree we need to perform what is called a split operation.

We can determine how to split the node by picking the middle key.

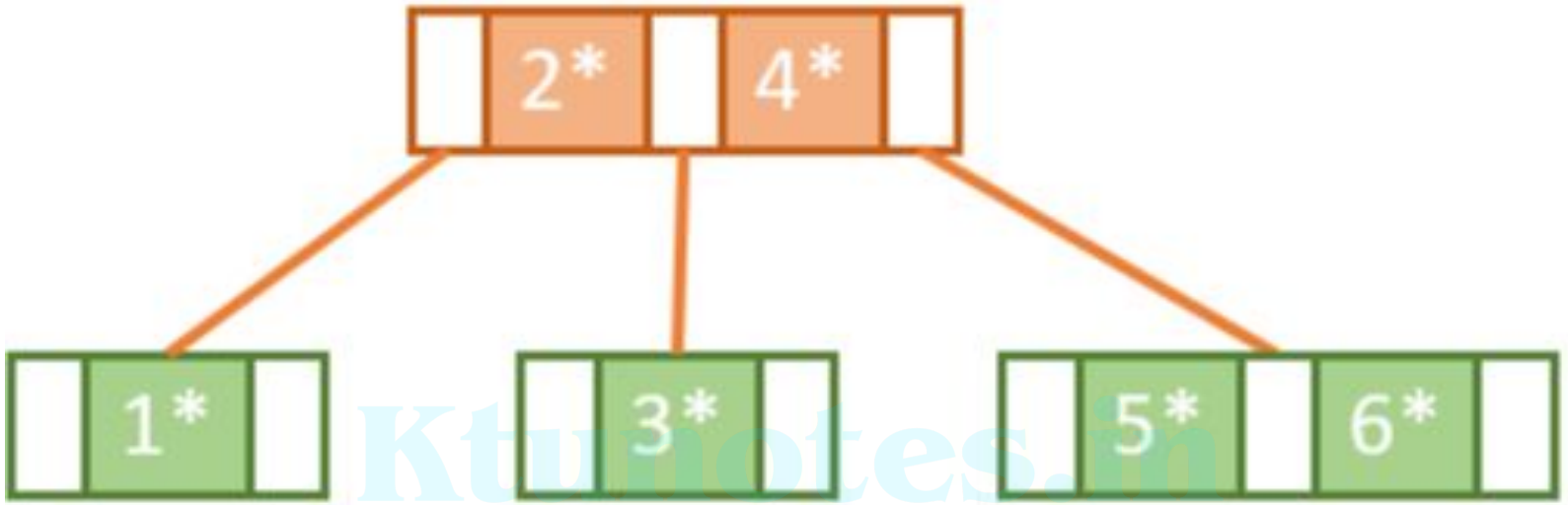




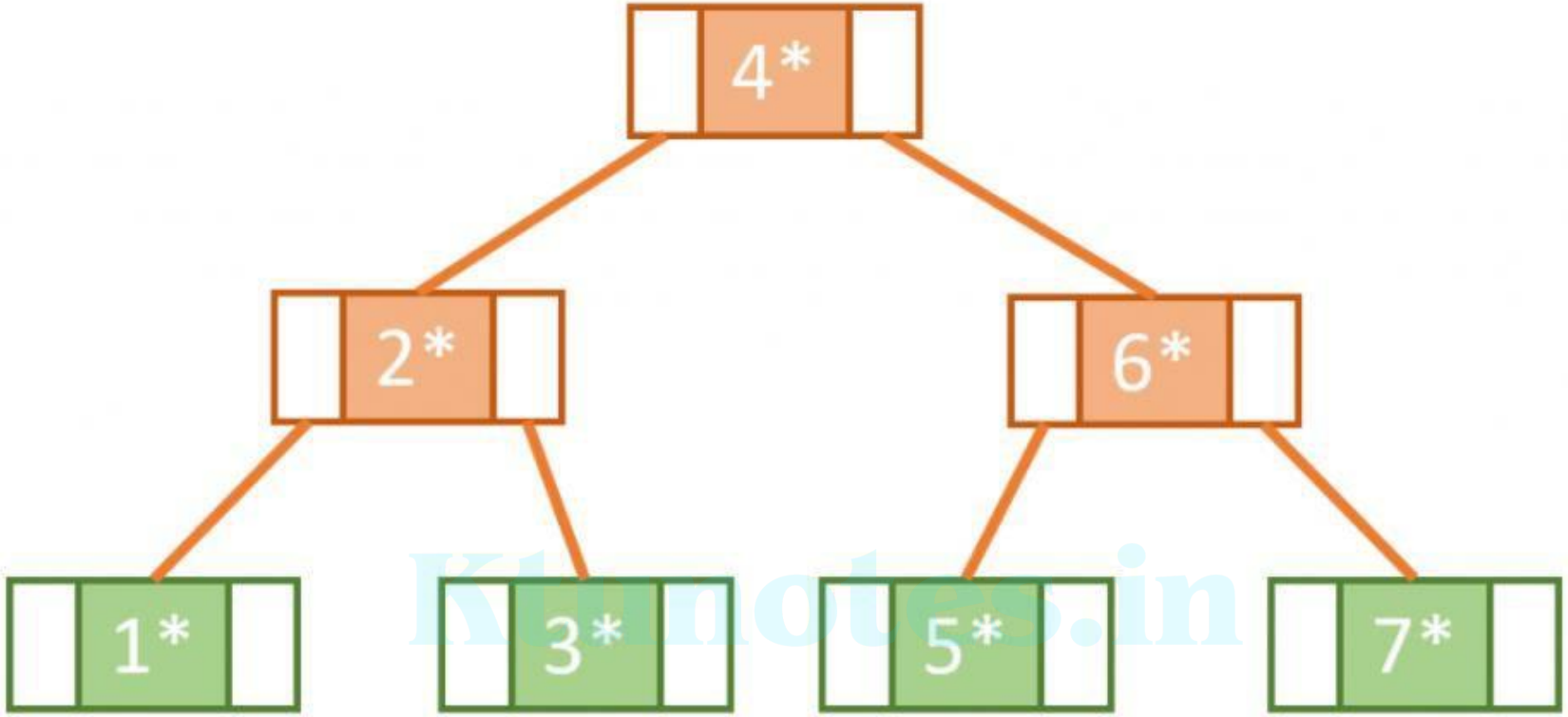
Now, let's insert 4. To determine where this needs to be placed we must remember that B-trees are organized such that sub-trees on the right have greater keys than sub-trees on the left. Consequently, Key 4 belongs in the right sub-tree. And since the right sub-tree still has the capacity, we can simply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



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.

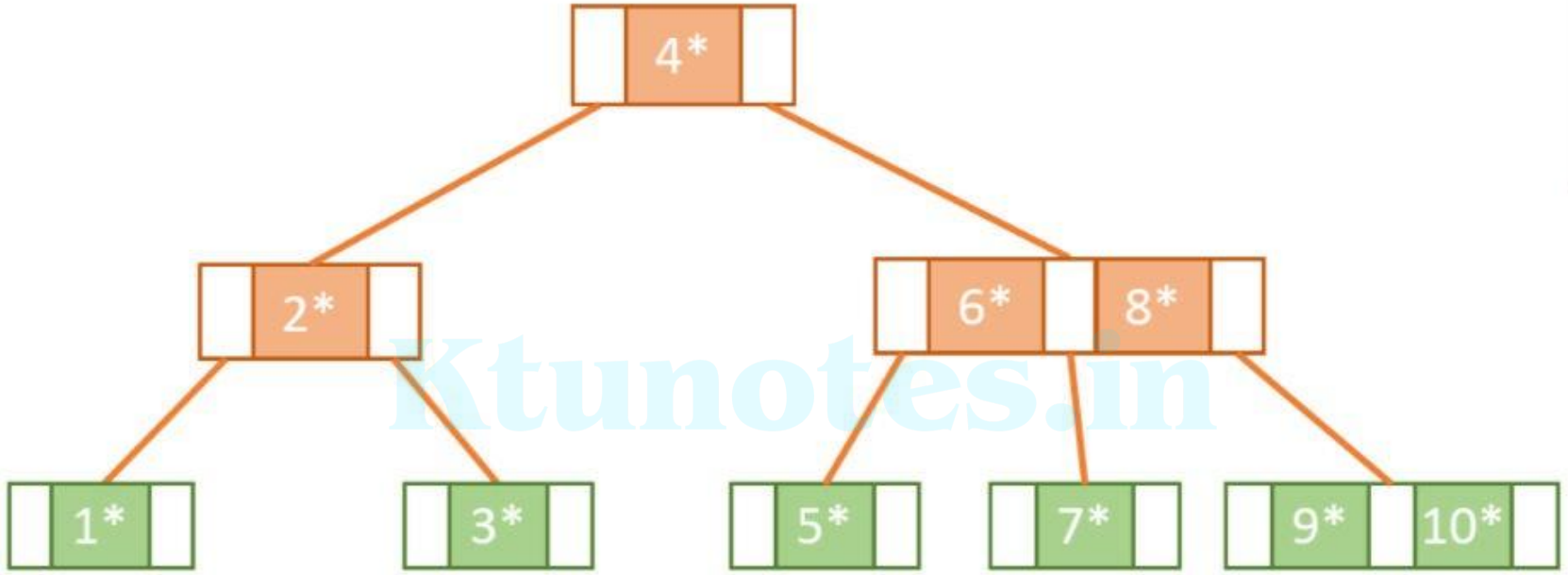


This rebalancing gives us space in the rightmost sub-tree to insert 6:



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.



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 <P1, K1, P2, K2, ..., Pq – 1, Kq –1, Pq> where q ≤ p and each Pi is a tree pointer.
  2. Within each internal node, K1 < K2 < ... < Kq–1.
  3. For all search field values X in the subtree pointed at by Pi, we have Ki–1 < X≤ Ki for 1 < i < q; X ≤ Ki for i = 1; and Ki–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 ⌈(p/2)⌉ tree pointers. The root node has at least two tree pointers if it is an internal node.
  6. An internal node with q pointers, q ≤ 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 <<K1, Pr1>, <K2, Pr2>, ..., <Kq–1, Prq–1>, Pnext> where q ≤ p, each Pri is a data pointer, and Pnext points to the next leaf node of the B+-tree.
  2. Within each leaf node, K1 ≤ K2 ... , Kq–1, q ≤ p.
  3. Each Pri is a data pointer that points to the record whose search field value is Ki or to a file block containing the record
  4. Each leaf node has at least ⌈(p/2)⌉ values.
  5. All leaf nodes are at the same level.

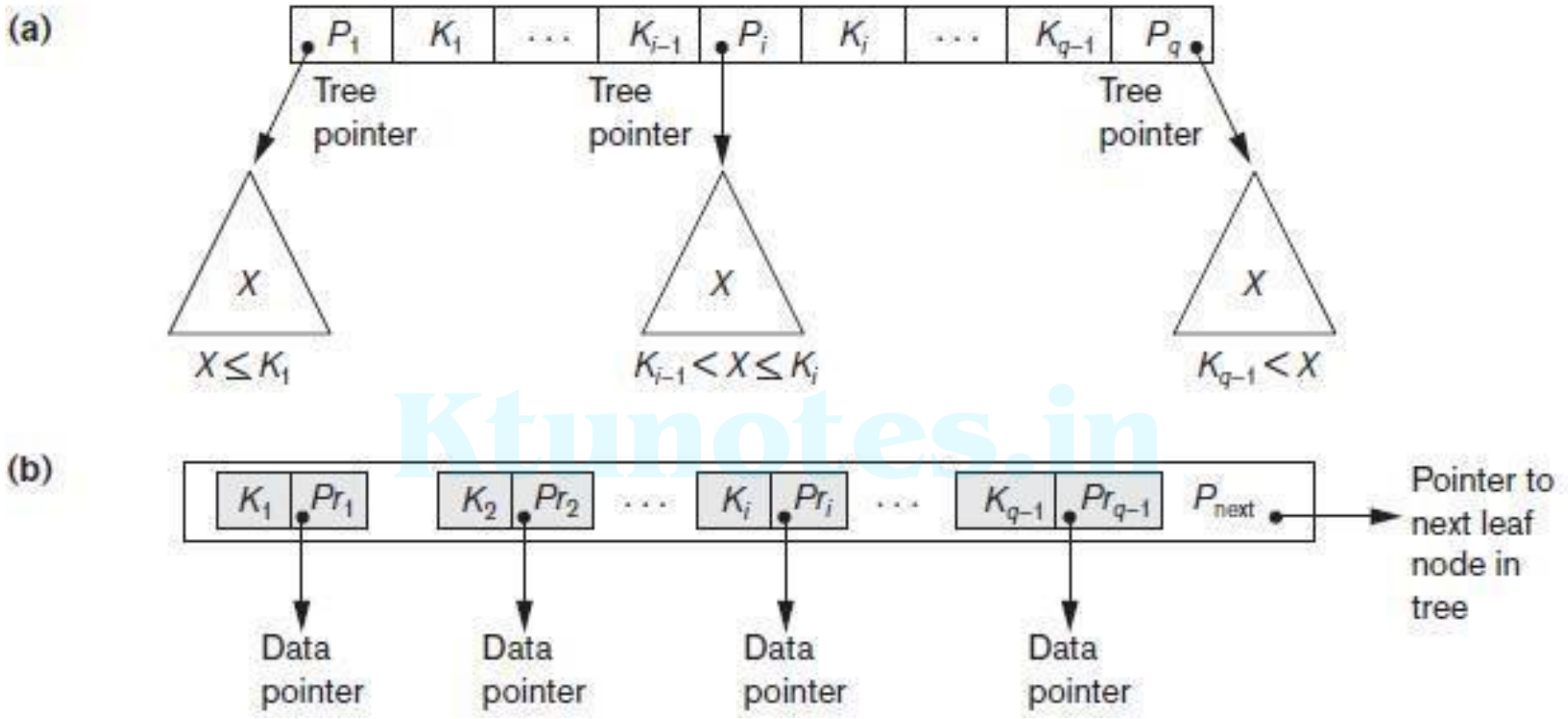
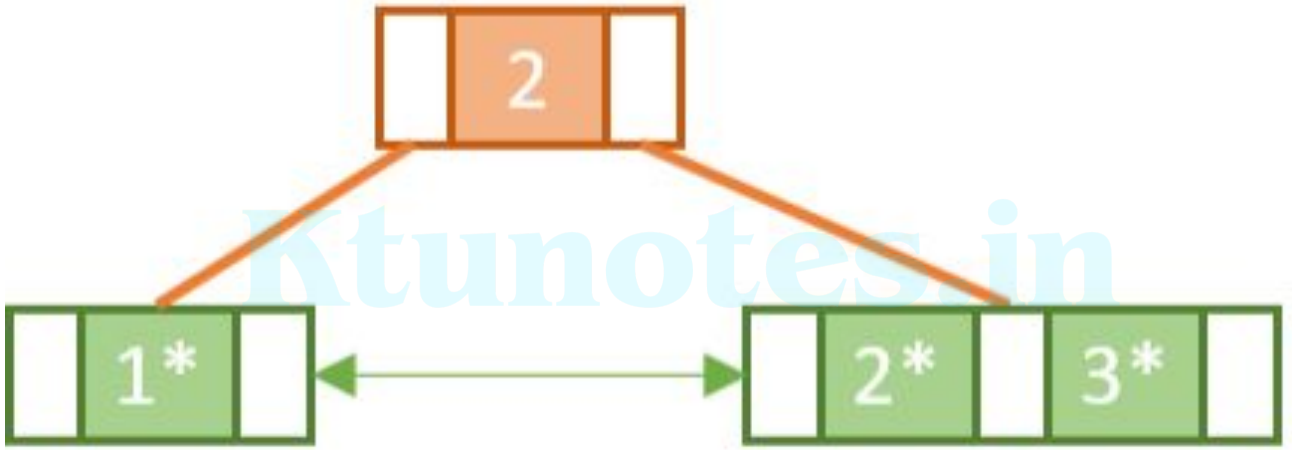


Figure 5.8: The nodes of a B+-tree. (a) Internal node of a B+-tree with $q - 1$ search values.
(b) Leaf node of a B+-tree with $q - 1$ search values and $q - 1$ data pointers.

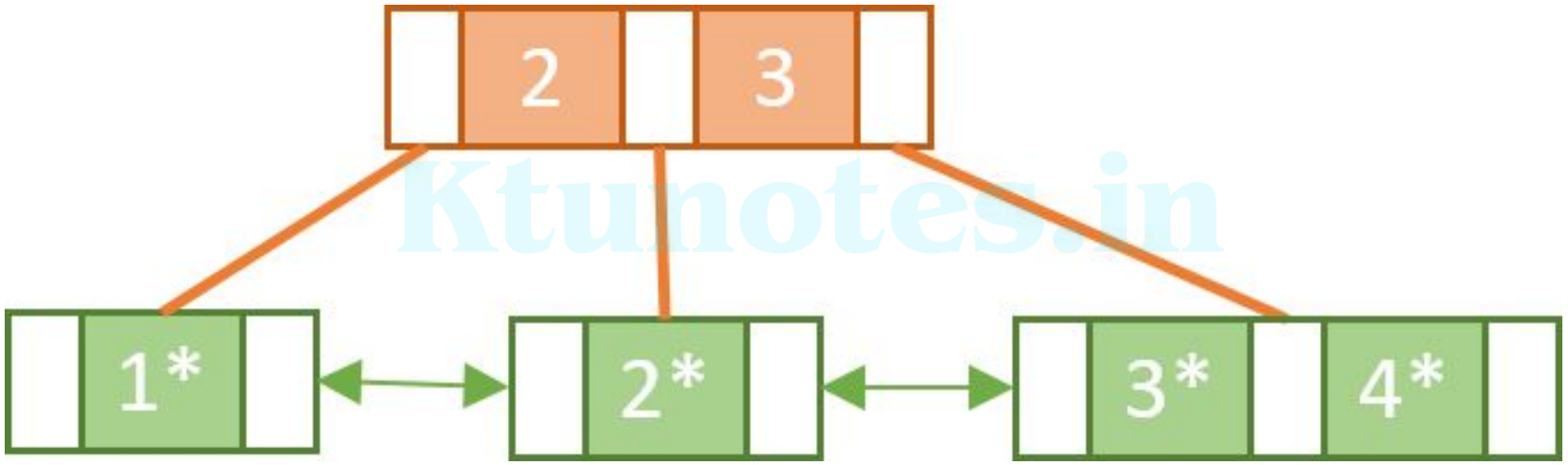
- When a leaf node is full and a new entry is inserted there, the node overflows and must be split.
- The first j = ⌈((pleaf + 1)/2)⌉ entries in the original node are kept there, and the remaining entries are moved to a new leaf node.
- The jth search value is replicated in the parent internal node, 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 Pj—the jth tree pointer after inserting the new value and pointer, where j = ⌊((p + 1)/2)⌋—are kept, while the jth search value is moved to the parent, not replicated.
- A new internal node will hold the entries from Pj+1 to the end of the entries in the node

Building a B+tree

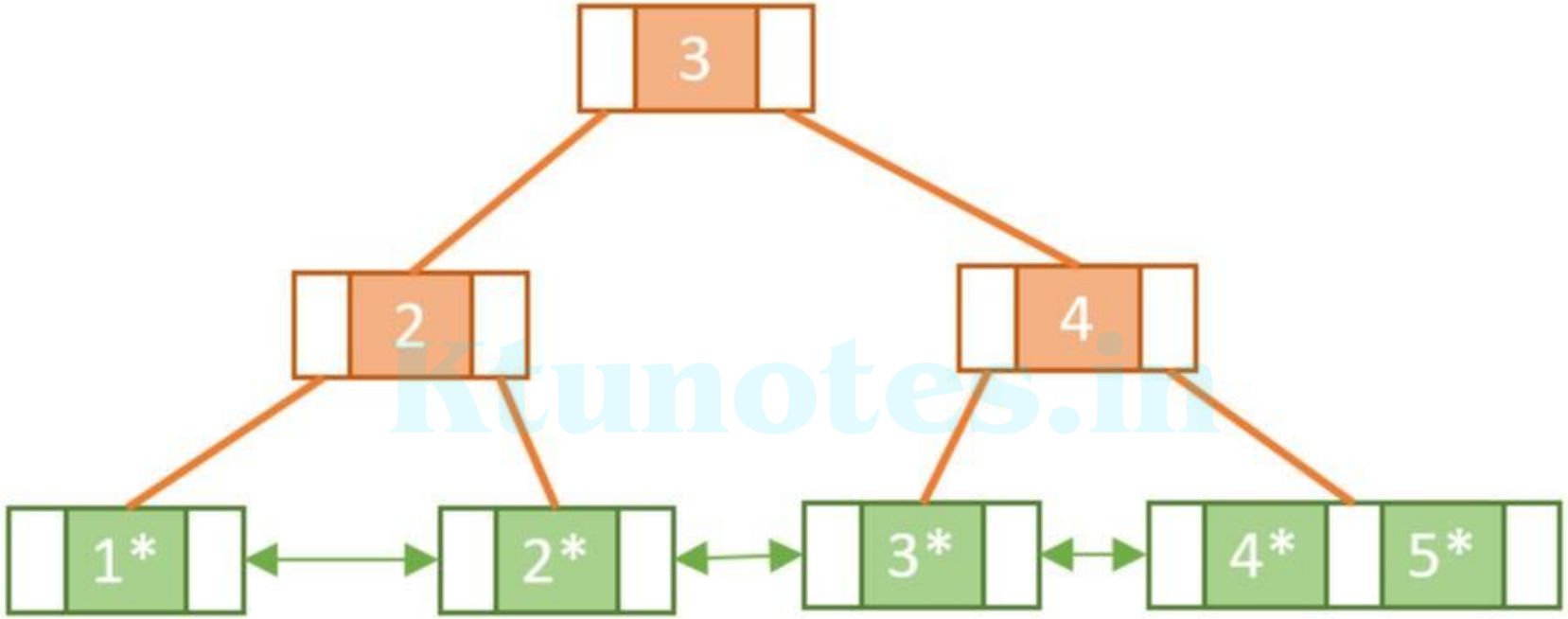
To start with, we'll insert Keys 1 and 2 into the root node in ascending order:



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:

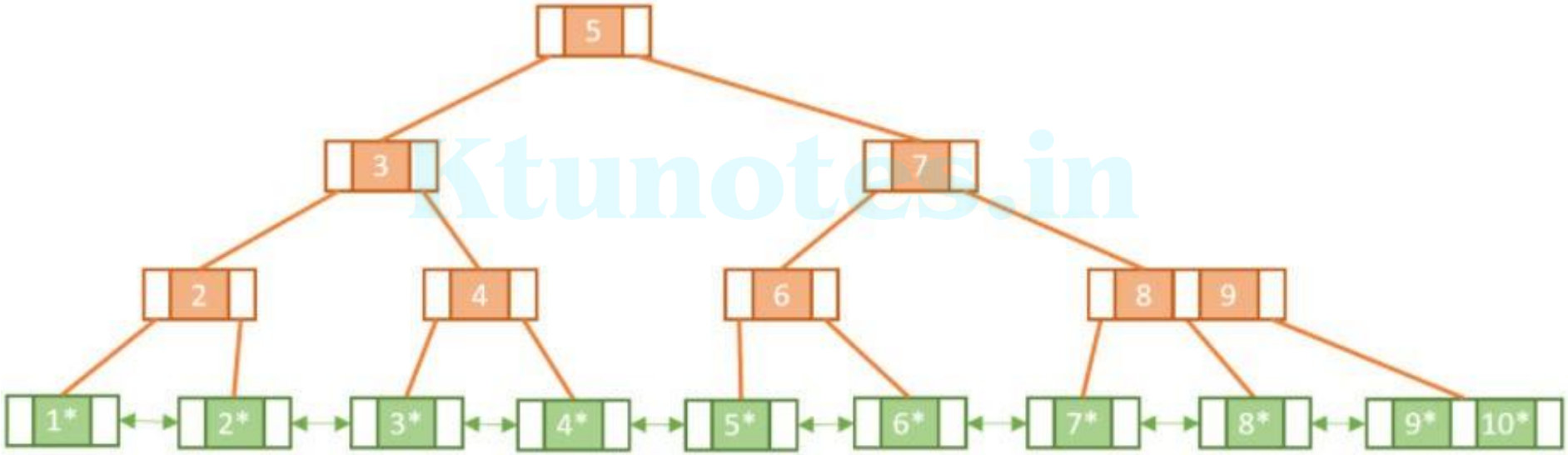


Next, we add Key 4 to the rightmost leaf node. Since it's full, we need to perform another a split operation and copy-up Key 3 to the root node:



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

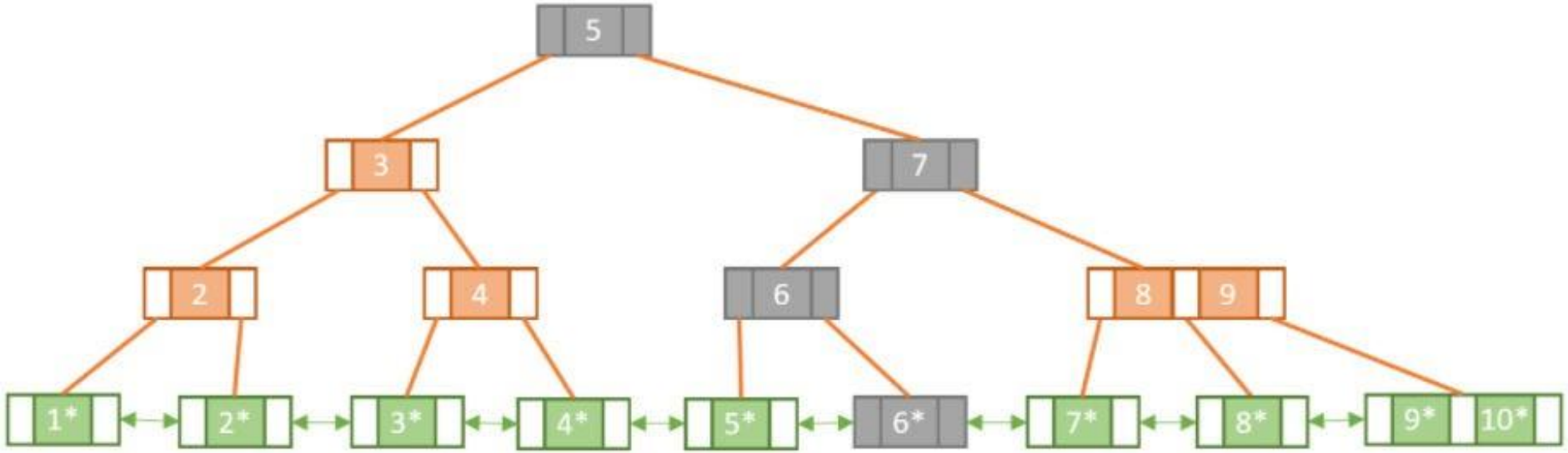
Notice the difference between splitting a leaf node and splitting an internal node. When we split the internal node in the second split operation we didn't copy-up Key 3.



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:

Searching a B+tree

- Searching for a specific key within a B+tree is very similar to searching for a key in a normal B-tree.
- Let's see what it would be like to search for Key 6 again but on the B+tree:



- The shaded nodes show us the path we have taken in order to find our match. Deduction tells us that searching within a B+tree means that we must go all the way down to a leaf node to get the satellite data. As opposed to B-trees where we could find the data at any level.
- In addition to exact key match queries, B+trees support range queries. This is enabled by the fact that the B+tree leaf nodes are all linked together. To perform a range query all we need to do is:
  - find an exact match search for the lowest key
  - and from there, follow the linked list until we reach the leaf node with the maximum key

Example of an Insertion in a B+-tree

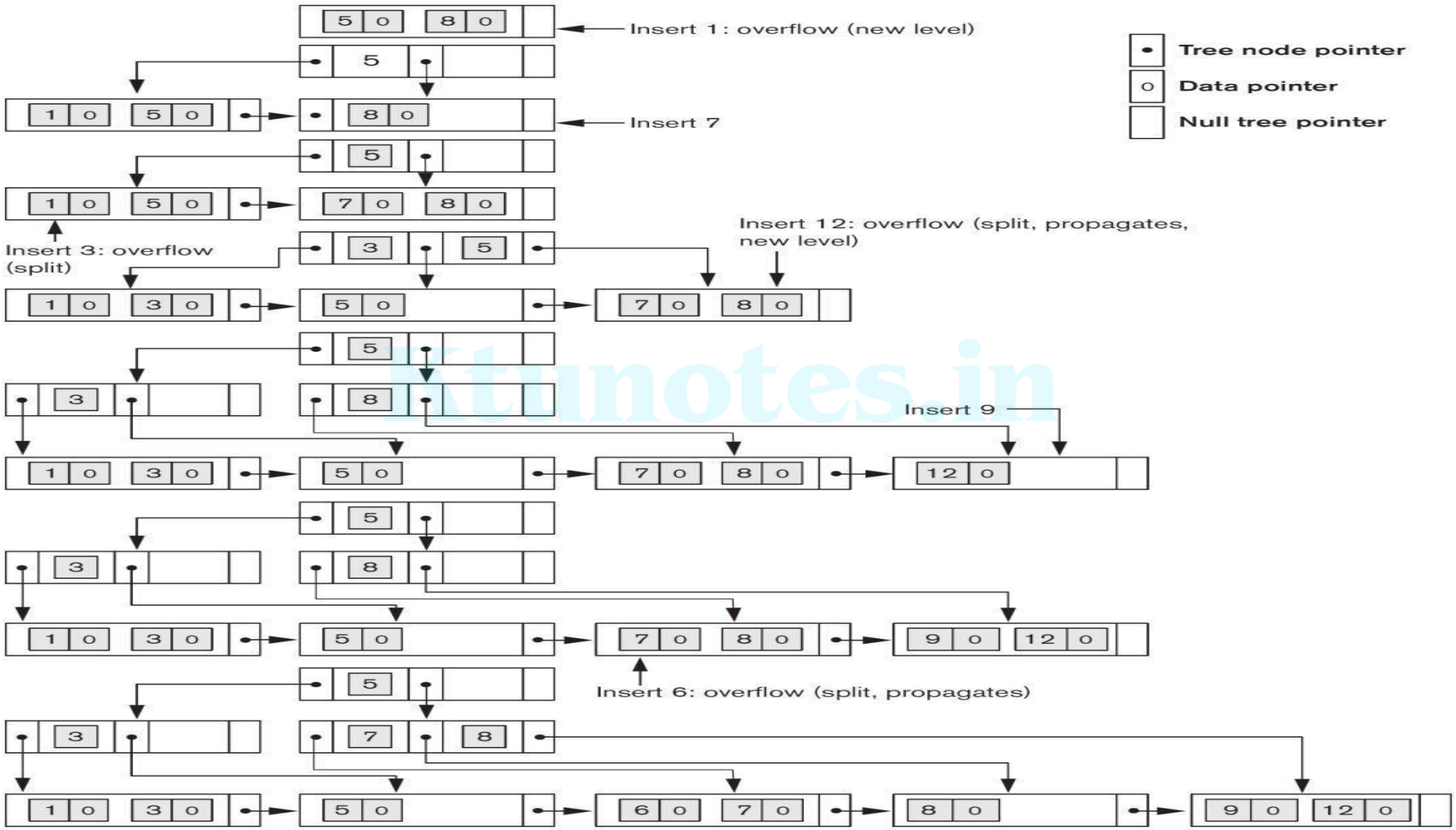


Figure 5.9: An example of insertion in a B+-tree with $p = 3$ and $p_{leaf} = 2$.
The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6

Example of a Deletion in a B +-tree

- When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there.
- In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree.
- Deletion may cause underflow by reducing the number of entries in the leaf node to below the minimum required.
- In this case, we try to find a sibling leaf node
  - a leaf node directly to the left or to the right of the node with underflow and
  - redistribute the entries among the node and its sibling so that both are at least half full;
  - otherwise, the node is merged with its siblings and the number of leaf

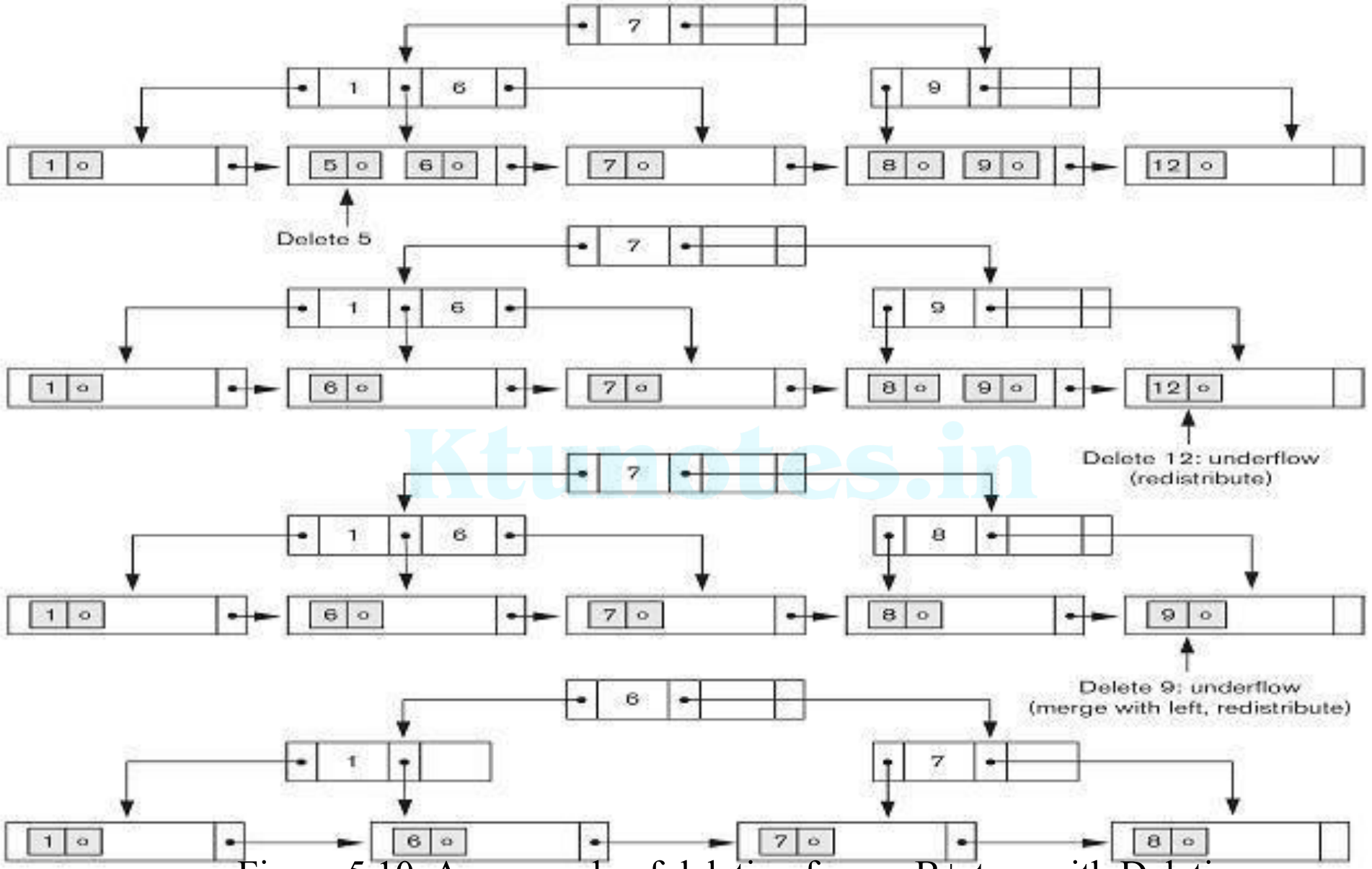


Figure 5.10: An example of deletion from a B+-tree with Deletion sequence: 5, 12, 9

- 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.

Query Optimization: Heuristics-based Query Optimization

- Query Optimization is the process of choosing a suitable execution strategy for processing a query.
- Process for heuristics optimization are:
  - The parser of a high-level query generates an initial internal representation.
  - Apply heuristics rules to optimize the internal representation.
  - A query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.
- The main heuristic is to apply first the operations that reduce the size of intermediate results.
- E.g, Apply SELECT and PROJECT operations before applying the JOIN or other binary operations.

Query tree

- A tree data structure that corresponds to a relational algebra expression.
- It represents the input relations of the query as leaf nodes of the tree, and represents the relational algebra operations as internal nodes.
- An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation.

Query graph

- A graph data structure that corresponds to a relational calculus expression.
- It does not indicate an order on which operations to perform first.
- There is only a single graph corresponding to each query.

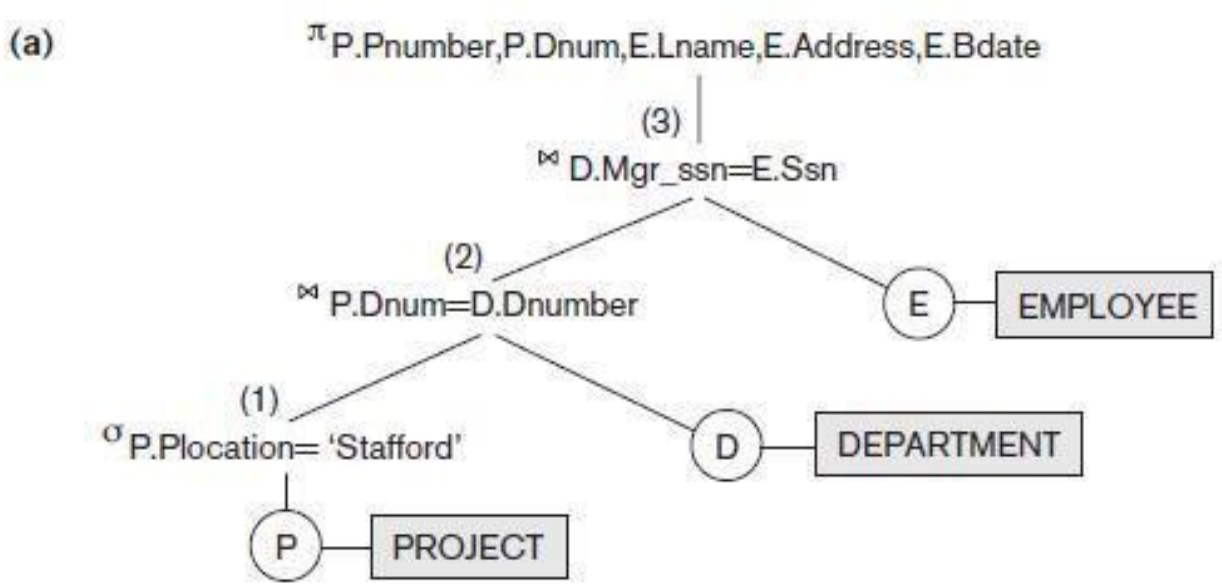
Example

- For every project located in 'Stafford', retrieve the project number, the controlling department number and the department manager's last name, address and birthdate.
- Relation algebra:

$$\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}(((\sigma_{PLOCATION='STAFFORD'}(PROJECT)) \quad DNUM=DNUMBER\ (DEPARTMENT)) \quad MGRSSN=SSN\ (EMPLOYEE))$$

SQL query:

```
Q2:   SELECT    P.NUMBER,P.DNUM,E.LNAME,
                E.ADDRESS, E.BDATE
      FROM      PROJECT AS P,DEPARTMENT AS D,
                EMPLOYEE AS E
      WHERE     P.DNUM=D.DNUMBER AND
                D.MGRSSN=E.SSN AND P.PLOCATION='STAFFORD';
```



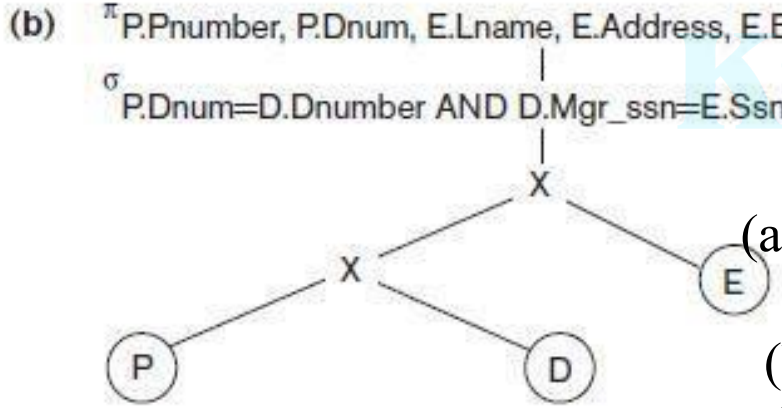
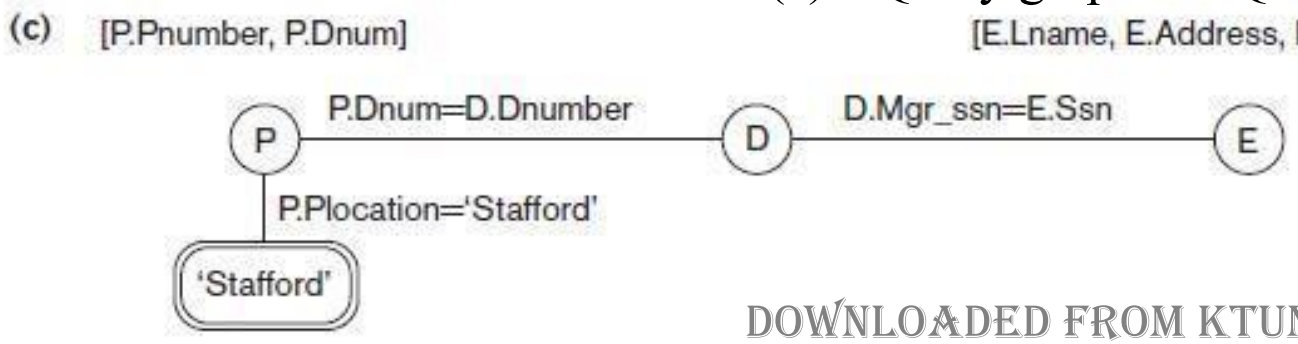


Figure 5.11: Two query trees for the query Q2.

(a)Query tree corresponding to the relational algebra expression for Q2.

(b) Initial (canonical) query tree for SQL query Q2.

(c) Query graph for Q2.



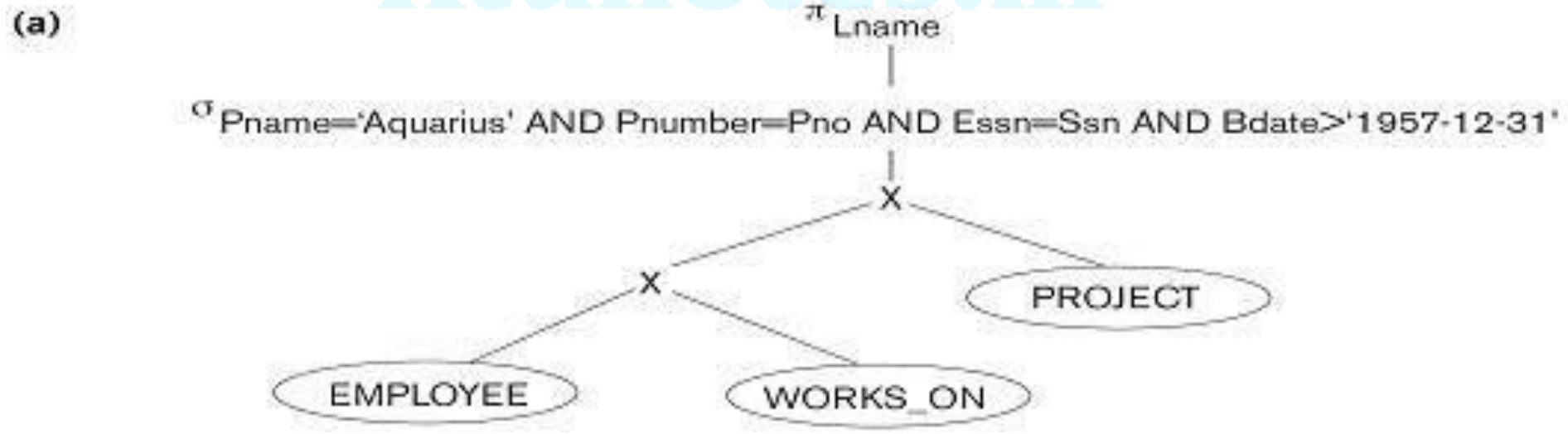
Heuristic Optimization of Query Trees

- The same query could correspond to many different relational algebra expressions and hence many different query trees.
- The task of heuristic optimization of query trees is to find a final query tree that is efficient to execute.

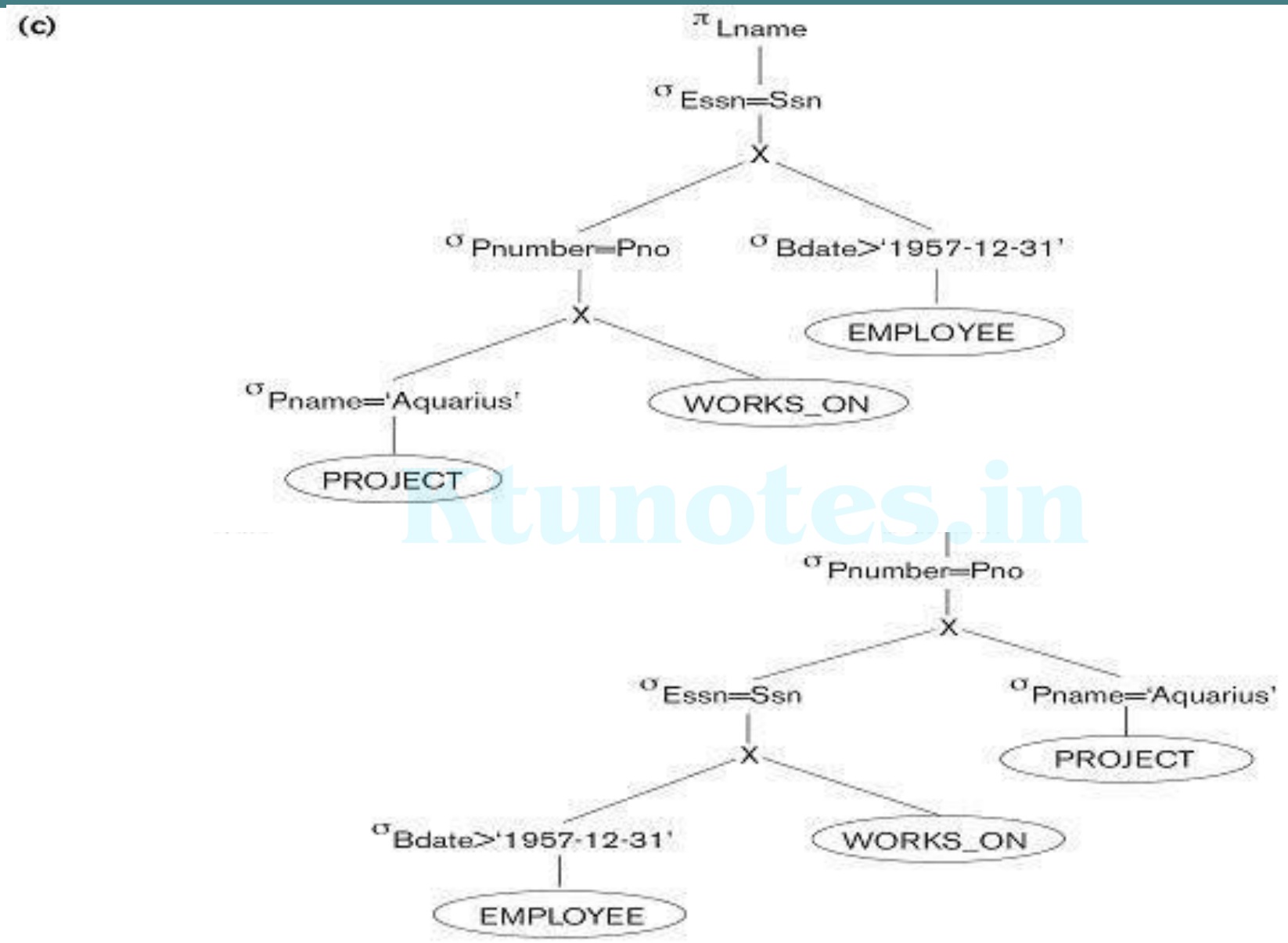
Example

```
Q:    SELECT    LNAME
          FROM      EMPLOYEE, WORKS_ON, PROJECT
          WHERE     PNAME = 'AQUARIUS' AND
                PNMUBER=PNO AND ESSN=SSN  AND BDATE > '1957-12-31';
```

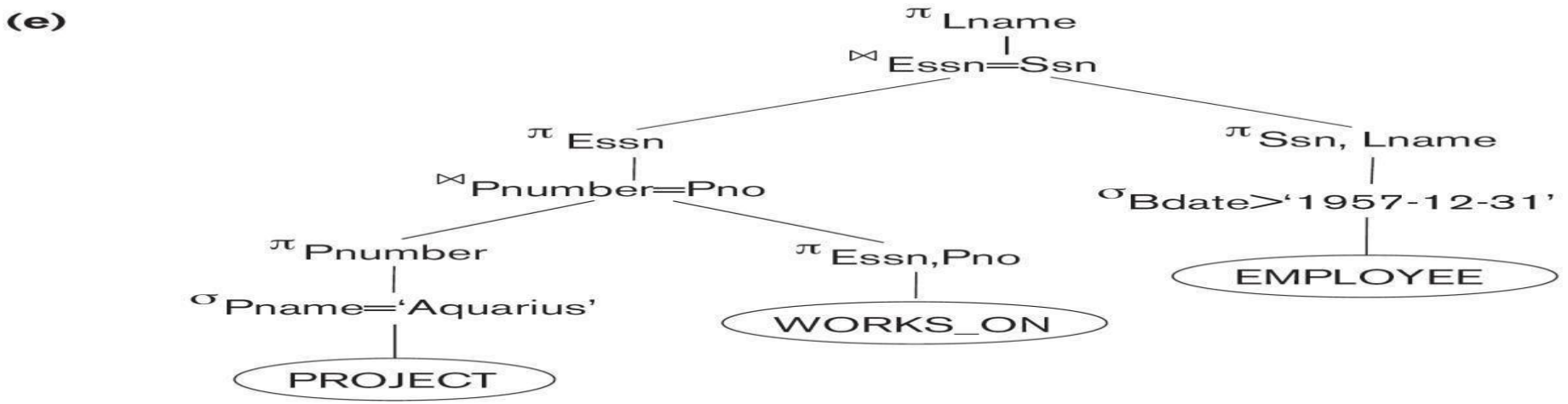
Figure 5.12: Steps in converting a query tree during heuristic optimization.



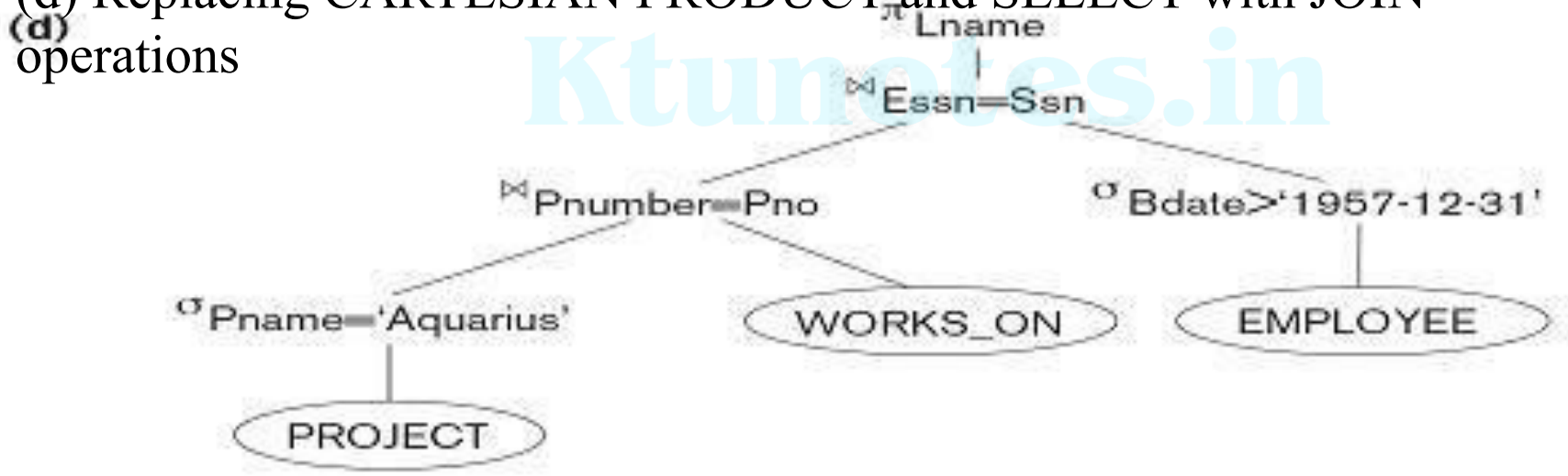
(a) Initial (canonical) query tree for SQL query Q.



(c) Applying the more restrictive SELECT operation first.



(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations



(e) Moving PROJECT operations down the query tree

Algebra Operations

1. **Cascade of σ** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$

2. **Commutativity of σ.** The σ operation is commutative:

   $$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$$

4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $$\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$$

5. **Commutativity of ⋈ (and ×).** The join operation is commutative, as is the × operation:

   $$R \bowtie_c S \equiv S \bowtie_c R$$

   $$R \times S \equiv S \times R$$

   Notice that although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning* is the same because the order of attributes is not important in the alternative definition of relation.

6. **Commuting σ with ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_c (R)) \bowtie S$$

Alternatively, if the selection condition c can be written as $(c_1$ AND $c_2)$, where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

$$\sigma_c (R \bowtie S) \equiv (\sigma_{c_1} (R)) \bowtie (\sigma_{c_2} (S))$$

The same rules apply if the ⋈ is replaced by a × operation.

7. **Commuting π with ⋈ (or ×).** Suppose that the projection list is $L = \{A_1, ..., A_n, B_1, ..., B_m\}$, where $A_1, ..., A_n$ are attributes of R and $B_1, ..., B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

$$\pi_L (R \bowtie_c S) \equiv (\pi_{A_1, ..., A_n} (R)) \bowtie_c (\pi_{B_1, ..., B_m} (S))$$

If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, ..., A_{n+k}$ of R and $B_{m+1}, ..., B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A_1, ..., A_n, A_{n+1}, ..., A_{n+k}} (R)) \bowtie_c (\pi_{B_1, ..., B_m, B_{m+1}, ..., B_{m+p}} (S)))$$

For ×, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with ×.

8. **Commutativity of set operations.** The set operations $\cup$ and $\cap$ are commutative but $-$ is not.

9. **Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$.** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

$$(R \ \theta \ S) \ \theta \ T \equiv R \ \theta \ (S \ \theta \ T)$$

10. **Commuting σ with set operations.** The σ operation commutes with $\cup$, $\cap$, and $-$. If θ stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c \ (R \ \theta \ S) \equiv (\sigma_c \ (R)) \ \theta \ (\sigma_c \ (S))$$

11. **The π operation commutes with $\cup$.**

$$\pi_L \ (R \cup S) \equiv (\pi_L \ (R)) \cup (\pi_L \ (S))$$

12. **Converting a $(\sigma, \times)$ sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ as follows:

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

There are other possible transformations. For example, a selection or join condition c can be converted into an equivalent condition by using the following standard rules from Boolean algebra (DeMorgan's laws):

$$\text{NOT} \ (c_1 \ \text{AND} \ c_2) \equiv (\text{NOT} \ c_1) \ \text{OR} \ (\text{NOT} \ c_2)$$

$$\text{NOT} \ (c_1 \ \text{OR} \ c_2) \equiv (\text{NOT} \ c_1) \ \text{AND} \ (\text{NOT} \ c_2)$$

Algorithm:

- Using rule 1, break up any select operations with conjunctive conditions into a cascade of select operations.
- Using rules 2, 4, 6, and 10 concerning the commutativity of select with other operations, move each select operation as far down the query tree as is permitted by the attributes involved in the select condition.
- Using rule 9 concerning associativity of binary operations, rearrange the leaf nodes of the tree so that the leaf node relations with the most restrictive select operations are executed first in the query tree representation.
- Using Rule 12, combine a Cartesian product operation with a subsequent select operation in the tree into a join operation.
- Using rules 3, 4, 7, and 11 concerning the cascading of project and the commuting of project with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new project operations as needed.
- Identify subtrees that represent groups of operations that can be executed by a single algorithm.

Summary of Heuristics for Algebraic Optimization

- The main heuristic is to apply first the operations that reduce the size of intermediate results.
- Perform select operations as early as possible to reduce the number of tuples and perform project operations as early as possible to reduce the number of attributes.
  - This is done by moving select and project operations as far down the tree as possible.
- The select and join operations that are most restrictive should be executed before other similar operations.
  - This is done by reordering the leaf nodes of the tree

Query Execution Plans

- An execution plan for a relational algebra query consists of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.
- Materialized evaluation: the result of an operation is stored as a temporary relation.
- Pipelined evaluation:
  - as the result of an operator is produced, it is forwarded to the next operator in sequence.

END